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

M. KANDEMIR
Syracuse University
P. BANERJEE and A. CHOU DHARY
Northwestern University
J. RAM ANUJAM
Louisiana State University
and
N. SHENOY
Northwestern University

Reducing communication overhead is extremely important in distributed-memory message-passing architectures. In this article, 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 it 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).
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) [Koelbel et al. 1994] and Fortran D [Hiranandani et al. 1992] 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 nonlocal data.

On such machines, the time (cost) to access nonlocal 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 [Hennessy and Patterson 1990]. Therefore, it is imperative that the frequency and volume of nonlocal 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 [Foster 1994]. 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 [Balasundaram et al. 1990; Banerjee et al. 1995; Bozkus et al. 1994; Hiranandani et al. 1992; Gerndt 1990]. In message vectorization, instead of naively inserting send and recv operations just before references to nonlocal 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 blockwise 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 [Hiranandani et al. 1992]) and guard insertion (a technique that guarantees correct execution of statements within loop nests). The notation $\text{send}\{B, p+1, 256\}, \text{recv}\{B, p-1, 255\}$ means that $n$ elements of array $B$ should be sent to processor $p$; $\text{recv}\{B, q, n\}$ 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 [Adve et al. 1997; Hiranandani et al. 1992] 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 nonlocal elements to get transferred to the memories of the processors that

```
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
```

```
DO j = 2, 255
DO i = 2, 256
   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
```

```
send \{B,p+1,256\}, 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
```

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

```
DO j = 2, 255
DO i = 2, 266
   C(i,j)=B(i,j-1)+C(i,j)
END DO
END DO
```

```
DO j = 2, 255
DO i = 2, 266
   C(i,j)=B(i,j-1)+C(i,j)
END DO
END DO
```

Fig. 1. (a) A code fragment. (b) Naive communication placement. (c) Message vectorization. (d) Global communication optimization.
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 interloop 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 this version, compared with the message-vectorized program in Figure 1(c), 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 [Chakrabarti et al. 1996; Gong et al. 1993; Gupta et al. 1995b; Kennedy and Sethi 1995; Yuan et al. 1997a; 1997b]. Most of these approaches use a variant of Regular Section Descriptors (RSD) introduced by Callahan and Kennedy [1998]. Two most notable representations are the Available Section Descriptor (ASD) [Gupta et al. 1995b] and Section Communication Descriptor (SCD) [Yuan et al. 1997a; 1997b]. 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 sometimes 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 blockwise 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 communication cannot be taken out of \( t \) loop because of a data dependence [Wolf 1996; Zima and Chapman 1991], the redundant communication will occur \( T \) times.

On the other hand, we represent these sets in our framework 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 library [Kelly et al. 1995], 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
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 article, 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, and 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. [1997], 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 [Kelly et al. 1995; Pugh 1992] 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 $O(2^{2^n})$, the library is much more efficient for the practical cases that arise in compilation.

---

Fig. 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.
We compare our approach both qualitatively and quantitatively to the previous work which focused on a single loop nest at a time.

The remainder of this article 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 article. 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-à-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 the number of communication messages, volume of communication, and execution time. Section 7 discusses related work, and Section 8 concludes the article.

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 article.

2.1 Control Flow Graph

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 maybe at the end [Aho et al. 1986]. 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 $G = (V, E)$ where each $v \in V$ represents either a basic block or a (reduced) interval that represents a loop, and each $e \in E$ represents an edge between blocks. In this article, 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 $n \in V$ lies on a path from $s$ to $t$. We define the sets of all successors
and predecessors of a node \( n \) as \( \text{succ}(n) = \{ m \mid (n, m) \in \mathcal{E} \} \) and \( \text{pred}(n) = \{ m \mid (m, n) \in \mathcal{E} \} \), 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 \( j \in \text{dom}(i) \). The CFGs we consider also have the following properties: (a) empty else branches are added to if/endif constructs; (b) all the nonlocal references in the loop bounds and if-conditions are taken just above the respective constructs; and (c) as in Gupta et al. [1995b], 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, eliminates all critical edges [Knoop et al. 1994].

2.2 Interval Analysis

We assume, that, prior to our analysis, the compiler has performed all loop-level transformations [Banerjee 1994; Wolfe 1996; Zima and Chapman 1991] 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 Allen and Cocke [1976], 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 [Aho et al. 1986]. We use a version of the interval detection algorithm that identifies Tarjan’s intervals [Tarjan 1974].

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 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 [Kennedy and Sethi 1997; van Hanxleden and Kennedy 1993; 1994].

It should be noted that since we assume that our input programs are structured, irreducible (intermediate) CFGs [Aho et al. 1986] cannot 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 antidependence 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 to the dependence is said to carry that dependence. In-depth discussion of data dependence analysis techniques is beyond the scope of this article and can be found elsewhere [Wolfe 1996; Zima and Chapman 1991].

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 Ancourt et al. [1997] 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 [Balasundaram et al. 1990; Gerndt 1990; Hiranandani et al. 1992]. In this article, 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 article.

Our approach uses the affine framework introduced by Ancourt et al. [1997]. In this framework, data arrays, templates, and processors are all declared as Cartesian grids as in HPF [Koelbel et al. 1994]. 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(¢) attribute indicates that the template (or array) dimension in question will be partitioned into blocks of size ¢, 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(¢) distribution.

```plaintext
real X (al; au)
!HPF$ template T (tl; tu)
!HPF$ processors PROC (pl; pu)
!HPF$ align X (j) with T (α*j + β)
!HPF$ distribute T (cyclic(¢)) onto PROC
DO i = il, iu
   X (γL*i + θL) = ... X (γR*i + θR) ...
END DO
```

Let R_L = X (γL * i + θL) and R_R = X (γR * i + θR). In the rest of the article, 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.

Own (X, q) = \{d | d ∈ X and is owned by q\}

Compute (X, R_L, q) = \{i | γL * i + θL ∈ Own (X, q) and il ≤ i ≤ iu\}

View (X, R_R, q) = \{d | ∃ i st. i ∈ Compute (X, R_L, q) and d = X(γR * i + θR) and il ≤ i ≤ iu\}

CommSet (X, R_R, p, q) = Own (X, q) ∩ View (X, R_R, p).

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 $R_{x}$ is given by $\text{Compute}(X, R_{x}, q)$. Of course, during the execution of this local iteration set, some elements (local or nonlocal) denoted by the RHS reference $R_{x}$ will be required; the set $\text{View}(X, R_{x}, q)$ defines these elements. Finally, $\text{CommSet}(X, R_{x}, p, q)$ defines the elements that should be communicated from processor $q$ to processor $p$ due to reference $R_{x}$.

It should be noted that in general there may be more than one RHS reference, and the computation may involve multidimensional arrays and a multilevel nest in which case $d$ and $i$ denote data and iteration vectors respectively. Also in the most general case, $\alpha$, $\gamma_{L}$, and $\gamma_{R}$ are matrices, and $\beta$, $\theta_{L}$, and $\theta_{R}$ are vectors.

