Optimizing Communication Using Global Dataflow Analysis

M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy

CPDC-TR-97-02

October 1997

Center for Parallel and Distributed Computing
Department of Electrical & Computer Engineering
Northwestern University
Technological Institute
2145 Sheridan Road, Evanston IL-60208

Optimizing Communication Using Global Dataflow Analysis

M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy

Abstract

In distributed-memory message passing architectures reducing communication cost is extremely important. In this paper, we present a technique to improve communication globally. Our approach is based on a combination of linear algebra framework and dataflow analysis, and can take arbitrary control flow into account. The distinctive feature of the algorithm is its accuracy in keeping communication set information, its support for general alignments and distributions including block-cyclic distributions, and capability of simulating some of the previous approaches by appropriate modifications. The method is currently being implemented in the PARADIGM compiler. We show how optimizations such as message vectorization, message coalescing, redundancy elimination can be supported by our new framework. Experimental results on an IBM SP-2 show that our technique is effective in reducing both the number as well as the volume of the communication.

1 Introduction

Distributed memory multiprocessors 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 very difficult task. The main objective behind the efforts such as High Performance Fortran (HPF) [17] and Fortran D [13] is to raise the level of programming by allowing the user 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. So it is imperative to reduce the frequency and volume of non-local accesses as much as possible. Several efforts have been aimed at reducing the communication overhead incurred. The most common optimization technique used by the previous researchers is message vectorization [22, 13, 7, 6]. 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 reducing the number of messages. For example, consider the program fragment shown in Figure 1(a), assuming all arrays are distributed across processors block-wise in the second dimension. Figures 1(b) and (c) show the naively inserted messages and message vectorization respectively,

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

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

  (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)
 ENDDO
ENDDO

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)
 ENDDO
ENDDO

    (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)
 ENDDO
ENDDO

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)
 ENDDO
ENDDO

  (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)
 ENDDO
ENDDO


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

    (d)
```

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

for a processor `p` before the loop bounds reduction. The notation `send{B,q,n}` means that `n` elements of array `B` should be sent to `q`; `recv{B,q,n}` is defined similarly. Notice that the version in Figure 1(c) reduces the message startup cost as well as the latency. Some of the researchers [13, 1] also considered message coalescing which 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.

The main problem with these optimizations 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 last version reduces both the number of messages and the communication volume.

Recently a number of authors have proposed techniques based on dataflow analysis to optimize communication across multiple loop nests [10, 12, 15, 9, 21]. Almost all of the approaches use a variant of Regular Section Descriptors (RSD) introduced by Callahan and Kennedy [8]. Two most notable representations are Available Section Descriptor (ASD) [12] and Section Communication Descriptor (SCD) [21]. For each array referenced in the program, an RSD is defined which describes the portion of the array that is 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 failing to support block-cyclic distributions, working with section descriptors may sometimes result in overestimation or underestimation of the communication sets. The inaccuracy resulting from these estimations may be linear with the number of dataflow formulations to be evaluated, thus defeating the purpose of global communication optimization.

In this paper, we show that the problem of global communication optimization 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 as well as global optimizations such as redundant communication elimination and communication hoisting. We present two different approaches—hoisting communication, and minimizing the number of messages—aimed to reducing communication overhead and show the tradeoff between these two. Using the linear algebra framework proposed by Ancourt et al. [5], our technique is able to handle the optimization problem at the granularity of individual array elements resulting in high accuracy. The task of code generation is made easier by our use of the Omega library [18] from the University of Maryland. We show that the global communication sets resulted from the dataflow analysis can easily be manipulated by the Omega system.

The rest of this paper is organized as follows. Section 2 briefly describes some important concepts such as interval analysis and linear algebra framework. We present our approach in detail in Section 3 and show how it uses both the linear algebra framework and dataflow analysis. Section 4 discusses the effect of hoisting communication vis-a-vis reducing the number of messages. In Section 5, we present the details of communication generation. Section 6 reports experimental results. Section 7 discusses related work and Section 8 concludes the paper.

2 Preliminaries

The main idea of this work is to show that global communication optimization problem can be put into a linear algebra framework and that doing is useful in practice. Our approach gives the compiler the capability 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 exploited in this paper.

2.1 Control Flow Graph (CFG)

We concentrate on structured programs with conditional statements and nested loops without arbitrary goto statements. We assume that subscript functions in array references in these programs are affine functions of enclosing loop indices.

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 [2]. 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 which 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

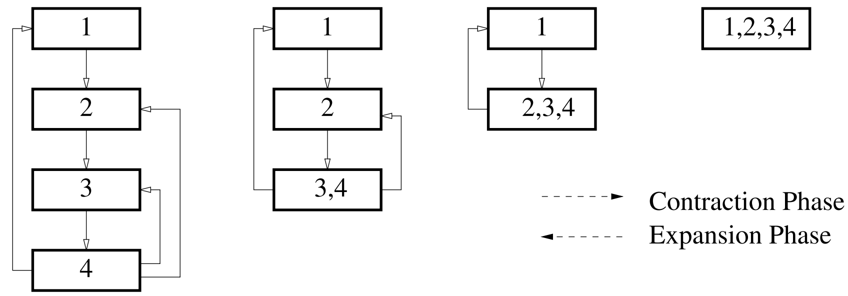

Figure 2: An example application of interval analysis.

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 [12], 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 eliminates all critical edges [16].

2.2 Interval Analysis

We assume that prior to our analysis, the compiler has performed all loop level transformations 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 [3], interval analysis consists of a *contraction* phase and an *elimination* 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 [2].

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 2 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 a loop which contains blocks 3 and 4.

2.3 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 [5] that linear algebra provides a powerful framework to generate code for distributed memory machines, taking into account compiler directives. The dataflow analysis described in this paper is being integrated into the PARADIGM compilation framework and PARADIGM already includes a linear algebra framework.

The PARADIGM compiler currently uses *owner-computes* rule, which assigns each computation to the processor that owns the data being computed [13, 22]. Our approach uses the affine framework introduced by Ancourt et al. [5]. In this framework, data arrays, templates and processors are all declared as Cartesian grids as in HPF [17]. Consider the following program fragment under a compilation scheme based on HPF-like directives and *owner-computes* rule.

