Proceedings of the 1998 International Conference on Parallel Processing (ICPP 98)

Minimizing Data and Synchronization Costs in One-Way Communication

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

Abstract

In contrast to the conventional send/receive model, the one-way communication model—using `Put` and `Synch`—allows the decoupling of message transmission from synchronization. This opens up new opportunities not only to further optimize communication but also to reduce synchronization overhead. In this paper, we present a general technique which uses a global dataflow framework to optimize communication and synchronization in the context of the one-way communication model. Our approach works with the most general data alignments and distributions in languages like HPF, and is more powerful than other current solutions for eliminating redundant synchronization messages. Preliminary results on several scientific benchmarks demonstrate that our approach is successful in minimizing the number of data and synchronization messages.

1 Introduction

Most of the current compilers for distributed memory machines rely on the `send` and `recv` primitives to implement communication. The impact of this approach is twofold. First, this technique 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 synchronization messages to maintain data integrity, separating synchronization messages from data messages may actually improve the performance of some programs. Second, the compiler has the difficult task of matching `send` and `recv` operations in order to guarantee correct execution.

In this paper, we focus on compilation of programs annotated by HPF-like directives with one-way communication operations `Put` and `Synch`, introduced by Gupta and Schonberg [6] and Hinrichs [8]. Let us 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 the data in that buffer. We note that the `Synch` operation is necessary for the repetition of this communication; that is, 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.

After briefly discussing the fundamental concepts used in this paper in Section 2, in Section 3 we show how the communication sets as well as the producers and the consumers manipulated by the `Put` operation can be implemented on top of the existing `send/recv` type of communication framework in our compiler [3]. Having determined those, the next issue is to minimize the number of `Put` communications as well as the communication volume. Section 4 presents an algorithm to achieve this goal. Our algorithm can take arbitrary control flow (excluding *goto* statements) into account and can optimize programs with all types of HPF-like alignments and distributions, including block-cyclic distributions. It is based on a linear algebra framework introduced by Ancourt et al. [2]; 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 associated predicates.

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 inserts 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 always necessary. The answer is no, and Section 5 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 occurring in the program. The basic idea is to use another message in the reverse direction between the same pair of processors in 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 very fast and more powerful than the previous work in synchronization elimination. This is because (1) it is very accurate in eliminating redundant synchronization, since it works at the granularity of a processor-pair using the Omega library [11]; (2) it can eliminate a synchronization message by using several data messages; (3) it handles block, cyclic, and block-cyclic distributions in a unified manner, whereas the previous approaches either work on virtual processor grids only or use an extension of regular section descriptors which are inherently inaccurate; and (4) it is preceded by a global communication optimization algorithm which itself eliminates a lot of synchronization 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 given 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).

In Section 6, we give preliminary results on several benchmark

[*] Elec. Engr. & Comp. Sci. Dept., Syracuse University, Syracuse, NY 13244. e-mail: `mtk@ece.nwu.edu`

[†] Supported in part by NSF Young Investigator Award CCR-9357840 and NSF grant CCR-9509143

[‡] Elec. & Comp. Engr. Dept., Northwestern University, Evanston, IL 60208. e-mail: {`nagaraj,banerjee,choudhar`}`@ece.nwu.edu`

[§] Supported in part by NSF under grant CCR-9526325 and in part by DARPA under contract DABT-63-97-C-0035.

[¶] Elec. & Comp. Engr. Dept., Louisiana State University, Baton Rouge, LA 70803. e-mail: `jxr@ee.lsu.edu`. Supported in part by NSF Young Investigator Award CCR-9457768 and NSF grant CCR-9210422.

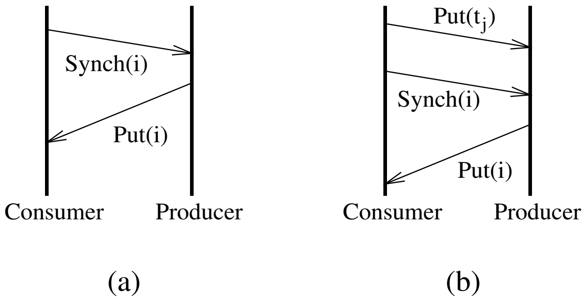

Figure 1: (a) one-way communication with `Put` operation. (b) elimination of a `Synch` message.