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

$$\text{Own}(X, q) = \{d| \exists t, c, l \text{ such that } t = \alpha \cdot d + \beta$$

and

$$t = \mathcal{C} \cdot p \cdot c + \mathcal{C} \cdot q + 1 \text{ and } a_{l} \leq d \leq a_{u}$$

and $p_{i} \leq q \leq p_{u}$ and $t_{l} \leq t \leq t_{u}$ and $0 \leq l \leq \mathcal{C} - 1\},$$

where $p = p_{u} - p_{l} + 1$. In this formulation, $t = \alpha \cdot d + \beta$ represents alignment information, and $t = \mathcal{C} \cdot p \cdot c + \mathcal{C} \cdot q + 1$ 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 $c = 0$ and $l = 0$, 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 the $c$ coordinate whereas the vertical dimension denotes $l$. For example, the 55th element of the (global) array is mapped onto Processor 1 with $c = 4$ and $l = 3$ as local coordinates.

The relation $t = \alpha \cdot d + \beta$ can be generalized by adding a replication matrix $\mathcal{V}$ which eliminates the replicated dimension from the equations: $\mathcal{V} \cdot t = \alpha \cdot d + \beta$. 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: $\mathcal{Y} \cdot t = \mathcal{C} \cdot p \cdot c + \mathcal{C} \cdot q + 1$. 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 article we assume (1) that identity alignment is used and (2) that arrays are directly distributed across processors. For an in-depth discussion of the linear algebra frame-
work for compiling distributed-memory programs, we refer the reader to Ancourt et al. [1997].

2.5 Parafrase-2 and Omega Library

Parafrase-2 [Polychronopoulos et al. 1989] 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 [Kelly et al. 1995]. 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 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

3.1.1 Communication Descriptors and Communication Sets. A communication descriptor can be defined as a pair $(\mathbb{R}, \mathbb{F})$, where $\mathbb{R}$ is an array identifier (name), and $\mathbb{F}$ is the communication set associated with $\mathbb{R}$. 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 $\{d| p \text{ owns } d \text{ and is required by } q\}$ except for the KILL set, which defines the set of elements written (killed) by $q$. In these set definitions $d$ refers to a multidimensional array element.

3.1.2 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 $\{d|P(d)\}$ where $P(.)$ is a predicate. Let $\{d|P(d)\}$ and $\{d|Q(d)\}$ be two communication sets. We define the operations $\cap$, $\cup$, and $\odot$ on communication sets as follows:

$$\{d|P(d)\} \cap_c \{d|Q(d)\} = \{d|P(d) \text{ or } Q(d)\}$$

$$\{d|P(d)\} -_c \{d|Q(d)\} = \{d|P(d) \text{ and not } Q(d)\}$$

$$\{d|P(d)\} \odot_c \{d|Q(d)\} = \{d|P(d) \text{ and not } Q(d)\}$$

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

3.1.3 Operations on Sets of Communication Descriptors. Let $\Xi = (\mathbb{R}, \mathbb{F})$ 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 $\mathcal{N}(\Xi) = \mathbb{R}$ and $\mathcal{M}(\Xi) = \mathbb{F}$.

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

$$\mathcal{D}_1 +_{\mathcal{D}} \mathcal{D}_2 = \{\Xi| \Xi \in \mathcal{D}_1 \text{ and } \forall \Xi' \in \mathcal{D}_2 \mathcal{N}(\Xi) \neq \mathcal{N}(\Xi')\}$$

$$\cup \{\Xi| \Xi \in \mathcal{D}_1 \text{ and } \forall \Xi' \in \mathcal{D}_2 \mathcal{N}(\Xi) \neq \mathcal{N}(\Xi')\}$$

$$\cup \{\Xi| \exists \Xi' \in \mathcal{D}_1, \Xi'' \in \mathcal{D}_2 \text{ st. } \mathcal{N}(\Xi)\}$$
When there is no ambiguity, we also use $\mathcal{c}$ and $\mathcal{d}$ instead of $\mathcal{c}$ and $\mathcal{d}$, respectively. It should be noted that although these operations are similar to those given by Gong et al. [1993], 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 $\{ [z] : \mathcal{Q}_1(z) \lor \mathcal{Q}_2(z) \}$ 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 rewritten in terms of equalities and inequalities, and then the $\text{simplify}$ utility of the Omega library is used to simplify them. Our experience shows that this approach requires a manageable 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 (Intrainterval) 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 $\mathcal{R}_x(i)$ and $\mathcal{R}_y(i)$ be the data elements obtained from references $\mathcal{R}_x$ and $\mathcal{R}_y$, 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 $\text{KILL}(i,q)$ be the set of elements written (killed) by processor $q$ in node $i$, and let $\text{Modified}(i,q)$ 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,

$$\text{KILL}(i,q) = \{ \vec{d} | \vec{d} \in \text{Own}(X,q) \land \exists \vec{i}, \mathcal{R}_x \text{ st. } \vec{d} = \mathcal{R}_x(\vec{i}) \text{ and } i_l \leq i \leq i_u \},$$

$$\text{Modified}(i,q) = \bigcup_{j \in \text{pred}(i)} \text{Modified}(j,q) \cup \text{KILL}(i,q)$$

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

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

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, 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

Fig. 6. An example program fragment. In this fragment, there are two intervals corresponding to the $i$ and $j$ loops, respectively.
similar manner). The analysis starts with the first node of the innermost interval (the \( j \) loop), and proceeds as follows:

\[
\begin{align*}
\text{KILL}(4, q) &= \{ \vec{d}|\vec{d} \in \text{Own}(X, q) \text{ and } \exists t, j \text{ st. } \vec{d} = X(t, j) \text{ and } i_l \leq t \leq i_u \text{ and } j_l \leq j \leq j_u \}.
\end{align*}
\]

\[
\begin{align*}
\text{Modified}(4, q) &= \text{KILL}(4, q) \\
\text{KILL}(5, q) &= \emptyset \\
\text{Modified}(5, q) &= \text{Modified}(4, q) \cup \text{KILL}(5, q) \\
\text{KILL}(6, q) &= \{ \vec{d}|\vec{d} \in \text{Own}(X, q) \text{ and } \exists t, j \text{ st. } \vec{d} = X(t - 1, j + 2) \text{ and } i_l \leq t \leq i_u \text{ and } j_l \leq j \leq j_u \}.
\end{align*}
\]

\[
\begin{align*}
\text{Modified}(6, q) &= \text{Modified}(5, q) \cup \text{KILL}(6, q) \\
\text{KILL}(7, q) &= \emptyset \\
\text{Modified}(7, q) &= \text{Modified}(6, q) \cup \text{KILL}(7, q) \\
\text{KILL}(9, q) &= \{ \vec{d}|\vec{d} \in \text{Own}(X, q) \text{ and } \exists t, j \text{ st. } \vec{d} = X(t + 1, j - 3) \text{ and } i_l \leq t \leq i_u \text{ and } j_l \leq j \leq j_u \}.
\end{align*}
\]

\[
\begin{align*}
\text{Modified}(9, q) &= \text{Modified}(5, q) \cup \text{KILL}(9, q) \\
\text{KILL}(10, q) &= \emptyset \\
\text{Modified}(10, q) &= [\text{Modified}(7, q) \cup \text{Modified}(9, q)] \cup \text{KILL}(10, q) \\
\text{KILL}(11, q) &= \emptyset \\
\text{Modified}(11, q) &= \text{Modified}(10, q) \cup \text{KILL}(11, q) \\
\text{KILL}(12, q) &= \emptyset \\
\text{Modified}(12, q) &= [\text{Modified}(7, q) \cup \text{Modified}(9, q)] \cup \text{KILL}(4, q) \cup \text{KILL}(6, q) \cup \text{KILL}(9, q) \\
&= \{ \vec{d}|\vec{d} \in \text{Own}(X, q) \text{ and } (\exists t, j \text{ st. } \vec{d} = X(t, j) \text{ or } \vec{d} = X(t - 1, j + 2) \text{ or } \vec{d} = X(t + 1, j - 3)) \text{ and } i_l \leq t \leq i_u \text{ and } j_l \leq j \leq j_u \}.
\end{align*}
\]