```
!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) ···
      enddo
```

Let $\mathcal{R_L}$ = $\mathtt{X(}\gamma_L\mathtt{*i+}\theta_L\mathtt{)}$ and $\mathcal{R_R}$ = $\mathtt{X(}\gamma_R\mathtt{*i+}\theta_R\mathtt{)}$. 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 $\mathtt{p}$ and $\mathtt{q}$ denote two processors, we define the following sets.

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

Intuitively, the set $\mathtt{Own(X,q)}$ refers to the elements mapped onto processor $\mathtt{q}$ through compiler directives. The set of iterations to be executed by $\mathtt{q}$ due to a LHS reference $\mathcal{R_L}$ is given by $\mathtt{Compute(X,}\mathcal{R_L}\mathtt{,q)}$. Of course, during the execution of this iteration set, some elements (local or non-local) denoted by RHS reference $\mathcal{R_R}$ will be required; the set $\mathtt{View(X,}\mathcal{R_R}\mathtt{,q)}$ defines these elements. Finally, $\mathtt{CommSet(X,}\mathcal{R_R}\mathtt{,p,q)}$ defines the elements that should be communicated from processor $\mathtt{q}$ to processor $\mathtt{p}$ 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 $\mathtt{d}$ and $\mathtt{i}$ 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 $\mathtt{Own}$ set above is rather informal. For a more precise definition, we take into account the block-cyclic distribution and define the $\mathtt{Own}$ set as

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

where P = $p_u - p_l + 1$. In this formulation, $\mathtt{t} = \alpha * \mathtt{d} + \beta$ represents alignment information and $\mathtt{t} = \mathcal{C} * P * \mathtt{c} + \mathcal{C} * \mathtt{q} + \mathtt{l}$ denotes the distribution information. In other words, each array element $\mathtt{d}$ is mapped onto a point in a two-dimensional array. This point can be represented by a pair ($\mathtt{c}$,$\mathtt{l}$) and gives the local address of the data item in a processor. Simple $\mathtt{BLOCK}$ and $\mathtt{CYCLIC(1)}$ distributions can easily be handled within this framework by setting $\mathtt{c}$=0 and $\mathtt{l}$=0, respectively.

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 into account, another projection matrix $\mathcal{Y}$ can be used: $\mathcal{Y} * \mathtt{t} = \mathcal{C} * P * \mathtt{c} + \mathcal{C} * \mathtt{q} + \mathtt{l}$. All the elements on a collapsed dimension are stored on a same processor. For an in-depth discussion of the linear algebra framework for compiling distributed-memory programs, we refer the reader to [5].

Currently the PARADIGM compiler generates code based on this linear algebra framework by using the Omega library [18, 14] which consists of a set of routines for manipulating linear constraints over integer variables, sets and formulas. The work described in this paper improves the communication generation part of the PARADIGM compiler and is currently being implemented in it.

3 Dataflow Analysis using a Linear Algebra Framework

In this section, we define our dataflow 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 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`.

Operations on Communication Sets Since we define a communication set as a list of equalities and inequalities (which 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 easily 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 communication descriptors. Three operations, namely $+_d$, $-_d$, and $\cap_d$, are defined on these sets as follows:

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

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

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

In this paper, we sometimes use $\cup_c$ and $\cup_d$ instead of $+_c$ and $+_d$, respectively. It should be noted that although these operations are similar to those presented by Gong et al. [10], there is an important difference.

