Appears in IEEE Trans. on Parallel and Distributed Systems, December 2000

Minimizing data and synchronization costs in one-way communication*

M. Kandemir[†] A. Choudhary[‡] P. Banerjee[§] J. Ramanujam[¶] N. Shenoy[||]

Abstract

Minimizing communication and synchronization costs is crucial to the realization of the performance potential of parallel programs. This paper presents a general technique which uses a global data-flow framework to optimize communication and synchronization in the context of the one-way communication model. In contrast to the conventional send/receive message-passing communication model, one-way communication is a new paradigm which decouples message transmission and synchronization. In parallel machines with appropriate low-level support, this may open up new opportunities not only to further optimize communication but also to reduce the synchronization overhead. We present optimization techniques using our framework for eliminating redundant data communication and synchronization operations. Our approach works with the most general data alignments and distributions in languages like High Performance Fortran (HPF), and uses a combination of the traditional data-flow analysis and polyhedral algebra. Empirical results for several scientific benchmarks on Cray T3E multiprocessor machine demonstrate that our approach is successful in reducing the number of data (communication) and synchronization messages, thereby reducing the overall execution times.

Keywords: one-way communication, message-passing, redundant synchronization, compiler optimizations, data-flow analysis, linear algebra techniques, data-parallel languages.

Corresponding Author:

Prof. Prith Banerjee
Department of Electrical and Computer Engineering
Northwestern University
L 497, Technological Institute
2145 Sheridan Road
Evanston, IL 60208-3118
Phone: (847) 491-4118, Fax: (847) 467-4144
Email: banerjee@ece.nwu.edu

* A preliminary version of this paper was presented at the 1998 International Conference on Parallel Processing.

[†] Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802. E-mail: kandemir@cse.psu.edu

[‡] Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208. E-mail: choudhar@ece.nwu.edu.

[§] Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208. E-mail: banerjee@ece.nwu.edu.

[¶] Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. E-mail: jxr@ee.lsu.edu.

[||] Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208. E-mail: nagaraj@ece.nwu.edu.

List of Figures

List of Tables

1 Introduction

Most of the current languages for distributed-memory architectures such as High Performance Fortran (HPF) [38], Fortran-D [31], and Vienna Fortran [11] provide data alignment and distribution directives to the users. Using these directives, the users can specify the data mappings and a compiler can derive the computation partitions automatically. The compilers for these languages have traditionally relied on send (`Send`) and receive (`Recv`) primitives to implement message-passing communication. The impact of this approach is twofold. First, such an approach combines synchronization with communication in the sense that data messages also carry *implicit* synchronization information. While this relieves the compiler of the job of inserting explicit synchronization messages to maintain data integrity and correct execution, separating synchronization messages from data messages may actually improve the performance of programs by giving compiler the option of optimizing the data and synchronization messages separately. In fact, recently O'Boyle and Bodin [42] and Tseng [52] have presented techniques to optimize synchronization messages on shared memory and distributed shared memory parallel architectures. These techniques can be applied to the synchronization messages of the programs compiled using separate data and synchronization messages. Second, the compiler has the task of matching send and receive operations in order to guarantee correct execution. This is a difficult job and limits the number of programs that can be compiled effectively for message-passing architectures.

Recently alternative communication and synchronization mechanisms called *one-way communication* have been offered. Stricker et al. [48] and Hayashi et al. [26] suggest that the separation of synchronization from data transfer is extremely useful for realizing good performance. In the context of distributed operating systems, a similar separation of data and control transfer has been suggested by Thekkath et al. [51]. Split C [13] offers one-way memory operations and active messages [53] provides a software implementation of one-way communication. One-way communication is also a part of the proposed Message Passing Interface standard [41]. The main characteristic of these techniques is that they separate interprocessor data transfers from producer-consumer synchronization. A number of (physically) distributed-memory machines such as the Fujitsu AP1000+ [26], the Cray T3D [12], the Cray T3E [47], and the Meiko CS-2 [8] already offer efficient low-level remote memory access (RMA) primitives which provide a processor with the capability of accessing the memory of another processor without the direct involvement of the latter. To preserve the original semantics, however, a synchronization protocol should be observed.

In this paper, we focus on the compilation of programs augmented with HPF-style data mapping directives using one-way communication operations `Put` (remote memory write) and `Synch` (synchronization). Although we present our techniques in the framework of (physically) distributed memory machines, they are readily applicable to uniform shared memory architectures as well. Gupta and Schonberg [19] show that compilers that generate code for one-way communication can exploit shared-memory architectures with flexible cache-coherence protocols (e.g., Wisconsin Typhoon [45] and Stanford FLASH [27]). To measure the benefits obtained from one-way communication (see Section 7) we used a Cray T3E [47] which is a logically shared, physically distributed memory multiprocessor that supports the PVM [15] and MPI [41] message-passing libraries as well as a simple one-sided communication library provided by Silicon Graphics Inc.

The `Put` primitive—executed by the producer of a data—transfers the data from the producer's memory to the consumer's memory. This operation is very similar to the execution of a `Send` primitive by the producer and the execution of a matching `Recv` primitive by the consumer. There is an important difference however: the consumer processor is not involved in the transfer directly and all the communication parameters are supplied by the producer [41]. As stated above, in order to ensure correctness, synchronization operations might be necessary. A large number of synchronization operations can be used to preserve the semantics of the program. These include barriers, point-to-point (or producer-consumer) synchronizations, and locks. The synchronization primitive used in this paper, namely `Synch`—executed by the producer of a data—, is a point-to-point communication primitive; however, our approach can be modified to work with other types of synchronizations as well. Note that both Stricker et al. [48] and Hayashi et al. [26] use barriers to implement synchronization. In contrast, our effort is aimed at reducing the total amount of synchronization using data-flow analysis, and using finer-granularity point-to-point primitives where possible.

Consider Figure 1(a); here, a consumer processor sends a `Synch` message to the producer informing that the producer can *put* data in a buffer physically located in the consumer's memory. After receiving the `Synch` message, the producer deposits (puts) the data in that buffer. It should be noted that the `Synch` operation is necessary if this communication between the producer and the consumer is going to *repeat*. This is because, when the producer wants to deposit new data into the buffer, it must know that the consumer has indeed consumed the old data in the buffer. To be precise, after the data has been deposited, the consumer should send an acknowledge (`Ack`) message; in order to keep the presentation simple, we will omit the `Ack` messages in this paper.

First, we briefly discuss the `Put/Synch` one-way communication framework and the fundamental concepts used in this paper in Sections 2 and 3, respectively. Then, in Section 4, we show how the communication sets as well as the producer and the consumer sets manipulated by the one-way communication mechanism can be implemented on top of the existing send/receive type of communication framework of a distributed-memory compiler. Having determined those, the next issue is to minimize the number of `Put` communications as well as the communication volume. Section 5 presents an algorithm to achieve this goal. Our algorithm can take control flow into account (except 'goto' constructs) and can optimize programs with all types of HPF-like alignments and distributions, including block-cyclic distributions. The solution is based on a linear algebra framework first introduced by Ancourt et al. [6]; in addition, our approach is quite general in the sense that several current solutions to the problem can be derived by a suitable definition of a predicate. Section 6 discusses how data-flow analysis can be used to optimize synchronization and presents a data-flow analysis for this purpose. In Section 7, we provide results on several benchmarks to demonstrate the efficacy of our technique. We measure the effect of our approach on the number of messages, communication volume, and the overall execution times. Section 8 discusses related work and Section 9 concludes the paper with a summary and discussion.

2 One-way communication

One-way communication (or remote memory access (RMA)) is a technique where a processor is allowed direct access to the memory of another processor. To preserve the integrity of data, however, some kind of synchronization should be established. In this section, we discuss the advantages and disadvantages of one-way communication over the traditional two-way (send/receive type of) communication and explain why we prefer the `Put` primitive over `Get`, another primitive that can be used in one-way communication scheme. We also discuss the synchronization elimination problem and informally explain our approach.

The communication mechanism based on the send/receive primitives is easy to understand. A data exchange requires the *active* involvement of both the producer and the consumer. The underlying protocol, however, needs to handle a number of problematic cases. One such case occurs when the data has been sent but the process running on the consumer processor has not issued a `Recv` command yet. In that case, the consumer should buffer the data until a `Recv` is posted. This requires expensive buffer copying and increases synchronization costs [19]. Another important issue is that the compiler should guarantee the correct ordering of the messages sent from one processor to another using message tags whenever necessary. Apart from these issues, combining data and synchronization messages prevents a compiler from optimizing them separately since each data communication involves an implicit synchronization whether it is needed or not.

Clearly, in a compilation framework based on the `Put` operation, the correct ordering of memory accesses has to be imposed by the compiler using the synchronization primitives. A straightforward approach would insert a `Synch` operation just before each `Put` operation as shown in Figure 1(a). The next question to be addressed then is whether or not every `Synch` operation inserted that way is necessary. The answer is *no,* and Section 6 proposes an algorithm to eliminate redundant synchronization messages. We refer to a `Synch` operation as *redundant* if its functionality can be fulfilled by other data communications or other `Synch` operations present in the program. The basic idea is to use another message in the reverse direction between the same pair of processors in the place of the `Synch` call as shown in Figure 1(b). In such a situation, we say that the communication $\mathtt{t}_j$ *kills* the synchronization requirement of communication `i`. We show that our algorithm is fast and very effective in reducing the number of synchronizations. This is because of the following: (i) it is very accurate in eliminating redundant synchronization, since it works at the granularity of a processor-pair using the Omega library [34, 44]; (ii) it can eliminate a synchronization message by using several data messages; (iii) it handles block, cyclic, and general block-cyclic distributions in a unified manner, whereas the previous approaches either work only for virtual processor grids or use an extension of regular section descriptors (RSDs) [9] which are inherently inaccurate; and (iv) it is preceded by a global communication optimization algorithm which itself eliminates many of the messages.

To show the idea behind the algorithm, we consider Figure 2(a), where eight processors (numbered 0 thru 7) are involved in a `Put` communication that repeats itself (as in a loop); processor i deposits data in the memory of processor $i - 1$ for $1 \leq i \leq 7$; the arrows indicate the direction of communication. Figure 2(b) shows the `Synch` messages required for the repetitions of this communication. Suppose that between successive repetitions of the communication pattern in Figure 2(a), subsets of processors are involved in communication patterns using `Put` shown in Figures 2(c) and 2(d). Our synchronization elimination algorithm can detect that the communications in Figures 2(c) and 2(d), *together*, kill the synchronization requirement of the first communication, i.e., kill the `Synch` messages shown in Figure 2(b).

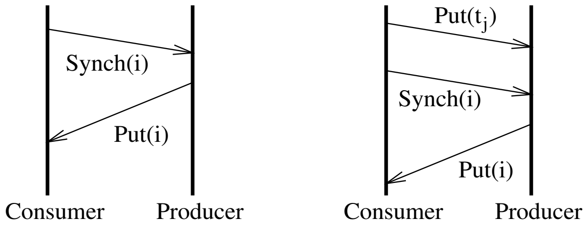
It should also be mentioned that in machine environments that support both one-way (`Put/Get`) and two-way (`Send/Recv`) communication mechanisms, it might be the case that two-way communication mechanisms are implemented using the one-way communication mechanisms. Because of this reason, one-way communication calls might have significantly lower startup latencies and higher bandwidths. Therefore, when used in a communication-intensive parts of a program, they can result in better scaling of that portion of the computation. This observation suggests that in some architectures we might be able to obtain better results using one-way communication. In fact, the MPI [41] and PVM [15] message-passing libraries on the Cray T3E are implemented using a one-sided communication library (SHMEM). In addition to that, with one-way communication messages, the remote processor is not interrupted during its computation, instead can continue to spend its precious cycles in actual computation. Of course, this flexibility comes with a certain cost: in those cases where the communication activity between processors needs to synchronized, explicit synchronization messages should be inserted in the code. This might be a problem for users. Therefore, in this paper, we propose the automatic insertion and minimization of synchronization calls. Also, we present preliminary results for several benchmark programs on the Cray T3E. Our experiments show that, by using one-way communication primitives, we are able to reduce the number of data (communication) messages and synchronization messages, and consequently obtain significant improvements in the overall execution times. We believe that these are also the first results from a comprehensive evaluation of synchronization elimination in one-way communication[1].

Finally, as discussed by Gupta and Schonberg [19], although both `Put` and `Get` can be implemented in terms of each other, we prefer to use `Put` because of the following two reasons. First, in general, the handshaking protocol for the `Get` primitive involves more messages than that of `Put` [19]. Secondly, the synchronizations originating from `Get` primitive is due to *flow-dependences* and are in general difficult to eliminate. On the other hand, the synchronization messages in case of the `Put` primitive are needed to satisfy *anti-dependences* (*pseudo-dependences*) and therefore easier to eliminate [19].