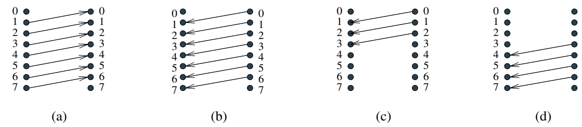

Figure 2: Communication and synchronization messages for the first loop of the program in Figure 6(a).

programs. Our experiments show that we are able to reduce data messages on the average by 37% and synchronization messages by 96%. We believe that these are also the first results from a comprehensive evaluation of synchronization elimination in one-way communication. We discuss related work in Section 7 and conclude the paper with a summary in Section 8.

2 Preliminaries

We focus on structured programs with conditional statements and nested loops but without arbitrary goto statements. A *basic block* is a sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end without the possibility of branching except perhaps at the end [1]. A *control flow graph* (CFG) is a directed graph constructed by 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. Our technique for minimizing the communication volume and the number of messages is based on *interval analysis* [1]. 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, a node (reduced interval) is expanded, and the information regarding the nodes in that interval is computed.

3 Producers and Consumers

We assume that all loop bounds and subscript expressions are affine functions of enclosing loop indices and symbolic variables. Under this condition, a loop nest, an array and a processor grid can all be represented as bounded polyhedra. Our compiler currently uses the owner-computes rule [9], which assigns each computation to the processor that owns the data being computed.

Consider the generic single loop i shown in Figure 3(a). Let $\mathcal{R}_{\mathcal{L}}(i) = \mathtt{X}(\gamma_L * i + \theta_L)$ and $\mathcal{R}_{\mathcal{R}}(i) = \mathtt{Y}(\gamma_R * i + \theta_R)$. Let p and q denote two processors. We define several sets shown in Figure 3(b) where `S` is the communication statement and $\vee$ and $\wedge$ are logical 'or' and 'and' operations respectively. 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. `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 sent 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`. The `CommSet(S)` is currently used in our compiler's communication generation portion to generate `send` and `recv` commands. The necessary projection functions can be implemented by using the Omega library [11], and are shown in Figure 3(c). For instance, $\mathtt{ConsumersFor}(\mathtt{S}, q)$ is obtained from `CommSet(S)` by projecting out d and substituting q for q'; that is, q is a parameter and $\mathtt{ConsumersFor}(\mathtt{S}, q)$ enumerates p' values in terms of q.

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

$$\begin{aligned} \mathtt{Own}(\mathtt{Y}, q) \quad = \quad & \{d \mid \exists t, c, l \text{ s. t. } (t = \alpha d + \beta = \mathcal{C}Pc + \mathcal{C}q + l) \\ \wedge \quad & (y_l \le d \le y_u) \wedge (p_l \le q \le p_u) \wedge (t_l \le t \le t_u) \\ \wedge \quad & (0 \le l \le \mathcal{C} - 1)\}, \end{aligned}$$

