A Global Communication Optimization Technique Based on Data-Flow Analysis and Linear Algebra

M. KANDEMIR* P. BANERJEE† A. CHOUDHARY‡ J. RAMANUJAM§ N. SHENOY¶

Abstract

Reducing communication overhead is extremely important in distributed-memory message-passing architectures. In this paper, we present a technique to improve communication that considers data access patterns of the entire program. Our approach is based on a combination of traditional data-flow analysis and a linear algebra framework, and works on structured programs with conditional statements and nested loops but without arbitrary goto statements. The distinctive features of the solution are the accuracy in keeping communication set information, support for general alignments and distributions including block-cyclic distributions and the ability to simulate some of the previous approaches with suitable modifications. We also show how optimizations such as message vectorization, message coalescing and redundancy elimination are supported by our framework. Experimental results on several benchmarks show that our technique is effective in reducing the number of messages (an average of 32% reduction), the volume of the data communicated (an average of 37% reduction), and the execution time (an average of 26% reduction).

Keywords: communication optimizations, message vectorization, distributed memory machines, parallelism, data-flow analysis, global optimizations.

*Dept. Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY 13244. (mtk@ece.nwu.edu)
†Dept. Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208. (banerjee@ece.nwu.edu)
‡Dept. Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208. (choudhar@ece.nwu.edu)
§Dept. Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. (jxr@ee.lsu.edu)
¶Dept. Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208. (nagaraj@ece.nwu.edu)

1 Introduction

Distributed memory multiprocessors such as the IBM SP-2 and the Intel Paragon are attractive for high performance computing in that they offer potentially high levels of flexibility, scalability and performance. But the need for explicit message passing resulting from the lack of a globally shared address space renders programming these machines a difficult task. The main objective behind the efforts such as High Performance Fortran (HPF) [39] and Fortran D [30] is to raise the level of programming by allowing the user to write programs with a shared address space view augmented with directives that specify data mapping. The compilers for such languages are responsible for partitioning the computation, inserting the necessary commands that implement the required message passing for access to non-local data.

On such machines, the time (cost) to access non-local data is usually orders of magnitude higher than accessing local data. For example, on the Intel Paragon the processor cycle time is 20 nanoseconds whereas the remote memory access time is between $10,000$ and $30,000$ nanoseconds depending on the distance between communicating processors [29]. Therefore, it is imperative that the frequency and volume of non-local accesses are reduced as much as possible. In particular, in message-passing programs, the startup cost for the messages can easily dominate the execution time. For example, on the Intel Paragon the message startup time is approximately $1,720$ times the transfer time per word; in the IBM SP-2 this figure is around 360 [16]. These figures indicate that optimizing communication is very important. Several software efforts have been aimed at reducing the communication overhead. The main goal of these optimizations is to increase the performance of programs by combining messages in various ways to reduce the overall communication overhead. The most common optimization technique used by previous researchers is message vectorization [51, 8, 30, 11, 10]. In message vectorization, instead of naively inserting *send* and *recv* operations just before references to non-local data, communication is hoisted to outer loops. Essentially this optimization replaces many small messages with one large message, thereby reducing the number of messages. For example, consider the program fragment shown in Figure 1(a) and assume that all arrays are distributed across processors block-wise in the second dimension. Figures 1(b) and (c) show naively inserted messages and message vectorization respectively, for a processor `p` before loop bounds reduction (a technique to allow processors to execute only those iterations which have assignments that write to local memory [30]) and guard insertion (a technique that guarantees correct execution of statements within loop nests). The notation `send{B,q,n}` means that `n` elements of array `B` should be sent to processor `q`; `recv{B,q,n}` is defined similarly. For this discussion, we are not concerned with exactly which elements are sent and received. Notice that the version in Figure 1(c) reduces the message startup cost as well as the latency. Some of the researchers [30, 2] also considered message coalescing which is a technique that combines messages due to different references to the same array, and message aggregation which combines messages due to references to different arrays to the same destination processor into a single message. In general, due to private physical memory spaces, generating communication code for message-passing architectures might be very difficult, because it requires the correct non-local elements to get transferred to the memories of the processors that will use them. Optimizing compilers for data-parallel languages automate this time consuming task of deriving node programs based on the data distribution specified by the programmer.

The main problem with the optimizations mentioned above is that they optimize communication for a single nest at a time. This restriction prevents a compiler from performing inter-loop optimizations such as global elimination of redundant communication. To see this, consider Figure 1(d) which shows the global optimization of the same program fragment via elimination of redundant communication. Notice that, compared with the message-vectorized program in Figure 1(c), this version reduces both the number of messages and the communication volume.

Recently a number of authors have proposed techniques based on data-flow analysis to optimize communication across multiple loop nests [18, 22, 35, 13, 49, 50]. Most of these approaches use a variant of Regular Section Descriptors (RSD) introduced by Callahan and Kennedy [12]. Two most notable representations are Available Section Descriptor (ASD) [22] and Section Communication Descriptor (SCD) [49, 50]. Associated with each array that is referenced in the program is an RSD that describes the portion of the array being referenced. Although this representation is convenient for simple array sections such as those found in pure block or cyclic distributions, it is hard to embed alignment and general distribution information into it. Apart from inadequate support for block-cyclic distributions, working with section descriptors may some-

```
DO j = 2, 255
 DO i = 1, 255
  A(i,j)=B(i,j)+B(i,j-1)
 END DO
END DO


DO j = 2, 255
 DO i = 2, 256
  C(i,j)=B(i,j-1)+C(i,j)
 END DO
END DO
```

(a)

```
DO j = 2, 255
 DO i = 1, 255
  send {B,p+1,1}, recv {B,p-1,1}
  A(i,j)=B(i,j)+B(i,j-1)
 END DO
END DO

DO j = 2, 255
 DO i = 2, 256
  send {B,p+1,1}, recv {B,p-1,1}
  C(i,j)=B(i,j-1)+C(i,j)
 END DO
END DO
```

(b)

```
send {B,p+1,255}, recv {B,p-1,255}
DO j = 2, 255
 DO i = 1, 255
  A(i,j)=B(i,j)+B(i,j-1)
 END DO
END DO

send {B,p+1,255}, recv {B,p-1,255}
DO j = 2, 255
 DO i = 2, 256
  C(i,j)=B(i,j-1)+C(i,j)
 END DO
END DO
```

(c)

```
send {B,p+1,256}, recv {B,p-1,256}
DO j = 2, 255
 DO i = 1, 255
  A(i,j)=B(i,j)+B(i,j-1)
 END DO
END DO


DO j = 2, 255
 DO i = 2, 256
  C(i,j)=B(i,j-1)+C(i,j)
 END DO
END DO
```

(d)

Figure 1: (a) A code fragment. (b) Naive communication placement. (c) Message vectorization. (d) Global communication optimization.

times result in overestimation of the communication sets, since regular sections are not closed under union and difference operators. The resulting inaccuracy may be linear with the number of data-flow formulations to be evaluated, thus defeating the purpose of global communication optimization.

This problem can be illustrated using the program fragment given in Figure 2(a) assuming that arrays `X` and `Y` are distributed block-wise across two processors, `0` and `1`. The RSDs corresponding to these two communications are also shown next to the loop statements. Notice that all communication is from processor `0` to processor `1`. The problem here is that a data-flow approach based on RSDs to combine these communications will be unable to represent the combined communication as an RSD. This means that even if all the communication can be hoisted above the i loop, the two communications can only be concatenated resulting in redundant communication as these two sets have some common elements. Moreover, since the commu-

```
real X(1000), Y(1000)

DO t = 1, T
  DO i = 0, 49 -------------> S_i = (1:200:4)
    Y(i+500) = X(4*i+1)
  END DO

  DO j = 17, 99 -------------> S_j = (50:300:3)
    Y(j+500) = X(3*j-1)
  END DO

  DO i = 1, 1000
    X(i) = f(Y(i),X(i))
  END DO
END DO
```

(a)

```
for (i = 1; i <= 49; i += 4) {
  process_element(i);
}
for (i = 50; i <= 197; i++) {
  if (Mod(i-1,4) == 0) {
    process_element(i);
  }
  if (Mod(i+1,3) == 0
       && -i-16 <= 12*Div(-i-10,12)) {
    process_element(i);
  }
}
for (i = 200; i <= 299; i += 3) {
  process_element(i);
}
```

(b)

Figure 2: (a) An example code fragment that shows the shortcomings of RSDs. (b) Code generated by using the Omega to enumerate the communication set in (a). `process_element()` is an implementation specific function to handle enumerated elements.

nication cannot be taken out of t loop because of a data dependence [52, 48], the redundant communication will occur `T` times.

On the other hand, we represent these sets in our framework as $S_i := \{[\mathtt{d}] : \exists(\alpha : \mathtt{d} = 1 + 4\alpha \text{ and } 1 \leq \mathtt{d} \leq 197)\}$ and $S_j := \{[\mathtt{d}] : \exists(\alpha : 1 + \mathtt{d} = 3\alpha \text{ and } 50 \leq \mathtt{d} \leq 299)\}$. Then by using the Omega library [31], we derive the code shown in Figure 2(b) which can enumerate all the elements in $S_i + S_j$. As a result, each element will be communicated once and only once. It should be stressed that the same problem with RSDs can occur with set difference ($-$) operations. For instance, the RSD difference between `(1:1000:3)` and `(1:1000:7)` cannot be represented as a single RSD. Unfortunately, the inaccuracies originating from the union (and difference) operations on the RSDs accumulate as the data-flow process proceeds, making the final communication sets imprecise.

In this paper, we make the following contributions:

(1) We show that the problem of *global* communication optimization for *regular* scientific codes can be cast in a linear algebra framework. This allows the compiler to easily apply traditional loop-based optimization techniques such as message vectorization, message coalescing, message aggregation as well as global optimizations such as redundant communication elimination and communication hoisting.

(2) We present two different approaches, primarily for hoisting communication and minimizing the number of messages, respectively, that are aimed at reducing communication overhead and show the trade-off between these two. Both these approaches are accurate; using the linear algebra framework proposed by Ancourt et al. [7], they are able to handle the optimization problem at the granularity of individual array elements.

(3) We show that the global communication sets resulting from our analysis can be enumerated by our use of the Omega library [42, 31] from the University of Maryland. Although the Omega library works on the Presburger formulas and the best known asymptotic upper bound of any algorithm for verifying the Presburger formulas is $O(2^{2^{2^n}})$, the library is much more efficient for the practical cases that arise in compilation.

(4) We compare our approach both qualitively and quantitively to the previous work which focused on a single loop nest at a time.

The remainder of this paper is organized as follows. Section 2 briefly describes some important concepts such as control flow graphs, interval analysis, dependence analysis and the linear algebra framework used throughout the paper. We present our approach in detail in Section 3 and show how it uses both the linear algebra framework and data-flow analysis. Section 4 discusses the effect of hoisting communication vis-a-vis reducing the number of messages. In Section 5, we present details of communication generation. Section 6 reports experimental results on a 16 node IBM SP-2 distributed-memory message-passing machine and shows that our technique is effective in reducing number of communication messages, volume of communication and execution time. Section 7 discusses related work and Section 8 concludes the paper.

2 Preliminaries

The main idea of this work is to show that a global communication optimization problem can be put into a linear algebra framework and that doing so might be useful in practice. Our approach gives the compiler the ability to represent communication sets globally as equalities and inequalities as well as to use polyhedron scanning techniques to perform optimizations such as redundant communication elimination and global message coalescing which were not possible under the loop-nest based communication optimization schemes. The following subsections give information about the basic concepts used throughout the paper.

2.1 Control Flow Graph (CFG)

We concentrate on structured programs with conditional statements and nested loops but without arbitrary goto statements. Our technique, however, can be extended to deal with jumps out of loops as well. We assume that array subscript functions, loop bounds and conditional expressions are affine functions of enclosing loop indices and symbolic constants. We also assume that the number of processors is known beforehand.

A *basic block* is a sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end without the possibility of branching except may be at the end [4]. A *control flow graph* (CFG) is a directed graph constructed by basic blocks and represents the flow-of-control information of the program.

For our purposes, the CFG can be thought of as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where each $v \in \mathcal{V}$ represents either a basic block or a (reduced) interval that represents a loop, and each $e \in \mathcal{E}$ represents an edge between blocks. In this paper, depending on the context, we use the term *node* interchangeably for a statement, a block or an interval. Two unique nodes s and t denote the start and terminal nodes, respectively, of a CFG. One might think of these nodes as dummy statements. It is assumed that every node $n \in \mathcal{V}$ lies on a path from s to t. We define the sets of all successors and predecessors of a node n as $succ(n) = \{m \mid (n, m) \in \mathcal{E}\}$ and $pred(n) = \{m \mid (m, n) \in \mathcal{E}\}$, respectively. We say node i *dominates* node j in the CFG, if every path from s to j goes through i. We write this relation as $j \in dom(i)$. The CFGs we consider have the following properties in addition: (a) empty *else* branches are added to *if/endif* constructs; (b) all the non-local references in the *loop bounds* and *if-conditions* are taken just above the respective constructs; and (c) like [22], any edge that goes directly from a block with more than one successor, to a block with more than one predecessor is split. This last transformation, shown in Figure 3,

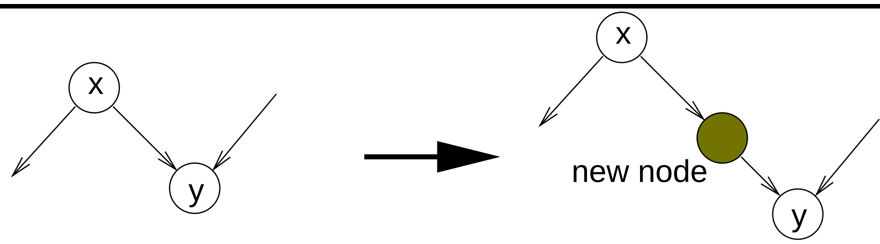

Figure 3: An example application of the edge-split transformation to eliminate critical edges-the edges going from a node with more than one successor to a node with more than one predecessor.

eliminates all critical edges [38].

2.2 Interval Analysis

We assume that prior to our analysis, the compiler has performed all loop level transformations [9, 48, 52] to enhance parallelism (e.g., loop permutation, loop distribution) and optimize communication. Our technique is based on *interval analysis* performed on the CFG. As explained in [5], the interval analysis consists of a *contraction* phase and an *expansion* phase. For programs written in a structured language, an interval corresponds to a loop, and there is a well defined algorithm to partition a CFG into disjoint intervals [4]. We use a version of the interval detection algorithm that identifies Tarjan's intervals [45].

The contraction phase collects information about what is generated and what is killed inside each interval. Then the interval is reduced to a single node and annotated with the information collected. This is a recursive procedure and stops when the reduced CFG contains no more cycles. In other words, the main purpose of this phase is to percolate the influence of each node to the outside into an increasingly more global context.

After the contraction phase, the expansion phase is run. In each step of this phase, a node (reduced interval) is expanded, and the information regarding the nodes in that interval is computed. In our case, at each step of the expansion phase, communication required for the intervals (loops) is determined.

Figure 4 shows the two phases of the interval analysis for an example CFG. In this figure, as shown by the dashed arrows, the contraction phase proceeds from left to right, whereas the expansion phase proceeds in the reverse direction. As an example, the block marked with $3, 4$ represents an interval (a loop) containing

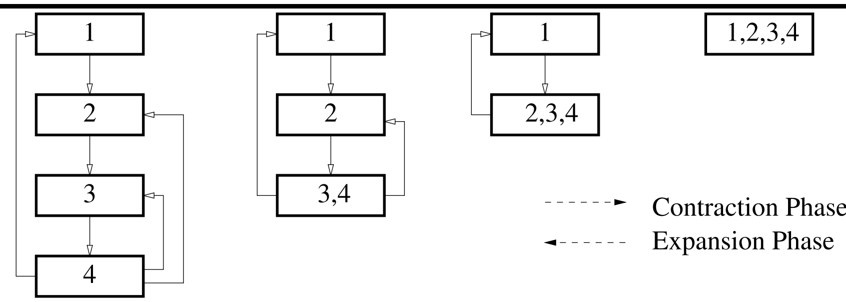

Figure 4: An example application of interval analysis based on Tarjan intervals. First, the contraction phase is run and then the expansion phase is executed.

blocks 3 and 4. It is also possible to adapt our approach to work with interval-flow graph, which is basically a CFG with an interval structure imposed on it [27, 28, 37].

It should be noted that since we assume that our input programs are structured, irreducible (intermediate) CFGs [4] can not occur during our analysis.

2.3 Data Dependence

Let S_x and S_y be two statements (not necessarily distinct) enclosed by nested loops. A *data dependence* determines which iterations of the loops can be executed in parallel. A *flow dependence* exists from statement S_x to statement S_y if S_x writes a value that is subsequently (in sequential execution) read by S_y. Such a dependence implies that instances of S_x and S_y must execute as if some of the nest levels must be executed sequentially. An *anti-dependence* exists between S_x and S_y if S_x reads a value that is subsequently modified by S_y. An *output dependence* exists between S_x and S_y if S_x writes a value that is subsequently written by S_y as well. Data dependences are *loop-independent* if the accesses to the same memory location occur in the same loop iteration; if the accesses occur in different loop iterations they are said to be *loop-carried*. Note that in that case not all loop nest levels need to contribute to the dependence. The outermost loop level that contributes the dependence is said to *carry* that dependence. In-depth discussion of data dependence analysis techniques is beyond the scope of this paper and can be found elsewhere [48, 52].