3 Preliminaries

In this section, we briefly describe the concepts used in this paper. We focus on structured programs with conditional statements and nested loops but without arbitrary goto statements. With appropriate modifications, our technique can handle arbitrary gotos as well. A *basic block* is defined as a sequence of consecutive

[1]Note, however, that in machines without low-level support for one-way communication, the performance of one-way communication its performance may be worse than that of the send/receive model.



(a) (b)

Figure 1: (a) One-way communication with `Put` operation. (b) elimination of a `Synch` message.

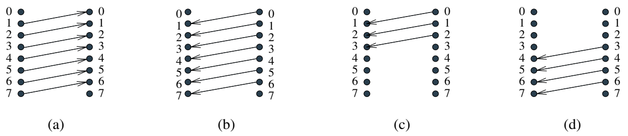


Figure 2: Communication and synchronization messages for the first loop of the program in Figure 7(a).

statements in which the flow of control enters at the beginning and leaves at the end without the possibility of branching except perhaps at the end [4]. A *control flow graph* (CFG) is a directed graph constructed from basic blocks and represents the flow-of-control information of the program. For the purpose of this paper, the CFG can be thought of as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where each $v \in \mathcal{V}$ represents either a basic block or a (reduced) interval that represents a loop, and each $e \in \mathcal{E}$ represents an edge between blocks. In this paper, depending on the context, we use the term *node* interchangeably for a statement, a block or an interval. Two unique nodes s and t denote the start and terminal nodes, respectively, of a CFG. One might think of these nodes as dummy statements. We define the sets of all successors and predecessors of a node n as $succ(n) = \{m \mid (n, m) \in \mathcal{E}\}$ and $pred(n) = \{m \mid (m, n) \in \mathcal{E}\}$, respectively. Node i *dominates* node j in the CFG (written as $j \in dom(i)$), if every path from s to j goes through i. We assume that prior to communication analysis, any edge that goes directly from a node with more than one successor to a node with more than one predecessor is split by introducing a dummy node [37]. Our technique for minimizing the communication volume and the number of messages is based on *interval analysis* [4]. Interval analysis consists of a *contraction phase* and an *expansion phase.* For programs written in a structured language, an interval corresponds to a loop. 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 each step of the expansion phase, on the other hand, a node (i.e., reduced interval) is expanded, and the information regarding the nodes in that interval is computed.

```
      REAL X(x_l:x_u), Y(y_l:y_u)
    !HPF$ TEMPLATE T(t_l:t_u)
    !HPF$ PROCESSORS PROC(p_l:p_u)
    !HPF$ ALIGN X(j), Y(j) WITH T(α*j+β)
    !HPF$ DISTRIBUTE T(CYCLIC(C)) ON TO PROC

     DO t = 1, TIMES /* time-step loop */
      ...
      < synchronization for Y using Synch operations >
  S:  < communication for Y using Put operations >
      DO i = i_l, i_u
       ...
       X(γ_L*i+θ_L) = ··· Y(γ_R*i+θ_R) ···
       ...
      END DO
      ...
     END DO
```

(a)

SET	PROJECTION FUNCTION
Producers(S)	$[p', q', d] \mapsto [q']$
Consumers(S)	$[p', q', d] \mapsto [p']$
ProducersFor(S, p)	$[p, q', d] \mapsto [q']$
ConsumersFor(S, q)	$[p', q, d] \mapsto [p']$
PutSet(S, p, q)	$[p, q, d] \mapsto [d]$
SendSet(S)	$[p', q', d] \mapsto [q', p']$
Pending(S)	$[p', q', d] \mapsto [p', q']$

(b)

Figure 3: (a) Generic loop. (b) Projection functions to manipulate sets (p and q are symbolic names).

4 Producers and consumers

We assume that all loop bounds, subscript expressions, and conditional expressions are affine functions of enclosing loop indices and the number of processor is known beforehand. Under these conditions, a loop nest, an array and a processor grid can all be represented as bounded polyhedra. Our approach uses the *owner-computes rule* [31, 57, 46], which assigns each computation to the processor that owns the data being computed. The technique presented here can be extended to handle cases where this rule is not adopted. Also, although we assume that data distributions across processors are specified using HPF-like directives, our techniques can also be used in conjunction with automatic data distribution frameworks such as those of Kremer [40], and Gupta and Banerjee [18].

Consider the one-dimensional generic loop i shown in Figure 3(a). Let $\mathcal{R}_{\mathcal{L}}(i)$ = $\mathtt{X}(\gamma_L{*}i{+}\theta_L)$ and $\mathcal{R}_{\mathcal{R}}(i)$ = $\mathtt{Y}(\gamma_R{*}i{+}\theta_R)$. Assuming symbols p and q denote two processors, we define the following sets where S is the communication statement and $\vee$ and $\wedge$ are the logical "or" and "and" operations, respectively:

$$
\begin{aligned}
\mathtt{Own}(\mathtt{X}, q) &= \{d \mid d \in \mathtt{X} \wedge \text{ is owned by } q\}\\
\mathtt{Producers}(\mathtt{S}) &= \{q' \mid \exists i, p' \text{ such that } \mathcal{R}_{\mathcal{R}}(i) \in \mathtt{Own}(\mathtt{Y}, q') \wedge \mathcal{R}_{\mathcal{L}}(i) \in \mathtt{Own}(\mathtt{X}, p') \wedge i_l \le i \le i_u \wedge q' \neq p'\}\\
\mathtt{Consumers}(\mathtt{S}) &= \{p' \mid \exists i, q' \text{ such that } \mathcal{R}_{\mathcal{R}}(i) \in \mathtt{Own}(\mathtt{Y}, q') \wedge \mathcal{R}_{\mathcal{L}}(i) \in \mathtt{Own}(\mathtt{X}, p') \wedge i_l \le i \le i_u \wedge q' \neq p'\}\\
\mathtt{ProducersFor}(\mathtt{S}, p) &= \{q' \mid \exists i \text{ such that } \mathcal{R}_{\mathcal{R}}(i) \in \mathtt{Own}(\mathtt{Y}, q') \wedge \mathcal{R}_{\mathcal{L}}(i) \in \mathtt{Own}(\mathtt{X}, p) \wedge i_l \le i \le i_u \wedge q' \neq p\}\\
\mathtt{ConsumersFor}(\mathtt{S}, q) &= \{p' \mid q \in \mathtt{ProducersFor}(\mathtt{S}, p')\}\\
\mathtt{PutSet}(\mathtt{S}, p, q) &= \{d \mid \exists i \text{ such that } d = \mathcal{R}_{\mathcal{R}}(i) \in \mathtt{Own}(\mathtt{Y}, q) \wedge \mathcal{R}_{\mathcal{L}}(i) \in \mathtt{Own}(\mathtt{X}, p) \wedge i_l \le i \le i_u \wedge q \neq p\}
\end{aligned}
$$

$$\texttt{SendSet(S)} = \{(q', p') \mid \exists d \text{ such that } d \in \texttt{PutSet}(\texttt{S}, p', q')\}$$

$$\texttt{Pending(S)} = \{(p', q') \mid (q', p') \in \texttt{SendSet(S)}\}.$$

The set `Own`(`X`,p) refers to the elements of array `X` mapped onto processor p through compiler directives. Similar `Own` sets are defined for other arrays as well. The sets `Producers(S)` and `Consumers(S)` denote, respectively, the processors that produce and consume data communicated in `S`. For a specific processor, `ProducersFor` and `ConsumersFor` give the set of processors that send data to and receive data from that processor, respectively. `PutSet`(`S`,p,q) is the set of elements that should be *put* (written) by processor q to memory of processor p without the active involvement of the latter. `SendSet(S)` is set of pairs (q', p') such that q' sends data to (write data in the memory of) p'. Finally, `Pending(S)` is the inverse of `SendSet(S)`, and gives set of pairs (p', q') such that p' should send a `Synch` message to q' for the repetitions of the communication (in each iteration of the time-step loop t) occurring in `S`. For a communication occurring in `i`, the set `Pending(i)` represents a list of individual `Synch` messages that should be communicated for the safe repetition of the data communication in `i`. That is, a `Synch` message is between just a pair of processors. For an `i` and a `Synch`, we say whether or not `Synch` $\in$ `Pending(i)`.

In fact, by using appropriate projection functions, all of those sets can be obtained from a single set called `CommSet(S)` containing triples (p', q', d) meaning that element d should be communicated from q' to p' in `S`. In general, `CommSet(S)` or a similar set is used in the communication optimization phase of distributed-memory compilers to generate the correct `Send` and `Recv` commands. The necessary projection functions can be implemented by using the Omega library [34, 44], and are shown in Figure 3(b). For instance, `ConsumersFor`$(\texttt{S}, q)$ is obtained from `CommSet(S)` by projecting out d and substituting q for q'; that is, q is a parameter and `ConsumersFor`$(\texttt{S}, q)$ enumerates p' values in terms of q. Notice that by building upon an existing framework, our approach can be extended to compile programs using a hybrid approach that consists of both `Put/Get` and `Send/Recv` primitives.

By taking into account the alignment and distribution information provided by the compiler directives, we can define the `Own` set more formally as

$$\begin{aligned}\texttt{Own}(\texttt{Y}, q) = \{d \mid \exists t, c, l \text{ such that } & (t = \alpha * d + \beta) \wedge (t = \mathcal{C} * P * c + \mathcal{C} * q + l) \wedge (y_l \leq d \leq y_u) \wedge \\ & (p_l \leq q \leq p_u) \wedge (t_l \leq t \leq t_u) \wedge (0 \leq l \leq \mathcal{C} - 1)\},\end{aligned}$$

Processor 0			Processor 1			Processor 2			Processor 3		
0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59
...	...	...	...	...	...	...	...	...	...	...	...

(a) Global addresses of array elements.

Processor 0			Processor 1			Processor 2			Processor 3		
(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)	(0,8)	(0,9)	(0,10)	(0,11)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)	(1,9)	(1,10)	(1,11)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)	(2,9)	(2,10)	(2,11)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)	(3,9)	(3,10)	(3,11)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)	(4,9)	(4,10)	(4,11)
...	...	...	...	...	...	...	...	...	...	...	...

(b) Two-dimensional view of global addresses. Addresses on processor q are $(c, \mathcal{C} * q + l)$.

Processor 0			Processor 1			Processor 2			Processor 3		
(0,0)	(0,1)	(0,2)	(0,0)	(0,1)	(0,2)	(0,0)	(0,1)	(0,2)	(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)	(1,0)	(1,1)	(1,2)	(1,0)	(1,1)	(1,2)	(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)	(2,0)	(2,1)	(2,2)	(2,0)	(2,1)	(2,2)	(2,0)	(2,1)	(2,2)
(3,0)	(3,1)	(3,2)	(3,0)	(3,1)	(3,2)	(3,0)	(3,1)	(3,2)	(3,0)	(3,1)	(3,2)
(4,0)	(4,1)	(4,2)	(4,0)	(4,1)	(4,2)	(4,0)	(4,1)	(4,2)	(4,0)	(4,1)	(4,2)
...	...	...	...	...	...	...	...	...	...	...	...

(c) Two-dimensional view of local addresses, (c, l).