```
do i = i_l, i_u
  X(i-2,i) = Y(i-1,i-1) + X(i,1)
  do j = j_l, j_u
    X(i,j) = Y(i-2,i+2)
    if (cond)
      X(i-1,j+2) = Y(i-2,j-2)
      Y(i,j) = ...
    else
      X(i+1,j-3) = Y(i+3,j-3)
    endif
    Z(i,j) = Y(i-4,j)
  enddo
enddo
```

Figure 3: An example program fragment.

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. Neither [10] nor [15] can optimize these kinds of messages mainly due to their representation of communication sets. Of course, it would not be practically feasible to perform these operations on communication sets each time a dataflow equation is computed. In order to alleviate this problem, our technique computes sets of communication descriptors as symbolic expressions consisting set names and postpones the real computation of sets until the end of the analysis. Next we present our data flow framework.

3.2 Local (Intra-Interval) Analysis

In order to make the dataflow 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 dataflow 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,

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

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. In order to see how the computation of the `KILL` set proceeds, consider Figure 3. 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 sets of other arrays proceeds 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}) \\
\mathtt{KILL}(10,\mathtt{q}) &= \emptyset \\
\mathtt{Modified}(10,\mathtt{q}) &= [\mathtt{Modified}(7,\mathtt{q}) \cup \mathtt{Modified}(9,\mathtt{q})] \cup \mathtt{KILL}(10,\mathtt{q}) \\
&= \mathtt{Modified}(7,\mathtt{q}) \cup \mathtt{Modified}(9,\mathtt{q}) \\
\mathtt{KILL}(11,\mathtt{q}) &= \emptyset \\
\mathtt{Modified}(11,\mathtt{q}) &= \mathtt{Modified}(10,\mathtt{q}) \cup \mathtt{KILL}(11,\mathtt{q}) \\
&= \mathtt{Modified}(10,\mathtt{q}) \\
&= [\mathtt{Modified}(7,\mathtt{q}) \cup \mathtt{Modified}(9,\mathtt{q})] \\
&= \mathtt{KILL}(4,\mathtt{q}) \cup \mathtt{KILL}(6,\mathtt{q}) \cup \mathtt{KILL}(9,\mathtt{q}) \\
&= \Big\{\vec{d} \mid \vec{d} \in \mathtt{Own}(\mathtt{X},\mathtt{q}) \text{ and } \Big(\exists \imath, \jmath \text{ st. } \vec{d} = \mathtt{X}(\imath, \jmath) \text{ or } \vec{d} = \mathtt{X}(\imath-1, \jmath+2) \text{ or } \vec{d} = \mathtt{X}(\imath+1, \jmath-3)\Big) \\
&\qquad \text{and } i_l \le \imath \le i_u \text{ and } j_l \le \jmath \le 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:

$$\mathtt{KILL}(3,\mathtt{q}) \quad = \quad \mathtt{Modified}(11,\mathtt{q}).$$

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

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

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

$$\mathtt{KILL}(1,\mathtt{q}) \quad = \quad \mathtt{Modified}(3,\mathtt{q}).$$

Although for the sake of presentation we show the analysis here in terms of communication sets, the dataflow 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 which refers to arrays $\mathcal{N}_i$ is as follows

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

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 before being referenced.[1]

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 define

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

For the sake of simplicity, we assume one RHS reference per node, and use GEN(i,p,q) in the following. 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 of i. Then, assuming that Comm(succ(last(i)),q) = $\emptyset$, we have

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

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

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

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. level(i) gives the nesting level of the interval (loop), with the value 1 corresponding to the outermost loop in the nest.[2]

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:

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

As an example consider Figure 3 once more, this time concentrating on the computation of GEN sets due to the 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):

$$\begin{aligned}
\mathtt{GEN}(\mathtt{11},\mathtt{p},\mathtt{q}) \quad &= \quad \Big\{\vec{d} \mid \exists \imath, \jmath \text{ st. } i_l \le \imath \le i_u \text{ and } j_l \le \jmath \le j_u \text{ and } \vec{d} \in \mathtt{Own}(\mathtt{Y},\mathtt{q}) \text{ and } \vec{d} = Y(\imath - 4, \jmath) \text{ and} \\
&\qquad Z(\imath,\jmath) \in \mathtt{Own}(\mathtt{Z},\mathtt{p}) \text{ and not } (\exists \imath', \jmath', \text{ st. } \vec{d} = Y(\imath', \jmath') \text{ and } i_l \le \imath' \le i_u \text{ and } j_l \le \jmath' \le j_u \\
&\qquad \text{and } (\imath' = \imath \text{ and } \jmath' < \jmath))\Big\}.
\end{aligned}$$

---

[1]Notice that this process involves considering all LHS references within an interval. But since the largest scope for this analysis is a single loop nest and, as mentioned earlier, prior to analysis we keep pointers to all LHS references within a loop nest, this cost is manageable.

[2]In fact, there are more subtle issues here as statement positions may also need be taken into account when GEN is computed. For the sake of simplicity, we omit the details.

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

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

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 \le \imath \le i_u \text{ and } j_l \le \jmath \le j_u \text{ and } \vec{d} \in \mathtt{Own}(\mathtt{Y},\mathtt{q}) \text{ and } \vec{d} = Y(\imath-4,\jmath) \text{ and}\\
&\ 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 \le \imath' \le i_u \text{ and } j_l \le \jmath' \le j_u\\
&\ \text{ and } (\imath' = \imath \text{ and } \jmath' < \jmath))\}\\
&\ \cup\{\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 \mathtt{Own}(\mathtt{Y},\mathtt{q}) \text{ and } \vec{d} = Y(\imath-2,\jmath+2) \text{ and}\\
&\ X(\imath,\jmath) \in \mathtt{Own}(\mathtt{X},\mathtt{p}) \text{ and not } (\exists \imath', \jmath',\ \text{st. } \vec{d} = Y(\imath',\jmath') \text{ and } i_l \le \imath' \le i_u \text{ and } j_l \le \jmath' \le j_u\\
&\ \text{ and } (\imath' = \imath \text{ and } \jmath' < \jmath))\})\\
&\ \cup\\
&\ (\{\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 \mathtt{Own}(\mathtt{Y},\mathtt{q}) \text{ and } \vec{d} = Y(\imath-2,\jmath-2) \text{ and}\\
&\ X(\imath-1,\jmath+2) \in \mathtt{Own}(\mathtt{X},\mathtt{p}) \text{ and not } (\exists \imath', \jmath',\ \text{st. } \vec{d} = Y(\imath',\jmath') \text{ and } i_l \le \imath' \le i_u \text{ and } j_l \le \jmath' \le j_u\\
&\ \text{ and } (\imath' = \imath \text{ and } \jmath' < \jmath))\}\\
&\ \cap\{\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 \mathtt{Own}(\mathtt{Y},\mathtt{q}) \text{ and } \vec{d} = Y(\imath+3,\jmath-3) \text{ and}\\
&\ X(\imath+1,\jmath-3) \in \mathtt{Own}(\mathtt{X},\mathtt{p}) \text{ and not } (\exists \imath', \jmath',\ \text{st. } \vec{d} = Y(\imath',\jmath') \text{ and } i_l \le \imath' \le i_u \text{ and } j_l \le \jmath' \le j_u\\
&\ \text{ and } (\imath' = \imath \text{ and } \jmath' < \jmath))\})
\end{aligned}
$$

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

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

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):

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

Backward Analysis:

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

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

Forward Analysis:

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

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