2.4 Linear Algebra Framework

HPF-like languages provide compiler directives that allow the user to perform data allocation onto local memories. The compiler then uses these distribution directives to partition computation across processors. It has been shown in [7] that linear algebra provides a powerful framework to generate code for distributed-memory message-passing machines, taking into account compiler directives.

Most of the compilers for distributed-memory message-passing machines use the *owner-computes* rule, which simply assigns each computation to the processor that owns the data being computed [30, 51, 8]. In this paper, we also assume the owner-computes rule; our framework, however, can be modified to handle the cases where this rule is relaxed. In such cases, the LHS references can also introduce communication. For clarity of the presentation, we do not consider relaxing the owner-computes rule in this paper.

Our approach uses the affine framework introduced by Ancourt et al. [7]. In this framework, data arrays, templates and processors are all declared as Cartesian grids as in HPF [39]. The data arrays are first aligned to templates and then these templates are distributed across the memories of the processors. Consider the following program fragment under a compilation scheme based on HPF-like directives and the owner-computes rule. A cyclic($\mathcal{C}$) attribute indicates that the template (or array) dimension in question will be partitioned into blocks of size $\mathcal{C}$ and these are assigned to processors in a round-robin fashion. The block and cyclic(1) are just two common cases for the general cyclic($\mathcal{C}$) distribution.

```
       real X(a_l:a_u)
!HPF$  template T(t_l:t_u)
!HPF$  processors PROC(p_l:p_u)
!HPF$  align X(j) with T(α*j+β)
!HPF$  distribute T(cyclic(C)) onto PROC

       DO i = i_l, i_u
          X(γ_L*i+θ_L) = ··· X(γ_R*i+θ_R) ···
       END DO
```

Let $\mathcal{R_L}$ = $\mathtt{X}(\gamma_L{*}i{+}\theta_L)$ and $\mathcal{R_R}$ = $\mathtt{X}(\gamma_R{*}i{+}\theta_R)$. In the rest of the paper, for the sake of simplicity, we will sometimes refer to the subscript expressions as data (array) elements when the intention is clear. Assuming p and q denote two processors, we define the following sets.

$$
\begin{aligned}
\mathtt{Own}(\mathtt{X},\mathtt{q}) &= \{\mathtt{d} \mid \mathtt{d} \in \mathtt{X} \text{ and is owned by } \mathtt{q}\} \\
\mathtt{Compute}(\mathtt{X},\mathcal{R_L},\mathtt{q}) &= \{i \mid \gamma_L * i + \theta_L \in \mathtt{Own}(\mathtt{X},\mathtt{q}) \text{ and } i_l \le i \le i_u\} \\
\mathtt{View}(\mathtt{X},\mathcal{R_R},\mathtt{q}) &= \{\mathtt{d} \mid \exists \imath \text{ st. } \imath \in \mathtt{Compute}(\mathtt{X},\mathcal{R_L},\mathtt{q}) \text{ and } \mathtt{d} = \mathtt{X}(\gamma_{\mathcal{R}} * \imath + \theta_{\mathcal{R}}) \text{ and } i_l \le \imath \le i_u\} \\
\mathtt{CommSet}(\mathtt{X},\mathcal{R_R},\mathtt{p},\mathtt{q}) &= \mathtt{Own}(\mathtt{X},\mathtt{q}) \cap \mathtt{View}(\mathtt{X},\mathcal{R_R},\mathtt{p}).
\end{aligned}
$$

Intuitively, the set Own(X,q) refers to the elements mapped onto processor q through compiler directives. The similar Own sets are defined for other arrays as well. The set of iterations to be executed by q due to a LHS reference $\mathcal{R_L}$ is given by Compute(X,$\mathcal{R_L}$,q). Of course, during the execution of this local iteration set, some elements (local or non-local) denoted by the RHS reference $\mathcal{R_R}$ will be required; the set View(X,$\mathcal{R_R}$,q) defines these elements. Finally, CommSet(X,$\mathcal{R_R}$,p,q) defines the elements that should be communicated from processor q to processor p due to reference $\mathcal{R_R}$.

It should be noted that in general there may be more than one RHS reference, and the computation may involve multi-dimensional arrays and a multi-level nest in which case d and i denote data and iteration vectors respectively. Also in the most general case, α, γ_L and γ_R are matrices, and β, θ_L and θ_R are vectors.

The definition of the Own set above is rather informal. For a more precise definition, we take into account the block-cyclic distribution and define the Own set as

$$
\begin{aligned}
\mathtt{Own}(\mathtt{X},\mathtt{q}) = \{\mathtt{d} \mid \exists \mathtt{t},\mathtt{c},\mathtt{l} \text{ such that } \mathtt{t} = \alpha * \mathtt{d} + \beta \text{ and } \mathtt{t} = \mathcal{C} * \mathtt{P} * \mathtt{c} + \mathcal{C} * \mathtt{q} + \mathtt{l} \\
\text{and } a_l \le \mathtt{d} \le a_u \text{ and } p_l \le \mathtt{q} \le p_u \text{ and } t_l \le \mathtt{t} \le t_u \text{ and } 0 \le \mathtt{l} \le \mathcal{C} - 1\},
\end{aligned}
$$

where P = $p_u - p_l + 1$. In this formulation, $\mathtt{t} = \alpha * \mathtt{d} + \beta$ represents alignment information and $\mathtt{t} = \mathcal{C} * \mathtt{P} * \mathtt{c} + \mathcal{C} * \mathtt{q} + \mathtt{l}$ denotes the distribution information. In other words, each array element d is mapped onto a point in a local two-dimensional array. This point can be represented by a pair (c,l) and gives the local address of the data item in a processor. Simple `block` and `cyclic(1)` distributions can easily be handled within this framework by setting $\mathtt{c} = 0$ and $\mathtt{l} = 0$, respectively. As an example, Figure 5(a) shows the global and local addresses of a one-dimensional array distributed in block-cyclic manner across three processors with $\mathcal{C} = 4$. Figures 5(b) and (c), on the other hand, illustrate two-dimensional views of the global and local addresses, respectively. For each processor, the horizontal dimension corresponds to `c` coordinate whereas the vertical dimension denotes `l`. For example, the 55^{th} element of the (global) array is mapped onto Processor 1 with $\mathtt{c} = 4$ and $\mathtt{l} = 3$ as local coordinates.

The relation $\mathtt{t} = \alpha * \mathtt{d} + \beta$ can be generalized by adding a replication matrix $\mathcal{V}$ which eliminates the replicated dimension from the equations: $\mathcal{V} * \mathtt{t} = \alpha * \mathtt{d} + \beta$. In the case where no replication is specified, $\mathcal{V}$ is the identity matrix. Also, in order to take the collapsed dimensions (the dimensions that are not distributed across processors) into account, another projection matrix $\mathcal{Y}$ can be used: $\mathcal{Y} * \mathtt{t} = \mathcal{C} * \mathtt{P} * \mathtt{c} + \mathcal{C} * \mathtt{q} + \mathtt{l}$. All the elements on a collapsed dimension are stored on the same processor. Notice that these projection matrices are only useful if we adhere to a matrix form for describing the relations. We do not need them if the relations are described on a per dimension basis. In the rest of the paper we assume that identity alignment is used and arrays are directly distributed across processors. For an in-depth discussion of the linear algebra framework for compiling distributed-memory programs, we refer the reader to Ancourt et al. [7].

2.5 Parafrase-2 and Omega Library

Parafrase-2 [41] is used as the front end in our compilation framework. It is a parallelizing compiler implemented as a source to source code restructurer that consists of several passes for analysis, transformation, parallelism detection and code generation. In order to obtain the loops that enumerate the elements in the ownership and communication sets, we use the Omega library [31]. This library is essentially a set of C++ classes for manipulating integer tuple relations and sets defined using Presburger formulas. We implemented a framework that obtains data access information from Parafrase-2 internal structures and feeds them into the Omega library; when all the required sets have been obtained the framework converts these sets back to

Processor 0				Processor 1				Processor 2			
0^{0}	1^{1}	2^{2}	3^{3}	4^{0}	5^{1}	6^{2}	7^{3}	8^{0}	9^{1}	10^{2}	11^{3}
12^{4}	13^{5}	14^{6}	15^{7}	16^{4}	17^{5}	18^{6}	19^{7}	20^{4}	21^{5}	22^{6}	23^{7}
24^{8}	25^{9}	26^{10}	27^{11}	28^{8}	29^{9}	30^{10}	31^{11}	32^{8}	33^{9}	34^{10}	35^{11}
36^{12}	37^{13}	38^{14}	39^{15}	40^{12}	41^{13}	42^{14}	43^{15}	44^{12}	45^{13}	46^{14}	47^{15}
48^{16}	49^{17}	50^{18}	51^{19}	52^{16}	53^{17}	54^{18}	55^{19}	56^{16}	57^{17}	58^{18}	59^{19}
60^{20}	61^{21}	62^{22}	63^{23}	64^{20}	65^{21}	66^{22}	67^{23}	68^{20}	69^{21}	70^{22}	71^{23}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

(a) Global and local addresses. Superscripts denote local addresses.

Processor 0				Processor 1				Processor 2			
$0,0$	$0,1$	$0,2$	$0,3$	$0,4$	$0,5$	$0,6$	$0,7$	$0,8$	$0,9$	$0,10$	$0,11$
$1,0$	$1,1$	$1,2$	$1,3$	$1,4$	$1,5$	$1,6$	$1,7$	$1,8$	$1,9$	$1,10$	$1,11$
$2,0$	$2,1$	$2,2$	$2,3$	$2,4$	$2,5$	$2,6$	$2,7$	$2,8$	$2,9$	$2,10$	$2,11$
$3,0$	$3,1$	$3,2$	$3,3$	$3,4$	$3,5$	$3,6$	$3,7$	$3,8$	$3,9$	$3,10$	$3,11$
$4,0$	$4,1$	$4,2$	$4,3$	$4,4$	$4,5$	$4,6$	$4,7$	$4,8$	$4,9$	$4,10$	$4,11$
$5,0$	$5,1$	$5,2$	$5,3$	$5,4$	$5,5$	$5,6$	$5,7$	$5,8$	$5,9$	$5,10$	$5,11$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

(b) Two-dimensional view of global addresses $(\mathtt{c}, \mathtt{p} * \mathcal{C} + \mathtt{l})$ for processor $\mathtt{p}$.

Processor 0				Processor 1				Processor 2			
$0,0$	$0,1$	$0,2$	$0,3$	$0,0$	$0,1$	$0,2$	$0,3$	$0,0$	$0,1$	$0,2$	$0,3$
$1,0$	$1,1$	$1,2$	$1,3$	$1,0$	$1,1$	$1,2$	$1,3$	$1,0$	$1,1$	$1,2$	$1,3$
$2,0$	$2,1$	$2,2$	$2,3$	$2,0$	$2,1$	$2,2$	$2,3$	$2,0$	$2,1$	$2,2$	$2,3$
$3,0$	$3,1$	$3,2$	$3,3$	$3,0$	$3,1$	$3,2$	$3,3$	$3,0$	$3,1$	$3,2$	$3,3$
$4,0$	$4,1$	$4,2$	$4,3$	$4,0$	$4,1$	$4,2$	$4,3$	$4,0$	$4,1$	$4,2$	$4,3$
$5,0$	$5,1$	$5,2$	$5,3$	$5,0$	$5,1$	$5,2$	$5,3$	$5,0$	$5,1$	$5,2$	$5,3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

(c) Two-dimensional view of local addresses $(\mathtt{c}, \mathtt{l})$ for processor $\mathtt{p}$.

Figure 5: Global and local addresses of the accessed elements of a one-dimensional array along with the two-dimensional view for a three-processor case with $\mathcal{C} = 4$. Each array element has a unique location $(\mathtt{c}, \mathtt{l})$ in a given processor. Non-local elements are accommodated by extending the local space along the $\mathtt{l}$ dimension.

internal Parafrase-2 structures.

3 Data-flow Analysis using a Linear Algebra Framework

In this section, we define our data-flow framework in detail. First, we introduce some important sets and operations on them.

3.1 Definition of Sets and Operations

Communication Descriptors and Communication Sets A *communication descriptor* can be defined as a pair $\langle \mathcal{R}, \mathcal{S} \rangle$, where $\mathcal{R}$ is an array identifier (name) and $\mathcal{S}$ is the *communication set* associated with $\mathcal{R}$. The exact definition of a communication set depends on the context in which it is used. Throughout our analysis, a communication set is defined as $\{\vec{d} \mid \vec{d}$ is owned by `q` and is required by (or should be transferred to or has already been transferred to) `p`$\}$ except for the `KILL` set, which defines the set of elements written (killed) by `q`. In these set definitions $\vec{d}$ refers to a multi-dimensional array element.

Operations on Communication Sets Since we define a communication set as a list of equalities and inequalities (this is how the Omega library represents a set), it can be represented as $\mathcal{S} = \{\vec{d} \mid \mathcal{P}(\vec{d})\}$ where $\mathcal{P}(.)$ is a predicate. Let $\{\vec{d} \mid \mathcal{P}(\vec{d})\}$ and $\{\vec{d} \mid \mathcal{Q}(\vec{d})\}$ be two communication sets. We define the operations $+_c$, $-_c$, and $\cap_c$ on communication sets as follows:

$$\begin{aligned}
\{\vec{d} \mid \mathcal{P}(\vec{d})\} +_c \{\vec{d} \mid \mathcal{Q}(\vec{d})\} &= \{\vec{d} \mid \mathcal{P}(\vec{d}) \text{ or } \mathcal{Q}(\vec{d})\} \\
\{\vec{d} \mid \mathcal{P}(\vec{d})\} -_c \{\vec{d} \mid \mathcal{Q}(\vec{d})\} &= \{\vec{d} \mid \mathcal{P}(\vec{d}) \text{ and not } (\mathcal{Q}(\vec{d}))\} \\
\{\vec{d} \mid \mathcal{P}(\vec{d})\} \cap_c \{\vec{d} \mid \mathcal{Q}(\vec{d})\} &= \{\vec{d} \mid \mathcal{P}(\vec{d}) \text{ and } \mathcal{Q}(\vec{d})\}
\end{aligned}$$

Note that the operations 'or', 'and' and 'not' can be performed by using the corresponding Omega operations on sets which contain equalities and inequalities.

Operations on Sets of Communication Descriptors Let $\mathcal{D} = \langle \mathcal{R}, \mathcal{S} \rangle$ be a communication descriptor. We define two functions: a function $\mathcal{N}$ from communication descriptors space to array identifiers space; and a function $\mathcal{M}$ from communication descriptors space to communication sets space such that $\mathcal{N}(\mathcal{D}) = \mathcal{R}$ and $\mathcal{M}(\mathcal{D}) = \mathcal{S}$.

Suppose $\mathcal{DS}_1$ and $\mathcal{DS}_2$ are two sets of communication descriptors. Three operations, namely $+_d$, $-_d$, and $\cap_d$, are defined on these sets as follows:

$$\begin{aligned}
\mathcal{DS}_1 +_d \mathcal{DS}_2 \quad = \quad & \{\mathcal{D} \mid \mathcal{D} \in \mathcal{DS}_1 \text{ and } \forall \mathcal{D}' \in \mathcal{DS}_2\ \mathcal{N}(\mathcal{D}) \neq \mathcal{N}(\mathcal{D}')\} \\
& \cup \{\mathcal{D} \mid \mathcal{D} \in \mathcal{DS}_2 \text{ and } \forall \mathcal{D}' \in \mathcal{DS}_1\ \mathcal{N}(\mathcal{D}) \neq \mathcal{N}(\mathcal{D}')\} \\
& \cup \{\mathcal{D} \mid \exists \mathcal{D}' \in \mathcal{DS}_1, \mathcal{D}'' \in \mathcal{DS}_2 \text{ st. } \mathcal{N}(\mathcal{D}) = \mathcal{N}(\mathcal{D}') = \mathcal{N}(\mathcal{D}'') \text{ and } \mathcal{M}(\mathcal{D}) = \mathcal{M}(\mathcal{D}') +_c \mathcal{M}(\mathcal{D}'')\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{DS}_1 -_d \mathcal{DS}_2 \quad = \quad & \{\mathcal{D} \mid \mathcal{D} \in \mathcal{DS}_1 \text{ and } \forall \mathcal{D}' \in \mathcal{DS}_2\ \mathcal{N}(\mathcal{D}) \neq \mathcal{N}(\mathcal{D}')\} \\
& \cup \{\mathcal{D} \mid \exists \mathcal{D}' \in \mathcal{DS}_1, \mathcal{D}'' \in \mathcal{DS}_2 \text{ st. } \mathcal{N}(\mathcal{D}) = \mathcal{N}(\mathcal{D}') = \mathcal{N}(\mathcal{D}'') \text{ and } \mathcal{M}(\mathcal{D}) = \mathcal{M}(\mathcal{D}') -_c \mathcal{M}(\mathcal{D}'')\}
\end{aligned}$$

$$\mathcal{DS}_1 \cap_d \mathcal{DS}_2 \quad = \quad \{\mathcal{D} \mid \exists \mathcal{D}' \in \mathcal{DS}_1, \mathcal{D}'' \in \mathcal{DS}_2 \text{ st. } \mathcal{N}(\mathcal{D}) = \mathcal{N}(\mathcal{D}') = \mathcal{N}(\mathcal{D}'') \text{ and } \mathcal{M}(\mathcal{D}) = \mathcal{M}(\mathcal{D}') \cap_c \mathcal{M}(\mathcal{D}'')\}$$

When there is no ambiguity, we also use $\cup_c$ and $\cup_d$ instead of $+_c$ and $+_d$, respectively. It should be noted that although these operations are similar to those given by Gong et al. [18], there is an important difference. Since we keep the communication sets accurately in terms of equalities and inequalities, we can optimize (e.g., coalesce) communication messages even if the messages do not have the same communication pattern (e.g., broadcast, point-to-point) or identical sender/receiver sets. Most of the previous approaches to global communication optimization cannot optimize these kinds of messages mainly due to their representation of communication sets.