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

\[
\text{KILL}(3, q) = \text{Modified}(11, q)
\]

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

\[
\begin{align*}
\text{KILL}(2, q) &= \{ \vec{d}|\vec{d} \in \text{Own}(X, q) \text{ and } \exists t \text{ st. } \vec{d} = X(t - 2, t) \text{ and } i_l \leq t \leq i_u \}.
\end{align*}
\]

\[
\begin{align*}
\text{Modified}(2, q) &= \text{KILL}(2, q) \\
\text{Modified}(3, q) &= \text{Modified}(2, q) \cup \text{KILL}(3, q)
\end{align*}
\]
\[ d \in \text{own}(X, q) \]

and \((\exists \iota, j \text{ s.t. } d = \chi(\iota, j) \text{ or } d = \chi(\iota - 1, j + 2)\)

or \(d = \chi(\iota + 1, j - 3) \text{ or } d = \chi(\iota - 2, \iota)\)

and \(i_l \leq \iota \leq i_u \text{ and } j_i \leq j \leq j_u\).

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

\[ \text{KILL}(1, q) = \text{Modified}(3, q). \]

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 \text{KILL} set for a program that refers to arrays \(N_i\) is as follows:

\[ \text{KILL}(i, q) = \{(N_1, \text{KILL}_{N_1}(i, q)), (N_2, \text{KILL}_{N_2}(i, q)), \cdots \} \]

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

\(\text{GEN}(i, p, q)\) 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 \text{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 \(\tilde{i} = (\iota_1, \ldots, \iota_n)\) and \(\tilde{i'} = (\iota'_1, \ldots, \iota'_n)\), let \(\tilde{i'} < \tilde{i}\) mean that \(\tilde{i'}\) is lexicographically less than or equal to \(\tilde{i}\); and let \(\tilde{i'} <_k \tilde{i}\) mean that \(\iota'_j = \iota_j\) for all \(j < k\), and \((\iota'_k, \ldots, \iota'_n) < (\iota_k, \ldots, \iota_n)\). Since a node can refer to multiple RHS references, we first define \(\text{gen}(i, R_{\tilde{R}}, p, q)\) as the set of elements to be sent by processor \(q\) to processor \(p\) at node \(i\) due to reference \(R_{\tilde{R}}\). In that case we can compute

\[ \text{GEN}(i, p, q) = \bigcup_{R_{\tilde{R}}} \text{gen}(i, R_{\tilde{R}}, p, q). \]
For the sake of explanation, we assume one RHS reference per node, and we use only \( \text{GEN}(i, p, q) \) in the following. The extension to the multiple RHS reference per node is straightforward. Let \( \text{Comm}(i, p, q) \) 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

\[
\text{GEN}(i, p, q) = \{ d | \exists \ i, Y \text{ st. } \bar{i}_l \leq \bar{i} \leq \bar{i}_u \text{ and } d \in \text{OWN}(X, q) \}
\]

and \( d = \mathcal{R}_R(\bar{i}) \) and

\[
\mathcal{R}_X(\bar{i}) \in \text{OWN}(Y, p) \text{ and not } (\exists \ j, \mathcal{R}'_X \text{ st. } \bar{i}_l \leq \bar{j} \leq \bar{i}_u)
\]

and \( d = \mathcal{R}'_X(\bar{j}) \) and \( \bar{j} < \text{level}(\bar{i}) \}, \)

\[
\text{Comm}(i, p, q) = \bigcap_{s \in \text{succ}(i)} \text{Comm}(s, p, q) \cup \text{GEN}(i, p, q).
\]

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

\[
\text{GEN}(i, p, q) = \text{Comm}(\text{First}(i), p, q)
\]

In the definition of \( \text{GEN} \), \( \mathcal{R}_R \) denotes the RHS reference, and \( \mathcal{R}_X \) denotes the LHS reference of the same statement. \( \mathcal{R}'_X \), on the other hand, refers to any LHS reference within the same interval. Notice that while \( \mathcal{R}'_X \) is a reference to the same array as \( \mathcal{R}_X \), \( \mathcal{R}'_X \) can be a reference to any array (e.g., array \( Y \) in the formulation above). \( \text{level}(i) \) 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 \( \text{GEN} \) set. In that case the formulation of the \( \text{GEN} \) 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 \( \text{GEN} \) set for it is recorded, and an operator \( \mathcal{F} \) is applied to the last part of this \( \text{GEN} \) set to propagate it to the outer interval:

\[
\mathcal{F}(\bar{j} < \bar{k}_l) = \bar{j} < \bar{(k-1)}_l
\]

For example, consider Figure 6 once more, this time concentrating on the computation of \( \text{GEN} \) 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):

ACM Transactions on Programming Languages and Systems, Vol. 21, No. 6, November 1999.
\[\mathrm{GEN}(11, p, q) = \{ d | \exists i, j \text{ st. } i_l \leq i \leq i_u \text{ and } j_l \leq j \leq j_u \text{ and } d \in \text{OWN}(Y, q) \]

and \( d = Y(u-4, j) \) and \( Z(u, j) \in \text{OWN}(Z, p) \) and not \( \exists \nu', j', \text{ st.} \)

\( d = Y(u', j') \) and \( i_l \leq \nu' \leq i_u \text{ and } j_l \leq j' \leq j_u \text{ and } (\nu' = \nu \text{ and } j' < j) \} \)

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

\[\text{Comm}(11, p, q) = \text{GEN}(11, p, q)\]

\[\text{GEN}(10, p, q) = \emptyset\]

\[\text{Comm}(10, p, q) = \text{Comm}(11, p, q) \cup \text{GEN}(10, p, q) = \text{Comm}(11, p, q)\]

\[\text{Comm}(9, p, q) = \text{Comm}(10, p, q) \cup \text{GEN}(9, p, q)\]

\[\text{GEN}(7, p, q) = \emptyset\]

\[\text{Comm}(7, p, q) = \text{Comm}(10, p, q) \cup \text{GEN}(7, p, q)\]

\[\text{Comm}(7, p, q) = \text{Comm}(10, p, q)\]

\[\text{Comm}(6, p, q) = \text{Comm}(7, p, q) \cup \text{GEN}(6, p, q)\]

\[\text{GEN}(5, p, q) = \emptyset\]

\[\text{Comm}(5, p, q) = \text{Comm}(6, p, q) \cap \text{Comm}(9, p, q)\]

\[\text{Comm}(4, p, q) = \text{Comm}(5, p, q) \cup \text{GEN}(4, p, q) = [\text{Comm}(6, p, q) \cap \text{Comm}(9, p, q)] \cup \text{GEN}(4, p, q)\]

\[= \left[ \left[ \text{GEN}(11, p, q) \cup \text{GEN}(6, p, q) \right] \cap \left[ \text{GEN}(11, p, q) \cup \text{GEN}(9, p, q) \right] \right] \cup \text{GEN}(4, p, q)\]

\[\cup \left[ \text{GEN}(11, p, q) \cup \text{GEN}(6, p, q) \cup \text{GEN}(4, p, q) \right] \cap \left[ \text{GEN}(11, p, q) \cup \text{GEN}(9, p, q) \cup \text{GEN}(4, p, q) \right]\]

\[\cup \left[ \text{GEN}(11, p, q) \cup \text{GEN}(4, p, q) \right] \cup \left[ \text{GEN}(6, p, q) \right]\]