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

Figure 4: Dataflow equations for optimizing communication.

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

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

Since `i` is the index of the outermost interval, there is no need to apply the $\mathcal{F}$ operator after reduction. We should emphasize that computing the `GEN` sets gives us all the communication that can be vectorized above a loop nest, i.e., our analysis easily handles message vectorization. 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:

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

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

3.3 Dataflow Equations

In our framework, any communication incurred is placed at the beginning of the nodes. Here, we concentrate on the computation of the `recv` sets only. Similar analysis applies to the `send` sets as well except for the fact that the definition of the communication set should be modified accordingly. Our dataflow 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 dataflow equations that we present are aggressive in hoisting communication. The input for the equations consists of the `GEN(i,p,q)`, `KILL(i,q)` and `POST_GEN(i,p,q)` sets as computed during the local analysis. The dataflow equations for the backward analysis are given by Equations (1) and (2) in Figure 4. Note that $\cap$ in this figure denotes $\cap_d$.

`SAFE_IN(i,p,q)` and `SAFE_OUT(i,p,q)` are the sets of communication descriptors which denote the elements that *can* safely be communicated at the beginning and end of block `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 dataflow analysis as well as some of the previous approaches like [15] adheres to. Equation (2), on the other hand, gives the set of elements that can safely be communicated at the beginning of block `i`, and makes use of the `GEN` and `KILL` sets. Intuitively, an element can be communicated at the beginning of block `i` if and only if it is either required (generated) by block `i` or it reaches at the end of block `i` 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 dataflow equations for the forward analysis are given by Equations (3), (4) and (5) in Figure 4; these equations observe the following rules:

- a node should not fetch data needed by a successor unless it dominates that successor; and
- 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 which contain the elements that *have been* communicated so far (at the beginning and end of the block `i` respectively) from `q` to `p`. On the other hand, `RECV(i,p,q)` denotes the set of communication descriptors which contain elements that *should* be communicated from `q` to `p` at the beginning of block `i` and is finally used by the communication generation portion of the compiler to generate the actual `recv` commands. 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 block except the ones that have already been communicated (`RECV_IN`). The elements that have been communicated at the end of block `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 which have been overwritten (killed) in `i`) and the elements communicated in `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 Dataflow 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

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

This assignment then triggers the next pass in the expansion phase. Of course, before the expansion phase starts $\mathtt{RECV\_IN}(\mathtt{i},\mathtt{p},\mathtt{q})^{1^{st} pass}$ is set to $\emptyset$. Figure 5 shows the overall algorithm `COMM-OPT` followed by our compiler to generate the `recv` sets. The `send` sets are generated analogously during the same process. Currently, our compiler inserts `send` and `recv` commands at the same point in the program.[3]

[3] It is also possible, albeit more complicated, to place asynchronous `send` and `recv` operations at the point of insertion of communication; and to insert a `wait` operation before the non-local data are referenced.

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

Figure 5: Communication optimization algorithm `COMM-OPT` based on dataflow analysis.

3.5 Example

We use the synthetic benchmark program shown in Figure 6(a) to illustrate our framework. We concentrate on the communication placement at the higher level CFG which contains no cycles. Figure 6(b) shows the message vectorized program with communication calls before the loop bounds reduction. The notation `send`{`B,q`} means that some elements of array `B` should be sent to `q`; `recv`{`B,q`} is defined similarly. In this example communication arises only due to references to array `B`. The loop-based communication analysis in PARADIGM places eight `send` and eight `recv` calls (in fact these are themselves loop nests) for eight RHS references marked as bold in Figure 6(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 Figure 6(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, we 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.[4]

For this example, our dataflow 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 6(b) are combined; that is our approach handles message coalescing easily.

[4] Notice that write to array `B` in the line `38` kills all the communication before it.

Table 1: Dataflow sets for the example shown in Figure 6.

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}	$\emptyset$

- 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 6(b).
- The communication in line `22` is reduced in volume.
- 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 6(c). It should be emphasized that the final communication sets are precise, i.e., there is no overestimation or underestimation of these sets. An important advantage is that these communication sets can be easily enumerated by Omega [14]. Notice that since all communication sets are enumerated in terms of abstract processors `p` and `q`; in general, if there are opportunities, message aggregation can also be performed easily.

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 as will be shown in Section 6. However, as also pointed out by others, hoisting the communication too eagerly can, under some circumstances, lead to excessive buffer requirement [15] and increase in the number of communication calls inserted [9]. One way to prevent these problems is to avoid hoisting the communication too high and reduce breaking of messages into smaller ones. Since the optimal placement of communication is NP-hard [9], we present a heuristic which stops accumulating communication sets as soon as it encounters a node that satisfies a predicate $\mathcal{P}(i)$. The content of this predicate depends on a specific implementation. A few alternatives are presented in Table 2. For example, in [15], the third alternative has been used. An implementation can, of course, employ a combination of these alternatives as well.

As an example, consider the predicate obtained by the conjunction of the first and second alternatives; i.e., $\mathcal{P}(i) = \{\texttt{KILL(i,q)} \neq \emptyset \texttt{ and } \texttt{GEN(i,p,q)} = \emptyset\}$. The dataflow equations given in Figure 8 are very similar to those presented in Figure 4. The only difference is in 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 each requiring a message of its own. This also eliminates some of the complexity of the resultant code. The situation is depicted in Figure 7. In the figure S_G and S_K denote the `GEN` and