Figure 4: HPF-style data mappings: global and local addresses of the accessed elements of a one-dimensional array along with the two-dimensional view for a four-processor case with $\mathcal{C} = 3$.

where $P = p_u - p_l + 1$. In this formulation, $t = \alpha * d + \beta$ represents alignment and $t = \mathcal{C} * P * c + \mathcal{C} * q + l$ denotes distribution. In other words, each array element d is mapped onto a point in a 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 4(a) shows the global addresses of a one-dimensional array distributed in block-cyclic manner across four processors with a blocking factor of $\mathcal{C} = 3$. Figures 4(b) and (c), on the other hand, illustrate a two-dimensional view 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, element 43 of the array is mapped onto Processor 2 with $c = 3$ and $l = 1$ as local coordinates.

$$
\begin{aligned}
\texttt{Producers}(2) &= \{q \mid 1 \le q \le 7\} \\
\texttt{Consumers}(2) &= \{p \mid 0 \le p \le 6\} \\
\texttt{ProducersFor}(2,p) &= \{q \mid (q = p+1) \wedge (0 \le p \le 6)\} \\
\texttt{ConsumersFor}(2,q) &= \{p \mid (p = q-1) \wedge (1 \le q \le 7)\} \\
\texttt{PutSet}(2,p,q) &= \{d \mid (d = 16q) \wedge (p+1 = q) \wedge (1 \le q \le 7)\} \\
\texttt{SendSet}(2) &= \{(q, q-1) \mid 1 \le q \le 7\} \\
\texttt{Pending}(2) &= \{(p, p+1) \mid 0 \le p \le 6\}
\end{aligned}
$$

(a)

$$
\begin{aligned}
\{\vec{d} \mid \mathcal{P}(\vec{d})\} \vee_c \{\vec{d} \mid \mathcal{Q}(\vec{d})\} &= \{\vec{d} \mid \mathcal{P}(\vec{d}) \vee \mathcal{Q}(\vec{d})\} \\
\{\vec{d} \mid \mathcal{P}(\vec{d})\} -_c \{\vec{d} \mid \mathcal{Q}(\vec{d})\} &= \{\vec{d} \mid \mathcal{P}(\vec{d}) \wedge \neg(\mathcal{Q}(\vec{d}))\} \\
\{\vec{d} \mid \mathcal{P}(\vec{d})\} \wedge_c \{\vec{d} \mid \mathcal{Q}(\vec{d})\} &= \{\vec{d} \mid \mathcal{P}(\vec{d}) \wedge \mathcal{Q}(\vec{d})\}
\end{aligned}
$$

(c)

$$
\begin{aligned}
\texttt{Producers}(2) &= \{q \mid 0 \le q \le 7\} \\
\texttt{Consumers}(2) &= \{q \mid 0 \le q \le 7\} \\
\texttt{ProducersFor}(2,p) &= \{q \mid q = 0 \wedge p = 7\} \\
&\quad \cup \{q \mid (q = p+1) \wedge (0 \le p \le 6)\} \\
\texttt{ConsumersFor}(2,q) &= \{p \mid p = 7 \wedge q = 0\} \\
&\quad \cup \{p \mid (p = q-1) \wedge (1 \le q \le 7)\} \\
\texttt{PutSet}(2,p,q) &= \{d \mid \exists\alpha \text{ such that } (d = 32\alpha) \wedge (q = 0) \wedge (p = 7) \\
&\quad \wedge (32 \le d \le 128)\} \cup \\
&\quad \{d \mid \exists\alpha \text{ such that } (d = 4 + 4p + 32\alpha) \wedge (q = p+1) \\
&\quad \wedge (0 \le p \le 6) \wedge (4p + 4 \le d \le 4p + 100)\} \\
\texttt{SendSet}(2) &= \{(0,7)\} \cup \{(q, q-1) \mid 1 \le q \le 7\} \\
\texttt{Pending}(2) &= \{(p, p+1) \mid 0 \le p \le 6\} \cup \{(7,0)\}
\end{aligned}
$$

(b)

$$
\begin{aligned}
\texttt{PENDING\_IN(i)} &= \bigsqcup_{\texttt{j} \in pred(\texttt{i})} \texttt{PENDING\_OUT(j)} \\
\texttt{PENDING\_OUT(i)} &= \mathcal{F}_i(\texttt{PENDING\_IN(i)})
\end{aligned}
$$

(d)

Figure 5: (a) Sets for the first loop in Figure 7(b) for `BLOCK` distribution. (b) Sets for the first loop in Figure 7(b) for `CYCLIC(4)` distribution. (c) Operations on `PutSets`. (d) Data-flow equations for optimizing synchronization messages.

These relations can easily be modified to accommodate replicated arrays and collapsed dimensions. The formulation given here can be generalized to multi-dimensional loops, arrays and processor grids [6, 49]. Consider the first i-loop in Figure 7(b). Figure 5(a) shows the sets for this loop assuming that the array bounds start from 0 in the transformed program. Notice that a processor q is in the `Producers` set if there exists a processor p such that $q \neq p$ and q puts data in p's memory. Similarly, a processor p is in the `Consumers` set if there exists a processor q such that $p \neq q$ and q puts data in p's memory. For this example, if distribution directive for the arrays is changed to `CYCLIC(4)`, then we have the sets shown in Figure 5(b). All of these sets can easily be represented and manipulated by the Omega library [34, 44] or a similar polyhedral tool. Notice that using the Omega sets to represent producer-consumer information, we are able to accommodate any type of HPF-style alignment and distribution in our framework through `Own` sets.

Let $\{\vec{d} \mid \mathcal{P}(\vec{d})\}$ and $\{\vec{d} \mid \mathcal{Q}(\vec{d})\}$ be two `PutSets` for the same multi-dimensional array where $\mathcal{P}(.)$ and $\mathcal{Q}(.)$ are two predicates and $\vec{d}$ refers to an array element. We define three operations, $\vee_c$, $-_c$, and $\wedge_c$ on these $\texttt{PutSets}$ as shown in Figure 5(c). In the remainder of this paper, $\bigvee$ and $\bigwedge$ symbols will also be used for $\vee_c$ and $\wedge_c$, respectively, when there is no confusion.

So far, we have described our method informally and defined the terminology that would be used in the remainder of this paper. Next, we explain our communication and synchronization optimization algorithms in detail.

5 Optimizing communication

The objective of our global communication optimization framework is to determine the set $\texttt{PutSet}(\texttt{i}, p, q)$ for each node $\texttt{i}$ in the program globally by taking into account all the nodes in the CFG that are involved in communication.

5.1 Local (intra-interval) analysis

In order to make the data-flow analysis task easier, the CFG of the program is traversed prior to the local analysis phase, and for each LHS reference a pointer is stored in the header of all enclosing loop nests. The local analysis part of our framework computes $\texttt{Kill}$, $\texttt{Gen}$ and $\texttt{Post\_Gen}$ sets defined below for each array and for each interval. Then the interval is reduced to a single node and annotated with these sets. With reference to Figure 3(a), for array $\texttt{X}$, the following sets are computed:

$$\begin{aligned}
\texttt{Kill}(\texttt{i}, q) &= \{\vec{d} \mid (\vec{d} \in \texttt{Own}(\texttt{X}, q)) \wedge (\exists \vec{\imath} \text{ such that } (\vec{d} = \mathcal{R}_{\mathcal{L}}(\vec{\imath})) \wedge (\vec{i_l} \leq \vec{\imath} \leq \vec{i_u}))\}, \\
\texttt{Modified}(\texttt{i}, q) &= [\bigvee_{\texttt{j} \in pred(\texttt{i})} \texttt{Modified}(\texttt{j}, q)] \vee_c \texttt{Kill}(\texttt{i}, q),
\end{aligned}$$

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

$\texttt{Kill}(\texttt{i}, q)$ is the set of elements owned and written (killed) by processor q locally in $\texttt{i}$, and $\texttt{Modified}(\texttt{i}, q)$ is the set of elements that may be killed along any path from the beginning of the interval to (and including) the node $\texttt{i}$. The computation of the $\texttt{Kill}$ set (and that of the $\texttt{Modified}(\texttt{i}, q)$ set) proceeds in the *forward direction;* that is, the nodes within the interval are traversed in topological sort order. If $\texttt{last(i)}$ is the last node in $\texttt{i}$, then

$$\texttt{Kill}(\texttt{i}, q) = \texttt{Modified}(\texttt{last}(\texttt{i}), q).$$

This last equation is used to reduce an interval into a node. The reduced interval is then annotated by its

Kill set.

$\mathtt{Gen}(\mathtt{i}, p, q)$ is the set of elements to be written by q into p's memory to satisfy the communication in i. The computation of the set 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 written by q into p's memory at the beginning of a node in the CFG are the elements required by p due to a RHS reference in the node except the ones that are written locally (killed) by q before being referenced by p. Note that this process involves considering all the LHS references within an interval. The cost incurred is manageable since the largest scope for this analysis is a single loop nest and, as mentioned earlier, prior to analysis we keep pointers to all LHS references within a loop nest.

Assuming $\vec{\imath} = (\imath_1, ..., \imath_n)$ and $\vec{\imath'} = (\imath'_1, ..., \imath'_n)$, let $\vec{\imath'} \prec \vec{\imath}$ mean that $\vec{\imath'}$ is lexicographically less than or equal to $\vec{\imath}$; and $\vec{\imath'} \prec_k \vec{\imath}$ mean that $\imath'_j = \imath_j$ for all $j < k$, and $(\imath'_k, ..., \imath'_n) \prec (\imath_k, ..., \imath_n)$. Let $\mathtt{Comm}(\mathtt{i}, p, q)$ be the set of elements that may be communicated at the beginning of interval i to satisfy communication requirements from the beginning of i to the last node of the interval which contains i. Then, from Figure 3(a), we have

$$\begin{aligned}
\mathtt{Gen}(\mathtt{i}, p, q) &= \{\vec{d} \mid \exists \vec{\imath} \text{ such that } (\vec{\imath_l} \le \vec{\imath} \le \vec{\imath_u}) \wedge (\vec{d} = \mathcal{R}_{\mathcal{R}}(\vec{\imath}) \in \mathtt{Own}(\mathtt{Y}, q)) \wedge (\mathcal{R}_{\mathcal{L}}(\vec{\imath}) \in \mathtt{Own}(\mathtt{X}, p)) \\
&\quad \wedge \neg(\exists \vec{\jmath}, \mathcal{R}_{\mathcal{L}}' \text{ such that } (\vec{\imath_l} \le \vec{\jmath} \le \vec{\imath_u}) \wedge (\vec{d} = \mathcal{R}_{\mathcal{L}}'(\vec{\jmath})) \wedge (\vec{\jmath} \prec_{\mathtt{level(i)}} \vec{\imath}))\}, \\
\mathtt{Comm}(\mathtt{i}, p, q) &= [\bigwedge_{\mathtt{s} \in succ(\mathtt{i})} \mathtt{Comm}(\mathtt{s}, p, q)] \vee_c \mathtt{Gen}(\mathtt{i}, p, q).
\end{aligned}$$

The negated condition eliminates all the elements written by q locally in an earlier iteration than the one in which p requires them. In addition, we use the following equation to reduce an interval into a single node

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

In the definition of Gen, $\mathcal{R}_{\mathcal{R}}$ denotes the RHS reference, and $\mathcal{R}_{\mathcal{L}}$ denotes the LHS reference of the same statement. $\mathcal{R}_{\mathcal{L}}'$, on the other hand, refers to any LHS reference within the same interval. Notice that while $\mathcal{R}_{\mathcal{L}}'$ is a reference to the same array as $\mathcal{R}_{\mathcal{R}}$, in general $\mathcal{R}_{\mathcal{L}}$ can be a reference to any array; level(i) gives the nesting level of the interval (loop) with a value 1 corresponding to the outermost loop in the nest. As a note, we should mention that there are more subtle issues here as lexical positions of the statements may also need be taken into account when Gen is computed. For the sake of simplicity of the presentation, we omit these details.

After the interval is reduced, the Gen set is recorded, and an operator $\mathcal{N}$ is applied to the last part of this

Gen set to propagate it to the outer interval:

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

It should be emphasized that computation of the Gen sets gives us all the communication that can be vectorized and hoisted out of a loop nest; i.e., our analysis easily handles what is known as *message vectorization* [31, 49] in Send/Recv based message-passing frameworks. A naive implementation may set $\mathtt{Put\_Set}(\mathtt{i}, p, q)$ to $\mathtt{Gen}(\mathtt{i}, p, q)$ for every i, p and q. But such an approach often retains redundant communication which would otherwise be eliminated.

Finally, $\mathtt{Post\_Gen}(\mathtt{i}, p, q)$ is the set of elements to be written by q into memory of p at node i with no subsequent local write to them by q:

$$\begin{aligned}\mathtt{Post\_Gen}(\mathtt{i}, p, q) \quad = \quad & \{\vec{d} \mid \exists\vec{\imath} \text{ such that } (\vec{i_l} \le \vec{\imath} \le \vec{i_u}) \wedge (\vec{d} = \mathcal{R}_{\mathcal{R}}(\vec{\imath}) \in \mathtt{Own}(\mathtt{Y}, q)) \wedge (\mathcal{R}_{\mathcal{L}}(\vec{\imath}) \in \mathtt{Own}(\mathtt{X}, p)) \\ & \wedge\neg(\exists\vec{\jmath}, \mathcal{R}_{\mathcal{L}}' \text{ such that } (\vec{i_l} \le \vec{\jmath} \le \vec{i_u}) \wedge (\vec{d} = \mathcal{R}_{\mathcal{L}}'(\vec{\jmath})) \wedge (\vec{\imath}\prec_{\mathtt{level(i)}}\vec{\jmath}))\}.\end{aligned}$$

The computation of $\mathtt{Post\_Gen}(\mathtt{i}, p, q)$ proceeds in the *forward direction.* Its computation is very similar to those of Kill and Gen sets, so we do not discuss it further.

5.2 Data-flow equations

In our framework, one-way communication calls are placed at the beginning of nodes in the CFG. Our data-flow analysis consists of a backward and a forward pass (performed for each array). In the backward pass, the compiler determines sets of data elements that can safely be communicated at specific points. The forward pass, on the other hand, eliminates redundant communication and determines the final set of elements that should be communicated (written by q into p's memory) at the beginning of each node i. The input for the equations consists of the Gen, Kill, and Post_Gen sets.