It should be noted that our analysis works with sets of equalities and inequalities. As compared with the previous approaches based on RSDs, our technique may be slower. In order to alleviate this problem, we do not operate on the contents of the sets in every data-flow equation to be evaluated; instead we represent the sets with symbolic names and postpone the real computation on them until the end of the analysis where the communication code should be generated. For example, suppose that a data-flow equation requires combining two sets $S_x = \{[x] : \mathcal{Q}_1(x)\}$ and $S_y = \{[y] : \mathcal{Q}_2(y)\}$ where $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are predicates consisting of equalities and inequalities. Instead of forming the set $\{[z] : \mathcal{Q}_1(z) \vee \mathcal{Q}_2(z)\}$ immediately and using it in subsequent computations, our approach represents the resulting set abstractly as $S_x + S_y$. When the whole process is finished, the resulting sets are re-written in terms of equalities and inequalities and the *simplify* utility of the Omega library is used to simplify them. Our experience shows that this approach requires a manageably small symbolic expression manipulation support and is fast in practice (see Section 6 for a cost analysis of the compilation time). Next we present our data flow framework.

3.2 Local (Intra-Interval) Analysis

In order to make the data-flow analysis task easier, the CFG of the program is traversed prior to the local analysis phase, and for each LHS reference a pointer is stored in the header of all enclosing loop nests. This allows the compiler to reach a LHS reference inside a loop quickly during the data-flow analysis. The local analysis part of our framework computes `KILL`, `GEN` and `POST_GEN` sets for each interval. Then the interval is reduced to a single node and annotated with this information.

Let $\mathcal{R}_{\mathcal{L}}(\vec{i})$ and $\mathcal{R}_{\mathcal{R}}(\vec{i})$ be the data elements obtained from references $\mathcal{R}_{\mathcal{L}}$ and $\mathcal{R}_{\mathcal{R}}$, respectively, with a specific iteration vector $\vec{i}$. The computation of the `KILL` set proceeds in the forward direction; that is, the nodes within the interval are traversed in topological sort order. Let `KILL(i,q)` be the set of elements written (killed) by processor `q` in node `i`, and `Modified(i,q)` be the set of elements that may be killed along any path from the beginning of the interval to node `i` (including node `i`). Then,

$$\begin{aligned}
\mathtt{KILL(i,q)} &= \left\{ \vec{d} \mid \vec{d} \in \mathtt{Own(X,q)} \text{ and } \exists \vec{\imath}, \mathcal{R}_{\mathcal{L}} \text{ st. } \vec{d} = \mathcal{R}_{\mathcal{L}}(\vec{\imath}) \text{ and } \vec{i_l} \le \vec{\imath} \le \vec{i_u} \right\}, \\
\mathtt{Modified(i,q)} &= \left(\bigcup_{\mathtt{j} \in pred(\mathtt{i})} \mathtt{Modified(j,q)} \right) \cup \mathtt{KILL(i,q)}
\end{aligned}$$

assuming that `Modified(pred(first(i)),q)` $= \emptyset$ where `first(i)` is the first node in `i`. If `last(i)` is the last node in `i`, then

$$\mathtt{KILL(i,q)} \;=\; \mathtt{Modified(last(i),q)}.$$

This last equation is used to reduce an interval into a node. Notice that `i` is used to denote a node in the CFG whereas $\vec{i}$ is used for an iteration vector. In order to see how the computation of the `KILL` set proceeds,

```
1    DO i = i_l, i_u
2      X(i-2,i) = Y(i-1,i-1) + X(i,1)
3      DO j = j_l, j_u
4        X(i,j) = Y(i-2,i+2)
5        IF (cond)
6          X(i-1,j+2) = Y(i-2,j-2)
7          Y(i,j) = ...
8        ELSE
9          X(i+1,j-3) = Y(i+3,j-3)
10       END IF
11       Z(i,j) = Y(i-4,j)
12     END DO
13   END DO
```

Figure 6: An example program fragment. In this fragment there are two intervals corresponding to the i and j loops, respectively.

consider Figure 6. In this example there are two intervals corresponding to the j and i loops. We concentrate only on the computation of the KILL sets for array X (the computation of the KILL sets of other arrays can be performed in a similar manner). The analysis starts with the first node of the innermost interval (the j loop), and proceeds as follows:

$$
\begin{aligned}
\mathtt{KILL}(4,\mathtt{q}) &= \{\vec{d} \mid \vec{d} \in \mathtt{Own}(\mathtt{X},\mathtt{q}) \text{ and } \exists \imath, \jmath \text{ st. } \vec{d} = \mathtt{X}(\imath, \jmath) \text{ and } i_l \le \imath \le i_u \text{ and } j_l \le \jmath \le j_u\}. \\
\mathtt{Modified}(4,\mathtt{q}) &= \mathtt{KILL}(4,\mathtt{q}) \\
\mathtt{KILL}(5,\mathtt{q}) &= \emptyset \\
\mathtt{Modified}(5,\mathtt{q}) &= \mathtt{Modified}(4,\mathtt{q}) \cup \mathtt{KILL}(5,\mathtt{q}) \\
&= \mathtt{Modified}(4,\mathtt{q}) \\
\mathtt{KILL}(6,\mathtt{q}) &= \{\vec{d} \mid \vec{d} \in \mathtt{Own}(\mathtt{X},\mathtt{q}) \text{ and } \exists \imath, \jmath \text{ st. } \vec{d} = \mathtt{X}(\imath - 1, \jmath + 2) \text{ and } i_l \le \imath \le i_u \text{ and } j_l \le \jmath \le j_u\}. \\
\mathtt{Modified}(6,\mathtt{q}) &= \mathtt{Modified}(5,\mathtt{q}) \cup \mathtt{KILL}(6,\mathtt{q}) \\
\mathtt{KILL}(7,\mathtt{q}) &= \emptyset \\
\mathtt{Modified}(7,\mathtt{q}) &= \mathtt{Modified}(6,\mathtt{q}) \cup \mathtt{KILL}(7,\mathtt{q}) \\
&= \mathtt{Modified}(6,\mathtt{q}) \\
\mathtt{KILL}(9,\mathtt{q}) &= \{\vec{d} \mid \vec{d} \in \mathtt{Own}(\mathtt{X},\mathtt{q}) \text{ and } \exists \imath, \jmath \text{ st. } \vec{d} = \mathtt{X}(\imath + 1, \jmath - 3) \text{ and } i_l \le \imath \le i_u \text{ and } j_l \le \jmath \le j_u\}. \\
\mathtt{Modified}(9,\mathtt{q}) &= \mathtt{Modified}(5,\mathtt{q}) \cup \mathtt{KILL}(9,\mathtt{q})
\end{aligned}
$$

$$
\begin{aligned}
\texttt{KILL}(10,\texttt{q}) &= \emptyset \\
\texttt{Modified}(10,\texttt{q}) &= [\texttt{Modified}(7,\texttt{q}) \cup \texttt{Modified}(9,\texttt{q})] \cup \texttt{KILL}(10,\texttt{q}) \\
&= \texttt{Modified}(7,\texttt{q}) \cup \texttt{Modified}(9,\texttt{q}) \\
\texttt{KILL}(11,\texttt{q}) &= \emptyset \\
\texttt{Modified}(11,\texttt{q}) &= \texttt{Modified}(10,\texttt{q}) \cup \texttt{KILL}(11,\texttt{q}) \\
&= \texttt{Modified}(10,\texttt{q}) \\
&= [\texttt{Modified}(7,\texttt{q}) \cup \texttt{Modified}(9,\texttt{q})] \\
&= \texttt{KILL}(4,\texttt{q}) \cup \texttt{KILL}(6,\texttt{q}) \cup \texttt{KILL}(9,\texttt{q}) \\
&= \{\vec{d} \mid \vec{d} \in \texttt{Own}(\texttt{X},\texttt{q}) \text{ and } (\exists \imath, \jmath \text{ st. } \vec{d} = \texttt{X}(\imath,\jmath) \text{ or } \vec{d} = \texttt{X}(\imath-1,\jmath+2) \text{ or } \vec{d} = \texttt{X}(\imath+1,\jmath-3)) \\
&\quad\ \text{and } \ i_l \leq \imath \leq i_u \ \text{ and } \ j_l \leq \jmath \leq j_u \ \}.
\end{aligned}
$$

Since `last(3,q) = 11`, at this point we can reduce the innermost interval into a single node and annotate it by its `KILL` set:

$$
\texttt{KILL}(3,\texttt{q}) \;=\; \texttt{Modified}(11,\texttt{q}).
$$

Then the analysis continues with the first node of the outer interval (i loop):

$$
\begin{aligned}
\texttt{KILL}(2,\texttt{q}) &= \{\vec{d} \mid \vec{d} \in \texttt{Own}(\texttt{X},\texttt{q}) \text{ and } \exists \imath \text{ st. } \vec{d} = \texttt{X}(\imath-2,\imath) \text{ and } i_l \leq \imath \leq i_u\}. \\
\texttt{Modified}(2,\texttt{q}) &= \texttt{KILL}(2,\texttt{q}) \\
\texttt{Modified}(3,\texttt{q}) &= \texttt{Modified}(2,\texttt{q}) \cup \texttt{KILL}(3,\texttt{q}) \\
&= \Big\{\vec{d} \mid \vec{d} \in \texttt{Own}(\texttt{X},\texttt{q}) \text{ and } \Big(\exists \imath, \jmath \text{ st. } \vec{d} = \texttt{X}(\imath,\jmath) \text{ or } \vec{d} = \texttt{X}(\imath-1,\jmath+2) \text{ or } \vec{d} = \texttt{X}(\imath+1,\jmath-3) \\
&\quad \text{or } \vec{d} = \texttt{X}(\imath-2,\imath)\Big) \text{ and } i_l \leq \imath \leq i_u \text{ and } j_l \leq \jmath \leq j_u\Big\}.
\end{aligned}
$$

Since `last(1,q) = 3`, at this point we can reduce the interval into a single node

$$
\texttt{KILL}(1,\texttt{q}) \;=\; \texttt{Modified}(3,\texttt{q}).
$$

Although for the sake of presentation we show the analysis here in terms of communication sets, the data-flow analysis is actually performed on sets of communication descriptors, since in general there may be accesses to several arrays. That is, the KILL set for a program that refers to arrays $\mathcal{N}_i$ is as follows

$$\mathtt{KILL(i,q)} \quad = \quad \{\langle \mathcal{N}_1, \mathtt{KILL}_{\mathcal{N}_1}(\mathtt{i,q})\rangle, \langle \mathcal{N}_2, \mathtt{KILL}_{\mathcal{N}_2}(\mathtt{i,q})\rangle, \cdots\}\,.$$

Since we concentrate on computation of the KILL set for a single array, we use KILL(i,q). Similar simplification will be used for presentation of the computation of the GEN(i,q,p) and POST_GEN(i,q,p) sets as well.

GEN(i,p,q) is the set of elements required by processor p from processor q at node i with no preceding write (assignment) to them. The computation of the GEN proceeds in the backward direction, i.e., the nodes within each interval are traversed in reverse topological sort order. The elements that can be communicated at the beginning of a node are the elements required by any RHS reference within the node except the ones that are written by the owner before being referenced. Notice that this process involves considering all the LHS references within an interval for a given RHS reference; this leads to an exponential cost. However, there are two factors that make the analysis affordable. First, the scope of the analysis is a single interval (loop nest). In practice the number of distinct references in a loop nest is a small value. Second, since, as mentioned earlier, prior to analysis we keep pointers to all LHS references within a loop nest, we do not have to traverse the parse tree once more to search for the LHS references.

Assuming $\vec{\imath} = (\imath_1, ..., \imath_n)$ and $\vec{\imath'} = (\imath'_1, ..., \imath'_n)$, let $\vec{\imath'} \prec \vec{\imath}$ mean that $\vec{\imath'}$ is lexicographically less than or equal to $\vec{\imath}$; and $\vec{\imath'} \prec_k \vec{\imath}$ mean that $\imath'_j = \imath_j$ for all $j < k$, and $(\imath'_k, ..., \imath'_n) \prec (\imath_k, ..., \imath_n)$. Since a node can refer to multiple RHS references, we first define $\mathtt{gen}(\mathtt{i}, \mathcal{R}_{\mathcal{R}}, \mathtt{p}, \mathtt{q})$ as the set of elements to be sent by processor q to processor p at node i due to reference $\mathcal{R}_{\mathcal{R}}$. In that case we can compute

$$\mathtt{GEN(i,p,q)} \quad = \quad \bigcup_{\mathcal{R}_{\mathcal{R}}} \mathtt{gen}(\mathtt{i}, \mathcal{R}_{\mathcal{R}}, \mathtt{p}, \mathtt{q}).$$

For the sake of explanation, we assume one RHS reference per node, and use only GEN(i,p,q) in the following. The extension to the multiple RHS reference per node is straightforward. Let Comm(i,p,q) be the set of elements that may be communicated at the beginning of interval i to satisfy communication requirements from the beginning of i to the last node in the interval that contains i. Then, for an array X, we have

$$\begin{aligned}
\texttt{GEN(i,p,q)} &= \Big\{\vec{d} \mid \exists \vec{\imath}, \texttt{Y} \text{ st. } \vec{i_l} \le \vec{\imath} \le \vec{i_u} \text{ and } \vec{d} \in \texttt{Own(X,q)} \text{ and } \vec{d} = \mathcal{R}_{\mathcal{R}}(\vec{\imath}) \text{ and} \\
&\quad \mathcal{R}_{\mathcal{L}}(\vec{\imath}) \in \texttt{Own(Y,p)} \text{ and not } \Big(\exists \vec{\jmath}, \mathcal{R}_{\mathcal{L}}{}' \text{ st. } \vec{i_l} \le \vec{\jmath} \le \vec{i_u} \text{ and } \vec{d} = \mathcal{R}_{\mathcal{L}}{}'(\vec{\jmath}) \text{ and } \vec{\jmath} \prec_{\texttt{level(i)}} \vec{\imath}\Big)\Big\}, \\
\texttt{Comm(i,p,q)} &= \left(\bigcap_{\texttt{s} \in succ(\texttt{i})} \texttt{Comm(s,p,q)}\right) \cup \texttt{GEN(i,p,q)}.
\end{aligned}$$

In addition, we use the following equation to reduce an interval into a single node:

$$\texttt{GEN(i,p,q)} \quad = \quad \texttt{Comm(First(i),p,q)}.$$

In the definition of GEN, $\mathcal{R}_{\mathcal{R}}$ denotes the RHS reference, and $\mathcal{R}_{\mathcal{L}}$ denotes the LHS reference of the same statement. $\mathcal{R}_{\mathcal{L}}{}'$, on the other hand, refers to any LHS reference within the same interval. Notice that while $\mathcal{R}_{\mathcal{L}}{}'$ is a reference to the same array as $\mathcal{R}_{\mathcal{R}}$, $\mathcal{R}_{\mathcal{L}}$ can be a reference to any array (e.g., array Y in the formulation above). `level(i)` gives the nesting level of the interval (loop), with the value 1 corresponding to the outermost loop in the nest. If the dependence is loop independent the textual positions of the references in the nest may also need to be taken into account when computing the GEN set. In that case the formulation of the GEN set should contain terms showing the precedence relations between references. For the sake of simplicity, we assume that all the dependences that we are dealing with are loop carried.

After the interval is reduced, the GEN set for it is recorded, and an operator $\mathcal{F}$ is applied to the last part of this GEN set to propagate it to the outer interval:

$$\mathcal{F}(\vec{\jmath} \prec_{\texttt{k}} \vec{\imath}) = \vec{\jmath} \prec_{(\texttt{k}-1)} \vec{\imath}.$$

As an example consider Figure 6 on page 17 once more, this time concentrating on the computation of GEN sets due to array Y. Notice that array Y is written only in statement (line) 7. The analysis starts with the last statement of the innermost interval (j loop):

$$\texttt{GEN(11,p,q)} \quad = \quad \{\vec{d} \mid \exists \imath, \jmath \text{ st. } i_l \le \imath \le i_u \text{ and } j_l \le \jmath \le j_u \text{ and } \vec{d} \in \texttt{Own(Y,q)} \text{ and } \vec{d} = Y(\imath - 4, \jmath) \text{ and}$$

$Z(\imath,\jmath) \in \mathtt{Own}(\mathtt{Z},\mathtt{p})$ and not $(\exists \imath',\jmath',$ st. $\vec{d} = Y(\imath',\jmath')$ and $i_l \leq \imath' \leq i_u$ and $j_l \leq \jmath' \leq j_u$ and $(\imath' = \imath$ and $\jmath' < \jmath))\}$.