ACM Transactions on Programming Languages and Systems, Vol. 21, No. 6, November 1999.
Since \( \text{first}(3) = 4 \), the innermost interval can now be reduced as follows:

\[
\text{GEN}(3,p,q) = \text{Comm}(4,p,q)
\]

\[
= \{d \mid \exists i, j \text{ st. } i_l \leq \iota \leq i_u \text{ and } j_l \leq j \leq j_u \text{ and } d \in \text{Own}(Y,q) \}
\]

and \( \tilde{d} = Y(\iota - 4, j) \) and \( Z(\iota, j) \in \text{Own}(Z,p) \)

and not \( (\exists \iota', j', \text{ st. } \tilde{d} = Y(\iota', j') \text{ and } i_l \leq \iota' \leq i_u \)

and \( j_l \leq j' \leq j_u \text{ and } (\iota' = \iota \text{ and } j' < j) \}) \}

\[
\cup \{d \mid \exists i, j \text{ st. } i_l \leq \iota \leq i_u \text{ and } j_l \leq j \leq j_u \text{ and } d \in \text{Own}(Y,q) \}
\]

and \( \tilde{d} = Y(\iota - 2, j + 2) \) and

\[
X(\iota, j) \in \text{Own}(X,p) \text{ and not } (\exists \iota', j', \text{ st. } \tilde{d} = Y(\iota', j') \text{ and } i_l \leq \iota' \leq i_u \)
\]

and \( j_l \leq j' \leq j_u \text{ and } (\iota' = \iota \text{ and } j' < j) \}) \}

\[
\cup \{(d) \mid \exists i, j \text{ st. } i_l \leq \iota \leq i_u \text{ and } j_l \leq j \leq j_u \}
\]

and \( \tilde{d} \in \text{Own}(Y,q) \) and \( \tilde{d} = Y(\iota - 2, j - 2) \) and

\[
X(\iota - 1, j + 2) \in \text{Own}(X,p)
\]

and not \( (\exists \iota', j', \text{ st. } \tilde{d} = Y(\iota', j') \text{ and } i_l \leq \iota' \leq i_u \)

and \( j_l \leq j' \leq j_u \text{ and } (\iota' = \iota \text{ and } j' < j) \}) \}

\[
\cap \{d \mid \exists i, j \text{ st. } i_l \leq \iota \leq i_u \text{ and } j_l \leq j \leq j_u \}
\]

and \( \tilde{d} \in \text{Own}(Y,q) \) and \( \tilde{d} = Y(\iota + 3, j - 3) \) and

\[
X(\iota + 1, j - 3) \in \text{Own}(X,p) \text{ and not }
\]

\[
(\exists \iota', j', \text{ st. } \tilde{d} = Y(\iota', j') \text{ and } i_l \leq \iota' \leq i_u \)
\]

and \( j_l \leq j' \leq j_u \text{ and } (\iota' = \iota \text{ and } j' < j) \}) \}

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

\[
(\iota' = \iota \text{ and } j' < j) \rightarrow (\iota' \leq \iota \text{ or } (\iota' = \iota \text{ and } j' < j)).
\]
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):

\[
\text{Comm}(3, p, q) = \text{GEN}(3, p, q)
\]
\[
\text{Comm}(2, p, q) = \text{Comm}(3, p, q) \cup \text{GEN}(2, p, q)
\]

Since \text{first}(1) = 2, the outer interval can now be reduced:

\[
\text{GEN}(1, p, q) = \text{Comm}(2, p, q)
\]

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 \text{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 [Hiranandani et al. 1992]. Finally, \text{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 $X$ we have

\[
\text{POST\_GEN}(i, p, q) = \{ \bar{d} | \exists i, y \text{ st. } \bar{i}_l \leq i \leq \bar{i}_u \text{ and } \bar{d} \in \text{Own}(X, q) \}
\]

and

\[
\bar{d} = \mathcal{R}_{\mathcal{B}}(\bar{i}) \text{ and } \mathcal{R}_{\mathcal{Y}}(\bar{i}) \subseteq \text{Own}(Y, p) \text{ and not } (\exists \bar{j}, \mathcal{R}_{\mathcal{X}} \text{ st. } \bar{i}_l \leq j \leq \bar{i}_u \text{ and } \bar{d} = \mathcal{R}_{\mathcal{X}}(\bar{j}) \text{ and } \bar{i} < \text{level}(\bar{j}))
\]

The computation of \text{POST\_GEN} ($i, p, q$) proceeds in the forward direction. Its computation is similar to those of the \text{KILL} ($i, q$) and \text{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 \text{RECV}. The actual send and recv sets used by the code generator are produced in a later pass of the compiler from the \text{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 nonlocal 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
Backward Analysis:

\[
\text{SAFE\_OUT}(i, p, q) = \bigcap_{s \in \text{succ}(i)} \text{SAFE\_IN}(s, p, q) 
\]
\[
\text{SAFE\_IN}(i, p, q) = (\text{SAFE\_OUT}(i, p, q) - \_d \text{KILL}(i, q)) + \_d \text{GEN}(i, p, q) \tag{2}
\]

Forward Analysis:

\[
\text{RECV\_IN}(i, p, q) = \bigcap_{j \in \text{pred}(i)} \text{RECV\_OUT}(j, p, q) \tag{3}
\]
\[
\text{RECV}(i, p, q) = \begin{cases} 
\text{GEN}(i, p, q) - \_d \text{RECV\_IN}(i, p, q) 
& \text{if } \exists k \in \text{succ}(i) \text{ and } k \notin \text{dom}(i) \\
\text{SAFE\_IN}(i, p, q) - \_d \text{RECV\_IN}(i, p, q) 
& \text{otherwise}
\end{cases} \tag{4}
\]
\[
\text{RECV\_OUT}(i, p, q) = \begin{cases} 
\text{RECV\_IN}(i, p, q) - \_d \text{KILL}(i, q) 
& \text{if } \exists k \in \text{succ}(i) \text{ and } k \notin \text{dom}(i) \\
(\text{RECV}(i, p, q) + \_d \text{RECV\_IN}(i, p, q)) 
& (\text{RECV\_OUT}(i, p, q) + \_d \text{POST\_GEN}(i, p, q)) 
& \text{otherwise}
\end{cases} \tag{5}
\]

Fig. 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.

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 Eqs. (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, as well as some of the previous approaches as in Kennedy and Sethi [1995], 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 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 Eqs. (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, that is, the elements communicated at the beginning of i provided that the condition in Eq. (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

\[
\text{RECV}_{\text{IN}}(i,p,q)^{k\text{th pass}} = \text{RECV}(i,p,q)^{k-1\text{th pass}}.
\]

This assignment then triggers the next pass in the expansion phase. Before the expansion phase starts RECV_{IN}(i,p,q)^{1\text{st pass}} is set to 0. Figure 8 shows the overall algorithm COMM-OPT followed by compiler to generate the send and recv sets. Notice, that, due to Eqs. (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 are used, and the correctness is ensured.