```
!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 6: A synthetic benchmark program with message vectorized and globally optimized versions.

Table 2: Possible predicates to control hoisting.

$\mathcal{P}(i)$	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

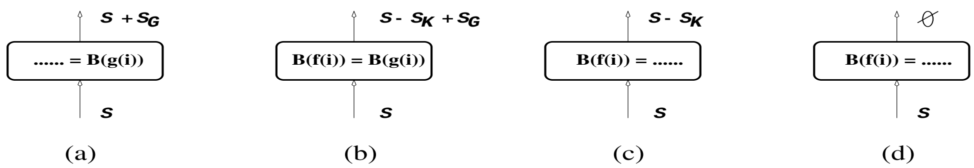

Figure 7: Handling of a node by different approaches: (a), (b), and (c) by the approach given in Figure 4; (a), (b), and (d) by the approach given in Figure 8.

KILL sets respectively for the node shown. The two approaches described in this paper behave similarly for the cases shown in Figures 7(a) and (b). But when a node performs only writes and no reads, the approach in Figure 4 still hoists the communication as shown in Figure 7(c) whereas the approach in Figure 8 stops hoisting as shown in Figure 7(d). That is, the new approach does not issue a communication call unless there are additional elements required by the node.

To compare our new approach with the previous one (Figure 4), we now consider the example given in Figure 9, a modified version of the last part of the program shown in Figure 6. 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) which uses message vectorization only. The bottom part of Table 3, on the other hand, presents SAFE_IN and RECV sets for our new approach. We observe that the number of messages is 1 and we have some 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 the reduced number of communication messages.

5 Communication Generation

Our communication code generator uses the Omega library from University of Maryland [18, 14]. After the SEND and RECV 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 6 (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 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 10 shows the communication sets for S_4, S_{11}, S_{33} and $S' = S_4 +_c S_{11} +_c S_{33}$ as

Backward Analysis:

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

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

Forward Analysis:

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

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

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

Figure 8: Dataflow equations for optimizing communication.

Table 3: Dataflow sets for the example shown in Figure 9.

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}	$\emptyset$

represented in Omega. A set element in this figure is represented as a quadruple `[q,p,`d_1`,`d_2`]` meaning that element `[`d_1, 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`. In Figure 10, 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 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 data space.