To keep the presentation simpler, we do not show the remaining GEN sets in this interval. The analysis proceeds as follows.

$$
\begin{aligned}
\mathtt{Comm}(11,\mathtt{p},\mathtt{q}) &= \mathtt{GEN}(11,\mathtt{p},\mathtt{q}) \\
\mathtt{GEN}(10,\mathtt{p},\mathtt{q}) &= \emptyset \\
\mathtt{Comm}(10,\mathtt{p},\mathtt{q}) &= \mathtt{Comm}(11,\mathtt{p},\mathtt{q}) \cup \mathtt{GEN}(10,\mathtt{p},\mathtt{q}) \\
&= \mathtt{Comm}(11,\mathtt{p},\mathtt{q}) \\
\mathtt{Comm}(9,\mathtt{p},\mathtt{q}) &= \mathtt{Comm}(10,\mathtt{p},\mathtt{q}) \cup \mathtt{GEN}(9,\mathtt{p},\mathtt{q}) \\
\mathtt{GEN}(7,\mathtt{p},\mathtt{q}) &= \emptyset \\
\mathtt{Comm}(7,\mathtt{p},\mathtt{q}) &= \mathtt{Comm}(10,\mathtt{p},\mathtt{q}) \cup \mathtt{GEN}(7,\mathtt{p},\mathtt{q}) \\
\mathtt{Comm}(7,\mathtt{p},\mathtt{q}) &= \mathtt{Comm}(10,\mathtt{p},\mathtt{q}) \\
\mathtt{Comm}(6,\mathtt{p},\mathtt{q}) &= \mathtt{Comm}(7,\mathtt{p},\mathtt{q}) \cup \mathtt{GEN}(6,\mathtt{p},\mathtt{q}) \\
\mathtt{GEN}(5,\mathtt{p},\mathtt{q}) &= \emptyset \\
\mathtt{Comm}(5,\mathtt{p},\mathtt{q}) &= \mathtt{Comm}(6,\mathtt{p},\mathtt{q}) \cap \mathtt{Comm}(9,\mathtt{p},\mathtt{q}) \\
\mathtt{Comm}(4,\mathtt{p},\mathtt{q}) &= \mathtt{Comm}(5,\mathtt{p},\mathtt{q}) \cup \mathtt{GEN}(4,\mathtt{p},\mathtt{q}) \\
&= [\mathtt{Comm}(6,\mathtt{p},\mathtt{q}) \cap \mathtt{Comm}(9,\mathtt{p},\mathtt{q})] \cup \mathtt{GEN}(4,\mathtt{p},\mathtt{q}) \\
&= [[\mathtt{GEN}(11,\mathtt{p},\mathtt{q}) \cup \mathtt{GEN}(6,\mathtt{p},\mathtt{q})] \cap [\mathtt{GEN}(11,\mathtt{p},\mathtt{q}) \cup \mathtt{GEN}(9,\mathtt{p},\mathtt{q})]] \cup \mathtt{GEN}(4,\mathtt{p},\mathtt{q}) \\
& \quad [\mathtt{GEN}(11,\mathtt{p},\mathtt{q}) \cup \mathtt{GEN}(6,\mathtt{p},\mathtt{q}) \cup \mathtt{GEN}(4,\mathtt{p},\mathtt{q})] \cap [\mathtt{GEN}(11,\mathtt{p},\mathtt{q}) \cup \mathtt{GEN}(9,\mathtt{p},\mathtt{q}) \cup \mathtt{GEN}(4,\mathtt{p},\mathtt{q})] \\
& \quad [\mathtt{GEN}(11,\mathtt{p},\mathtt{q}) \cup \mathtt{GEN}(4,\mathtt{p},\mathtt{q})] \cup [\mathtt{GEN}(6,\mathtt{p},\mathtt{q}) \cap \mathtt{GEN}(9,\mathtt{p},\mathtt{q})]
\end{aligned}
$$

Since `first(3) = 4`, the innermost interval can now be reduced as follows.

$$
\begin{aligned}
\mathtt{GEN}(3,\mathtt{p},\mathtt{q}) &= \mathtt{Comm}(4,\mathtt{p},\mathtt{q}) \\
&= (\{\vec{d} \mid \exists \imath,\jmath \text{ st. } i_l \leq \imath \leq i_u \text{ and } j_l \leq \jmath \leq j_u \text{ and } \vec{d} \in \mathtt{Own}(\mathtt{Y},\mathtt{q}) \text{ and } \vec{d} = Y(\imath-4,\jmath) \text{ and} \\
& \quad Z(\imath,\jmath) \in \mathtt{Own}(\mathtt{Z},\mathtt{p}) \text{ and not } (\exists \imath',\jmath', \text{ st. } \vec{d} = Y(\imath',\jmath') \text{ and } i_l \leq \imath' \leq i_u \text{ and } j_l \leq \jmath' \leq j_u
\end{aligned}
$$

and $(\imath' = \imath$ and $\jmath' < \jmath))\}$

$\cup\{\vec{d} \mid \exists\imath,\jmath$ st. $i_l \leq \imath \leq i_u$ and $j_l \leq \jmath \leq j_u$ and $\vec{d} \in \mathtt{Own(Y,q)}$ and $\vec{d} = Y(\imath-2,\jmath+2)$ and

$X(\imath,\jmath) \in \mathtt{Own(X,p)}$ and not $(\exists\imath',\jmath',$ st. $\vec{d} = Y(\imath',\jmath')$ and $i_l \leq \imath' \leq i_u$ and $j_l \leq \jmath' \leq j_u$

and $(\imath' = \imath$ and $\jmath' < \jmath))\})$

$\cup$

$(\{\vec{d} \mid \exists\imath,\jmath$ st. $i_l \leq \imath \leq i_u$ and $j_l \leq \jmath \leq j_u$ and $\vec{d} \in \mathtt{Own(Y,q)}$ and $\vec{d} = Y(\imath-2,\jmath-2)$ and

$X(\imath-1,\jmath+2) \in \mathtt{Own(X,p)}$ and not $(\exists\imath',\jmath',$ st. $\vec{d} = Y(\imath',\jmath')$ and $i_l \leq \imath' \leq i_u$ and $j_l \leq \jmath' \leq j_u$

and $(\imath' = \imath$ and $\jmath' < \jmath))\}$

$\cap\{\vec{d} \mid \exists\imath,\jmath$ st. $i_l \leq \imath \leq i_u$ and $j_l \leq \jmath \leq j_u$ and $\vec{d} \in \mathtt{Own(Y,q)}$ and $\vec{d} = Y(\imath+3,\jmath-3)$ and

$X(\imath+1,\jmath-3) \in \mathtt{Own(X,p)}$ and not $(\exists\imath',\jmath',$ st. $\vec{d} = Y(\imath',\jmath')$ and $i_l \leq \imath' \leq i_u$ and $j_l \leq \jmath' \leq j_u$

and $(\imath' = \imath$ and $\jmath' < \jmath))\})$

After `GEN(3,p,q)` is recorded, the compiler applies $\mathcal{F}$ operator to `GEN(3,p,q)`. The effect of this operator for this example is

$$(\imath' = \imath \text{ and } \jmath' < \jmath) \quad \rightsquigarrow \quad (\imath' \leq \imath \text{ or } (\imath' = \imath \text{ and } \jmath' < \jmath)).$$

That is, at this point the compiler takes into account flow dependences carried by the i loop as well. Then we continue with the last statement of the outer interval (i loop):

$$\begin{aligned} \mathtt{Comm}(3,\mathtt{p},\mathtt{q}) &= \mathtt{GEN}(3,\mathtt{p},\mathtt{q}) \\ \mathtt{Comm}(2,\mathtt{p},\mathtt{q}) &= \mathtt{Comm}(3,\mathtt{p},\mathtt{q}) \cup \mathtt{GEN}(2,\mathtt{p},\mathtt{q}). \end{aligned}$$

Since `first(1) = 2`, the outer interval can now be reduced:

$$\mathtt{GEN}(1,\mathtt{p},\mathtt{q}) = \mathtt{Comm}(2,\mathtt{p},\mathtt{q}).$$

Since i is the index of the outermost interval, there is no need to apply the $\mathcal{F}$ operator after this reduction. We should emphasize that computing the `GEN` sets gives us all the communication that can be vectorized or coalesced above a loop nest; i.e., our analysis easily handles message vectorization and message coalescing [30]. Finally, POST_GEN(i,p,q) is the set of elements required by processor p from processor q at node i with no subsequent write to them. For an array X:

$$\begin{aligned}\mathtt{POST\_GEN(i,p,q)} \quad = \quad & \Big\{\vec{d} \mid \exists\vec{\imath}, \mathtt{Y} \text{ st. } \vec{i_l} \le \vec{\imath} \le \vec{i_u} \text{ and } \vec{d} \in \mathtt{Own(X,q)} \text{ and } \vec{d} = \mathcal{R}_{\mathcal{R}}(\vec{\imath}) \text{ and} \\ & \mathcal{R}_{\mathcal{L}}(\vec{\imath}) \in \mathtt{Own(Y,p)} \text{ and not } (\exists\vec{\jmath}, \mathcal{R}_{\mathcal{L}}{}' \text{ st. } \vec{i_l} \le \vec{\jmath} \le \vec{i_u} \text{ and } \vec{d} = \mathcal{R}_{\mathcal{L}}{}'(\vec{\jmath}) \text{ and } \vec{\imath} \prec_{\mathtt{level(i)}} \vec{\jmath})\Big\}.\end{aligned}$$

The computation of POST_GEN(i,p,q) proceeds in the forward direction. Its computation is similar to those of the KILL(i,q) and GEN(i,p,q) sets, so we do not discuss it in detail.

3.3 Data-flow Equations

In our framework, any communication incurred is placed at the beginning of the nodes. Here, we concentrate on the computation of a communication set called RECV. The actual *send* and *recv* sets used by the code generator are produced in a later pass of the compiler from the RECV sets discussed here using two projection functions as explained in Section 5. Our data-flow analysis framework consists of a backward and a forward pass. In the backward pass, the compiler determines sets of data elements that can safely be communicated at specific points. The forward pass eliminates redundant communication and determines the final set of elements (if any) that should be communicated at the beginning of each node i. The data-flow equations that we present here are aggressive in the sense that a communication incurred by a non-local reference is hoisted to the highest point possible in the CFG. Later in Section 4 we discuss how to refine this approach to control communication hoisting. The input for the equations consists of the GEN(i,p,q), KILL(i,q) and POST_GEN(i,p,q) sets for each i as computed during the local analysis.

The data-flow equations for the backward analysis are given by Equations (1) and (2) in Figure 7. The symbol $\cap$ in this figure denotes $\cap_d$. SAFE_IN(i,p,q) and SAFE_OUT(i,p,q) are the sets of communication descriptors; these denote the elements that *can* safely be communicated at the beginning and end of node i, respectively. Equation (1) says that an element should be communicated at a point if and only if it will be used in all of the following paths in the CFG. This is the fundamental rule that our data-flow analysis

Backward Analysis:

$$\mathtt{SAFE\_OUT}(\mathtt{i},\mathtt{p},\mathtt{q}) = \bigcap_{\mathtt{s}\in succ(\mathtt{i})} \mathtt{SAFE\_IN}(\mathtt{s},\mathtt{p},\mathtt{q}) \tag{1}$$

$$\mathtt{SAFE\_IN}(\mathtt{i},\mathtt{p},\mathtt{q}) = (\mathtt{SAFE\_OUT}(\mathtt{i},\mathtt{p},\mathtt{q}) -_{\mathtt{d}} \mathtt{KILL}(\mathtt{i},\mathtt{q})) +_{\mathtt{d}} \mathtt{GEN}(\mathtt{i},\mathtt{p},\mathtt{q}) \tag{2}$$

Forward Analysis:

$$\mathtt{RECV\_IN}(\mathtt{i},\mathtt{p},\mathtt{q}) = \bigcap_{\mathtt{j}\in pred(\mathtt{i})} \mathtt{RECV\_OUT}(\mathtt{j},\mathtt{p},\mathtt{q}) \tag{3}$$

$$\mathtt{RECV}(\mathtt{i},\mathtt{p},\mathtt{q}) = \begin{cases} \mathtt{GEN}(\mathtt{i},\mathtt{p},\mathtt{q}) -_{\mathtt{d}} \mathtt{RECV\_IN}(\mathtt{i},\mathtt{p},\mathtt{q}) & \text{if } \exists\, \mathtt{k} \in succ(\mathtt{i}) \text{ and } \mathtt{k} \notin dom(\mathtt{i}) \\ \mathtt{SAFE\_IN}(\mathtt{i},\mathtt{p},\mathtt{q}) -_{\mathtt{d}} \mathtt{RECV\_IN}(\mathtt{i},\mathtt{p},\mathtt{q}) & \text{otherwise} \end{cases} \tag{4}$$

$$\mathtt{RECV\_OUT}(\mathtt{i},\mathtt{p},\mathtt{q}) = \begin{cases} \mathtt{RECV\_IN}(\mathtt{i},\mathtt{p},\mathtt{q}) -_{\mathtt{d}} \mathtt{KILL}(\mathtt{i},\mathtt{q}) & \text{if } \exists\, \mathtt{k} \in succ(\mathtt{i}) \text{ and } \mathtt{k} \notin dom(\mathtt{i}) \\ ((\mathtt{RECV}(\mathtt{i},\mathtt{p},\mathtt{q}) +_{\mathtt{d}} \mathtt{RECV\_IN}(\mathtt{i},\mathtt{p},\mathtt{q})) -_{\mathtt{d}} \mathtt{KILL}(\mathtt{i},\mathtt{q})) +_{\mathtt{d}} \mathtt{POST\_GEN}(\mathtt{i},\mathtt{p},\mathtt{q}) & \text{otherwise} \end{cases} \tag{5}$$

Figure 7: Data-flow equations for optimizing communication. The optimization process involves a backward analysis followed by a forward analysis. At the end, for each `i`, the `RECV(i,p,q)` set is computed.

as well as some of the previous approaches like [35] adheres to. Equation (2), on the other hand, gives the set of elements that can safely be communicated at the beginning of node `i`, and makes use of the `GEN` and `KILL` sets. Intuitively, an element can be communicated at the beginning of node `i` if and only if it is either required (generated) by node `i` or it reaches the end of node `i` (in the backward analysis) and is not overwritten (killed) in it. It should be noted that if the elements contained in `SAFE_IN` sets are directly communicated without any further analysis, there would be significant amounts of redundant communication. The task of the forward analysis phase is to eliminate redundant communication.

The data-flow equations for the forward analysis are given by Equations (3), (4) and (5) in Figure 7; these equations observe the following two rules:

(1) a node should not fetch data needed by a successor unless it dominates that successor; and

(2) a successor should ignore what a predecessor has received so far unless that predecessor dominates it.

`RECV_IN(i,p,q)` and `RECV_OUT(i,p,q)` denote the set of communication descriptors containing the elements that *have been* communicated so far (at the beginning and end of the node `i`, respectively) from `q` to `p`. On the other hand, `RECV(i,p,q)` denotes the set of communication descriptors containing the elements that *should* be communicated from `q` to `p` at the beginning of node `i` and is finally used by the communication generation portion of the compiler to generate the actual *send* and *recv* commands as explained in Section 5. Equation (3) simply says that the communication set arriving in a join node can be found by intersecting the sets for all the joining paths. Equation (4) is used to compute the RECV set which corresponds to the elements that can be communicated at the beginning of the node except the ones that have already been communicated (RECV_IN). The elements that have been communicated at the end of node i (that is, RECV_OUT set) are simply the union of the elements communicated up to the beginning of i; the elements communicated at the beginning of i provided that the condition in equation (5) is not satisfied (except the ones that have been overwritten (killed) in i) and the elements communicated within i and not written subsequently (POST_GEN), again provided that the condition in the equation is not satisfied. It should be emphasized that all these sets are communication descriptor sets, and the order of operations as indicated by the parenthesis is important.

3.4 Global Data-flow Analysis

Our approach starts by computing the GEN, KILL and POST_GEN sets for each node. Then the contraction phase of the analysis reduces the intervals from innermost to outermost and annotates them with GEN, KILL and POST_GEN sets. When a reduced CFG with no cycles is reached, the expansion phase starts and RECV sets for each interval is computed, this time from outermost to innermost. There is one important point to note: before starting to process the next inner graph, the RECV_IN set of the first node in this graph is set to the RECV set of the interval that contains it. More formally, in the expansion phase, we set

$$\mathtt{RECV\_IN}(\mathtt{i},\mathtt{p},\mathtt{q})^{k^{th}pass} = \mathtt{RECV}(\mathtt{i},\mathtt{p},\mathtt{q})^{(k-1)^{th}pass.} \qquad (6)$$

This assignment then triggers the next pass in the expansion phase. Before the expansion phase starts $\mathtt{RECV\_IN}(\mathtt{i},\mathtt{p},\mathtt{q})^{1^{st}pass}$ is set to $\emptyset$. Figure 8 shows the overall algorithm COMM-OPT followed by compiler to generate the *send* and *recv* sets. Notice that due to Equations (1) and (2) in Figure 7 a datum can only be communicated when it is *safe* to do so (i.e., the semantics of the program is preserved). In the forward analysis, the RECV sets contain only the elements needed to be communicated; therefore no stale data is used

INPUT: A connected CFG.
OUTPUT: A processed CFG with optimized communication calls.