3.5 Example

We use the synthetic benchmark program in Figure 9(a) to illustrate our framework. We concentrate on the communication placement at the higher-
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.

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

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

ACM Transactions on Programming Languages and Systems, Vol. 21, No. 6, November 1999.
Fig. 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 itemwise messages are combined and are lifted out of the enclosing loops. The globally optimized version is generated using the approach discussed in this article.
Without loss of generality, assume after local analysis that the **GEN** and **KILL** sets are obtained as shown in the second and third column of Table I respectively. The corresponding line numbers are shown in the first column. Notice for this example that \( \text{POST}_\text{GEN}(i,p,q) \) is \( \emptyset \) for every \( i \) (column 4). The fifth column in Table I 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 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 the 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 communication in lines 7, 8, 29, and 48 are entirely eliminated.

—Overall, for a single processor, 16 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 [Kelly et al. 1995]. Notice that 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 Interprocedural Analysis

It is relatively straightforward to extend our analysis to work interprocedurally. In a simple interprocedural setting, our approach can be used as follows. We first build a call graph [Aho et al. 1986] where each node corresponds to a procedure and where there is a directed edge between two nodes \( P_1 \) and \( P_2 \) if and only if \( P_1 \) calls \( P_2 \). 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 \( P_k \) 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:

Table I. Data-Flow Sets for the Example Shown in Figure 9. The \texttt{GEN}, \texttt{KILL}, and \texttt{POST\_GEN} sets are obtained after local analysis, and the \texttt{SAFE\_IN} and \texttt{RECV} sets are obtained after global analysis.

<table>
<thead>
<tr>
<th>Line</th>
<th>GEN</th>
<th>KILL</th>
<th>POST_GEN</th>
<th>SAFE_IN</th>
<th>RECV</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$S_4$</td>
<td>0</td>
<td>0</td>
<td>$((S_{51} + c S_{45}) - c S_{38} + c S_{33} + c S_{11}) + c S_4$</td>
<td>$S_4 + c S_{11} + c S_{33}$</td>
</tr>
<tr>
<td>9</td>
<td>$S_{11}$</td>
<td>0</td>
<td>0</td>
<td>$((S_{51} + c S_{45}) - c S_{38} + c S_{23} + c S_{11})$</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>$S_{25}$</td>
<td>0</td>
<td>0</td>
<td>$((S_{51} + c S_{45}) - c S_{38} + c S_{33} + c S_{25})$</td>
<td>$(S_{25} + c S_{33}) - c (S_4 + c S_{11} + c S_{33})$</td>
</tr>
<tr>
<td>30</td>
<td>$S_{33}$</td>
<td>0</td>
<td>0</td>
<td>$((S_{51} + c S_{45}) - c S_{38} + c S_{33})$</td>
<td>0</td>
</tr>
<tr>
<td>36</td>
<td>0</td>
<td>$S_{38}$</td>
<td>0</td>
<td>$(S_{51} + c S_{45}) - c S_{38}$</td>
<td>0</td>
</tr>
<tr>
<td>43</td>
<td>$S_{45}$</td>
<td>0</td>
<td>0</td>
<td>$S_{51} + c S_{45}$</td>
<td>$S_{51} + c S_{45}$</td>
</tr>
<tr>
<td>49</td>
<td>$S_{51}$</td>
<td>0</td>
<td>0</td>
<td>$S_{51}$</td>
<td>0</td>
</tr>
</tbody>
</table>
GEN, KILL, and POST_GEN. Notice that these three sets completely define the communication behavior of $P_k$. Subsequently, $P_k$ is transformed to a new single node and is 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 $\text{RECV}$ sets for each node of it.

It should be noted that there are several interprocedural communication optimization algorithms (e.g., Creusillet and Irigoin [1995], and Hall et al. [1992; 1995]) with different degrees of sophistication, and the detailed analysis of communication optimization across procedure boundaries is beyond the scope of this article. However, we believe that for most of the algorithms found in the literature, the summarized communication information represented by $\text{GEN}$, $\text{KILL}$, and $\text{POST\_GEN}$ would be sufficient to optimize communication interprocedurally.

4. HOISTING COMMUNICATION VERSUS MINIMIZING THE NUMBER OF MESSAGES

The approach explained so far is focused on hoisting communication as far as possible, and in general, it results in reduction in communication volume as well as in the number of messages. However, as also pointed out by others, hoisting the communication too eagerly can, under some circumstances, lead to an excessive buffer requirement [Kennedy and Sethi 1995] and an increase in the number of communication calls inserted [Chakrabarti et al. 1996]. 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 [Kennedy and Sethi 1997]. 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 [Garey and Johnson 1979], we present a simple heuristic that 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 II. For example, Kennedy and Sethi [1995] use the third
alternative. An implementation can also employ a combination of these alternatives. For example, consider the predicate obtained by the conjunction of the first and second alternatives, i.e., \( \mathcal{P}(i) = \{ \text{KILL}(i, q) \neq \emptyset \text{ and } \text{GEN}(i, p, q) = \emptyset \} \).

The data-flow equations given in Figure 11 are very similar to those shown in Figure 7. 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 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 article 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, a modified version of the last part of the program shown in Figure 9. Columns two, three, and four of the top part of Table III 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 the table 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 III, 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 a 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, the two approaches discussed behave similarly in practice.
5. COMMUNICATION GENERATION

Our communication code generator uses the Omega library from the University of Maryland [Kelly et al. 1995; Pugh 1992]. After the \( \text{RECV}(i,p,q) \) 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 \( \text{send} \) and \( \text{recv} \) loops.

Let us now consider the example given in Figure 9 (and Table I) 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

\begin{align*}
\text{SAFE\_OUT}(i,p,q) &= \bigcap_{s \in \text{succ}(i)} \text{SAFE\_IN}(s,p,q) \quad (7) \\
\text{SAFE\_IN}(i,p,q) &= \begin{cases} 
\text{GEN}(i,p,q) \\
(\text{SAFE\_OUT}(i,p,q) \land \neg \text{KILL}(i,q)) + \text{GEN}(i,p,q) 
\end{cases} 
\text{if } P(i) \\
\text{otherwise} \quad (8)
\end{align*}

\begin{align*}
\text{RECV\_IN}(i,p,q) &= \bigcap_{j \in \text{pred}(i)} \text{RECV\_OUT}(j,p,q) \quad (9) \\
\text{RECV}(i,p,q) &= \begin{cases} 
\text{GEN}(i,p,q) \land \neg \text{RECV\_IN}(i,p,q) \\
\text{SAFE\_IN}(i,p,q) \land \neg \text{RECV\_IN}(i,p,q) 
\end{cases} 
\text{if } \exists k \in \text{succ}(i) \text{ and } k \notin \text{dom}(i) \\
\text{otherwise} \quad (10)
\end{align*}

\begin{align*}
\text{RECV\_OUT}(i,p,q) &= \begin{cases} 
\text{RECV\_IN}(i,p,q) \land \neg \text{KILL}(i,q) \\
(\text{RECV}(i,p,q) \land \text{RECV\_IN}(i,p,q)) \land \neg \text{KILL}(i,q) \land \text{POST\_GEN}(i,p,q) 
\end{cases} 
\text{for } \exists k \in \text{succ}(i) \text{ and } k \notin \text{dom}(i) \\
\text{otherwise} \quad (11)
\end{align*}