The data-flow equations for the backward analysis are given by Equations (1) and (2) in Figure 6. Basically, they are used to combine and hoist communication. $\mathtt{Safe\_In}(\mathtt{i}, p, q)$ and $\mathtt{Safe\_Out}(\mathtt{i}, p, q)$ are sets of elements that *can* safely be communicated at the beginning and at the 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. Equation (2), on the other hand, gives the set of elements that can safely be communicated at the beginning of i. Intuitively, an element can be written by q into p's memory at

Backward Analysis:

$$\texttt{Safe\_Out}(\texttt{i},p,q) = \bigwedge_{\texttt{s}\in succ(\texttt{i})} \texttt{Safe\_In}(\texttt{s},p,q) \tag{1}$$

$$\texttt{Safe\_In}(\texttt{i},p,q) = \begin{cases} \texttt{Gen}(\texttt{i},p,q) & \text{if } \mathcal{P}(\texttt{i}) \\ (\texttt{Safe\_Out}(\texttt{i},p,q) -_c \texttt{Kill}(\texttt{i},q)) \vee_c \texttt{Gen}(\texttt{i},p,q) & \text{otherwise} \end{cases} \tag{2}$$

Forward Analysis:

$$\texttt{Put\_In}(\texttt{i},p,q) = \bigwedge_{\texttt{j}\in pred(\texttt{i})} \texttt{Put\_Out}(\texttt{j},p,q) \tag{3}$$

$$\texttt{PutSet}(\texttt{i},p,q) = \begin{cases} \texttt{Gen}(\texttt{i},p,q) -_c \texttt{Put\_In}(\texttt{i},p,q) & \text{if } \exists\texttt{k} \text{ such that } \texttt{k} \in succ(\texttt{i}) \text{ and } \texttt{k} \notin dom(\texttt{i}) \\ \texttt{Safe\_In}(\texttt{i},p,q) -_c \texttt{Put\_In}(\texttt{i},p,q) & \text{otherwise} \end{cases} \tag{4}$$

$$\texttt{Put\_Out}(\texttt{i},p,q) = \begin{cases} \texttt{Put\_In}(\texttt{i},p,q) -_c \texttt{Kill}(\texttt{i},q) & \text{if } \exists\texttt{k} \text{ such that } \texttt{k} \in succ(\texttt{i}) \text{ and } \texttt{k} \notin dom(\texttt{i}) \\ ((\texttt{PutSet}(\texttt{i},p,q) +_c \texttt{Put\_In}(\texttt{i},p,q)) -_c \texttt{Kill}(\texttt{i},q)) +_c \texttt{Post\_Gen}(\texttt{i},p,q) & \text{otherwise} \end{cases} \tag{5}$$

Figure 6: Data-flow equations for optimizing communication (data) messages.

the beginning of $\texttt{i}$ if and only if it is either required by p in $\texttt{i}$ or it reaches at the end of $\texttt{i}$ (in the backward analysis) and is not overwritten (killed) by the owner (q) in it. The predicate $\mathcal{P}(\texttt{i})$ is used to control communication hoisting. If $\mathcal{P}(\texttt{i})$ is true, communication is not hoisted to the beginning of $\texttt{i}$. $\mathcal{P}(\texttt{i})$=*false* implies aggressive communication combining and hoisting. An algorithm can include a condition to test the compatibility between the sets $\texttt{Gen(i},p,q\texttt{)}$ and $\texttt{Safe\_Out(i},p,q\texttt{)}$. For example, two left-shift communications are compatible whereas a left-shift and a right-shift are incompatible. [10]. The other possibilities include avoiding message splitting ($\texttt{Kill(i},q\texttt{)} \neq \emptyset$), clustering ($\texttt{Gen(i},p,q\texttt{)} = \emptyset$), and avoiding the buffer pressure [36]. In the experiments we did, we combined and hoisted communication aggressively for codes except for `tomcatv`, where we checked the compatibility before combining. Note that a re-definition of $\mathcal{P}(\texttt{i})$ can totally change the behavior of the algorithm as well as the empirical results. It should also be noted that for some applications ignoring the control predicates may affect the correctness of the resulting code as well [36]. An in-depth discussion of different control predicates, however, is beyond the scope of this paper.

The task of the forward analysis phase, which makes use of Equations (3), (4) and (5) in Figure 6, is to eliminate redundant communication by observing the following: (i) a node in the CFG should not have a non-local datum which is exclusively needed by a successor unless it dominates that successor; and (ii) a successor should ignore what a predecessor has so far unless that predecessor dominates it. The sets $\texttt{Put\_In}(\texttt{i},p,q)$ and $\texttt{Put\_Out}(\texttt{i},p,q)$ denote the set of elements that *have been* written so far (at the

```
 REAL A(128), B(128), C(128)
     D(128), E(128), F(128)
!HPF$ PROCESSORS PROC(0:7)
!HPF$ DISTRIBUTE BLOCK ON TO
     PROC ::  A, B, C, D, E

  DO t = 1, TIMES
   DO i = 1, 127
S:  A(i) = B(i+1)
   END DO
   DO i = 1, 126
    D(i) = A(i) + 1
    A(i) = B(i+1) + B(i+2)
   END DO
   IF (A(1).EQ.B(1))
    DO i = 2, 64
     C(i) = D(i-1)
    END DO
   ELSE
    DO i = 2, 64
     A(i) = C(i) + D(i-1)
    END DO
   END IF
   DO i = 1, 127
    F(i) = A(i) - 1
    B(i) = A(i) * A(i)
   END DO
   DO i = 65, 128
    E(i) = F(i-1)
   END DO
  END DO
```

(a)

```
 REAL A(128), B(128), C(128)
     D(128), E(128), F(128)
!HPF$ PROCESSORS PROC(0:7)
!HPF$ DISTRIBUTE BLOCK ON TO
     PROC ::  A, B, C, D, E

DO t = 1, TIMES
 Synch; Put{B};
 DO i = 1, 127
  A(i) = B(i+1)
 END DO
 Synch; Put{B};
 Synch; Put{B};
 DO i = 1, 126
  D(i) = A(i) + 1
  A(i) = B(i+1) + B(i+2)
 END DO
 IF (A(1).EQ.B(1))
  Synch; Put{D};
  DO i = 2, 64
   C(i) = D(i-1)
  END DO
 ELSE
  Synch; Put{D};
  DO i = 2, 64
   A(i) = C(i) + D(i-1)
  END DO
 END IF
 DO i = 1, 127
  F(i) = A(i) - 1
  B(i) = A(i) * A(i)
 END DO
 Synch; Put{F};
 DO i = 65, 128
  E(i) = F(i-1)
 END DO
END DO
```

(b)

```
 REAL A(128), B(128), C(128)
     D(128), E(128), F(128)
!HPF$ PROCESSORS PROC(0:7)
!HPF$ DISTRIBUTE BLOCK ON TO
     PROC ::  A, B, C, D, E

  DO t = 1, TIMES
1:  Synch; Put{B};
   DO i = 1, 127
    A(i) = B(i+1)
   END DO
   DO i = 1, 126
    D(i) = A(i) + 1
    A(i) = B(i+1) + B(i+2)
   END DO
2:  Synch; Put{D};
   IF (A(1).EQ.B(1))
    DO i = 2, 64
     C(i) = D(i-1)
    END DO
   ELSE
    DO i = 2, 64
     A(i) = C(i) + D(i-1)
    END DO
   END IF
   DO i = 1, 127
    F(i) = A(i) - 1
    B(i) = A(i) * A(i)
   END DO
3:  Synch; Put{F};
   DO i = 65, 128
    E(i) = F(i-1)
   END DO
  END DO
```

(c)

Figure 7: (a) A synthetic benchmark. (b) Message vectorized translation. (c) Global communication optimization.

beginning and end of node i, respectively) by q into memory of p. Equation (3) conservatively 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 PutSet set which corresponds to the elements that can be communicated at the beginning of the node except the ones that have already been communicated (Put_In). The elements that have been communicated at the end of node i (that is, Put_Out set) are simply the union of the elements communicated up to the beginning of i (that is, Put_In set), the elements communicated at the beginning of i (that is, PutSet set) (except the ones which have been killed in i), and the elements communicated in i and not killed subsequently (that is, Post_Gen set).

5.3 Overall interval 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 the innermost loop to the outermost loop and annotates them with the Gen, Kill and Post_Gen sets. When a reduced CFG with no cycles is reached, the expansion phase starts and the PutSets for each node is computed from the outermost loop to the innermost loop. There is one important point to note: before starting to process the next graph in the expansion phase, the Put_In set of the first node in this graph is set to the PutSet of the interval that contains it to avoid redundant communication. More formally, in the expansion phase, we set $\texttt{Put\_In}(\texttt{i}, p, q)^{k^{th} pass} = \texttt{PutSet}(\texttt{i}, p, q)^{(k-1)^{th} pass.}$. This assignment then triggers the next pass in the expansion phase. Before the expansion phase starts $\texttt{Put\_In}(\texttt{i}, p, q)^{1^{st} pass}$ is set to the empty set. Note that the whole data-flow procedure operates on sets of equalities and inequalities which can be manipulated by the Omega library [34] or a similar polyhedral tool. Note also that the SendSet and Pending sets should be updated accordingly after global communication optimization.

5.4 Discussion

It should be noted that the analysis presented so far works with sets of equalities and inequalities. Compared to 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. For example, suppose that a data-flow equation requires combining two sets $S_x = \{[x] : \mathcal{Q}_1(x)\}$ and

$S_y = \{[y] : \mathcal{Q}_2(y)\}$ where $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are predicates consisting of equalities and inequalities. Instead of forming the set $\{[z] : \mathcal{Q}_1(z) \vee \mathcal{Q}_2(z)\}$ immediately, our approach represents the resulting set abstractly as $S_x + S_y$. When the whole process is finished, the resulting sets are re-written in terms of equalities and inequalities and the *simplify* utility of the Omega library [34] is used to simplify them. Our experience shows that this approach requires a reasonable support for the manipulation of symbolic expressions and is fast in practice.

5.5 Example

Consider the synthetic benchmark given in Figure 7(a). In this example, communication occurs for three arrays, B, D and F. A communication optimization scheme based on message vectorization alone, can place the communications and the associated synchronization as shown in Figure 7(b) before the loop bounds reduction and guard insertion. Note that a Synch message in that figure in fact represents a number of point-to-point synchronization messages. In particular, for a communication occurring at i, a processor p must get synchronized with every processor q that satisfies the condition: $(p, q) \in \texttt{Pending}(\texttt{i})$. An application of our global communication optimization method generates the program shown in Figure 7(c). Compared to the message vectorized version, there is a 50% reduction (from 28 to 14) in the number of messages and 40% in the communication volume (from 35 to 21) across all processors. We note that we can optimize this program even when the distribution directive is changed to CYCLIC(K) for any K. Most of the previous approaches are not able to handle the global communication optimization for $\texttt{K} \geq 2$, mostly due to the representations they use for the communication sets. In fact, when the distribution directive is CYCLIC(4), we have a 50% reduction (from 48 to 24) in the number of messages and 32% reduction (from 139 to 94) in the communication volume across all processors. Note that our approach here reduces the original number of synchronization messages as well (from 28 to 14 for the BLOCK distribution and from 48 to 24 for the CYCLIC(4) case). We investigate the conditions under which the synchronization messages can be further decreased in the following section.

6 Optimizing synchronization

In this section, we assume that the compiler has conducted the data-flow analysis described in Section 5 and determined the optimal communication points and communication sets. Assuming that these communications will be implemented using `Put` operations, we present a data-flow analysis to minimize the number of `Synch` messages. We assume that communication patterns (i.e., producer-consumer relationships) are *identical* for each repetition of a communication during the loop execution. For example, in Figure 7(c), the producer-consumer pattern for the communication occurring in `1` is identical for every repetition of time-step loop `t`. The problem formulation and the conditions under which the redundant synchronization can be eliminated for the *varying* producer-consumer case can also be obtained by extending our approach. Our experience, however, shows that in that case only a small fraction of the `Synch` messages can be eliminated.

Our approach first makes a single pass over the current interval and determines some synchronizations that cannot be eliminated by the analysis to be described. Informally, if a synchronization is repeated in the program in different places with the same producer-consumer and is associated with (the part of a same) buffer and there is no intervening communication in the synchronization direction, then that synchronization cannot be eliminated. We call the set of synchronizations (associated with a node `i`) that cannot be eliminated `SynchFix(i)`.