Step (a) Pre-processing phase:

- **(a.1)** The CFG is traversed and in each loop a pointer for each LHS it encloses is stored;
- **(a.2)** The CFG is traversed to add empty else branches to "if" constructs and to eliminate the critical edges;
- **(a.3)** The "dominance" relation for each node in the CFG is computed.

Step (b) Initialization phase: For each node in the initial CFG, `KILL`, `GEN` and `POST_GEN` sets are computed in terms of symbolic set names;

Step (c) Contraction phase: Until a CFG with no cycles is reached, recursively each CFG is handled by reducing its intervals and annotating each interval by its `KILL`, `GEN` and `POST_GEN` sets;

Step (d) Expansion phase: For each intermediate CFG, the following is repeated:

- **(d.1)** Using data-flow Equations (1) and (2) in Figure 7, the `SAFE_IN` sets are computed in backward direction;
- **(d.2)** Using data-flow Equations (3), (4) and (5) in Figure 7, the `RECV` sets are computed in forward direction;
- **(d.3)** The CFG is expanded; the equation (6) is used to trigger the data-flow activity in the new CFG;

Step (e) Substitution phase: The symbolic set names in the resultant `RECV` sets are replaced with actual sets consisting of equalities and inequalities;

Step (f) Set generation phase: The Omega library is called to generate *send* and *recv* sets used by the code generator from the `RECV` sets.

Figure 8: Communication optimization algorithm `COMM-OPT` based on data-flow analysis. This algorithm computes the *send* and *recv* sets.

and the correctness is ensured.

3.5 Example

We use the synthetic benchmark program shown in Figure 9(a) on page 29 to illustrate our framework. We concentrate on the communication placement at the higher level CFG that is acyclic. Figure 9(b) shows the message vectorized program with communication calls before the loop bounds reduction and guard insertion. The notation `send{B,q}` means that some elements of array `B` should be sent to `q`; `recv{B,q}` is defined similarly. We omitted from the figure the number of elements communicated to make the code look clear. In this example communication arises only due to references to array `B`. A loop-based communication analysis places eight *send* and eight *recv* calls (in fact these are themselves loop nests) for eight RHS references marked as bold in Figure 9(b). The communication points for these references are just above the corresponding loop nests. For example, communication required due to reference `B(i-1,j-1)` in line `33` in

Table 1: Data-flow sets for the example shown in Figure 9. The GEN, KILL, and POST_GEN sets are obtained after local analysis; and the SAFE_IN and RECV sets are obtained after global analysis.

Line	GEN	KILL	POST_GEN	SAFE_IN	RECV
2	S_4	$\emptyset$	$\emptyset$	$((((S_{51} +_c S_{45}) -_c S_{38}) +_c S_{33}) +_c S_{11}) +_c S_4$	$S_4 +_c S_{11} +_c S_{33}$
9	S_{11}	$\emptyset$	$\emptyset$	$(((S_{51} +_c S_{45}) -_c S_{38}) +_c S_{33}) +_c S_{11}$	$\emptyset$
23	S_{25}	$\emptyset$	$\emptyset$	$(((S_{51} +_c S_{45}) -_c S_{38}) +_c S_{33}) +_c S_{25}$	$(S_{25} +_c S_{33}) -_c (S_4 +_c S_{11} +_c S_{33})$
30	S_{33}	$\emptyset$	$\emptyset$	$((S_{51} +_c S_{45}) -_c S_{38}) +_c S_{33}$	$\emptyset$
36	$\emptyset$	S_{38}	$\emptyset$	$(S_{51} +_c S_{45}) -_c S_{38}$	$\emptyset$
43	S_{45}	$\emptyset$	$\emptyset$	$S_{51} +_c S_{45}$	$S_{51} +_c S_{45}$
49	S_{51}	$\emptyset$	$\emptyset$	S_{51}	$\emptyset$

Figure 9(b) would be performed in line 29. Notice that in this example array B is written only once (in line 38).

Without loss of generality, assume that after local analysis, the GEN and KILL sets are obtained as shown in the second and third column of Table 1 respectively. The corresponding line numbers are shown in the first column. Notice that for this example POST_GEN(i,p,q) is $\emptyset$ for every i (column 4). The fifth column in Table 1 shows the SAFE_IN sets for array B after backward analysis corresponding to the lines given in the first column of the same table. Notice that the communication set S_{25} cannot be hoisted above the line 22 due to the conditional branch. The sixth column, on the other hand, shows the final RECV sets for the same array after the forward analysis and simplifications. Notice that write to array B in line 38 kills all the communication before it.

For this example, the data-flow analysis framework achieves the following:

- The communication sets due to references B(i-1,j) and B(i-1,j+1) in line 45 of Figure 9(b) are combined; that is our approach handles message coalescing easily.
- Communication due to reference B(i-1,j-1) in line 51 is combined with the communication in line 45; and this combined communication can be performed above line 43.
- Similarly, the communication sets due to references in lines 33, 11 and 4 can be combined and performed above line 2 in Figure 9(b).
- The communication in line 22 is reduced in volume (from $4,032$ elements per processor to $3,024$ elements per processor).

- The communications in lines `7`, `8`, `29` and `48` are entirely eliminated.

- Overall, for a single processor, sixteen communication calls (eight *send* and eight *recv*) are replaced by six communication calls.

The resulting optimized program is shown in Figure 9(c). It should be emphasized that the final communication sets are precise, i.e., there is no overestimation. Moreover, these communication sets can be enumerated using the Omega library [31]. Notice that since all communication sets are enumerated in terms of abstract processors `p` and `q`; in general, if desired, message aggregation can also be performed easily.

3.6 Extension for Inter-procedural Analysis

It is relatively straightforward to extend our analysis to work inter-procedurally. In a simple inter-procedural setting, our approach can be used as follows. We first build a call graph [4] where each node corresponds to a procedure and there is a directed edge between two nodes $\mathtt{P}_1$ and $\mathtt{P}_2$ if and only if $\mathtt{P}_1$ calls $\mathtt{P}_2$. We assume that there is no recursive procedure call. We then traverse the call graph in two steps corresponding to backward and forward analyses. In the backward analysis, we traverse the graph in such a way that a node is visited only after all of the nodes it calls have been visited. When a node $\mathtt{P}_k$ is visited, the compiler runs our algorithm for the backward analysis. After the algorithm terminates, we summarize this node's communication by using three sets: `GEN`, `KILL`, and `POST_GEN`. Notice that these three sets completely define the communication behavior of $\mathtt{P}_k$. Subsequently, $\mathtt{P}_k$ is transformed to a new single node, and annotated by these sets (of course, all formal parameters are replaced with actual parameters). When the whole program is reduced to a single node, the forward analysis starts. This time we traverse the call graph in such a way that a node is visited only after all the nodes that call it have been visited. During the visit of a node, we compute the `RECV` sets for each node of it.

It should be noted that there are several inter-procedural communication optimization algorithms (e.g., [25],[26],[15]) with different degrees of sophistication, and the detailed analysis of communication optimization across procedure boundaries is beyond the scope of this paper. However, we believe that for most of the algorithms found in the literature, the summarized communication information represented by `GEN`,

```
!HPF$ processors PROC(0:3)
!HPF$ distribute (cyclic(4),*)
         onto PROC ::  A, B, C, D
  implicit none
  integer i, j, cond
  real A(128,128), B(128,128),
         C(128,128), D(128,128)
   DO i = 2, 127
    DO j = 64, 127
     C(i,j)=C(i,j)+B(i-1,j-1)+1
    END DO
   END DO
   DO i = 2, 127
    DO j = 2, 31
     A(i,j)=(B(i-1,j-1)+B(i-1,j-1)
          +B(i,j))/3.0
    END DO
   END DO
   IF(cond .GT. 0.0) THEN
    DO i = 2, 127
     DO j = 2, 127
      C(i,j)=A(i,j)+C(i,j)+D(i,j)+1
     END DO
    END DO
   ELSE
    DO i = 2, 127
     DO j = 2, 127
      A(i,j)=B(i-1,j-1)+1
     END DO
    END DO
   ENDIF
   DO i = 2, 31
    DO j = 2, 127
     D(i,j)=(B(i,j)*B(i,j))
          +B(i-1,j-1)+1
    END DO
   END DO
   DO i = 1, 127
    DO j = 1, 127
     B(i,j)=B(i,j)-C(i,j)+A(i,j)
    END DO
   END DO
   DO i = 2, 127
    DO j = 2, 127
     A(i,j)=(B(i-1,j)
          +B(i-1,j+1))/2.0
    END DO
   END DO
   DO i = 2, 127
    DO j = 2, 127
     C(i,j)=(B(i,j)+B(i-1,j-1))/2.0
    END DO
   END DO
   END
```

(a) Original program.

```
!HPF$ processors PROC(0:3)
!HPF$ distribute (cyclic(4),*)
         onto PROC ::  A, B, C, D
  implicit none
  integer i, j, cond
  real A(128,128), B(128,128),
         C(128,128), D(128,128)
   [send{B,p+1}], [recv{B,p-1}]
   DO i = 2, 127
    DO j = 64, 127
     C(i,j)=C(i,j)+B(i-1,j-1)+1
    END DO
   END DO
   [send{B,p+1}], [recv{B,p-1}]
   [send{B,p+1}], [recv{B,p-1}]
   DO i = 2, 127
    DO j = 2, 31
     A(i,j)=(B(i-1,j-1)+B(i-1,j-1)
          +B(i,j))/3.0
    END DO
   END DO
   IF(cond .GT. 0.0) THEN
    DO i = 2, 127
     DO j = 2, 127
      C(i,j)=A(i,j)+C(i,j)+D(i,j)+1
     END DO
    END DO
   ELSE
    [send{B,p+1}], [recv{B,p-1}]
    DO i = 2, 127
     DO j = 2, 127
      A(i,j)=B(i-1,j-1)+1
     END DO
    END DO
   ENDIF
   [send{B,p+1}], [recv{B,p-1}]
   DO i = 2, 31
    DO j = 2, 127
     D(i,j)=(B(i,j)*B(i,j))
          +B(i-1,j-1)+1
    END DO
   END DO
   DO i = 1, 127
    DO j = 1, 127
     B(i,j)=B(i,j)-C(i,j)+A(i,j)
    END DO
   END DO
   [send{B,p+1}], [recv{B,p-1}]
   [send{B,p+1}], [recv{B,p-1}]
   DO i = 2, 127
    DO j = 2, 127
     A(i,j)=(B(i-1,j)+B(i-1,j+1))/2.0
    END DO
   END DO
   [send{B,p+1}], [recv{B,p-1}]
   DO i = 2, 127
    DO j = 2, 127
     C(i,j)=(B(i,j)+B(i-1,j-1))/2.0
    END DO
   END DO
   END
```

(b) Message vectorized program.

```
!HPF$ processors PROC(0:3)
!HPF$ distribute (cyclic(4),*)
         onto PROC ::  A, B, C, D
  implicit none
  integer i, j, cond
  real A(128,128), B(128,128),
         C(128,128), D(128,128)
   [send{B,p+1}], [recv{B,p-1}]
   DO i = 2, 127
    DO j = 64, 127
     C(i,j)=C(i,j)+B(i-1,j-1)+1
    END DO
   END DO
   DO i = 2, 127
    DO j = 2, 31
     A(i,j)=(B(i-1,j-1)+B(i-1,j-1)
          +B(i,j))/3.0
    END DO
   END DO
   IF(cond .GT. 0.0) THEN
    DO i = 2, 127
     DO j = 2, 127
      C(i,j)=A(i,j)+C(i,j)+D(i,j)+1
     END DO
    END DO
   ELSE
    [send{B,p+1}], [recv{B,p-1}]
    DO i = 2, 127
     DO j = 2, 127
      A(i,j)=B(i-1,j-1)+1
     END DO
    END DO
   ENDIF
   DO i = 2, 31
    DO j = 2, 127
     D(i,j)=(B(i,j)*B(i,j))
          +B(i-1,j-1)+1
    END DO
   END DO
   DO i = 1, 127
    DO j = 1, 127
     B(i,j)=B(i,j)-C(i,j)+A(i,j)
    END DO
   END DO
   [send{B,p+1}], [recv{B,p-1}]
   DO i = 2, 127
    DO j = 2, 127
     A(i,j)=(B(i-1,j)
          +B(i-1,j+1))/2.0
    END DO
   END DO
   DO i = 2, 127
    DO j = 2, 127
     C(i,j)=(B(i,j)+B(i-1,j-1))/2.0
    END DO
   END DO
   END
```

(c) Global communication optimization.

Figure 9: A synthetic benchmark program (a) with message vectorized (b) and globally optimized (c) versions. The message vectorized program is obtained using the popular vectorization approach based on dependence-analysis. After determining the outermost loop at which the vectorization can be applied, the item wise messages are combined and are lifted out of the enclosing loops. The globally optimized version is generated using the approach discussed in this paper.

Table 2: Possible $\mathcal{P}(\texttt{i})$ predicates to control communication hoisting. Any logical combination of those predicates can also be used. Note that changing $\mathcal{P}(\texttt{i})$ changes behavior of the optimization algorithm completely. With appropriate $\mathcal{P}(\texttt{i})$ predicates most of the previous optimization algorithms can be simulated.

$\mathcal{P}(\texttt{i})$	COMMENT
KILL(i,q) $\neq \emptyset$	avoids message splitting
GEN(i,p,q) $= \emptyset$	avoids hoisting too far - clustering
Buffer_Length(i) $\geq$ limit	avoids protocol delays and hot spots
Number_of_Buffers(i) $\geq$ limit	avoids buffer pressure

KILL, and POST_GEN would be sufficient to optimize communication inter-procedurally.

4 Hoisting Communication vs. Minimizing the Number of Messages

The approach explained so far is focused on hoisting communication as far as possible, and in general, results in reduction in communication volume as well as number of messages. However, as also pointed out by others, hoisting the communication too eagerly can, under some circumstances, lead to excessive buffer requirement [35] and an increase in the number of communication calls inserted [13]. In particular, failing to take resource constraints into account may affect the correctness of the communication placement. For example, if the buffer requirements exceed the maximum available buffer, the program may stall [37]. One way to prevent these problems is to avoid hoisting communication aggressively and to reduce breaking of messages into smaller ones. Since the optimal placement of communication is NP-hard [17], we present a simple heuristic that stops accumulating communication sets as soon as it encounters a node that satisfies a predicate $\mathcal{P}(\texttt{i})$. The content of this predicate depends on a specific implementation. A few alternatives are presented in Table 2. For example, in [35], the third alternative has been used. An implementation can also employ a combination of these alternatives. As an example, consider the predicate obtained by the conjunction of the first and second alternatives; i.e., $\mathcal{P}(\texttt{i}) = \{\texttt{KILL(i,q)} \neq \emptyset \textbf{ and } \texttt{GEN(i,p,q)} = \emptyset\}$.

The data-flow equations given in Figure 11 on page 31 are very similar to those shown in Figure 7 on page 24. The only difference is in the computation of the SAFE_IN(i,p,q) set in which the predicate is taken into account. The reason for this is to prevent a communication set from breaking into smaller sets

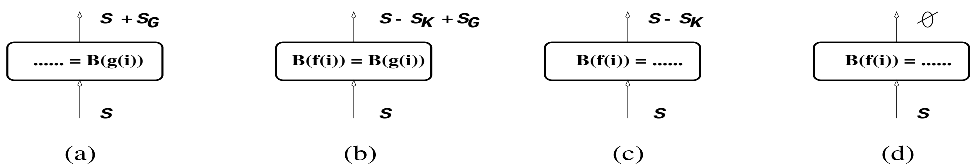

Figure 10: Handling communication during backward analysis in a node using different approaches: (a), (b), and (c) by the approach given in Figure 7; (a), (b), and (d) by the approach given in Figure 11.