Fig. 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 \( P(i) \) predicate to control communication hoisting.

\begin{verbatim}
!HPFS$ processors PROC(0:3)
!HPFS$ distribute (cyclic(4),*) onto PROC :: A, B, C, D
real A(128,128), B(128,128), C(128,128), D(128,128)
1 DO i = 32, 63
2 DO j = 1, 63
3 B(i,j)=B(i,j)+2
4 END DO
5 END DO
6 DO i = 1, 31
7 DO j = 1, 63
8 B(i,j)=B(i,j)-C(i,j)+A(i,j)
9 END DO
10 END DO
11 DO i = 2, 127
12 DO j = 2, 127
13 A(i,j)=(B(i-1,j)+B(i-1,j+1))/2.0
14 END DO
15 END DO
16 DO i = 2, 127
17 DO j = 2, 127
18 C(i,j)=(B(i,j)+B(i-1,j-1))/2.0
19 END DO
20 END DO
\end{verbatim}

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

5. COMMUNICATION GENERATION

Our communication code generator uses the Omega library from the University of Maryland [Kelly et al. 1995; Pugh 1992]. After the \( \text{RECV}(i,p,q) \) 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 \( \text{send} \) and \( \text{recv} \) loops.

Let us now consider the example given in Figure 9 (and Table I) 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
Table III. Data-Flow Sets for the Example Shown in Figure 12. Top: Aggressive communication hoisting results in an excessive number of communication messages. Bottom: Communication hoisting is controlled to minimize the number of messages.

<table>
<thead>
<tr>
<th>Line</th>
<th>GEN</th>
<th>KILL</th>
<th>POST_GEN</th>
<th>SAFE_IN</th>
<th>RECV</th>
<th>Loop Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>$S_3$</td>
<td>0</td>
<td>$(S_{18} + c S_{13}) - c S_8) - c S_3$</td>
<td>$(S_{18} + c S_{13}) - c S_8) - c S_3$</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>$S_8$</td>
<td>0</td>
<td>$(S_{18} + c S_{13}) - c S_8$</td>
<td>$(S_{18} + c S_{13}) - c ((S_{18} + c S_{13}) - c S_8) - c S_3$</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>$S_{13}$</td>
<td>0</td>
<td>0</td>
<td>$S_{18} + c S_{13}$</td>
<td>$(S_{18} + c S_{13}) - c ((S_{18} + c S_{13}) - c S_8) - c S_3$</td>
<td>$S_{13}$</td>
</tr>
<tr>
<td>16</td>
<td>$S_{18}$</td>
<td>0</td>
<td>0</td>
<td>$S_{18}$</td>
<td>0</td>
<td>$S_{18}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>SAFE_IN</th>
<th>RECV</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>$S_{18} + c S_{13}$</td>
<td>$S_{18} + c S_{13}$</td>
</tr>
<tr>
<td>16</td>
<td>$S_{18}$</td>
<td>0</td>
</tr>
</tbody>
</table>
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 $[q, p, d_1, d_2]$ meaning that the array element indexed by $[d_1, d_2]$ should be transferred from $q$ to $p$. Later in the code generation, the projection function $\text{proj}_R := \{(q, p, d_1, d_2) \rightarrow [q, d_1, d_2]\}$ is applied to this set to generate the recv set, and similarly the projection function $\text{proj}_S := \{(p, p, d_1, d_2) \rightarrow [p, d_1, d_2]\}$ 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 (nonlocal) elements; and the entire procedure works on the local address space similar to the one shown in Figure 5(c).

After the projection functions are applied, the code generator part of the Omega library is called to generate the loops to enumerate $[q, d_1, d_2]$ and $[p, d_1, d_2]$ 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) as C code for the send set and in Figure 14(b) for the recv set. In these codes, $\text{process}()$ 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 IV. addx and eflux are two 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 1/4th of the original values. The REFS column shows the number of references in the program in question whereas the CREFS 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 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 an asterisk denotes a nondistributed dimension as in HPF [Koelbet et al. 1994].

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 distribu-

ACM Transactions on Programming Languages and Systems, Vol. 21, No. 6, November 1999.
Table IV. 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 a program are distributed. A “D” in a dimension means that the dimension is distributed across processors, while an asterisk denotes a nondistributed dimension.

<table>
<thead>
<tr>
<th>Program</th>
<th>Source</th>
<th>Arrays</th>
<th>REFS</th>
<th>C REFS</th>
<th>DISTR</th>
<th>SIZE</th>
<th>ITER</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hydro</td>
<td>Livermore</td>
<td>nine 2D</td>
<td>52</td>
<td>10</td>
<td>(*,D)</td>
<td>400</td>
<td>20</td>
<td>2D hydrodynamics</td>
</tr>
<tr>
<td>hydro_m</td>
<td>Livermore</td>
<td>nine 2D</td>
<td>52</td>
<td>10</td>
<td>(*,D)</td>
<td>400</td>
<td>20</td>
<td>modified hydro</td>
</tr>
<tr>
<td>adi</td>
<td>Livermore</td>
<td>three 3D, three 1D</td>
<td>33</td>
<td>6</td>
<td>(<em>,D,</em>)</td>
<td>400</td>
<td>10</td>
<td>iterative method</td>
</tr>
<tr>
<td>tomcatv</td>
<td>Spec92</td>
<td>seven 2D, two 1D</td>
<td>75</td>
<td>20</td>
<td>(D,*)</td>
<td>400</td>
<td>10</td>
<td>2D mesh generation</td>
</tr>
<tr>
<td>swim</td>
<td>Spec92</td>
<td>fourteen 2D</td>
<td>196</td>
<td>43</td>
<td>(D,*)</td>
<td>513</td>
<td>20</td>
<td>water equation solver</td>
</tr>
<tr>
<td>addx</td>
<td>Perfect Club</td>
<td>five 3D, one 2D</td>
<td>72</td>
<td>32</td>
<td>(D,<em>,</em>)</td>
<td>194</td>
<td>1</td>
<td>mesh related comp.</td>
</tr>
<tr>
<td>eflux</td>
<td>Perfect Club</td>
<td>four 3D, one 2D</td>
<td>76</td>
<td>13</td>
<td>(D,<em>,</em>)</td>
<td>5000</td>
<td>10</td>
<td>mesh related comp.</td>
</tr>
<tr>
<td>tred2</td>
<td>Eispack</td>
<td>two 2D, two 1D</td>
<td>42</td>
<td>22</td>
<td>(D,*)</td>
<td>60/120</td>
<td>1</td>
<td>matrix reduction</td>
</tr>
</tbody>
</table>
tions where most of the previous techniques fail. Two cyclic factors, namely 4 and 7, are selected arbitrarily, one being a power of two whereas the other one is prime. Gupta and Banerjee [1992] note for tred2 that the block-cyclic distribution is the best choice. We also found in addx that 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 [Palermo et al. 1994]. 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^p 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 \( X(i, j) = Y(i, j - 1) + Z(i, j - 1) \), arrays \( Y \) and \( Z \) have the same communication structure; therefore, we can generate communication loops only once and reuse the communication patterns 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 \( \mathcal{P}(i) \) predicates) and extension to an interprocedural 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 128MB memory, 2GB disk, and an IBM Power2 processor.