The code enumerating the `[q,`d_1, d_2`]` triples for $(S_{25} +_c S_{33}) -_c (S_4 +_c S_{11} +_c S_{33})$ is shown in Figure 11:(a) as C code for the `send` set and in Figure 11:(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.

```
!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 9: An example program fragment to show solution to the problem due to aggressive hoisting.

6 Implementation and Results

6.1 PARADIGM and its Communication Optimization Framework

The PARADIGM compiler [6] accepts a sequential Fortran program and generates an SPMD node program with explicit message passing. The compilation process consists of several passes, including program analysis, data partitioning, computation partitioning, communication analysis and processor mapping. One of the most important steps in the compilation framework is to determine the communication sets and the associated senders and receivers. Currently, communication analysis pass can handle block, cyclic and block-cyclic distributions for a restricted set of cases. For block and cyclic distributions, the compiler uses processor tagged descriptors (PTD) [20] to provide a uniform representation of partitioning for each processor. For block-cyclic distributions, on the other hand, the compiler employs either of two *engines* to enumerate arbitrary iteration and data sets. The first engine is an implementation [19] of Fourier-Motzkin elimination technique within Mathematica, which can manipulate symbolic as well as non-linear equalities and inequalities. The second one is based on the Omega library [18] from University of Maryland. The original communication analysis in PARADIGM was based on a single loop-nest, i.e., it handles one loop nest at a time and attempts to optimize it by using standard analysis and optimization techniques such as message vectorization. Restricting the scope of the communication optimization to a single loop nest at a time however prevents cross-loop optimizations such as redundant communication elimination. The communication framework described in this paper is currently being implemented in the PARADIGM compiler. When it is fully implemented it will replace the current loop nest based framework.

6.2 Results

In this section, we report experimental results for four programs including our synthetic benchmark in Figure 6, whose characteristics are given in Table 4.

$$
\begin{aligned}
S_4 &:= \{[q,p,d_1,d_2] : \exists(i,c_1,l_1,c_2,l_2,c_3,l_3 : 2 \le i \le 31 \wedge 16c_1+4q+l_1 = d_1 \wedge 0 \le q \le 3 \wedge 0 \le l_1 \le 3 \wedge i-1 = d_1 \\
&\quad \wedge 16c_3+4p+l_3 = i \wedge 0 \le l_3 \le 3 \wedge 16c_2+4p+l_2 = d_1 \wedge -1 \le l_2 \le 3 \wedge 0 \le p \le 3 \wedge 2 \le d_2+1 \le 127 \wedge p \ne q)\}; \\
S_{11} &:= \{[q,p,d_1,d_2] : \exists(i,c_1,l_1,c_2,l_2,c_3,l_3 : 2 \le i \le 127 \wedge 16c_1+4q+l_1 = d_1 \wedge 0 \le q \le 3 \wedge 0 \le l_1 \le 3 \wedge i-1 = d_1 \\
&\quad \wedge 16c_3+4p+l_3 = i \wedge 0 \le l_3 \le 3 \wedge 16c_2+4p+l_2 = d_1 \wedge -1 \le l_2 \le 3 \wedge 0 \le p \le 3 \wedge 2 \le d_2+1 \le 31 \wedge p \ne q)\}; \\
S_{33} &:= \{[q,p,d_1,d_2] : \exists(i,c_1,l_1,c_2,l_2,c_3,l_3 : 2 \le i \le 127 \wedge 16c_1+4q+l_1 = d_1 \wedge 0 \le q \le 3 \wedge 0 \le l_1 \le 3 \wedge i-1 = d_1 \\
&\quad \wedge 16c_3+4p+l_3 = i \wedge 0 \le l_3 \le 3 \wedge 16c_2+4p+l_2 = d_1 \wedge -1 \le l_2 \le 3 \wedge 0 \le p \le 3 \wedge 64 \le d_2+1 \le 127 \wedge p \ne q)\}; \\
S' &:= \{[q,q+1,d_1,d_2] : \exists(\alpha : d_1 = 3+4q+16\alpha \wedge 0 \le q \le 2 \wedge 1 \le d_2 \le 126 \wedge 4q+3 \le d_1 \le 4q+19)\} \\
&\quad \cup\{[q,p,d_1,d_2] : 4p+15 \le d_1 \le 4q+3 \wedge 1 \le d_2 \le 126 \wedge q \le 3 \wedge 0 \le p\} \\
&\quad \cup\{[q,p,d_1,d_2] : \exists(\alpha : 1 \le d_1 \le 4q+99 \wedge 1 \le d_2 \le 30 \wedge q \le 3 \wedge 13+d_1 \le 4q+16\alpha \wedge 4p+16\alpha \le 1+d_1 \wedge 0 \le p)\} \\
&\quad \cup\{[q,q+1,d_1,d_2] : \exists(\alpha : d_1 = 3+4q+16\alpha \wedge 0 \le q \le 2 \wedge 1 \le d_2 \le 30 \wedge 4q+3 \le d_1 \le 4q+115)\} \\
&\quad \cup\{[q,p,d_1,d_2] : \exists(\alpha : 1 \le d_1 \le 4q+99 \wedge 63 \le d_2 \le 126 \wedge q \le 3 \wedge 13+d_1 \le 4q+16\alpha \wedge 4p+16\alpha \le 1+d_1 \wedge 0 \le p)\} \\
&\quad \cup\{[q,q+1,d_1,d_2] : \exists(\alpha : d_1 = 3+4q+16\alpha \wedge 0 \le q \le 2 \wedge 63 \le d_2 \le 126 \wedge 4q+3 \le d_1 \le 4q+115\}
\end{aligned}
$$

Figure 10: Omega Relations corresponding to for the example shown in Figure 6.