Backward Analysis:

$$\mathtt{SAFE\_OUT}(\mathtt{i},\mathtt{p},\mathtt{q}) = \bigcap_{\mathtt{s}\in succ(\mathtt{i})} \mathtt{SAFE\_IN}(\mathtt{s},\mathtt{p},\mathtt{q}) \tag{7}$$

$$\mathtt{SAFE\_IN}(\mathtt{i},\mathtt{p},\mathtt{q}) = \begin{cases} \mathtt{GEN}(\mathtt{i},\mathtt{p},\mathtt{q}) & \text{if } \mathcal{P}(\mathtt{i}) \\ (\mathtt{SAFE\_OUT}(\mathtt{i},\mathtt{p},\mathtt{q}) -_{\mathtt{d}} \mathtt{KILL}(\mathtt{i},\mathtt{q})) +_{\mathtt{d}} \mathtt{GEN}(\mathtt{i},\mathtt{p},\mathtt{q}) & \text{otherwise} \end{cases} \tag{8}$$

Forward Analysis:

$$\mathtt{RECV\_IN}(\mathtt{i},\mathtt{p},\mathtt{q}) = \bigcap_{\mathtt{j}\in pred(\mathtt{i})} \mathtt{RECV\_OUT}(\mathtt{j},\mathtt{p},\mathtt{q}) \tag{9}$$

$$\mathtt{RECV}(\mathtt{i},\mathtt{p},\mathtt{q}) = \begin{cases} \mathtt{GEN}(\mathtt{i},\mathtt{p},\mathtt{q}) -_{\mathtt{d}} \mathtt{RECV\_IN}(\mathtt{i},\mathtt{p},\mathtt{q}) & \text{if } \exists\, \mathtt{k} \in succ(\mathtt{i}) \text{ and } \mathtt{k} \notin dom(\mathtt{i}) \\ \mathtt{SAFE\_IN}(\mathtt{i},\mathtt{p},\mathtt{q}) -_{\mathtt{d}} \mathtt{RECV\_IN}(\mathtt{i},\mathtt{p},\mathtt{q}) & \text{otherwise} \end{cases} \tag{10}$$

$$\mathtt{RECV\_OUT}(\mathtt{i},\mathtt{p},\mathtt{q}) = \begin{cases} \mathtt{RECV\_IN}(\mathtt{i},\mathtt{p},\mathtt{q}) -_{\mathtt{d}} \mathtt{KILL}(\mathtt{i},\mathtt{q}) & \text{if } \exists\, \mathtt{k} \in succ(\mathtt{i}) \text{ and } \mathtt{k} \notin dom(\mathtt{i}) \\ ((\mathtt{RECV}(\mathtt{i},\mathtt{p},\mathtt{q}) +_{\mathtt{d}} \mathtt{RECV\_IN}(\mathtt{i},\mathtt{p},\mathtt{q})) -_{\mathtt{d}} \mathtt{KILL}(\mathtt{i},\mathtt{q})) +_{\mathtt{d}} \mathtt{POST\_GEN}(\mathtt{i},\mathtt{p},\mathtt{q}) & \text{otherwise} \end{cases} \tag{11}$$

Figure 11: Data-flow equations for optimizing communication. These equations are very similar to those presented in Figure 7. The only difference is the use of the $\mathcal{P}(\mathtt{i})$ predicate to control communication hoisting.

each requiring a message of its own. This also eliminates some of the complexity of the resultant code. A possible impact of the new approach is shown in Figure 10. In this figure S_G and S_K denote the GEN and KILL sets respectively for the node shown. The two approaches described in this paper behave similarly for the cases shown in Figures 10(a) and (b). But when a node performs only writes and no reads, the approach in Figure 7 still hoists the communication as shown in Figure 10(c) whereas the approach in Figure 11 stops hoisting as shown in Figure 10(d). That is, the new approach does not issue a communication call unless there are additional elements required by the node. This, in turn, reduces the number of communication calls.

To compare our new approach with the previous one (Figure 7), consider the example program fragment given in Figure 12 on page 33, a modified version of the last part of the program shown in Figure 9.

Table 3: Data-flow sets for the example shown in Figure 12. Top: Aggressive communication hoisting results in excessive number of communication messages. Bottom: Communication hoisting is controlled to minimize the number of messages.

Line	GEN	KILL	POST_GEN	SAFE_IN	RECV	Loop Based
1	$\emptyset$	S_3	$\emptyset$	$((S_{18} +_c S_{13}) -_c S_8) -_c S_3$	$((S_{18} +_c S_{13}) -_c S_8) -_c S_3$	$\emptyset$
6	$\emptyset$	S_8	$\emptyset$	$(S_{18} +_c S_{13}) -_c S_8$	$((S_{18} +_c S_{13}) -_c S_8) -_c (((S_{18} +_c S_{13}) -_c S_8) -_c S_3)$	$\emptyset$
11	S_{13}	$\emptyset$	$\emptyset$	$S_{18} +_c S_{13}$	$(S_{18} +_c S_{13}) -_c ((S_{18} +_c S_{13}) -_c S_8)$	S_{13}
16	S_{18}	$\emptyset$	$\emptyset$	S_{18}	$\emptyset$	S_{18}

Line	SAFE_IN	RECV
1	$\emptyset$	$\emptyset$
6	$\emptyset$	$\emptyset$
11	$S_{18} +_c S_{13}$	$S_{18} +_c S_{13}$
16	S_{18}	$\emptyset$

Columns two, three and four of the top part of Table 3 shows the `GEN`, `KILL` and `POST_GEN` sets respectively corresponding to the line numbers given in the first column. The fifth and sixth columns of the top part of Table 3 show the `SAFE_IN` and `RECV` sets respectively of the previous approach. Although we obtain some reduction in communication volume, the number of messages is three which is larger than that of the loop based approach (column 7) that uses message vectorization alone. The bottom part of Table 3, on the other hand, presents `SAFE_IN` and `RECV` sets obtained by our new approach. In that case the number of messages is 1 and we have reduction in communication volume as well.

The main advantages of the new approach are less computation time during the compilation, less complex *send*/*recv* loops and reduced number of communication messages. However, in real programs when a communicated array is written by the owner processor, it is usually written entirely; therefore, two approaches discussed behave similarly in practice.

5 Communication Generation

Our communication code generator uses the Omega library from University of Maryland [42, 31]. After the `RECV(i,p,q)` sets are obtained in terms of symbolic expressions, they are rewritten in terms of equalities and inequalities. Then the Omega library is called to generate the *send* and *recv* loops.

Let us now consider the example given in Figure 9 (and Table 1) once more to show how the communication sets are generated. We first concentrate on the computation of $S_4 +_c S_{11} +_c S_{33}$. The compiler keeps

```
!HPF$ processors PROC(0:3)
!HPF$ distribute (cyclic(4),*) onto PROC ::  A, B, C, D
 real A(128,128), B(128,128), C(128,128), D(128,128)
   DO i = 32, 63
    DO j = 1, 63
     B(i,j)=D(i,j)+2
    END DO
   END DO
   DO i = 1, 31
    DO j = 1, 63
     B(i,j)=B(i,j)-C(i,j)+A(i,j)
    END DO
   END DO
   DO i = 2, 127
    DO j = 2, 127
     A(i,j)=(B(i-1,j)+B(i-1,j+1))/2.0
    END DO
   END DO
   DO i = 2, 127
    DO j = 2, 127
     C(i,j)=(B(i,j)+B(i-1,j-1))/2.0
    END DO
   END DO
```

Figure 12: An example program fragment to show solution to the problem due to aggressive hoisting. Aggressive communication hoisting does not work for this example.

this set as a symbolic expression until the code generation phase where it inserts equalities and inequalities corresponding to S_4, S_{11}, and S_{33}, and then calls the Omega library to enumerate the elements. Figure 13 shows the communication sets for S_4, S_{11}, S_{33} and $S' = S_4 +_c S_{11} +_c S_{33}$ as represented in Omega. A set element in this figure is represented as a quadruple $[\mathtt{q},\mathtt{p},\mathtt{d}_1,\mathtt{d}_2]$ meaning that the array element indexed by $[\mathtt{d}_1,\mathtt{d}_2]$ should be transferred from q to p. Later in code generation, the projection function $proj_R := \{[\mathtt{q},\mathtt{P},\mathtt{d}_1,\mathtt{d}_2] \rightarrow [\mathtt{q},\mathtt{d}_1,\mathtt{d}_2]\}$ is applied to this set to generate the *recv* set, and similarly the projection function $proj_S := \{[\mathtt{P},\mathtt{p},\mathtt{d}_1,\mathtt{d}_2] \rightarrow [\mathtt{p},\mathtt{d}_1,\mathtt{d}_2]\}$ is applied to generate the *send* set, for a particular processor P. Notice that deriving *send* and *recv* sets from a common set ensures correctness. In Figure 13, l_1 and c_1 denote the coordinates of an element to be communicated in the source (sending) processor whereas l_2 and c_2 denote its coordinates in the target (receiving) processor. l_3 and c_3, on the other hand, refer to coordinates of the LHS reference in the same statement. Notice that the bounds on l_2 are adjusted in the appropriate directions to accommodate the received (non-local) elements; and the entire procedure works on the local address space similar to the one shown in Figure 5(c) on page 13.

After the projection functions are applied, the code generator part of the Omega library is called to

$$
\begin{aligned}
S_4 \;&:=\; \{[\mathtt{q},\mathtt{p},\mathtt{d}_1,\mathtt{d}_2] : \exists(i,c_1,l_1,c_2,l_2,c_3,l_3 : 2 \le i \le 31 \wedge 16c_1+4\mathtt{q}+l_1=\mathtt{d}_1 \wedge 0\le \mathtt{q}\le 3 \wedge 0 \le l_1 \le 3 \wedge i-1=\mathtt{d}_1\\
&\qquad \wedge 16c_3+4\mathtt{p}+l_3=i \wedge 0\le l_3\le 3 \wedge 16c_2+4\mathtt{p}+l_2=\mathtt{d}_1 \wedge -1\le l_2\le 3 \wedge 0\le \mathtt{p}\le 3 \wedge 2\le \mathtt{d}_2+1\le 127 \wedge \mathtt{p}\ne\mathtt{q})\};\\
S_{11} \;&:=\; \{[\mathtt{q},\mathtt{p},\mathtt{d}_1,\mathtt{d}_2] : \exists(i,c_1,l_1,c_2,l_2,c_3,l_3 : 2 \le i \le 127 \wedge 16c_1+4\mathtt{q}+l_1=\mathtt{d}_1 \wedge 0\le \mathtt{q}\le 3 \wedge 0 \le l_1 \le 3 \wedge i-1=\mathtt{d}_1\\
&\qquad \wedge 16c_3+4\mathtt{p}+l_3=i \wedge 0\le l_3\le 3 \wedge 16c_2+4\mathtt{p}+l_2=\mathtt{d}_1 \wedge -1\le l_2\le 3 \wedge 0\le \mathtt{p}\le 3 \wedge 2\le \mathtt{d}_2+1\le 31 \wedge \mathtt{p}\ne\mathtt{q})\};\\
S_{33} \;&:=\; \{[\mathtt{q},\mathtt{p},\mathtt{d}_1,\mathtt{d}_2] : \exists(i,c_1,l_1,c_2,l_2,c_3,l_3 : 2 \le i \le 127 \wedge 16c_1+4\mathtt{q}+l_1=\mathtt{d}_1 \wedge 0\le \mathtt{q}\le 3 \wedge 0 \le l_1 \le 3 \wedge i-1=\mathtt{d}_1\\
&\qquad \wedge 16c_3+4\mathtt{p}+l_3=i \wedge 0\le l_3\le 3 \wedge 16c_2+4\mathtt{p}+l_2=\mathtt{d}_1 \wedge -1\le l_2\le 3 \wedge 0\le \mathtt{p}\le 3 \wedge 64\le \mathtt{d}_2+1\le 127 \wedge \mathtt{p}\ne\mathtt{q})\};\\
S' \;&:=\; \{[\mathtt{q},\mathtt{q}+1,\mathtt{d}_1,\mathtt{d}_2] : \exists(\alpha : \mathtt{d}_1 = 3+4\mathtt{q}+16\alpha \wedge 0\le\mathtt{q}\le 2 \wedge 1\le\mathtt{d}_2\le 126 \wedge 4\mathtt{q}+3\le\mathtt{d}_1\le 4\mathtt{q}+19)\}\\
&\qquad \cup\{[\mathtt{q},\mathtt{p},\mathtt{d}_1,\mathtt{d}_2] : 4\mathtt{p}+15\le\mathtt{d}_1\le 4\mathtt{q}+3 \wedge 1\le\mathtt{d}_2\le 126 \wedge \mathtt{q}\le 3 \wedge 0\le\mathtt{p}\}\\
&\qquad \cup\{[\mathtt{q},\mathtt{p},\mathtt{d}_1,\mathtt{d}_2] : \exists(\alpha : 1\le\mathtt{d}_1\le 4\mathtt{q}+99 \wedge 1\le\mathtt{d}_2\le 30 \wedge \mathtt{q}\le 3 \wedge 13+\mathtt{d}_1\le 4\mathtt{q}+16\alpha \wedge 4\mathtt{p}+16\alpha\le 1+\mathtt{d}_1 \wedge 0\le\mathtt{p})\}\\
&\qquad \cup\{[\mathtt{q},\mathtt{q}+1,\mathtt{d}_1,\mathtt{d}_2] : \exists(\alpha : \mathtt{d}_1 = 3+4\mathtt{q}+16\alpha \wedge 0\le\mathtt{q}\le 2 \wedge 1\le\mathtt{d}_2\le 30 \wedge 4\mathtt{q}+3\le\mathtt{d}_1\le 4\mathtt{q}+115)\}\\
&\qquad \cup\{[\mathtt{q},\mathtt{p},\mathtt{d}_1,\mathtt{d}_2] : \exists(\alpha : 1\le\mathtt{d}_1\le 4\mathtt{q}+99 \wedge 63\le\mathtt{d}_2\le 126 \wedge \mathtt{q}\le 3 \wedge 13+\mathtt{d}_1\le 4\mathtt{q}+16\alpha \wedge 4\mathtt{p}+16\alpha\le 1+\mathtt{d}_1 \wedge 0\le\mathtt{p})\}\\
&\qquad \cup\{[\mathtt{q},\mathtt{q}+1,\mathtt{d}_1,\mathtt{d}_2] : \exists(\alpha : \mathtt{d}_1 = 3+4\mathtt{q}+16\alpha \wedge 0\le\mathtt{q}\le 2 \wedge 63\le\mathtt{d}_2\le 126 \wedge 4\mathtt{q}+3\le\mathtt{d}_1\le 4\mathtt{q}+115\}
\end{aligned}
$$

Figure 13: Omega Relations corresponding to for the example shown in Figure 9. The actual *send* and *recv* sets are derived from these Omega representations using projection functions.

generate the loops to enumerate $[\mathtt{q},\mathtt{d}_1,\mathtt{d}_2]$ and $[\mathtt{p},\mathtt{d}_1,\mathtt{d}_2]$ triples. Finally, the loops are converted to Fortran and the internal data structures of the compiler are updated. As an example, the code enumerating the triples for $(S_{25} +_c S_{33}) -_c (S_4 +_c S_{11} +_c S_{33})$ is shown in Figure 14(a) on page 35 as C code for the *send* set and in Figure 14(b) for the *recv* set. In these codes, `process(.)` is an implementation-specific function that handles the resulting elements. These codes enumerate the elements and only the elements that should be communicated between `q` and `p`. The remaining sets are computed and enumerated similarly. Notice that redundant equalities and inequalities can be eliminated before the code generation phase by using the 'simplify' utility provided by the Omega library.

As a final note, although our use of Omega library increases the compilation time as compared to the previous approaches based on RSDs, this increase was not an issue for the programs we experimented with and was more than compensated by the run-time gains due to optimized communication as explained in the next section.

6 Experiments

In this section we report experimental results for eight programs that exhibit regular communication behavior. The salient characteristics of these programs are given in Table 4 on page 36. `addx` and `eflux` are two

```
if (P == 3) {
  for (j = 31; j <= 111; j += 16) {
    for (k = 31; k <= 62; k++) {
      process_element(0,j,k);
    }
  }
}
if (P >= 0 && P <= 2) {
  for (j = 4*P+35; j <= 4*P+115; j += 16) {
    for (k = 31; k <= 62; k++) {
      process_element(P+1,j,k);
    }
  }
}
```

(a) *send* set.

```
if (P >= 1 && P <= 3) {
  for (j = 4*P+31; j <= 4*P+111; j += 16) {
    for (k = 31; k <= 62; k++) {
      process_element(P-1,j,k);
    }
  }
}
if (P == 0) {
  for (j = 31; j <= 111; j += 16) {
    for (k = 31; k <= 62; k++) {
      process_element(3,j,k);
    }
  }
}
```

(b) *recv* set.

Figure 14: Codes for enumerating $(S_{25} +_c S_{33}) -_c (S_4 +_c S_{11} +_c S_{33})$ for the example shown in Figure 9 for a specific processor P. `process_element()` is an implementation specific function that handles the set of elements to be communicated.

subprograms from the Perfect Club Benchmarks. The `hydro_m` code is a modified version of `hydro`. To obtain this version two modifications have been made to the program aimed at highlighting the difference between our two global optimization techniques. First, the second loop nest is distributed over its statements. Second, the loop bounds in the first loop nest are reduced to `1/4`th of the original values. The `REFS` column shows the number of references in the program in question whereas the `C REFS` column gives the number of references that require communication. The `ITER` column shows how many times the outermost timing loop has been iterated for each program. Except for some hard-coded (small) values of array dimensions, the size of each dimension of an array used in the experiments is set to the value shown in the `SIZE` column. In `tred2` for 8 and 16 processors we used 60 and 120, respectively, as the `SIZE` parameter. The `DISTR` column shows how the highest dimensional arrays in the program are distributed. A '`D`' in a dimension means that the dimension is distributed across processors while a '`*`' denotes a non-distributed dimension as in HPF [39].

The distributed dimensions shown in the table are the *best* distributions for these programs as far as the communication is concerned. For example, selecting a `(*,D)` distribution for `tomcatv` would prevent message-vectorization. For each distributed dimension we experimented with *four different distributions*: block (`BLK`), cyclic (`CYC`), cyclic(4) (`CYC(4)`), and cyclic(7) (`CYC(7)`). The last two distributions are taken into account to demonstrate the effectiveness of our approach with block-cyclic distributions where most

Table 4: Programs in our experiment set and their characteristics. The REFS column shows the number of references in the program whereas the C REFS column gives the number of references that require communication. The ITER column shows how many times the outermost timing loop has been iterated for each program. Each dimension of an array used in the experiments is set to the value shown in the SIZE column. The DISTR column shows how the highest dimensional arrays in the program are distributed. A 'D' in a dimension means that the dimension is distributed across processors while a '*' denotes a non-distributed dimension.