Tables V through XII give the number of communications, the communication volume, and the execution times (in seconds) for our programs for the base and opt versions. Table XIII 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 XIV 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 XV 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

<table>
<thead>
<tr>
<th># of PROCES = 8</th>
<th># of PROCES = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>(+,BLK) (+,CYC) (+,CYC(4)) (+,CYC(7))</td>
</tr>
<tr>
<td>base</td>
<td>1,540 1,604 1,604 1,604</td>
</tr>
<tr>
<td>opt</td>
<td>1,110 1,227 1,227 1,227</td>
</tr>
<tr>
<td>opt*</td>
<td>1,110 1,180 1,180 1,180</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of PROCES = 8</th>
<th># of PROCES = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>(+,BLK) (+,CYC) (+,CYC(4)) (+,CYC(7))</td>
</tr>
<tr>
<td>base</td>
<td>2.0 75.4 19.1 11.0</td>
</tr>
<tr>
<td>opt</td>
<td>1.8 17.0 4.2 3.1</td>
</tr>
<tr>
<td>opt*</td>
<td>1.8 5.9 3.2 2.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of PROCES = 8</th>
<th># of PROCES = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>(+,BLK) (+,CYC) (+,CYC(4)) (+,CYC(7))</td>
</tr>
<tr>
<td>base</td>
<td>3.65 4.96 3.97 3.99</td>
</tr>
<tr>
<td>opt</td>
<td>2.67 3.05 3.14 2.98</td>
</tr>
<tr>
<td>opt*</td>
<td>2.30 2.82 2.95 2.81</td>
</tr>
</tbody>
</table>

Table VI. Results for hydro_m on an IBM SP-2. Top: Number of communications for different versions. Middle: Communication volume in megabytes for different versions. Bottom: Execution times in seconds for the different versions.

Table VII. Results for adi on an IBM SP-2. Top: Number of communications for different versions. Middle: Communication volume in kilobytes for different versions. Bottom: Execution times in seconds for the different versions.
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 through a global analysis that 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 VIII and IX:

**Table VIII.** Results for **tomcatv** on an IBM SP-2. Top: Number of communications for different versions. Middle: Communication volume in megabytes for different versions. Bottom: Execution times in seconds for the different versions.

<table>
<thead>
<tr>
<th># of PROCES</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
</tr>
<tr>
<td>base</td>
<td>140</td>
<td>6,396</td>
</tr>
<tr>
<td>opt</td>
<td>56</td>
<td>124</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of PROCES</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
</tr>
<tr>
<td>base</td>
<td>0.35</td>
<td>19.2</td>
</tr>
<tr>
<td>opt</td>
<td>0.06</td>
<td>6.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of PROCES</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
</tr>
<tr>
<td>base</td>
<td>1.39</td>
<td>3.19</td>
</tr>
<tr>
<td>opt</td>
<td>1.06</td>
<td>1.31</td>
</tr>
</tbody>
</table>

**Table IX.** Results for **swim** on an IBM SP-2. Top: Number of communications for different versions. Middle: Communication volume in megabytes for different versions. Bottom: Execution times in seconds for the different versions.

<table>
<thead>
<tr>
<th># of PROCES</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
</tr>
<tr>
<td>base</td>
<td>3,967</td>
<td>120,142</td>
</tr>
<tr>
<td>opt</td>
<td>3,678</td>
<td>84,182</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of PROCES</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
</tr>
<tr>
<td>base</td>
<td>7.4</td>
<td>240.2</td>
</tr>
<tr>
<td>opt</td>
<td>7.1</td>
<td>163.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of PROCES</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
</tr>
</tbody>
</table>
XVI and XVII. All the compilation times shown in the rest of the article have been obtained on a Model 712/60 HP workstation with a 132MHz PA RISC processor, 64KB first-level cache, 1MB second-level cache, and a 256MB memory.

Table XVI 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

Table X. Results for addx on an IBM SP-2. Top: Number of communications for different versions. Middle: Communication volume in megabytes for different versions. Bottom: Execution times in seconds for the different versions.

<table>
<thead>
<tr>
<th># of PROCS = 8</th>
<th># of PROCS = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
</tr>
<tr>
<td>base</td>
<td>91,470</td>
</tr>
<tr>
<td>opt</td>
<td>57,266</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of PROCS = 8</th>
<th># of PROCS = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
</tr>
<tr>
<td>base</td>
<td>0.37</td>
</tr>
<tr>
<td>opt</td>
<td>0.23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of PROCS = 8</th>
<th># of PROCS = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
</tr>
<tr>
<td>base</td>
<td>5.18</td>
</tr>
<tr>
<td>opt</td>
<td>3.33</td>
</tr>
</tbody>
</table>

Table XI. Results for eflux on an IBM SP-2. Top: Number of communications for different versions. Middle: Communication volume in megabytes for different versions. Bottom: Execution times in seconds for the different versions.

<table>
<thead>
<tr>
<th># of PROCS = 8</th>
<th># of PROCS = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
</tr>
<tr>
<td>base</td>
<td>203</td>
</tr>
<tr>
<td>opt</td>
<td>84</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of PROCS = 8</th>
<th># of PROCS = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
</tr>
<tr>
<td>base</td>
<td>0.07</td>
</tr>
<tr>
<td>opt</td>
<td>0.04</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of PROCS = 8</th>
<th># of PROCS = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>version</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
</tr>
<tr>
<td>base</td>
<td>5.90</td>
</tr>
<tr>
<td>opt</td>
<td>5.76</td>
</tr>
</tbody>
</table>
generate communication 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 XVII 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 on the average that 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. For the case where 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 different versions.

Table XII. Results for tred2 on an IBM SP-2. Top: Number of communications for different versions. Middle: Communication volume in megabytes for different versions. Bottom: Execution times in seconds for the different versions.