The data-flow technique described here starts with the deepest loops and works its way through loops in a bottom up manner, handling one loop at a time. It then reduces the loop to a node and annotates it with its final synchronization requirements that cannot be eliminated. Since the approach starts from the deepest loops, it tends to eliminate the synchronization requirements from the most frequently executed parts of the program first. The synchronization is introduced at the least executed parts of the code only to preserve correctness. The procedure works on an augmented CFG, where each communication loop is represented by a single node. In the following discussion, the symbol `i` refers to such a node.

For a given `i`, we can define the set `SynchSet` as a set of processor pairs that should be synchronized after our analysis. In a straightforward implementation, `SynchSet(i) = Pending(i)` for each `i`. We would like to reduce the cardinality of `SynchSet(i)` for each `i`. We say that no synchronization is required for a communication `i`, if `SynchSet(i)` becomes an empty set after our data-flow analysis.

If the compiler wants to eliminate a `Synch` message for communication `i` from p to q, it needs to find a message $\mathtt{t}_j$ for another communication from p to q and use it as synchronization. Such a message should occur *between* repetitions of `i` and *after* the data value communicated at `i` is consumed. Suppose that a specific producer q and a consumer p are involved in a communication in `i`. Consider all k communications $\mathtt{t}_j$ ($1 \leq j \leq k$) occurring after the value communicated in `i` is consumed by p and before the next repetition of `i`. Then, if the following holds, the `Synch` message from p to q can be eliminated:

$$(\exists j \mid (1 \leq j \leq k) \wedge q \in \texttt{ConsumersFor}(\texttt{t}_j, p)).$$

An interesting case occurs when all the `Synch` messages contained in the set `Pending(i)` for a specific `i` are eliminated. We can formalize this condition as

$$\forall p \forall q (((p,q) \in \texttt{Pending(i)}) \quad \Rightarrow \quad \exists j ((p,q) \in \texttt{SendSet}(\texttt{t}_j))), \tag{6}$$

assuming $1 \leq j \leq k$. Notice that the j values can be different for each q. If we additionally require that all j values should be the same for all q values, then we obtain

$$\forall p \forall q \exists j (((p,q) \in \texttt{Pending(i)}) \quad \Rightarrow \quad ((p,q) \in \texttt{SendSet}(\texttt{t}_j))) \tag{7}$$

We note that Condition (7) leads to the synchronization elimination algorithms offered by Gupta and Schonberg [19], and Hinrichs [30].

Claim: *The synchronizations eliminated by Condition (7) are a subset of the synchronizations that can be eliminated by Condition (6).*

Proof:

Let the set of synchronizations eliminated by scheme (6) be S_1 and those eliminated by scheme (7) be S_2. We need to show that $S_2 \subseteq S_1$.

Consider a synchronization, $(p,q) \in S_2$. Since $(p,q) \in S_2$, condition (7) above implies that $\exists j$ s.t. $q \in ConsumersFor(t_j, p)$. From condition (6) above this implies that $(p,q) \in S_1$.

Now consider a case where there are two synchronization requirements (p_1, q_1) and (p_2, q_2). Let $j_1, j_2 (j_1 \neq j_2)$ be such that $q_1 \in ConsumersFor(t_{j_1}, p_1)$ and $q_2 \in ConsumersFor(t_{j_2}, p_2)$ and there is no other j which meets this condition. By condition (6), $\{(p_1,q_1),(p_2,q_2)\} \in S_1$. However, since $j_1 \neq j_2$, $\{(p_1,q_1),(p_2,q_2)\} \notin S_2$ by condition (7). Hence $S_2 \subseteq S_1$. $\square$

Even if a `Pending(i)` set cannot be totally eliminated, we can reduce its cardinality by eliminating as many Synch messages as possible from it. That is, after our analysis, for every i,

$$\texttt{SynchSet(i)} \quad = \quad \texttt{Pending(i)} -_c \{(p,q) \mid \exists j \text{ such that } q \in \texttt{ConsumersFor}(\texttt{t}_j, p)\} \vee_c \texttt{SynchFix(i)}. \qquad (8)$$

As an example, let us consider the program shown in Figure 7(c). In this program, communication occurs at three points: 1, 2 and 3. A straightforward implementation inserts three sets of Synch operations, corresponding to 1, 2 and 3 as shown in that figure. Let us now focus on the communication in 1. In fact, this communication is combination of communications due to references B(i+1), B(i+1) and B(i+2) in lines 4 and 10 in Figure 7(b). Figure 2(a) shows the messages sent (Put operations) for this communication. Figure 2(b), on the other hand, shows the required synchronization messages for the repetitions of this communication. Finally, Figures 2(c) and 2(d) show the communication messages in 2 and 3, respectively. Notice that, by using the condition given in (6), the synchronization requirements for 1 can be eliminated; i.e., the communications occurring in 2 and 3 together kill the synchronization requirements for the repetitions of the communication in 1. If we consider Figure 2(c) and Figure 2(d) separately for the condition given by (7), however, none of them individually can eliminate the synchronization for 1. Using a similar argument, it can be concluded that the communications in 2 and 3 do not need any synchronization either, as their synchronization requirements are killed by the communication taking place in 1.

6.1 Data-flow analysis

We now present our data-flow analysis for eliminating redundant synchronization messages. The data flow equations are shown in Figure 5(d). Our analysis consists of iterative forward passes on the augmented CFG. Let us first concentrate on the second equation in that figure and explain the functionality of $\mathcal{F}_i$. In that equation, PENDING_IN(i) represents the synchronization requirements of all the communications traversed so far up to the beginning of i. PENDING_OUT(i) is defined analogously for the end of i.

Assuming that P_k is the synchronization requirement (in terms of pairs of processors) for node k up to i in the analysis and $\texttt{PENDING\_IN(i)} = \{P_1, P_2, ..., P_{i-1}, P_i, P_{i+1}, ..., P_m\}$, we can define $\mathcal{F}_i$ as

$$\mathcal{F}_i(\texttt{PENDING\_IN(i)}) \quad = \quad \{f_i(P_1), f_i(P_2), ..., f_i(P_{i-1}), P_i, f_i(P_{i+1}), ..., f_i(P_m)\},$$

where $f_i(P_k) = P_k -_c \texttt{SendSet(i)}$. In other words, when a node i is visited, the synchronization requirements for all *other* communications are checked to see whether or not any synchronization can be eliminated by using the communication occurring at `i`. Prior to the analysis, P_i is set to `Pending(i)` for the *first* node. After a fixed state is reached, `PENDING_OUT` set of the *last* node gives the synchronization requirements to be satisfied. In fact, except for the statements in the conditionally executed blocks, in steady state, all `PENDING_OUT` sets should be the same. The resulting `PENDING_OUT` is then reduced to a single set and is used to represent the synchronization requirements of this loop to the next upper level.

We now consider the first equation of Figure 5(d) and explain the $\bigsqcup$ operator appearing there. In the join nodes, the compiler takes a conservative approach by computing the union of the synchronization requirements for the same communication. Suppose that for each $j \in pred(i)$, `PENDING_OUT(j)` = $\{P_{j1}, P_{j2}, ..., P_{jm}\}$, where P_{jk} is the synchronization requirement of communication k up to the end of `j`. Assuming then that the resulting `PENDING_IN(i)` = $\{P_1, P_2, ..., P_m\}$, each $P_l \in$ `PENDING_IN(i)` can be computed as $P_l = \vee_c P_{jl}$ where $\vee_c$ is performed over the j values. We note that this algorithm is more accurate and faster than those proposed by others [19, 29, 30]; the approach presented by Gupta and Schonberg [19] converges within three iterations, whereas Hinrichs does not provide a bound for her approach [29, 30].

Claim: *The data-flow procedure defined by equations given in Figure 5(d) can reach a steady state in at most after two iterations.*

Proof:

Consider an arbitrary statement i in the interval which needs synchronization (p, q). Assume that this can be eliminated by a communication (p, q) in any statement t_j by condition (6). There are two possibilities.

Case 1. $j > i$: *Since we are doing a forward pass over the interval, we would visit node j sometime in the first iteration itself and eliminate (p, q)*

Case 2. $j < i$: *We will miss node j in the first iteration, but surely we will visit it in the second iteration.* $\square$

Example The upper part of Table 1 shows the `SendSet` and `Pending` sets (in an open form rather than in terms of equalities and inequalities) for the program shown in Figure 7(c). Note that `SendSet(1)` and `Pending(1)` are computed from the `SendSet` and `Pending` sets, respectively given in Figure 5(a). The sets for 2 and 3 are computed similarly. Before the data-flow analysis starts, `PENDING_IN(1)` is initialized as follows:

$$\begin{aligned}\texttt{PENDING\_IN}(1) &= \{\texttt{Pending}(1), \texttt{Pending}(2), \texttt{Pending}(3)\}\\ &= \{\{(0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,7)\},\{(1,0),(2,1),(3,2)\},\{(4,3),(5,4),(6,5),(7,6)\}\}.\end{aligned}$$

The lower part of Table 1 demonstrates application of our data-flow algorithm to this example. After the fixed state is reached, an examination of `PENDING_OUT(3)` reveals that the program can be executed without any synchronization. We note that many frequently occurring computation patterns such as stencil computations, reduction and prefix operations that involve reciprocal producer-consumer relationships between processors can benefit significantly from synchronization elimination.

6.2 Eliminating Synch messages using other Synch messages

In some cases, when all communication is in the same direction, the `Synch` messages cannot be eliminated by data messages. However, they can still be eliminated by other `Synch` messages. In our experiments, this happened with the `stfrg` benchmark whose message-vectorized version contains six similar communications in the same direction. The algorithm that we use for eliminating `Synch` messages with `Synch` messages is rather straightforward. We check all the possible subsets of `Synch` messages and determine how many `Synch` messages are eliminated if we use that subset. We then choose the subset that eliminates the most number of messages. In the case of a tie, we choose the alternative with the least number of elements, in an attempt to reduce the code complexity. Note that although this approach requires a kind of exhaustive search (i.e., it is exponential in number of `Pending` sets), since a large portion of the `Synch` messages will be eliminated by global communication analysis, the cost is manageable.

7 Experiments

In order to examine the applicability and usefulness of the optimization framework presented in this paper, we applied it to a number of floating-point codes. The application codes used in our study and their important characteristics are listed in Table 2. Most of the codes are from standard benchmarks such as Spec and Perfect Club and run with standard inputs. We experimented with `BLOCK`, `B-CYC` (block-cyclic with $\mathcal{C} = 4$) and `CYCLIC` distributions on 8 processors to measure the static improvements in communication volume, number of data messages, and number of synchronization messages as well as the runtime (dynamic) im-

Table 1: Data-flow sets and PENDING_OUT sets for the example in Figure 7(c).

communication in	SendSet	Pending
1	{(1,0),(2,1),(3,2),(4,3),(5,4),(6,5),(7,6)}	{(0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,7)}
2	{(0,1),(1,2),(2,3)}	{(1,0),(2,1),(3,2)}
3	{(3,4),(4,5),(5,6),(6,7)}	{(4,3),(5,4),(6,5),(7,6)}

PENDING_OUT for	iteration 1	iteration 2
1	$\{\{(0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,7)\},\{\emptyset\},\{\emptyset\}\}$	$\{\{\emptyset\},\{\emptyset\},\{\emptyset\}\}$
2	$\{\{(3,4),(4,5),(5,6),(6,7)\},\{\emptyset\},\{\emptyset\}\}$	$\{\{\emptyset\},\{\emptyset\},\{\emptyset\}\}$
3	$\{\{\emptyset\},\{\emptyset\},\{\emptyset\}\}$	$\{\{\emptyset\},\{\emptyset\},\{\emptyset\}\}$

provements in execution times. We also investigate the scalability of our approach using different processor sizes.

Our experimental platform is a 256-node Cray T3E multiprocessor [47]. The Cray T3E is a scalable shared-memory multiprocessor based on the DEC Alpha 21164 microprocessors running at 300 MHz. This machine includes a number of novel architectural features designed to tolerate latency, enhance scalability, and deliver high performance. It provides a shared address space over a 3D torus interconnect. There are two levels of caching on a DEC Alpha 21164 chip: 8 KB L1 instruction and data caches and a unified, three-way associative, 96 KB L2 cache. The on-chip caches are kept coherent with local memory through an external back-map. I/O is performed through the GigaRing channel, with sustainable bandwidths of 267MB/per second input and output for every four processors. The Cray T3E network latency for the MPI library is 14 microseconds and the sustainable bandwidth is 260 MB/sec. The programs written using HPF-style data alignment and distribution directives are translated automatically using Parafrase-2 [43] and the resulting codes are then compiled using the native compiler on the Cray T3E. Parafrase-2 is a parallelizing compiler implemented as a source-to-source code re-structurer that consists of several passes for analysis, transformation, parallelism detection, and code generation. Our framework is implemented as a separate pass in Parafrase-2. In order to obtain the loops that enumerate the elements in the ownership, producer, consumer, and communication sets, we use the Omega library [34]. Currently, our implementation works on a single procedure at a time and does not use any inter-procedural analysis [22, 23]. In our experiments, we measured that on the average approximately 41% of the total compilation time is spent in our global communication and synchronization approach. However, it should be mentioned that nearly 24% of the compilation time is spent in the Omega library itself. We also compared the compilation time taken by our Omega-based approach with that of an approach based on processor-tagged descriptors (PTDs) [49], 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. Our results show that using the Omega library instead of an RSD-based representation scheme increases compilation time by approximately 16%. Given the fact that with the Omega library we are able to keep communication sets much more accurately and able to compile the general block-cyclic distributions, we believe that this additional compile time overhead is bearable.