PROGRAM	SOURCE	ARRAYS	REFS	C REFS	DISTR	SIZE	ITER	BRIEF DESCRIPTION
hydro	Livermore	nine 2D	52	10	(*,D)	400	20	2D hydrodynamics
hydro_m	Livermore	nine 2D	52	10	(*,D)	400	20	modified hydro
adi	Livermore	three 3D, three 1D	33	6	(*,D,*)	400	10	iterative method
tomcatv	Spec92	seven 2D, two 1D	75	20	(D,*)	400	10	2D mesh generation
swim	Spec92	fourteen 2D	196	43	(D,*)	513	20	water equation solver
addx	Perfect Club	five 3D, one 2D	72	32	(D,*,*)	194	1	mesh related comp.
eflux	Perfect Club	four 3D, one 2D	76	13	(D,*,*)	5000	10	mesh related comp.
tred2	Eispack	two 2D, two 1D	42	22	(D,*)	60/120	1	matrix reduction

of the previous techniques fail. Two cyclic factors, namely 4 and 7, are selected arbitrarily, one being power of two whereas the other one is prime. Gupta and Banerjee [21] note that for tred2 the block-cyclic distribution is the best choice. We also found that in addx block-cyclic distribution performs best (depending on the number of processors used).

We have found that except hydro_m for all of these programs our two global optimization approaches given in Figures 7 and 11 result in the same optimized code. For each program except hydro_m we experiment with two different versions of the code. The base version does not perform any global communication optimization but does perform message-vectorization. In fact, a direct application of the owner-computes rule without any optimization results in run-time resolution. In run-time resolution the ownership and communication for each reference are computed at run-time. Since each processor must execute the entire iteration space to compute ownership, this method results in large amounts of overhead. Communication for resolution programs is also very inefficient as it involves transmission of a large number of small messages [40]. Instead we considered the message-vectorized version with loop bounds reduction as the base version. Since most of the compilers for message-passing architectures apply some kind of message-vectorization, we felt that it would be unfair to compare our method against run-time resolution without loop bounds reduction. Notice however, even in a single loop nest our global optimization approach subsumes most local optimizations including message-vectorization, message-coalescing, and message-aggregation. For all the programs except `hydro_m` we refer to the globally optimized version as `opt`. In the `hydro_m` code, `opt` refers to the approach given in Figure 7 whereas `opt*` denotes the approach given in Figure 11. For all the programs and the versions, we also applied an optimization that we call *communication pattern reuse*. For example, assuming a `(*,D)` distribution for all arrays, in a statement such as $\texttt{X}(i,j) = \texttt{Y}(i,j-1) + \texttt{Z}(i,j-1)$, arrays `Y` and `Z` have the same communication structure; therefore, we can generate communication loops only once and reuse it with a different name for each array. This optimization has not been fully implemented yet.

We now briefly discuss the implementation status of our framework. We have finished the implementation of local communication analysis, Omega–Parafrase data structure interfacing, and communication loop generation parts. Currently, the global communication analysis part and communication pattern reuse optimizations are being implemented. Experimenting with different message combining techniques (different $\mathcal{P}(\texttt{i})$ predicates) and extension to an inter-procedural setting are in our future plans. Below, we present the first results from our implementation.

We measure the effectiveness of our approach in terms of three different but correlated parameters: number of communication messages across *all* processors, data volume to be communicated across *all* processors and execution time. The number of messages and the communication volume are counted dynamically during the execution. The execution times are obtained on a 16 node IBM SP-2 at the Center for Parallel and Distributed Computing at Northwestern University. Each node of this machine has 128 MB memory, 2 GB disk, and an IBM Power2 processor.

Tables 5 through 12 give the number of communications, the communication volume and the execution times (in seconds) for our programs for the `base` and `opt` versions. Table 13 on page 43 summarizes the improvement in number of messages for our programs. For `hydro_m` there are two rows corresponding to our two methods (`opt` and `opt*` from top). Overall there is a 32% reduction in the number of messages. Improvement with 16 processors is slightly higher than that with 8 processors. This is because with the 16 processors in general there are more communication messages to optimize. It is also interesting to note that our optimization technique achieves 33% improvement with block-cyclic distribution (CYC(4) and CYC(7)) where most of the previous techniques fail. As expected, for hydro_m our second approach which controls communication hoisting performs better than aggressive hoisting.

Table 14 shows the percentage improvement in communication volume across all processors. We note that in both 8 and 16 processor cases we have on average 37% improvement over the base version. Considering block-cyclic distributions alone, we have a 40% improvement. As mentioned earlier these counts are collected dynamically at run-time using the performance analysis tools available on the SP-2. Also it should be emphasized that most of the improvements on adi and tomcatv result from a single nest, meaning that an aggressive loop level optimizer that applies a combination of vectorization, coalescing, and aggregation could also obtain similar improvements.

Finally, Table 15 gives the improvement in execution times. We note that the performance improvement for some programs such as hydro, adi, tomcatv, and swim is very good whereas for eflux and tred2 the improvement is only modest. This is due to the fact that the communication for this second group of codes is either small compared to the total execution time or difficult to optimize. Therefore, there is not much opportunity for improvement. Overall we have 26% improvement. Our approach improves performance in all cases, and more importantly we see a 27% improvement in block-cyclic distributions showing that through a global analysis it is possible to optimize communication globally even in the existence of block-cyclic distributions.

Having established the benefits of our global optimization approach, we now quantify the additional costs incurred by our approach at compile-time and run-time. The results of our cost analysis are summarized in Tables 16 and 17. All the compilation times shown in the rest of the paper have been obtained on a Model 712/60 HP workstation with a 132 MHz PA RISC processor, 64 KB first-level cache, 1 MB second-level cache and a 256 MB memory.

Table 16 shows the compilation times in milliseconds for our programs under different distributions. For each distribution the compilation time is divided into three components: GLO is the time it takes for our global data-flow analysis to run; OME is the time the Omega library takes to generate communication

Table 5: Results for `hydro` on IBM SP-2.

(a): # of communications for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	1,410	1,440	1,440	1,440	3,005	3,880	3,880	3,880
opt	1,120	1,280	1,280	1,280	2,424	2.560	2,560	2,560

(b): Communication volume in MBytes for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	2.1	108.9	27.1	14.8	5.1	111.0	28.8	17.1
opt	1.9	97.0	24.2	12.0	4.3	99.1	26.0	13.9

(c): Execution times in secs for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	4.12	4.75	5.09	4.83	2.81	3.33	3.83	3.06
opt	3.07	3.74	3.87	3.37	2.11	2.75	2.94	2.80

Table 6: Results for `hydro_m` on IBM SP-2.

(a): # of communications for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	1,540	1,604	1,604	1,604	3,318	3,810	3,810	3,810
opt	1,110	1,227	1,227	1,227	2,400	2.611	2,611	2,611
opt*	1,110	1,180	1,180	1,180	2,330	2.555	2,555	2,555

(b): Communication volume in MBytes for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	2.0	75.4	19.1	11.0	4.9	85.2	19.8	11.7
opt	1.8	17.0	4.2	3.1	4.1	17.8	4.3	3.9
opt*	1.8	5.9	3.2	2.8	3.9	6.6	4.4	4.1

(c): Execution times in secs for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	3.65	4.96	3.97	3.99	2.40	2.82	3.10	2.88
opt	2.67	3.05	3.14	2.98	1.90	1.99	1.97	1.98
opt*	2.30	2.82	2.95	2.81	1.73	1.80	1.87	1.78

Table 7: Results for `adi` on IBM SP-2.

(a): # of communications for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	710	968	968	968	1,640	1,922	1,922	1,922
opt	288	480	480	480	644	960	960	960

(b): Communication volume in KBytes for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	20.1	187.2	46.8	27.0	84.0	187.2	46.8	27.0
opt	11.4	94.0	23.3	13.0	53.9	94.0	23.3	13.0

(c): Execution times in secs for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	0.58	0.79	0.64	1.11	1.08	1.13	1.10	1.98
opt	0.43	0.52	0.47	0.58	0.81	0.87	0.88	0.84

Table 8: Results for `tomcatv` on IBM SP-2.

(a): # of communications for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	140	6,396	1.612	932	300	6,498	1,694	998
opt	56	124	124	124	120	252	252	252

(b): Communication volume in MBytes for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	0.35	19.2	4.9	2.8	0.75	19.2	4.9	2.8
opt	0.06	6.1	1.5	0.86	0.11	6.1	1.5	0.86

(c): Execution times in secs for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	1.39	3.19	2.39	2.41	1.30	2.25	2.26	2.29
opt	1.06	1.31	1.44	1.34	0.88	1.06	1.09	1.07

Table 9: Results for `swim` on IBM SP-2.

(a): # of communications for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	3,967	120,142	31,598	17,593	8,215	125,142	34,598	21,593
opt	3,678	84,182	22,358	13,753	7,615	88,182	26,358	19,753

(b): Communication volume in MBytes for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	7.4	240.2	62.2	37.9	15.8	248.0	68.0	42.4
opt	7.1	163.8	44.4	26.0	14.2	168.0	48.5	30.1

(c): Execution times in secs for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	11.31	36.55	21.14	19.64	7.03	19.22	12.20	12.24
opt	10.48	25.47	18.47	16.13	6.71	11.12	10.78	10.31

Table 10: Results for `addx` on IBM SP-2.

(a): # of communications for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	91,470	92,014	91,462	93,646	98,078	98,622	98,614	101,342
opt	57,266	57,538	57,190	58,626	61,426	61,698	61,690	63,466

(b): Communication volume in MBytes for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	0.37	0.38	0.37	0.37	0.40	0.40	0.40	0.41
opt	0.23	0.23	0.23	0.24	0.26	0.26	0.26	0.27

(c): Execution times in secs for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	5.18	6.65	5.10	5.44	5.99	6.48	6.89	5.70
opt	3.33	4.94	4.12	3.16	3.08	4.79	4.07	3.36

Table 11: Results for `eflux` on IBM SP-2.

(a): # of communications for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	203	470	470	470	435	950	950	950
opt	84	408	408	408	180	816	816	816

(b): Communication volume in MBytes for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	0.07	30.3	7.4	4.3	0.15	30.3	7.4	4.3
opt	0.04	30.1	7.0	4.0	0.09	30.1	7.0	4.0

(c): Execution times in secs for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	5.90	13.10	13.10	12.96	4.22	7.46	8.78	7.41
opt	5.76	12.90	12.97	12.01	3.99	7.28	6.98	6.72

Table 12: Results for `tred2` on IBM SP-2.

(a): # of communications for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	1,706	1,759	1,740	1,721	2,028	2,300	2,286	2,280
opt	1,650	1,719	1,718	1,711	1,988	2,015	2,004	1,996

(b): Communication volume in MBytes for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	11.5	12.1	11.9	11.9	14.2	15.5	15.1	15.2
opt	11.1	11.8	11.4	11.3	13.7	15.0	14.7	14.7

(c): Execution times in secs for different versions.

version	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
base	0.83	0.90	0.80	0.76	0.91	1.16	0.90	0.89
opt	0.78	0.78	0.73	0.69	0.90	1.05	0.82	0.79

Table 13: Percentage (%) improvements in # of messages.

program	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
hydro	21	11	11	11	19	34	34	34
hydro_m	28	24	24	24	28	31	31	31
hydro_m	28	26	26	26	30	33	33	33
adi	59	50	50	50	61	50	50	50
tomcatv	60	98	92	87	60	96	85	75
swim	7	30	29	22	7	29	24	9
addx	37	37	37	37	37	37	37	37
eflux	47	13	13	13	59	14	14	14
tred2	3	2	1	1	2	12	12	12
average	33	33	32	30	33	37	35	33

Table 14: Percentage (%) improvements in communication volume.

program	# of PROCS = 8				# of PROCS = 16			
	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	(*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
hydro	10	11	11	19	16	11	10	19
hydro_m	10	77	78	72	16	79	78	67
hydro_m	10	92	83	75	20	92	77	65
adi	43	50	50	52	36	50	50	52
tomcatv	83	68	69	69	85	68	69	69
swim	4	32	29	31	10	32	29	29
addx	38	39	38	35	35	35	35	34
eflux	43	1	5	7	40	1	5	7
tred2	3	2	4	5	4	3	3	3
average	27	42	41	40	29	41	39	38

loops; and REM is the remaining time in compilation including parsing and code generation. The extra time required to write intermediate code into disk files is excluded from these figures.

The first three columns of Table 17 show the percentages for GLO, OME and REM in compilation time considering all the distributions used in the programs. The GLP column gives us the sum of the columns GLO and OME and represents the percentage of the compilation time that our global optimization approach takes (global analysis + generating communication loops). We can see that on the average 64% of the compilation time is spent on our global approach. However it is also important to observe how much compilation time the base version using the Omega library would take. If we do not use any global optimization but still use an Omega-based loop-level optimization, the percentages of compilation time the Omega library takes to generate communication loops are shown under column LOP. We see that even if we do not use the

Table 15: Percentage (%) improvements in execution time.

program	# of PROCS = 8 (*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))	# of PROCS = 16 (*,BLK)	(*,CYC)	(*,CYC(4))	(*,CYC(7))
hydro	25	21	24	30	25	17	23	8
hydro_m	27	39	21	25	21	29	36	30
hydro_m	37	43	26	30	28	36	40	36
adi	26	34	27	48	25	23	20	58
tomcatv	24	59	40	44	32	53	52	53
swim	7	30	13	18	5	42	12	16
addx	36	26	19	41	49	32	41	41
eflux	2	2	1	7	5	2	21	9
tred2	6	13	9	9	1	9	9	11
average	21	30	20	28	21	27	29	29

global framework, just using the Omega library takes 50% of the compilation time on the average. The DIF column shows the difference (GLP−LOP) between the global optimization approach and the loop nest based optimization approach, both using the Omega library. We see that the additional burden of our framework over the existing framework is only 14%.

We can conclude that a hypothetical global optimization approach using RSDs to represent communication sets may be able to eliminate at most 64% of the compilation time. This is a theoretical bound as we do not know of any RSD based framework with *zero cost* that can handle *block-cyclic* distributions *globally*. Given the gains in execution time, we believe that the extra overhead that our approach incurs at compile-time is tolerable. In general, over several runs, the extra compilation time will be amortized. Moreover, we can expect the Omega-like tools to be much faster in the future.

The RUN column shows the percentages of execution times spent on executing the communication loops (without communication statements). On the average, only 7% of the execution time is spent on communication loops; therefore, the overhead incurred by our Omega-based approach at run-time is reasonable.

We also compared the compilation time taken by our Omega-based global approach with that of an approach based on processor-tagged descriptors (PTDs) [44], an enhanced form of RSDs built on top of Parafrase-2. PTDs provide an efficient way of describing distributed sets of iterations and regions of data, and are based on a single set representation parameterized by the processor location for each dimension of a virtual mesh. Table 18 shows the overall compilation times of the Omega-based approach (OME), the

Table 16: Compilation times in milliseconds for different distributions. For a given distribution, the compilation time is divided into three components: GLO is the time it takes for our global data-flow analysis to run; OME is the time the Omega library takes to generate communication loops; and REM is the remaining time in compilation including parsing and code generation.

	BLK			CYC			CYC(4)			CYC(7)		
program	GLO	OME	REM	GLO	OME	REM	GLO	OME	REM	GLO	OME	REM
hydro	157	1,883	1,414	211	1,945	1,466	213	2,267	1,176	213	2,280	1,466
hydro_m	166	1,900	1,290	188	1,906	1,400	199	2,444	1,176	180	2,200	1,366
adi	161	955	1,100	161	970	1,134	174	984	1,100	176	976	1,132
tomcatv	167	2,308	1,100	200	2,616	1,232	217	3,008	1,200	217	2,867	1,186
swim	300	2,967	1,800	284	3,817	1,834	266	3,767	1,834	384	3,783	1,834
addx	200	1,283	1,155	254	1,367	1,184	198	1,417	1,184	242	1,555	1,180
eflux	183	2,017	1,104	184	2,082	1,106	187	2,300	1,134	187	2,117	1,130
tred2	180	2,417	1,334	183	2,584	1,334	184	2,466	1,366	187	2,484	1,360
average	189	1,966	1,245	208	2,161	1,274	205	2,332	1,271	223	2,283	1,269

PTD-based approach (PTD), and the percentage increase (INC) when going from PTD to OME for pure block (BLK) and pure cyclic (CYC) distributions, as the PTDs cannot compile for general block-cyclic distributions. The results show that using Omega instead of an RSD-like approach increases the compilation time 7% to 27%, averaging on 19% for both block and cyclic distributions.

7 Related Work

Several papers have address the problem of generating local address and communication sets for HPF programs where arrays are distributed using the general block-cyclic distributions [7, 14, 24, 33, 34, 46, 47]. Of these, Ancourt et al. [7] use a linear algebra framework; this renders their approach general. The rest of the approaches are very efficient for a restricted class of mappings. Considering the lack of generality of these approaches, their use in the communication optimizations of the kind discussed in this paper appears to be limited.

Most of the previous efforts considered communication optimization at loop level. Although each approach has its own unique features, the general idea has been the use of an appropriate combination of message vectorization, message coalescing and message aggregation [10, 11, 30, 51, 8, 52].