```
if (P == 3) {
  for (j = 31; j <= 111; j += 16) {
    for (k = 31; k <= 62; k++) {
      process(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(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(P-1,j,k);
    }
  }
}
if (P == 0) {
  for (j = 31; j <= 111; j += 16) {
    for (k = 31; k <= 62; k++) {
      process(3,j,k);
    }
  }
}
```

(b) `recv` set.

Figure 11: Codes for enumerating $(S_{25} +_c S_{33}) -_c (S_4 +_c S_{11} +_c S_{33})$ for the example shown in Figure 6.

We measure the effectiveness of our approach statically in terms of number of messages and communication volume across all processors. For our synthetic benchmark we also report the execution times obtained on IBM SP-2. Table 5 shows the four different versions of a program.

Table 6 presents the results of static analysis and run-time measurements on IBM SP-2 for our synthetic benchmark program given in Figure 6 for 4 and 8 processors. For this program, both of our approaches (Figures 4 and 8) result in the same optimized code given in Figure 6(c). The results show that there is an average of 44% reduction in the number of messages, and 49% reduction in the communication volume over all processors. These reductions in turn cause an average of 15% reduction in the execution time as shown in the bottom part of Table 6. Notice that as compared with the pure block distribution, the reduction in the execution time for block-cyclic distribution is much higher.

Table 7 presents the results of static analysis for a single outermost iteration of the `hydro` benchmark on 32 processors with an input size of 512. Before we run our optimizer, two modifications have been made to the program: First, the second loop nest is distributed over its statements. Second, the loop bounds in the first loop nest are reduced to 1/4th of the original values. As can be seen from Table 7, both of our approaches outperform the vectorized program, averaging a 38% reduction in the number of messages, and a

Table 4: Programs in our experiment set and their characteristics.

PROGRAM	SUITE	NUMBER OF LINES	NUMBER OF ARRAYS	DESCRIPTION
synthetic benchmark	-	52	five 2D	stencil-like computation
hydro	livermore	38	nine 2D	2D hydrodynamics
tomcatv	spec	190	seven 2D, two 1D	2D mesh generation
swim256	spec	428	fourteen 2D	shallow water equation solver

Table 5: Different versions used in the experiments.

VERSION	EXPLANATION
base	message vectorization
priv	message vectorization + privatization
opt1	optimized (using the equations in Figure 4)
opt2	optimized (using the equations in Figure 8)

22% reduction in the communication volume. We note that opt2 is the most successful version at minimizing the number of messages. We would like to remind the reader that none of the previous approaches were able to optimize communication for block-cyclic distributions. This is unfortunate, because in general the communication volume in block-cyclic distributions is much higher than that of the pure block distributions.

For tomcatv and swim256 benchmarks, opt1 and opt2 result in the same optimized program. Table 8 presents the results of static analysis for a single outermost iteration of tomcatv on 8 and 32 processors with an input size of 512. (CYCLIC(4),*)T corresponds to a version of the program in which the first dimension of each two dimensional array is aligned with a template such that an element k in this dimension is mapped onto element 7k in the template. After the alignment, the template dimension is distributed in CYCLIC(4) fashion. From Table 8, we see that, there is an average of 75% reduction in the number of messages and in the communication volume as compared to the priv version. Notice also that for the cases (CYCLIC(4),*), (CYCLIC,*), (CYCLIC(4),*)T and (CYCLIC(5),*) our approach performs very well reducing both the communication volume and number of messages.

Finally, Table 9 presents the results of static analysis for 40 iterations of the swim256 benchmark on 8 processors with an input size of 512 for (BLOCK,*), (CYCLIC(4),*) and (CYCLIC,*) distributions. On the average, there is a 47% reduction in the number of messages, and a 44% reduction in the communication volume. It should be mentioned that, during the optimization of this program, the compiler did not perform procedure inlining.

Overall, we can conclude that our approach is successful at reducing the number of communication calls, the communication volume, and the execution time.

7 Related Work

Most of the previous efforts considered communication optimization at loop level. Although each approach has its own unique features, the general idea is to apply an appropriate combination of message vectorization, message coalescing and message aggregation [22, 13, 7, 6].

More recently some researchers have proposed techniques based on dataflow analysis in order to optimize communication across multiple loop nests. Amarasinghe and Lam [4] present several algorithms to

Table 6: Results of the static and run-time analyses for our synthetic benchmark.

program	number of messages				communication volume (in array elements)			
	p = 4		p = 8		p = 4		p = 8	
	(BLOCK,*)	(CYCLIC(4),*)	(BLOCK,*)	(CYCLIC(4),*)	(BLOCK,*)	(CYCLIC(4),*)	(BLOCK,*)	(CYCLIC(4),*)
base	44	56	88	112	4,288	40,288	8,576	40,288
opt1	24	32	48	64	2,784	15,240	5,568	14,168
% imprv.	45%	43%	45%	43%	36%	62%	35%	64%
opt2	24	32	48	64	2,784	15,240	5,568	14,168
% imprv.	45%	43%	45%	43%	36%	62%	35%	64%

execution times (in milliseconds) on IBM SP-2:

program	p = 4		p = 8	
	(BLOCK,*)	(CYCLIC(4),*)	(BLOCK,*)	(CYCLIC(4),*)
base	5.24	8.18	3.21	4.85
opt1	4.82	6.26	3.07	3.73
% imprv.	8%	23%	4%	23%
opt2	4.82	6.26	3.07	3.73
% imprv.	8%	23%	4%	23%

Table 7: Results of the static analysis for `hydro` on `32` nodes.

program	number of messages		communication volume	
	(BLOCK,*)	(CYCLIC(3),*)	(BLOCK,*)	(CYCLIC(3),*)
base	256	512	103,680	576,695
opt1	184	288	81,717	441,301
% imprv.	28%	44%	21%	23%
opt2	176	256	81,717	441,301
% imprv.	31%	50%	21%	23%

Table 8: Results of the static analysis for `tomcatv`.

number of messages:

program	p = 8				p = 32	
	(BLOCK,*)	(CYCLIC(4),*)	(CYCLIC,*)	(CYCLIC(4),*)T	(BLOCK,*)	(CYCLIC(5),*)
base	420	480	480	960	1,860	1,920
priv	112	128	128	256	496	512
opt1	28	32	32	64	124	128
% imprv.	75%	75%	75%	75%	75%	75%
opt2	28	32	32	64	124	128
% imprv.	75%	75%	75%	75%	75%	75%

communication volume (in array elements):

program	p = 8				p = 32	
	(BLOCK,*)	(CYCLIC(4),*)	(CYCLIC,*)	(CYCLIC(4),*)T	(BLOCK,*)	(CYCLIC(5),*)
base	214,200	3,890,280	15,606,000	15,606,000	948,600	3,121,200
priv	57,120	1,040,000	4,161,600	4,161,600	252,960	832,320
opt1	14,436	261,120	1,044,480	1,044,480	63,488	208,896
% imprv.	75%	75%	75%	75%	75%	75%
opt2	14,436	261,120	1,044,480	1,044,480	63,488	208,896
% imprv.	75%	75%	75%	75%	75%	75%

Table 9: Results of the static analysis for `swim256` on `8` nodes.

program	number of messages			communication volume (in array elements)		
	(BLOCK,*)	(CYCLIC(4),*)	(CYCLIC,*)	(BLOCK,*)	(CYCLIC(4),*)	(CYCLIC,*)
base	4,691	5,212	5,212	2,188,032	34,277,622	135,778,800
opt1	2,489	2,770	2,770	1,270,230	18,936,460	73,517,520
% imprv.	47%	47%	47%	42%	45%	46%
opt2	2,489	2,770	2,770	1,270,230	18,936,460	73,517,520
% imprv.	47%	47%	47%	42%	45%	46%

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 dataflow information, they do not allow loop nests within conditionals.

Granston and Veidenbaum [11] present 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. [10] describe optimizations that reduce communication overhead and execution time. Their optimizations include elimination of redundant communication and combining messages. However their approach works for only singly nested loops and unidimensional arrays.

Gupta et al. [12] present a framework to optimize communication based on dataflow 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.

The communication optimization method described in [15] is reference-oriented and communication sets can be represented by references. For example, a communication set that is originated from an RHS reference such as `X[i-1]` can be represented by `X[i-1]`. An assignment to an array to be communicated can restrict the placement of a `send` (or `recv`) primitive even if only a portion of the elements is overwritten. That is, the dataflow procedure used by [15] is coarse-grained. In comparison, our approach works on granularity level of individual array elements. The preciseness of communication sets is maintained by keeping them as linear equalities and inequalities.

The approach described by [9] 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. [21] present a communication optimization approach based on array dataflow 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.

Earlier in the paper we discussed the difficulty of embedding alignment and distribution information in the RSDs. This results in lack of support for general block-cyclic distributions. Apart from this, array dataflow analysis based on RSDs suffer from the fact that regular sections are not closed under union and difference operations. To see the problem, consider the program fragment given in Figure 12(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 of the communication is from processor `0` to processor `1`. The problem here is that a dataflow approach based on RSDs trying to

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

do t = 1, T
  do i = 0, 49 -------------> Si = (1:200:4)
    Y(i+500) = X(4*i+1)
  enddo

  do j = 17, 99 -------------> Sj = (50:300:3)
    Y(j+500) = X(3*j-1)
  enddo

  do i = 1, 1000
    X(i) = f(Y(i),X(i))
  enddo
enddo
```