The experiments to be presented shortly investigate two main issues. First, we would like to see how one-way communication performs against the more traditional two-way send/receive type of communication. In order to measure this, we first compiled and ran two versions of each code in our suite on the Cray T3E machine [47]. The send/receive version uses the MPI message-passing library [41] whereas the one-way version uses the SHMEM library [47]. In both the versions, we employed *message-vectorization* as the only communication optimization. In the one-way version both communication messages and synchronization messages are vectorized. With the SHMEM library, the data sending involves only one processor in that the source processor simply puts that data into the memory of the destination processor. Likewise, a processor can read data from another processor's memory without interrupting the latter. The target processor is not made aware that its memory has been read or written. In our framework, the compiler provides the necessary synchronization. As mentioned earlier, MPI is implemented on the T3E using SHMEM calls, so intuitively the one-way version should incur less overhead. In the best possible scenario, every MPI_SEND - MPI_RECV pair can be replaced by a SHMEM_PUT call, thereby reducing the number of messages by half. In reality, however, the one-way communication version needs synchronization messages to ensure correct execution (Note that `Synch` calls are also implemented using SHMEM_PUT calls). In the experiments, using one-way communication, we were able to reduce the total number of messages by 11% to 50% as compared to the send/receive version. The execution times (in *seconds*) for the message-passing (send/receive) and one-way communication versions are presented in Table 3. This table also gives the percentage reduction (improvement) in execution times obtained using one-way communication instead of more conventional two-way send/receive type of message-passing. We see that on the average we get 13.4%, 12.7%, and 9.8% improvements for the `BLOCK`, `B-CYC`, and `CYCLIC` distributions, respectively. These results show that the one-way communication scheme is able to bring decent improvements over the classical send/receive message-passing paradigm on the Cray T3E. Of course, whether similar results can be obtained on other architectures depends largely on how efficiently one-way and two-way communications are implemented in those architectures.

The second issue that we investigate is the performance of our scheme to eliminate redundant communication and synchronization messages, with respect to a (relatively) unoptimized one-way communication framework that uses message-vectorization alone (the execution time results for this framework are given above). First, Table 4 shows the communication volume (total number of elements transferred between processors) and number of data messages for the *message-vectorized version* of one-way communication scheme. Note that the execution times for this version are given in Table 3. From now on we refer to this version as `base`. This is the one-way communication version optimized using message-vectorization alone. We use this message-vectorized version as the base version, because, today, almost every compiler for data-parallel languages uses some form of message-vectorization. Table 5 shows the percentage improvements obtained by using our framework that minimizes the number of data messages and communication volume (i.e., the framework presented in Section 5) over the `base` version. These results are promising and indicate that on the average we have a 26.8% reduction in communication volume and a 25.8% reduction in number of data messages. These improvements reflect on execution times as a 19.7% reduction, considering all three types of data distributions experimented with. Although not presented here, applying two additional communication optimizations (message coalescing [31] and message aggregation [31]) brought only an additional 6.1% improvement in execution time (of the message-passing send/receive version) for the benchmarks in our experimental suite.

It is also useful to compare the results of this globally optimized code with the results of a send/receive communication framework that also uses global optimizations. In order to achieve this, we generated two different send/receive codes for each benchmark using the *same* data communication optimization algorithm (of Section 5): a version that uses the Omega library (called `version-1`) and another version that uses a variant of section descriptors [49] to represent communication and ownership sets (called `version-2`). Note that `version-1` is different from our optimized code as it does not contain explicit synchronization messages and the synchronization activity is captured in the data messages (i.e., it uses MPI). Therefore, the data messages are more costly in `version-1`. Also note that `version-1` is more powerful than many of the previous techniques proposed for optimizing communication globally as it uses the Omega library to determine the communication sets more accurately, uses global data-flow analysis to optimize the inter-nest communication, and can handle block-cyclic distributions. Table 6 presents the percentage improvements provided by our optimized codes over the codes generated by `version-1` and `version-2`. The table shows that our optimized version improves performance by 13.2% over `version-1`. This says that, even with comparable communication optimizations, the one-way communication paradigm might be the choice of communication model in Cray T3E. It should be noted, however, that these results may change if we move to a different platform, because they are largely dependent on how these two communication mechanisms are implemented on a given platform and whether there is low-level support in the architecture for one-way communication. Compared to `version-2`, our approach brings even more improvement (19.7% on average). This is due to the fact that the processor-tagged descriptors (PTDs) (an enhanced form of RSDs) used in `version-2` cannot provide the same accuracy (in obtaining the communication sets and hoisting the communications) as our polyhedral tool.

Having shown that our approach is able to reduce execution times significantly as compared to a message-vectorized version and as compared to a globally-optimized send/receive version, we now turn into the problem of minimizing the number of *synchronization messages* and impact of this in the overall execution times. The left part of Table 7 presents the reductions in the number of synchronization messages obtained through our redundant synchronization elimination technique (i.e., the framework presented in Section 6) over our 'one-way' communication optimization approach whose improvements are given in Table 5. The results show that we have approximately 45% improvement (reduction in the number of synchronization messages) on the average. The impact of these improvements in the overall execution times is presented on the right part of Table 7. We see that we have an additional 14.5% reduction in execution times. These results reveal that our synchronization elimination technique is very successful in practice and brings additional improvements over a technique that optimizes only data messages. As a side note, if we use barrier synchronizations instead of point-to-point synchronizations in our framework, the overall improvements in execution times are 9.1%, 8.5%, and 8.3% for `BLOCK`, `B-CYC`, and `CYCLIC` distributions, respectively. This indicates that using point-to-point synchronization messages instead of barrier synchronization allows small differences in execution times of individual processors to even out and results in better overall execution times.

Finally, we consider the effect of our approach on the scalability of the codes in our suite. For this purpose, we use different processor sizes on the Cray T3E machine. The results, given in Table 8, are the *average values* over the following data distributions: `BLOCK`, `CYCLIC(1)`, `CYCLIC(4)`, `CYCLIC(7)`, and

`CYCLIC(16)`. For each code and processor-size combination, we present two speedups: that of the `base` version and that of our optimized version (denoted `opt`). Note that the speedups are measured over the best sequential version of each code. The results illustrate that the codes in our suite significantly benefit from our optimizations and our framework is robust across different types of data distributions. In most cases, our approach is able to first reduce the number of data messages and then reduce the number of synchronization messages, thereby improving the overall scalability of applications.

8 Related work

Several methods have been presented for optimizing communication on distributed-memory message-passing machines. Most of the efforts have considered communication optimization at the loop (or array assignment statement) level. Although each approach has its own unique features, the general idea is to apply an appropriate combination of message vectorization, message coalescing and message aggregation [7, 31, 57].

More recently some researchers have proposed techniques based on data-flow analysis in order to optimize communication across multiple loop nests for the two-way (send/receive) communication model. Several works [1, 2, 3, 10, 16, 20, 36, 55, 56] present similar frameworks to optimize the send/receive communications globally. Almost all these approaches (except for the work of Adve et al. [1, 2]) use a variant of Regular Section Descriptors (RSD) introduced by Callahan and Kennedy [9]. For each array referenced in the program, an RSD is defined which describes the portion of the array that is referenced. Although this representation is convenient for simple array sections such as those found in pure block or cyclic distributions, it is hard to embed alignment and general distribution information into it. Apart from inadequate support for block-cyclic distributions, working with section descriptors may result in overestimation of the communication sets. The inaccuracy resulting from these estimations may be linear with the number of data-flow formulations to be evaluated, thus defeating the purpose of global communication optimization.

For example, Gupta et al. [20] present a framework to optimize two-way 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. Similarly, Hanxleden and Kennedy [24, 25] 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 [35] offer a global data-flow analysis technique using bit vectors. Although this approach is efficient, it is not as precise as the approach presented in this paper. They do not give any information about how their method can be extended to handle general type block-cyclic distributions. Kennedy and Sethi [36] show the necessity of incorporating resource constraints into a global communication optimization framework. Their approach takes constraints such as limited buffer size and uses strip-mining where necessary to improve communication placement. Although their approach works with multiple nests, it does not handle general block-cyclic distributions. Since they do not give any experimental results, a direct quantitative comparison of this work with ours is not possible. They do not use a linear algebra framework; later work from the dHPF project at Rice [1, 2] makes use of the Omega library for message optimizations. The IBM pHPF compiler [10, 21] achieves both redundancy elimination and message combining globally. But message combining is feasible only if the messages have identical patterns, or one pattern is a subset of another. The general block-cyclic distributions, however, can lead to complicated data access patterns and communication sets which, we believe, more precisely can be represented within a linear algebra framework. Yuan, Gupta, and Melhem [55, 56] present a communication optimization approach based on array data-flow analysis. The cost of the analysis is managed by partitioning the optimization problem into subproblems, and solving the subproblems one at a time. Since that approach is also based on RSDs, it has difficulty in handling block-cyclic distributions. Adve et al. [1, 2] describe an integer set based approach for analysis and code generation for data parallel programs that uses the Omega library [34]. 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. Note that non of these techniques considers optimizing synchronization separately. Our results in this paper show that separately optimizing data and synchronization messages in a one-way communication framework may bring additional improvements on a machine like Cray T3E. Our approach is based on a linear algebra framework, and can represent all HPF-like alignment and distribution information accurately. It should be emphasized that the *data communication* optimization part of our approach can also be used for optimizing send/receive type of communication on distributed memory machines [32, 33].

Duesterwald et al. [14] present a data-flow framework for array reference analysis that provides information which can be used in communication optimization. The main limitations of their approach are that sometimes they have to make conservative assumptions in performing *kill analysis* and that in order to handle multidimensional arrays they may first need to linearize them; this can affect subsequent compiler analyses severely. Also, they have not extended their techniques to handle coupled array subscripts as in $A(i+j, i-j)$ where both i and j are loop index variables.

The approaches of Gupta and Schonberg [19] and Hinrichs [29, 30] examine the problem of eliminating redundant synchronization operations by piggy-backing them on data messages. While Hincrichs [29, 30] uses barrier synchronizations, Gupta and Schonberg [19] focus on point-to-point synchronization messages. Gupta and Schonberg's technique [19] as well as Hinrichs' solution [29, 30] work with alignment information (virtual processors), and take distribution into account only for simple distributions such as pure block distributions. Their techniques do not handle general block–cyclic distributions.

Subhlok [50] addresses the problem of synchronization in Fortran programs with parallel loop constructs. His approach determines whether the current synchronization points in the program are sufficient. Instead, we employ a communication analysis that can aggressively optimize synchronization even in shared memory systems.

Hayashi et al. [26] and Stricker et al. [48] discuss separation of data transmission and synchronization and give reasons for that. Our approach presented in this paper supports their claims.

O'Boyle and Bodin [42] and Tseng [52] focus on synchronization elimination problem on shared virtual memory and distributed shared memory systems, respectively. There is an important difference between our work and theirs. They eliminate synchronizations which are introduced by the insufficient communication analysis performed by the shared memory compilers. A compiler approach based on distributed-memory paradigm (like ours) does not insert those synchronizations in the first place. In our case, we start with an unoptimized program in which those types of artificial synchronizations are non-existent to begin with. We rather focus on elimination of synchronizations that are caused by *mandatory data communications.* Such types of synchronizations are not eliminated by using either O'Boyle and Bodin's solution [42] or Tseng's approach [52], although Tseng's solution [52] replaces barriers with software-implemented counters in certain cases leading to improvements in performance.