More recently some researchers have proposed techniques based on data-flow analysis in order to op-

Table 17: Cost analysis of our approach over *all* distribution types. On the average, half the compilation time is spent in generating the communication loops. All the values are in percentages of the total compilation time (except the RUN column). The run-time overhead of executing these loops is not very high.

program	BREAKDOWN (%)			GLP	LOP	DIF	RUN
	GLO	OME	REM				
hydro	5	57	38	62	48	14	6
hydro_m	5	59	36	64	48	16	6
adi	7	43	50	50	36	14	4
tomcatv	5	66	29	71	47	24	5
swim	5	63	32	68	58	10	9
addx	8	50	42	58	54	4	8
eflux	5	63	32	68	54	14	7
tred2	3	62	35	65	55	10	9
average	6	58	36	64	50	14	7

Table 18: Total compilation times (in milliseconds) of the Omega-based approach and the PTD-based approach. The OME column and the PTD column gives the compilation times obtained using the Omega-based and the PTD-based approaches, respectively. The INC column shows the percentage increase when going from PTD to OME.

program	BLK			CYC		
	OME	PTD	INC	OME	PTD	INC
hydro	3,454	2,715	27	3,622	2,927	24
hydro_m	3,356	2,644	25	3,494	2,801	25
adi	2,216	2,044	8	2,265	2,086	9
tomcatv	3,575	3,148	14	4,048	3,290	23
swim	5,067	4,426	15	5,935	5,015	18
addx	2,638	2,241	18	2,805	2,615	7
eflux	3,304	2,650	25	3,372	2,814	20
tred2	3,931	3,355	17	4,101	3,390	21
average	3,443	2,903	19	3,705	3,117	19

timize communication across multiple loop nests. Agrawal and Saltz [1] present a framework for partial redundancy elimination for communication optimization in data-parallel programs with irregular data access patterns. Amarasinghe and Lam [6] present several algorithms to optimize communication on machines with distributed address spaces. Their approach uses the *last write tree* representation to eliminate redundant messages within a single loop nest. Although, their technique is also based on data-flow information, they do not allow loop nests within conditionals.

Granston and Veidenbaum [19] propose an algorithm that applies combined flow and dependence analysis to programs with parallel constructs. Their algorithm detects partial redundancies across loop nests and in the presence of conditionals. However their approach is not directly applicable to programs with general data distributions.

Gong et al. [18] describe optimizations that reduce communication overhead and execution time. Their optimizations include elimination of redundant communication and combining messages. However their approach cannot handle general types of distributions, and they offer no optimizations to eliminate the excessive number of communication calls due to split operations.

Gupta et al. [22] present a framework to optimize communication based on data-flow analysis and available section descriptors. Their approach is aggressive in exploiting the locally available data but fails to support general block-cyclic distributions, and the representation that they use makes it difficult to embed alignment and distribution information. Moreover, the communication set information they compute may not be precise.

Hanxleden and Kennedy [27, 28] present a code placement framework for optimizing communication caused by irregular array references. Although the framework provides global data-flow analysis, it treats arrays as indivisible entities; thus, it is limited in exploiting the information available in compile-time. In contrast, Kennedy and Nedeljkovic [32] offer a global data-flow analysis technique using bit vectors. Although this approach is efficient, it is not as precise as the approach presented in this paper. They do not give any clue how their method can be extended to handle general type block-cyclic distributions.

Kennedy and Sethi [35, 36, 37] show the necessity of incorporating resource constraints into a global communication optimization framework. They take into account limited buffer size constraint and illustrate how strip-mining improves the efficacy of the communication placement. Their approach works with multiple nests but not for general block-cyclic distributions. Since they do not give any experimental results, a direct quantitive comparison of this work with ours is not possible. Their work defines a data-flow variable called `SAFE` which can be used in a similar manner as our predicate $\mathcal{P}(\mathtt{i})$. Kennedy and Sethi [35, 36, 37] do not use a linear algebra framework; later work from the dHPF project at Rice [2, 3] includes the use of the Omega library for message optimizations.

The IBM pHPF compiler [13, 23] achieves both redundancy elimination and message combining globally. But message combining is feasible only if the messages have identical patterns, or one pattern is a subset of another. The general block-cyclic distributions, however, can lead to complicated data access patterns and communication sets which, we believe, more precisely can be represented within a linear algebra framework.

Yuan et al. [49, 50] present a communication optimization approach based on array data-flow analysis. The cost of the analysis is managed by partitioning the optimization problem into subproblems, and solving the subproblems one at a time. Since that approach is also based on RSDs, it has difficulty in handling block-cyclic distributions.

Adve et al. [2, 3] describe an integer set based approach for analysis and code generation for data parallel programs that uses the Omega library [31]. They consider performing message vectorization and message coalescing for general access patterns. Their method can also work with computation decomposition schemes that are not based on the owner-computes rule. These papers do not show how their techniques handle global communication optimization for multiple loop nests in the case of block-cyclic distributions.

Interval analysis used in this paper was first introduced by Allen and Cocke [5]. They used it to solve several data-flow problems; the analysis was then extended by Gross and Steenkiste [20] to array sections. The approach proposed by Gupta et al. [22] mentioned above refines the technique by Gross and Steenkiste using loop-carried dependences.

In this paper we used ideas from the linear algebra framework [7] and data-flow analysis [5, 4] developed for performing optimizations on the CFG representation of the programs. We have shown that these two techniques blend together in a nice manner, which makes dealing with the global communication optimization problem feasible even in the presence of general block-cyclic distributions. We should emphasize that the data-flow equations given by Figures 7 and 11 are only two representative solutions to show how the global communication problem can be put into the linear algebra framework. We believe most of the previous approaches can also be put into this framework by re-defining the communication and ownership sets in terms of equalities and inequalities. This would not only give those approaches the capability to handle arbitrary alignments and distributions, but also provides high accuracy in manipulating the communication sets.

8 Summary

Management of accesses to non-local data to minimize communication costs is critical for scaling performance on distributed-memory message-passing machines. In this paper, we presented a global communication optimization scheme based on two complementary techniques: data-flow analysis and linear algebra framework. The combination of these techniques allows us to optimize communication globally and use polyhedron scanning techniques to enumerate global communication sets effectively for HPF-like alignments and distributions including block-cyclic distributions. Our framework takes into account control flow and achieves message vectorization, message coalescing, message aggregation and redundant communication elimination all in a unified framework. The cost of the analysis is managed by keeping the communication sets symbolically until the end of the data-flow analysis where the Omega library is called to generate actual sets in terms of equalities and inequalities. The experimental results demonstrate the effectiveness of our approach in reducing the number of messages and the volume of the data to be communicated. Future work will address the development of performance models to provide the compiler with the ability to estimate the profitability of message aggregation and coalescing globally.

Acknowledgments

The authors would like to thank Evan Rosser for his help in installing the Omega library and Omega calculator, a high level interface to the Omega library. The material presented in this paper is based on research supported in part by Prof. A. Choudhary's NSF Young Investigator Award CCR-9357840, the NSF grant CCR-9509143, DOE AV-6193 and the Air Force Materials Command under contract F30602-97-C-0026. The work of P. Banerjee is supported in part by the NSF under grant CCR-9526325 and in part by the DARPA under contract DABT-63-97-C-0035. The work of J. Ramanujam is supported in part by the NSF Young Investigator Award CCR-9457768 and the NSF grant CCR-9210422.

References

[1] G. AGRAWAL and J. SALTZ. Inter-procedural data flow based optimizations for distributed memory compilation. *Software Practice and Experience,* 27(5): 519–545, 1997.

[2] V. ADVE, J. MELLOR-CRUMMEY, and A. SETHI. An integer set framework for HPF analysis and code generation. Technical Report TR97-275, Computer Science Dept., Rice University, 1997.

[3] V. ADVE and J. MELLOR-CRUMMEY. Advanced code generation for High Performance Fortran. In *Languages, Compilation techniques, and Run-time Systems for Scalable Parallel Systems,* S. Pande and D. Agrawal (Eds.), Chapter 18, Lecture Notes in Computer Science Series, Springer-Verlag, 1998 (to appear).

[4] A. V. AHO, R. SETHI, and J. ULLMAN. *Compilers: Principles, techniques, and tools.* Addison-Wesley, Reading, MA, second edition, 1986.

[5] F. E. ALLEN and J. COCKE. A program data flow analysis procedure. *Communications of the ACM,* 19(3):137–147, March 1976.

[6] S. AMARASINGHE and M. LAM. Communication optimization and code generation for distributed memory machines. In *Proc. SIGPLAN'93 Conference on Programming Language Design and Implementation,* pages 126–138, Albuquerque, NM, June 1993.

[7] A. ANCOURT, F. COELHO, F. IRIGOIN, and R. KERYELL. A linear algebra framework for static HPF code distribution. *Scientific Programming,* 6(1):3–28, Spring 1997.

[8] V. BALASUNDARAM, G. FOX, K. KENNEDY, and U. KREMER. An interactive environment for data partitioning and distribution. In *5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[9] U. BANERJEE. *Loop parallelization.* Kluwer Academic Publishers, 1994.

[10] P. BANERJEE, J. A. CHANDY, M. GUPTA, E. W. HODGES IV, J. G. HOLM, A. LAIN, D. J. PALERMO, S. RAMASWAMY, and E. SU. The PARADIGM compiler for distributed-memory multicomputers. *IEEE Computer,* 28(10):37–47, October 1995.

[11] Z. BOZKUS, A. CHOUDHARY, G. FOX, T. HAUPT, and S. RANKA. A compilation approach for Fortran 90D/HPF compilers. *Languages and Compilers for Parallel Computing,* U. Banerjee et al. (Eds.), Lecture Notes in Computer Science, Volume 768, pages 200–215, 1994.

[12] D. CALLAHAN and K. KENNEDY. Analysis of inter-procedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing,* 5(5):517–550, October 1988.

[13] S. CHAKRABARTI, M. GUPTA, and J.-D. CHOI. Global communication analysis and optimization. In *Proc. ACM SIGPLAN'96 Conference on Programming Language Design and Implementation,* pages 68–78, Philadelphia, PA, May 1996.

[14] S. CHATTERJEE, J. GILBERT, F. LONG, R. SCHREIBER, and S. TENG. Generating local addresses and communication sets for data-parallel programs. *Journal of Parallel and Distributed Computing,* 26(1):72–84, April 1995.

[15] B. CREUSILLET and F. IRIGOIN. Inter-procedural array region analyses. In *Proc. 8th International Workshop on Languages and Compilers for Parallel Computers*, pages 46–60, Columbus, Ohio, August 1995.

[16] I. FOSTER. Designing and building parallel programs. Addison-Wesley Publishing Company, Reading, MA 01867.

[17] M. GAREY and D. JOHNSON. *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman and Company, 1979.

[18] C. GONG, R. GUPTA, and R. MELHEM. Compilation techniques for optimizing communication on distributed-memory systems. In *Proc. International Conference on Parallel Processing,* Volume II, pages 39–46, St. Charles, IL, August 1993.

[19] E. GRANSTON and A. VEIDENBAUM. Detecting redundant accesses to array data. In *Proc. Supercomputing'91,* pages 854–865, Albuquerque, NM, November 1991.

[20] T. GROSS and P. STEENKISTE. Structured data-flow analysis for arrays and its use in an optimizing compiler. In *Software-Practice and Experience*, vol 20, no 2, pages 133–155, February 1990.

[21] M. GUPTA and P. BANERJEE. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[22] M. GUPTA, E. SCHONBERG, and H. SRINIVASAN. A unified data-flow framework for optimizing communication. *Languages and Compilers for Parallel Computing,* K. Pingali et al. (Eds.), Lecture Notes in Computer Science, Volume 892, pages 266–282, 1995.

[23] M. GUPTA, S. MIDKIFF, E. SCHONBERG, V. SESHADRI, D. SHIELDS, K. WANG, W. CHING, and T. NGO. An HPF compiler for the IBM SP-2. In *Proc. Supercomputing 95,* San Diego, CA, December 1995.

[24] S. K. S. GUPTA, S. D. KAUSHIK, C.-H. HUANG, and P. SADAYAPPAN. Compiling array expressions for efficient execution on distributed-memory machines. *Journal of Distributed and Parallel Computing,* 32(2):155–172, February 1996.

[25] M. W. HALL, S. HIRANANDANI, K. KENNEDY, and C.-W. TSENG. Inter-procedural compilation of Fortran D for MIMD distributed-memory machines. In *Proc. Supercomputing'92*, Minneapolis, MN, November 1992

[26] M. W. HALL, B. MURPHY, S. AMARASINGHE, S. LIAO, and M. LAM. Inter-procedural analysis for parallelization. In *Proc. 8th International Workshop on Languages and Compilers for Parallel Computers*, pages 61–80, Columbus, Ohio, August 1995.

[27] R. v. HANXLEDEN and K. KENNEDY. A code placement framework and its application to communication generation. Technical Report CRPC-TR93337-S, CRPC, Rice University, October 1993.

[28] R. v. HANXLEDEN and K. KENNEDY. Give-n-take – a balanced code placement framework. In *Proc. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation,* Orlando, FL, June 1994.

[29] J. L. HENNESSY and D. A. PATTERSON. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[30] S. HIRANANDANI, K. KENNEDY, and C. TSENG. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM,* 35(8):66–80, August 1992.

[31] W. KELLY, V. MASLOV, W. PUGH, E. ROSSER, T. SHPEISMAN, and DAVID WONNACOTT. The Omega Library interface guide. Technical Report CS-TR-3445, CS Dept., University of Maryland, College Park, March 1995.

[32] K. KENNEDY and N. NEDELJKOVIC. Combining dependence and data-flow analyses to optimize communication. In *Proc. 9th International Parallel Processing Symposium,* pages 340–346, Santa Barbara, CA, April 1995.

[33] K. KENNEDY, N. NEDELJKOVIC, and A. SETHI. A linear-time algorithm for computing the memory access sequence in data parallel programs. In *Proc. the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* Santa Barbara, CA, pages 102–111, July 1995.

[34] K. KENNEDY, N. NEDELJKOVIC, and A. SETHI. Communication generation for cyclic(k) distributions. In *Languages, Compilers, and Run-Time Systems for Scalable Computers,* B. Szymanski and B. Sinharoy (Eds.), Chapter 14, Kluwer Academic Publishers, 1996.

[35] K. KENNEDY and A. SETHI. A constrained-based communication placement framework, Technical Report CRPC-TR95515-S, CRPC, Rice University, 1995.

[36] K. KENNEDY and A. SETHI. A communication placement framework with unified dependence and data-flow analysis. *Proc. 3rd International Conference on High Performance Computing,* Trivandrum, India, December 1996.

[37] K. KENNEDY and A. SETHI. Resource-based communication placement analysis. *Languages and Compilers for Parallel Computing,* D. Sehr et al. (Eds.), Lecture Notes in Computer Science, Volume 1239, pages 369–388, Springer-Verlag, 1997.

[38] J. KNOOP, O. RUTHING, and B. STEFFEN. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems,* 16(4):1117–1155, July 1994.

[39] C. KOELBEL, D. LOVEMEN, R. SCHREIBER, G. STEELE, and M. ZOSEL. *High Performance Fortran Handbook*. The MIT Press, 1994.

[40] D. J. PALERMO, E. SU, J. A. CHANDY, and P. BANERJEE. Communication optimizations used in the PARADIGM compiler for distributed-memory multicomputers. In *Proc. International Conference on Parallel Processing*, St. Charles, IL, August 1994.

[41] C. POLYCHRONOPOULOS, M. B. GIRKAR, M. R. HAGHIGHAT, C. L. LEE, B. P. LEUNG, and D. A. SCHOUTEN. Parafrase-2: an environment for parallelizing, partitioning, synchronizing, and

scheduling programs on multiprocessors. In *Proc. the International Conference on Parallel Processing*, St. Charles IL, August 1989, pages II 39–48.

[42] W. PUGH. A practical algorithm for exact array dependence analysis. *Communications of the ACM,* 35(8):102–114, August 1992.

[43] E. SU, A. LAIN, S. RAMASWAMY, D. J. PALERMO, E. W. HODGES IV, and P. BANERJEE. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *Proc. 9th ACM International Conference on Supercomputing,* pages 424–433, Barcelona, Spain, July 1995.

[44] E. SU, D. J. PALERMO, and P. BANERJEE. Processor tagged descriptors: a data structure for compiling for distributed-memory multicomputers. In *Proc. Conf. on Parallel Architectures and Compilation Techniques*, Montreal, Canada, August 1994.

[45] R. E. TARJAN. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[46] A. THIRUMALAI and J. RAMANUJAM. Efficient computation of address sequences in data-parallel programs using closed forms for basis vectors. *Journal of Parallel and Distributed Computing,* 38(2):188–203, November 1996.

[47] A. VENKATACHAR, J. RAMANUJAM, and A. THIRUMALAI. Communication generation for block-cyclic distributions. *Parallel Processing Letters,* 7(2):195–202, June 1997.

[48] M. WOLFE. *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company, CA, 1996.

[49] X. YUAN, R. GUPTA, and R. MELHEM. An array data flow analysis based communication optimizer. In *Proc. 10th Annual Workshop on Languages and Compilers for Parallel Computing,* Minneapolis, Minnesota, August 1997.

[50] X. YUAN, R. GUPTA, and R. MELHEM. Demand-driven data flow analysis for communication optimization. *Parallel Processing Letters,* 7(4):359–370, December 1997.

[51] M. GERNDT. Updating distributed variables in local computations. In *Concurrency – Practice and Experience*, 2(3), pages 171–193, September 1990.

[52] H. ZIMA and B. CHAPMAN. *Supercompilers for parallel and vector computers,* ACM Press, 1991.