<table>
<thead>
<tr>
<th>Version</th>
<th># of PROCS = 8</th>
<th># of PROCS = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
</tr>
<tr>
<td>Base</td>
<td>1,706</td>
<td>1,721</td>
</tr>
<tr>
<td></td>
<td>1,759</td>
<td>2,028</td>
</tr>
<tr>
<td></td>
<td>1,740</td>
<td>2,300</td>
</tr>
<tr>
<td></td>
<td>1,721</td>
<td>2,286</td>
</tr>
<tr>
<td></td>
<td>2,028</td>
<td>2,280</td>
</tr>
<tr>
<td>Opt</td>
<td>1,650</td>
<td>1,711</td>
</tr>
<tr>
<td></td>
<td>1,719</td>
<td>1,988</td>
</tr>
<tr>
<td></td>
<td>1,718</td>
<td>2,015</td>
</tr>
<tr>
<td></td>
<td>1,711</td>
<td>2,004</td>
</tr>
<tr>
<td></td>
<td>1,996</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th># of PROCS = 8</th>
<th># of PROCS = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
</tr>
<tr>
<td>Base</td>
<td>11.5</td>
<td>14.2</td>
</tr>
<tr>
<td></td>
<td>12.1</td>
<td>15.5</td>
</tr>
<tr>
<td></td>
<td>11.9</td>
<td>15.1</td>
</tr>
<tr>
<td></td>
<td>11.9</td>
<td>15.2</td>
</tr>
<tr>
<td>Opt</td>
<td>11.1</td>
<td>13.7</td>
</tr>
<tr>
<td></td>
<td>11.8</td>
<td>15.0</td>
</tr>
<tr>
<td></td>
<td>11.4</td>
<td>14.7</td>
</tr>
<tr>
<td></td>
<td>11.3</td>
<td>14.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th># of PROCS = 8</th>
<th># of PROCS = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
</tr>
<tr>
<td>Base</td>
<td>0.83</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>0.90</td>
<td>1.16</td>
</tr>
<tr>
<td></td>
<td>0.80</td>
<td>0.90</td>
</tr>
<tr>
<td></td>
<td>0.76</td>
<td>0.90</td>
</tr>
<tr>
<td></td>
<td>0.91</td>
<td>1.05</td>
</tr>
<tr>
<td></td>
<td>0.69</td>
<td>0.82</td>
</tr>
<tr>
<td></td>
<td>0.73</td>
<td>0.79</td>
</tr>
<tr>
<td>Opt</td>
<td>0.78</td>
<td>1.05</td>
</tr>
<tr>
<td></td>
<td>0.78</td>
<td>1.05</td>
</tr>
<tr>
<td></td>
<td>0.73</td>
<td>0.82</td>
</tr>
<tr>
<td></td>
<td>0.69</td>
<td>0.79</td>
</tr>
</tbody>
</table>

Table XIII. Percentage Improvements in Number of Messages

<table>
<thead>
<tr>
<th>Version</th>
<th># of PROCS = 8</th>
<th># of PROCS = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
<td>(<em>,BLK) (</em>,CYC) (<em>,CYC(4)) (</em>,CYC(7))</td>
</tr>
<tr>
<td>Hydro</td>
<td>21</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>Hydro_m</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>31</td>
</tr>
<tr>
<td>Hydro_m</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>33</td>
</tr>
<tr>
<td>Tomcatv</td>
<td>59</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Swim</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>98</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>92</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>87</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>87</td>
<td>60</td>
</tr>
<tr>
<td>Addx</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>Ef1ux</td>
<td>47</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>Tred2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>Average</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>35</td>
<td>33</td>
</tr>
</tbody>
</table>

generate communication 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.
Even if we do not use the global framework, we see that just using the Omega library takes 50% of the compilation time on the average. The DIF column shows the difference (GLP \( \text{LOP} \)) 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) [Su et al. 1994], 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 XVIII shows the overall compilation times of the Omega-based approach (OME), the 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 the use of Omega instead of an RSD-like approach increases the compilation time 7% to 27%, averaging on 19% for both block and cyclic distributions.

Table XV. 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.

<table>
<thead>
<tr>
<th>program</th>
<th>BLK</th>
<th>CYC</th>
<th>CYC (4)</th>
<th>CYC (7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hydro</td>
<td>157</td>
<td>1,883</td>
<td>1,414</td>
<td>211</td>
</tr>
<tr>
<td>hydro_m</td>
<td>166</td>
<td>1,900</td>
<td>1,290</td>
<td>188</td>
</tr>
<tr>
<td>adi</td>
<td>161</td>
<td>955</td>
<td>1,100</td>
<td>161</td>
</tr>
<tr>
<td>tomcatv</td>
<td>167</td>
<td>2,308</td>
<td>1,100</td>
<td>200</td>
</tr>
<tr>
<td>swim</td>
<td>300</td>
<td>2,967</td>
<td>1,800</td>
<td>284</td>
</tr>
<tr>
<td>addx</td>
<td>200</td>
<td>1,283</td>
<td>1,155</td>
<td>254</td>
</tr>
<tr>
<td>eflux</td>
<td>183</td>
<td>2,017</td>
<td>1,104</td>
<td>184</td>
</tr>
<tr>
<td>tred2</td>
<td>180</td>
<td>2,417</td>
<td>1,334</td>
<td>183</td>
</tr>
<tr>
<td>average</td>
<td>189</td>
<td>1,966</td>
<td>1,245</td>
<td>208</td>
</tr>
</tbody>
</table>

Table XVI. Cost Analysis of Our Approach Over All Distribution Types. On 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.

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

We compared the compilation time taken by our Omega-based global approach with that of an approach based on processor-tagged descriptors (PTDs) [Su et al. 1994], 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 XVIII shows the overall compilation times of the Omega-based approach (OME), the 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 the use of 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 addressed the problem of generating local address and communication sets for HPF programs where arrays are distributed using the general block-cyclic distributions [Ancourt et al. 1997; Chatterjee et al. 1995; Gupta et al. 1996; Kennedy et al. 1995; 1996; Thirumalai and Ramanujam 1996; Venkatachar et al. 1997]. Of these, Ancourt et al. 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 article appears to be limited.

Most of the previous efforts considered communication optimization at the 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 [Balasundaram et al. 1990; Banerjee et al. 1995; Bozkus et al. 1994; Gerndt 1990; Hiranandani et al. 1992; Zima and Chapman 1991].

More recently some researchers have proposed techniques based on data-flow analysis in order to optimize communication across multiple loop nests. Agrawal and Saltz [1997] present a framework for partial redundancy elimination for communication optimization in data-parallel programs with irregular data access patterns. Amarasinghe and Lam [1993] 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 [1991] 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. [1993] 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. [1995b] 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 [1993; 1994] 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 [1995] 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 article. They do not give any clue how their method can be extended to handle general type block-cyclic distributions.

Kennedy and Sethi [1995; 1996; 1997] 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 quantitative comparison of this work with ours is not possible. Their work defines a data-flow variable called $\text{SAFE}$ which can be used in a similar manner as our predicate $\mathcal{P}(i)$. Kennedy and Sethi [1995; 1996; 1997] do not use a linear algebra framework; later work from the dHPF project at Rice [Adve and Mellor-Crummey 1998; Adve et al. 1997] includes the use of the Omega library for message optimizations.

The IBM pHPF compiler [Chakrabarti et al. 1996; Gupta et al. 1995a] achieves both redundancy elimination and message combining globally. But message combining is feasible only if the messages have identical patterns, or if 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. [1997a; 1997b] 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
then 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. [Adve and Mellor-Crummey 1998; Adve et al. 1997] describe an integer-set-based approach for analysis and code generation for data-parallel programs that uses the Omega library [Kelly et al. 1995]. 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 article was first introduced by Allen and Cocke [1976]. They used it to solve several data-flow problems; the analysis was then extended by Gross and Steenkiste [1990] to array sections. The approach proposed by Gupta et al. [1995b] mentioned above refines the technique by Gross and Steenkiste using loop-carried dependences.

In this article we used ideas from the linear algebra framework [Ancourt et al. 1997] and data-flow analysis [Aho et al. 1986; Allen and Cocke 1976] 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 redefining 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 nonlocal data to minimize communication costs is critical for scaling performance on distributed-memory message-passing machines. We presented here a global communication optimization scheme based on two complementary techniques: data-flow analysis and a 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

ACM Transactions on Programming Languages and Systems, Vol. 21, No. 6, November 1999.
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.

REFERENCES


ACM Transactions on Programming Languages and Systems, Vol. 21, No. 6, November 1999.


Received March 1998; revised July 1998; accepted November 1998