9 Summary

Minimizing communication and synchronization costs is crucial to realizing the performance potential of parallel machines. In this paper, we have presented an approach based on data-flow analysis combined with a linear algebra framework to minimize synchronization and data transfer costs. We have presented data-flow algorithms to reduce the number of data messages, communication volume, and number of synchronization messages in a communication generation framework based on one-way communication using `Put/Synch` primitives. The reduction in these parameters has resulted in substantial savings in execution times on the Cray T3E multiprocessor machine. Our approach argues for separation of data transfer and synchronization and for optimization each of them using data-flow analysis techniques. Several machines such as the Cray T3D [12], the Cray T3E [47], the Fujitsu AP1000+ [26], and the Meiko CS-2 [8] offer remote memory access primitives that allow efficient implementation of the `Put/Synch` primitives. In addition, one-way communication is a key part of the proposed extensions to the Message Passing Interface standard [41]. Therefore, our solution to the problem of minimizing synchronization and data transfer costs will be useful in practice.

The data-flow algorithms we have presented represent the communication requirement exactly with Presburger formulae using the Omega library [34]. As a result, our analysis is more powerful and more accurate than many other methods proposed previously. Experimental results reveal that our approach is quite successful in practice, leading to a significant reduction in the number of data and synchronization messages for message-vectorized programs. These results are on standard benchmarks from scientific computing codes. We plan to develop techniques for compiling data-parallel programs with `Get` primitives. In addition, we will focus on the elimination of synchronization and potential deadlocks in programs compiled under hybrid approaches that employ both `Put/Get` and `Send/Recv` primitives.

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.

Alok Choudhary was supported in part by NSF Young Investigator Award CCR-9357840 and NSF grant CCR-9509143. The work of Prith Banerjee was supported in part by NSF under grant CCR-9526325 and in

Table 2: Programs in our experiment set and their important characteristics. In the last three entries the problem sizes (in bytes) are given in parentheses after the program name; for the other programs the standard input sizes are used.

Program	Source	Lines	Description
2Dhydro	Livermore	38	2D Hydrodynamics
adi	Livermore	30	ADI Fragment
stfrg	Livermore	17	State Fragment Equation
nasa7	Spec92	1,105	Nasa Ames Fortran Kernels
fpppp	Spec92	2,718	2 Electron Integral Derivative
su2cor	Spec92	2,414	Quantum Physics
tomcatv	Spec95	190	2D Mesh Generation
swim256	Spec95	428	Shallow Water Model
tis	Perfect Club	485	Electron Integral Transform
lws	Perfect Club	1,217	Molecular Dynamics of Water
srs	Perfect Club	3,786	2D Fluid Flow Solver
sms	Perfect Club	2,712	Depth Migration
shallow(512)	-	167	Shallow Water Model
jacobi(1K)	-	25	Jacobi Solver
sor(256)	-	25	Successive Over-Relaxation

Table 3: Execution times (in *seconds*) of message-vectorized `send/recv` and message-vectorized `one-way` (put) communication versions and percentage (%) improvements (the `imprv` column).

Program	BLOCK			B-CYC			CYCLIC		
	send/recv	one-way	imprv	send/recv	one-way	imprv	send/recv	one-way	imprv
2Dhydro	4.12	3.71	10.0	5.09	4.80	5.7	4.75	4.46	6.1
adi	0.58	0.53	8.6	0.64	0.61	4.7	0.79	0.73	9.2
stfrg	0.27	0.23	14.8	0.41	0.35	14.6	0.39	0.37	5.1
nasa7	20.18	14.35	28.9	24.03	17.00	29.3	22.58	15.95	29.4
fpppp	40.02	36.24	9.4	45.50	39.18	13.9	44.98	39.04	13.2
su2cor	37.21	33.01	11.2	42.10	36.28	13.8	45.06	36.92	18.1
tomcatv	1.39	1.16	16.5	2.39	1.68	29.7	3.20	2.08	35.0
swim256	11.31	7.13	36.9	21.44	16.03	25.2	36.55	30.36	16.9
tis	11.57	11.28	2.5	19.80	20.04	-1.2	20.46	20.38	0.4
lws	25.06	20.13	19.7	32.13	25.86	19.5	32.72	26.14	20.1
srs	50.82	46.05	9.4	55.34	49.92	9.8	52.76	51.00	3.3
sms	43.14	42.05	2.5	50.40	48.96	2.9	51.17	48.35	5.5
shallow(512)	9.67	7.07	26.9	17.95	13.98	22.1	30.64	24.72	19.3
jacobi(1K)	1.28	1.26	1.6	1.65	1.69	-2.4	1.72	1.77	-2.9
sor(256)	3.27	3.21	1.8	5.15	5.02	2.5	5.04	5.30	-5.2
Average:			13.4			12.7			9.8

Table 4: Communication volume and number of messages for the message-vectorized one-way communication (base version).

Program	Communication Volume			Number of Messages		
	BLOCK	B-CYC	CYCLIC	BLOCK	B-CYC	CYCLIC
2Dhydro	57.1	1,036.3	4,169.8	112	128	128
adi	10.7	96.8	387.1	42	48	48
stfrg	0.2	4.6	6.1	42	64	48
nasa7	359.0	4,822.3	20,621.0	728	1,216	1,018
fpppp	716.4	7,428.1	38,624.4	1,224	1,896	1,790
su2cor	700.4	7,200.8	36,955.2	1,098	1,604	1,638
tomcatv	57.1	1,040.0	8,323.2	112	128	128
swim256	57.6	914.4	3,649.1	128	142	142
tis	56.9	872.0	872.0	112	178	186
lws	400.2	6,100.8	24,130.6	910	1,604	1,724
srs	811.0	8,295.8	38,128.0	1,630	2,024	2,024
sms	794.7	8,005.1	34,301.6	1,438	2,248	2,428
shallow(512)	53.9	900.5	3,586.2	128	142	142
jacobi(1K)	14.4	522.2	2,088.9	14	16	16
sor(256)	21.3	193.5	774.2	84	96	96

Table 5: Improvements (%) on communication volume, number of *communication* messages, and execution times.

Program	Communication Volume			Number of Messages			Execution Times		
	BLOCK	B-CYC	CYCLIC	BLOCK	B-CYC	CYCLIC	BLOCK	B-CYC	CYCLIC
2Dhydro	50.0	50.0	50.0	50.0	51/0	50.0	31.8	32.1	31.8
adi	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
stfrg	71.0	64.1	0.0	83.2	74.2	0.0	62.1	50.5	0.0
nasa7	57.0	60.8	60.8	27.4	28.2	28.2	44.6	48.1	48.1
fpppp	16.6	12.8	12.7	10.4	9.8	9.5	9.9	8.0	7.8
su2cor	18.2	13.5	11.3	19.3	16.0	14.2	16.0	11.7	10.1
tomcatv	74.2	74.2	75.0	74.2	74.0	75.0	59.3	59.2	59.6
swim256	37.4	41.3	42.9	44.2	41.7	41.7	28.5	28.5	28.4
tis	12.6	20.4	19.0	18.4	20.4	18.9	8.7	9.5	9.3
lws	7.8	13.7	13.8	7.0	13.2	13.7	7.0	11.4	11.9
srs	16.5	15.9	15.5	20.1	23.4	25.7	12.5	11.8	11.9
sms	16.8	16.3	16.3	16.3	16.0	16.0	12.0	10.0	9.9
shallow(512)	39.3	44.6	47.3	46.3	42.5	42.5	29.2	29.2	29.0
jacobi(1K)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
sor(256)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Average:	27.8	28.5	24.3	27.7	27.4	22.4	21.4	20.6	17.2

Table 6: Improvements (%) in execution times over two globally optimized send/receive versions.

Program	version-1			version-2		
	BLOCK	B-CYC	CYCLIC	BLOCK	B-CYC	CYCLIC
2Dhydro	11.3	6.4	6.0	11.0	10.4	9.9
adi	8.2	4.3	8.6	16.5	10.7	16.6
stfrg	16.0	15.3	9.3	30.3	28.8	21.5
nasa7	30.2	31.7	32.5	42.3	44.0	47.6
fpppp	8.2	14.0	13.1	13.4	20.2	22.5
su2cor	11.0	13.4	16.5	16.3	15.2	18.7
tomcatv	21.3	34.3	39.0	29.9	45.0	49.7
swim256	34.8	21.2	17.0	49.3	33.7	28.1
tis	1.2	-0.3	0.5	1.3	-0.4	0.2
lws	20.3	21.5	21.6	28.2	30.0	32.1
srs	10.1	12.2	5.8	10.7	12.9	6.6
sms	1.4	2.0	3.8	2.1	4.5	6.2
shallow(512)	27.2	24.3	19.7	38.4	33.0	29.3
jacobi(1K)	1.0	-3.0	-3.1	3.5	-1.0	-1.0
sor(256)	3.4	1.9	-1.4	3.9	1.9	-2.2
Average:	13.7	13.3	12.6	20.5	19.3	19.1

Table 7: Improvements (%) in the number of *synchronization* messages and execution times.

Program	Number of Messages			Execution Times		
	BLOCK	B-CYC	CYCLIC	BLOCK	B-CYC	CYCLIC
2Dhydro	100.0	100.0	100.0	18.4	19.0	18.4
adi	100.0	100.0	100.0	25.2	20.8	19.0
stfrg	0.0	0.0	0.0	0.0	0.0	0.0
nasa7	33.6	31.5	31.5	19.4	19.3	19.3
fpppp	10.5	12.2	12.2	8.2	8.0	8.0
su2cor	10.9	11.4	11.4	9.9	10.5	10.5
tomcatv	100.0	100.0	100.0	20.6	21.2	21.0
swim256	89.2	89.2	89.2	30.3	31.6	30.9
tis	8.9	9.0	9.0	5.5	5.6	5.5
lws	10.6	10.6	10.6	4.2	4.0	4.2
srs	12.5	11.7	11.7	10.2	8.7	8.2
sms	12.1	12.4	12.4	10.1	10.3	11.9
shallow(512)	90.4	90.4	90.4	31.7	30.9	32.0
jacobi(1K)	0.0	0.0	0.0	0.0	0.0	0.0
sor(256)	100.0	100.0	100.0	23.8	26.9	28.7
Average:	45.2	45.2	44.6	14.5	14.4	14.5

Table 8: Speedups for the message-vectorized (base) version and our version.

Program	Number of Processors							
	4		8		16		32	
	base	opt	base	opt	base	opt	base	opt
2Dhydro	3.0	3.5	6.0	7.0	10.0	12.7	14.8	23.8
adi	2.5	3.6	5.1	7.2	8.6	13.0	10.6	19.5
stfrg	2.8	3.4	5.5	7.2	8.9	13.4	9.9	20.5
nasa7	2.9	3.5	5.7	7.0	7.0	11.9	6.2	19.7
fpppp	2.9	3.7	4.0	7.5	7.2	14.0	7.0	19.5
su2cor	1.9	2.9	3.1	6.8	7.0	12.5	8.0	23.0
tomcatv	2.9	3.7	4.1	7.4	3.9	13.5	3.6	23.8
swim256	2.5	3.7	5.9	7.5	9.0	14.1	10.1	22.6
tis	1.8	2.8	3.2	6.8	5.8	12.5	5.4	16.9
lws	2.0	3.5	5.3	7.6	6.5	13.3	5.3	20.1
srs	3.2	3.8	4.8	7.5	4.4	14.2	4.0	20.2
sms	2.8	3.5	6.4	7.2	5.8	13.5	5.3	23.8
shallow(512)	3.1	3.8	6.1	7.0	9.6	12.9	5.2	20.2
jacobi(1K)	3.2	3.8	6.4	7.0	6.0	13.3	12.4	23.0
sor(256)	2.3	3.5	5.4	6.9	5.1	12.4	4.7	14.8
Average:	2.6	3.5	5.1	7.2	6.9	13.1	7.4	20.8

part by DARPA under contract DABT-63-97-C-0035. J. Ramanujam was supported in part by NSF Young Investigator Award CCR-9457768 and NSF grant CCR-9210422.

References

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

[2] V. Adve, and J. Mellor-Crummey. Advanced code generation for High Performance Fortran. In *Languages, Compilation techniques, and Run-time Systems for Scalable Parallel Systems,* Chapter 18, Lecture Notes in Computer Science Series, Springer-Verlag, 2001 (to appear).

[3] G. Agrawal and J. Saltz. Interprocedural data flow based optimizations for distributed memory compilation. *Software Practice and Experience,* 27(5): 519–545, 1997.

[4] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, techniques, and tools.* Addison-Wesley, Reading, MA, Second edition, 1986.

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

[6] A. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. *Scientific Programming,* 6(1):3–28, Spring 1997.

[7] P. Banerjee, J. Chandy, M. Gupta, E. Hodges, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. The PARADIGM compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.

[8] E. Barton, J. Cownie, and M. McLaren. Message passing on the Meiko CS-2. *Parallel Computing,* 20(4): 497–507, April 1994.

[9] D. Callahan, and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing,* 5(5):517–550, October 1988.

[10] S. Chakrabarti, M. Gupta, and J.-D. Choi. Global communication analysis and optimization. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation,* pages 68–78, Philadelphia, PA, May 1996.