(a)

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

(b)

Figure 12: (a) An example code fragment which shows shortcoming of RSDs. (b) Code generated by using the Omega to enumerate the communication set in (a).

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

Our framework, on the other hand, can represent these sets as $S_i := \{[d] : \exists(\alpha : d = 1{+}4\alpha \text{ and } 1 \leq d \leq 197)\}$ and $S_j := \{[d] : \exists(\alpha : 1 + d = 3\alpha \text{ and } 50 \leq d \leq 299)\}$. Then by using the Omega, the code shown in Figure 12(b) 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 $(-)$ operation as well. 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 dataflow process proceeds, making the final communication sets highly inaccurate.

Adve et al. [1] describe an integer set based approach for analysis and code generation for data parallel programs. Although they also consider performing message vectorization and coalescing for general patterns, they do not attempt to optimize cross-loop communication.

In this paper we used ideas from the linear algebra framework [5] and dataflow analysis [3, 2] developed for performing optimizations on the CFG representation of the programs. We have shown that these two techniques can be combined in a manner which makes handling the global communication optimization problem easy even in the presence of general block-cyclic distributions. We should emphasize that the dataflow equations given by Figures 4 and 8 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 redefining the sets in terms of equalities and inequalities. This would not only give an approach 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: dataflow 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. To our knowledge, this is the first attempt to use a linear algebra framework to optimize communication globally. 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 dataflow analysis where the Omega library is called to generate actual sets in terms of equalities and inequalities. Our technique is currently being implemented within the PARADIGM compilation framework. 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.

References

[1] Vikram Adve, John Mellor-Crummey, and Ajay Sethi. An integer set framework for HPF analysis and code generation. Technical Report TR97-275, Computer Science Dept., Rice University, 1997.

[2] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.

[3] F. E. Allen, and J. Cocke. A Program data flow analysis procedure. In *Communications of the ACM*, 19(3), pp. 137-147, March 1976.

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

[5] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. In *Proc. 4th International Workshop on Compilers for Parallel Computers*, Delft, the Netherlands, 1993.

[6] 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. In *IEEE Computer*, Vol. 28, No. 10, pages 37-47, October 1995.

[7] Z. Bozkus, A. Choudhar, G. Fox, T. Haupt, and S. Ranka. A compilation approach for Fortran 90D/HPF compilers on distributed memory MIMD computers. In *Proc. 6th Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 1993.

[8] D. Callahan, and K. Kennedy. Analysis of inter-procedural side effects in a parallel programming environment. In *J. Parallel Dstrib. Comput. 5*, 5 (Oct. 1988), pp. 517-550.

[9] S. Chakrabarti, M. Gupta, and J.-D. Choi. Global communication analysis and optimization, In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.

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

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

[12] M. Gupta, E. Schonberg, and H. Srinivasan. A unified dataflow framework for optimizing communication. In *Proc. 7th Workshop on Languages and Compilers for Parallel Computing*, Ithaca NY, August 1994.

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

[14] 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.

[15] K. Kennedy, and A. Sethi. A constrained-based communication placement framework, Technical Report CRPC-TR95515-S, CRPC, Rice University, 1995.

[16] J. Knoop, O. Ruthing, and B. Steffen. Optimal code motion: Theory and practice. In *ACM Transactions on Programming Languages and Systems*, 16(4):1117-1155, July 1994.

[17] C. Koelbel, D. Lovemen, R. Schreiber, G. Steele, and M. Zosel. *High Performance Fortran handbook*. The MIT Press, 1994.

[18] W. Pugh. A practical algorithm for exact array dependence analysis. In *Communications of the ACM*, 35(8):102-114, August 1992.

[19] 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. the 9th ACM International Conference on Supercomputing*, pages 424-433, Barcelona, Spain, July 1995.

[20] E. Su, D. J. Palermo, and P. Banerjee. Processor Tagged Descriptors: A data structure for compiling for distributed-memory multicomputers. In *Proc. the 1994 International Conference on Parallel Architectures and Compilation Techniques*, pages 123-132, Montreal, Canada, August 1994.

[21] 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.

[22] H. Zima, H. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. In *Parallel Computing*, 6:1-18, 1988.