[11] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming,* 1(1):31–50, Fall 1992.

[12] Cray Research Inc. Cray T3D system architecture overview. 1993.

[13] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. *Proc. Supercomputing'93,* Portland, OR, November 1993.

[14] E. Duesterwald, R. Gupta, and M. L. Soffa. A practical data-flow framework for array reference analysis and its application in optimizations. In Proc. *ACM SIGPLAN'93 Conference on Programming Language Design and Implementation,* pages 68–77, Albuquerque, New Mexico, June 1993.

[15] A. Geist. PVM : Parallel Virtual Machine : A users' guide and tutorial for networked parallel computing. Scientific and Engineering Computation Series, 1994.

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

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

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

[19] M. Gupta, and E. Schonberg. Static analysis to reduce synchronization costs in data-parallel programs. In *Proc. ACM Conference on Principles of Programming Languages*, pages 322–332, St. Petersburg, FL, 1996.

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

[21] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Proc. Supercomputing 95,* San Diego, CA, December 1995.

[22] M. W. Hall, S. Hiranandani, K. Kennedy, and C.-W. Tseng. Inter-procedural compilation of Fortran D for MIMD distributed-memory machines. In *Proc. Supercomputing'92*, Minneapolis, MN, November 1992

[23] M. W. Hall, B. Murphy, S. Amarasinghe, S. Liao, and M. Lam. Inter-procedural analysis for parallelization. In *Proc. 8th International Workshop on Languages and Compilers for Parallel Computers*, pages 61–80, Columbus, Ohio, August 1995.

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

[25] R. v. Hanxleden, and K. Kennedy. GIVE-N-TAKE – a balanced code placement framework. In *Proc. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation,* Orlando, FL, June 1994.

[26] K. Hayashi, T. Doi, T. Horie, Y. Koyanagi, O. Shiraki, N. Imamura, T. Shimizu, H. Ishihata, and T. Shindo. AP1000+: Architectural support of put/get interface for parallelizing compiler. In *Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems,* San Jose, CA, pages 196–207, October 1994.

[27] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integrating message passing in the Stanford FLASH multiprocessor. In *Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems,* San Jose, CA, October 1994.

[28] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming,* 2(1-2):1–170, 1993.

[29] S. Hinrichs. Compiler-directed architecture-dependent communication optimizations. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1995.

[30] S. Hinrichs. Synchronization elimination in the deposit model. In *Proc. 1996 International Conference on Parallel Processing,* pages 87–94, St. Charles, IL, 1996.

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

[32] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy. A generalized framework for global communication optimization. In Proc. *International Parallel Processing Symposium (IPPS'98),* pages 69–73, March 1998.

[33] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy. A global communication optimization technique based on data flow analysis and linear algebra. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 21(6):1251–1297, November 1999.

[34] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and David Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, Department of Computer Science, University of Maryland, College Park, March 1995.

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

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

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

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

[39] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.H. Lim. Integrating message passing and shared memory: Early experience. In *Proc. 4th ACM Symposium on Principles and Practice of Parallel programming (PPOPP'93)*, San Diego, CA, May 1993.

[40] U. Kremer. Automatic data layout for distributed memory machines. Ph.D. Thesis. Technical Report CRPC-TR95559-S, Center for Research on Parallel Computation, Rice University, Houston, TX, October 1995.

[41] Message Passing Interface Forum. *MPI-2: Extensions to the message-passing interface.* July 1997.

[42] M. O'Boyle, and F. Bodin. Compiler reduction of synchronization in shared virtual memory systems. In *Proc. International Conference on Supercomputing*, pages 318–327, Barcelona, Spain, July 1995.

[43] C. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten. Parafrase-2: an environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Proc. the International Conference on Parallel Processing*, St. Charles IL, August 1989, pages II 39–48.

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

[45] S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: User-level shared memory. In *Proc. 21th International Symposium on Computer Architecture,* pages 325–336, April 1994.

[46] A. Rogers, and K. Pingali. Process decomposition through locality of reference. In *Proc. SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 69–80, Portland, OR, June 1989.

[47] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In Proc. *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96),* pages 26–36, October 1996.

[48] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. Decoupling synchronization and data transfer in message passing systems of parallel computers. In *Proc. 9th ACM International Conference on Supercomputing*, Barcelona, Spain, pages 1–10, July 1995.

[49] E. Su. *Compiler framework for distributed memory multicomputers.* Ph.D. thesis, University of Illinois at Urbana-Champaign, March 1997.

[50] J. Subhlok. *Analysis of synchronization in a parallel programming environment.* Ph.D. thesis, Rice University, 1990.

[51] C. Thekkath, H. Levy, and E. Lazowska. Separating data and control transfer in distributed operating systems. In *Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems,* San Jose, CA, pages 1–12, October 1994.

[52] C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proc. the 5th ACM Symposium on Principles and Practice of Parallel programming (PPOPP'95)*, Santa Barbara, CA, July 1995.

[53] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proc. 19th International Symposium on Computer Architecture,* pages 256–266, May 1992.

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

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

[56] X. Yuan, R. Gupta, and R. Melhem. Demand-driven data flow analysis for communication optimization. *Parallel Processing Letters,* 7(4):359–370, December 1997.

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

AUTHOR BIOGRAPHIES

Mahmut Kandemir

Mahmut Kandemir received the B.Sc. and M.Sc. degrees in control and computer engineering from Istanbul Technical University, Istanbul, Turkey, in 1988 and 1992, respectively. He received the Ph.D. from Syracuse University, Syracuse, New York in electrical engineering and computer science, in 1999. He has been an assistant professor in the Computer Science and Engineering Department at the Pennsylvania State University since August 1999. His main research interests are optimizing compilers, I/O intensive applications, and power-aware computing. He is a member of the IEEE and the ACM.

Alok Choudhary

Alok Choudhary received his Ph.D. from University of Illinois, Urbana-Champaign, in Electrical and Computer Engineering, in 1989, M.S. from University of Massachusetts, Amherst, in 1986 and B.E.(Hons.) from Birla Institute of Technology and Science, Pilani, India in 1982.

He is a professor of Electrical and Computer Engineering at Northwestern University. From 1993 to 1996 he was an associate professor in the ECE department at Syracuse University and from 1989 to 1993 he was an assistant professor in the same department. He has worked in industry for computer consultants prior to 1984.

Alok Choudhary received the National Science Foundation's Young Investigator Award in 1993 (1993-1999). He has also received an IEEE Engineering Foundation award, an IBM Faculty Development award and an Intel Research Council award.

His main research interests are in high-performance computing and communication systems and their applications in many domains including multimedia systems, information processing and scientific computing. In particular, his interests lie in the design and evaluation of architectures and software systems (from system software such as runtime systems, compilers and programming languages to applications), high-performance servers, high-performance databases and input-output.

He has published more than 130 papers in various journals and conferences in the above areas. He has also written a book and several book chapters on the above topics.

His research has been sponsored by (past and present) DARPA, NSF, NASA, AFOSR, ONR, DOE, Intel, IBM, and TI.

Alok Choudhary served as the conference co-chair for the International Conference on Parallel Processing and as a program chair and general chair for the International Workshop on I/O Systems in Parallel and Distributed Systems. He also served a program vice-chair for HiPC'1999. He is an editor of the Journal of Parallel and Distributed Computing, and an associate editor of IEEE Transactions on parallel and Distributed Systems. He has also served as a guest editor for IEEE Computer and IEEE Parallel and Distributed Technology. He serves (or has served) on the program committee of many International conferences in architectures, parallel computing, multimedia systems, performance evaluation, distributed computing etc. He is a member of the IEEE Computer Society and the ACM. He also serves in the High-Performance Fortran Forum, a forum of Academia, Industry and Government Labs working on standardizing programming languages for portable programming on parallel computers.

Prithviraj Banerjee

Prithviraj Banerjee received his B.Tech. degree in Electronics and Electrical Engineering from the Indian Institute of Technology, Kharagpur, India, in August 1981, and the M.S. and Ph.D degrees in Electrical Engineering from the University of Illinois at Urbana-Champaign in December 1982 and December 1984 respectively.

Dr. Banerjee is currently the Walter P. Murphy Professor and Chairman of the Department of Electrical and Computer Engineering, and Director of the Center for Parallel and Distributed Computing. at Northwestern University in Evanston, Illinois. Prior to that he was the Director of the Computational Science and Engineering program, and Professor of Electrical and Computer Engineering and the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign.

Dr. Banerjee's research interests are in Parallel Algorithms for VLSI Design Automation, Distributed Memory Parallel Compilers, and Compilers for Adaptive Computing, and is the author of over 270 papers in these areas. He leads the PARADIGM compiler project for compiling programs for distributed memory multicomputers, the ProperCAD project for portable parallel VLSI CAD applications, and the MATCH project on a MATLAB compilation environment for adaptive computing. He is also the author of a book entitled *Parallel Algorithms for VLSI CAD* published by Prentice Hall, Inc., 1994. He has supervised 27

Ph.D. and 30 M.S. student theses so far.

Dr. Banerjee has received numerous awards and honors during his career. He became a Fellow of the ACM in 2000. He was the recipient of the 1996 Frederick Emmons Terman Award of ASEE's Electrical Engineering Division sponsored by Hewlett-Packard. He was elected to the Fellow grade of IEEE in 1995. He received the University Scholar award from the University of Illinois for in 1993, the Senior Xerox Research Award in 1992, the IEEE Senior Membership in 1990, the National Science Foundation's Presidential Young Investigators' Award in 1987, the IBM Young Faculty Development Award in 1986, and the President of India Gold Medal from the Indian Institute of Technology, Kharagpur, in 1981.

Dr. Banerjee has served as the Program Chair of the High-Performance Computing Conference in 1999, and Program Chair of the Int. Conf. on Parallel Processing for 1995. He has served as General Chairman of the International Conference on Parallel and Distributed Computing Systems in 1997, and the International Workshop on Hardware Fault Tolerance in Multiprocessors, 1989. He has served on the Program and Organizing Committees of the 1988, 1989, 1993 and 1996 Fault Tolerant Computing Symposia, the 1992, 1994, 1995, 1996 and 1997 International Parallel Processing Symposium, the 1991, 1992, 1994 and 1998 International Symposia on Computer Architecture, the 1998 International Conference on Architectural Support of Programming Languages and Operating Systems, the 1990, 1993, 1994, 1995, 1996, 1997 and 1998 International Symposium on VLSI Design, the 1994, 1995, 1996, 1997, 1998 and 2000 International Conference on Parallel Processing, and the 1995, 1996 and 1997 International Conference on High-Performance Computing. He is an Associate Editor of the IEEE Transactions on Parallel and Distributed Systems, and IEEE Transactions on Computers. In the past he has served as Associate Editor of the Journal of Parallel and Distributed Computing, the IEEE Transactions on VLSI Systems, and the Journal of Circuits, Systems and Computers. He has been a consultant to many companies and was on the Technical Advisory Board of Ambit Design Systems.

J. Ramanujam

J. Ramanujam (Ram) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras, India, in 1983, and the M.S. and Ph.D. degrees in computer science from The Ohio State University, Columbus, Ohio, in 1987 and 1990, respectively. He is currently an an associate professor of electrical and computer engineering at Louisiana State University, Baton Rouge, Louisiana. His research interests are in compilers for high-performance computer systems, software optimizations for low-power computing, embedded systems, high-level hardware synthesis, parallel architectures and algorithms. He has published over 80 papers in refereed journals and conferences in these areas in addition to several book chapters.

Dr. Ramanujam received the National Science Foundation's Young Investigator Award in 1994. He has served on the Program Committees of several conferences such as the International Conference on Parallel Architectures and Compilation Techniques (PACT 2000), the International Symposium on High Performance Computing (HiPC 99) and 1997 International International Conference on Parallel Processing. He is co-organizing a workshop on *Compilers and Operating Systems for Low Power,* to be held in conjunction with PACT 2000 in October 2000. He has taught tutorials on compilers for high-performance computers at several conferences such as the International Conference on Parallel Processing (1998, 1996), Supercomputing 94, Scalable High-Performance Computing Conference (SHPCC 94), and the International Symposium on Computer Architecture (1993 and 1994). He has been a frequent reviewer for several journals and conferences.

Nagaraj Shenoy

Nagaraj Shenoy received his MS and PhD in com-puter science from the Indian Institute of Science. He is currently a research associate professor in Electrical and Computer Engineering at Northwestern University. Prior to that he was with the Center for Development of Advanced Computing (C-DAC), an Indian national research center. He headed several research groups at C-DAC from 1989-1997. His main research areas include parallel systems, parallel environments and tools, and automatic parallelization for parallel adaptive computing systems. He is a member of the IEEE.