where $P = p_u - p_l + 1$. In this formulation, $t = \alpha d + \beta$ represents alignment information and $t = \mathcal{C}Pc + \mathcal{C}q + l$ denotes distribution information. In other words, each array element d is mapped onto a point in a two-dimensional array. This point can be represented by a

```
   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)	$[p', q', d] \mapsto [p', q']$

(c)

$$
\begin{aligned}
\mathtt{Own}(\mathtt{X}, q) &= \{d \mid d \in \mathtt{X} \wedge \text{ is owned by } q\} \\
\mathtt{Producers}(\mathtt{S}) &= \{q' \mid \exists i, p' \text{ s. t. } \mathcal{R}_{\mathcal{R}}(i) \in \mathtt{Own}(\mathtt{Y}, q') \wedge \mathcal{R}_{\mathcal{L}}(i) \in \mathtt{Own}(\mathtt{X}, p') \wedge i_l \leq i \leq i_u \wedge q' \neq p'\} \\
\mathtt{Consumers}(\mathtt{S}) &= \{p' \mid \exists i, q' \text{ s. t. } \mathcal{R}_{\mathcal{R}}(i) \in \mathtt{Own}(\mathtt{Y}, q') \wedge \mathcal{R}_{\mathcal{L}}(i) \in \mathtt{Own}(\mathtt{X}, p') \wedge i_l \leq i \leq i_u \wedge q' \neq p'\} \\
\mathtt{ProducersFor}(\mathtt{S}, p) &= \{q' \mid \exists i \text{ s. t. } \mathcal{R}_{\mathcal{R}}(i) \in \mathtt{Own}(\mathtt{Y}, q') \wedge \mathcal{R}_{\mathcal{L}}(i) \in \mathtt{Own}(\mathtt{X}, p) \wedge i_l \leq i \leq i_u \wedge q' \neq p\} \\
\mathtt{ConsumersFor}(\mathtt{S}, q) &= \{p' \mid q \in \mathtt{ProducersFor}(\mathtt{S}, p')\} \\
\mathtt{PutSet}(\mathtt{S}, p, q) &= \{d \mid \exists i \text{ s. t. } d = \mathcal{R}_{\mathcal{R}}(i) \in \mathtt{Own}(\mathtt{Y}, q) \wedge \mathcal{R}_{\mathcal{L}}(i) \in \mathtt{Own}(\mathtt{X}, p) \wedge i_l \leq i \leq i_u \wedge q \neq p\} \\
\mathtt{SendSet}(\mathtt{S}) &= \{(q', p') \mid \exists d \text{ s. t. } d \in \mathtt{PutSet}(\mathtt{S}, p', q')\} \\
\mathtt{Pending}(\mathtt{S}) &= \{(p', q') \mid (q', p') \in \mathtt{SendSet}(\mathtt{S})\}.
\end{aligned}
$$

(b)

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

pair (c, l) and gives the local address of the data item in a processor. Simple BLOCK and CYCLIC(1) distributions can be handled within this framework by setting $c = 0$ and $l = 0$, respectively. The formulation given here can be generalized to multi-dimensional loops, arrays and processor grids [2]. Consider the first i-loop in Figure 6(b). Figure 4(a) shows the sets for this loop assuming that in the transformed program array bounds start from 0. 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 4(b). All of these sets can easily be represented and manipulated by the Omega library [11]. Notice that using the Omega sets to represent producer-consumer information, we are able to accommodate any kind of HPF-like alignment and distribution data 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 a same multi-dimensional array where $\mathcal{P}(.)$ and $\mathcal{Q}(.)$ denote two predicates and $\vec{d}$ refers to an array element. We define three operations, $\vee_c$, $-_c$, and $\wedge_c$ on these PutSets as shown in Figure 4(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.

4 Optimizing Communication

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

Local (Intra-Interval) Analysis The local analysis part of our framework computes Kill, Gen and Post_Gen sets defined below for each interval. Then the interval is reduced to a single node and annotated with these sets. With reference to Figure 3(a),

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

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

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

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

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 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.

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

(a)

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

(b)

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

(c)

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

(d)

Figure 4: (a) Sets for the first loop in Figure 6(b) for `BLOCK` distribution. (b) Sets for the first loop in Figure 6(b) for `CYCLIC(4)` distribution. (c) Operations on `PutSets`. (d) Dataflow equations for optimizing synchronization messages.

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 `i`. Then, from Figure 3(a), assuming that $\mathtt{Comm}(\mathtt{succ}(\mathtt{last}(\mathtt{i})), q) = \emptyset$, we have

$$
\begin{aligned}
\mathtt{Gen}(\mathtt{i}, p, q) &= \{\vec{d} \mid \exists \vec{\imath} \text{ s. t. } (\vec{i_l} \le \vec{\imath} \le \vec{i_u}) \wedge \\
&\quad (\vec{d} = \mathcal{R}_{\mathcal{R}}(\vec{\imath}) \in \mathtt{Own}(\mathtt{Y}, q)) \wedge (\mathcal{R}_{\mathcal{L}}(\vec{\imath}) \in \mathtt{Own}(\mathtt{X}, p)) \\
&\quad \wedge \neg(\exists \vec{\jmath}, \mathcal{R}_{\mathcal{L}}{}' \text{ s. t. } (\vec{i_l} \le \vec{\jmath} \le \vec{i_u}) \wedge (\vec{d} = \mathcal{R}_{\mathcal{L}}{}'(\vec{\jmath})) \\
&\quad \wedge (\vec{\jmath} \prec_{\mathtt{level(i)}} \vec{\imath}))\}, \\
\mathtt{Comm}(\mathtt{i}, p, q) &= [\bigwedge_{s \in succ(i)} \mathtt{Comm}(\mathtt{s}, \mathtt{p}, \mathtt{q})] \vee_c \mathtt{Gen}(\mathtt{i}, \mathtt{p}, \mathtt{q}).
\end{aligned}
$$

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), 1 corresponding to the outermost loop in the nest.

After the interval is reduced, the `Gen` set for it 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 `Gen` sets gives us all the communication that can be vectorized above a loop nest; that is, our analysis easily handles message vectorization [9]. 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:

$$
\begin{aligned}
\mathtt{Post\_Gen}(\mathtt{i}, p, q) &= \{\vec{d} \mid \exists \vec{\imath} \text{ s. t. } (\vec{i_l} \le \vec{\imath} \le \vec{i_u}) \wedge \\
&\quad (\vec{d} = \mathcal{R}_{\mathcal{R}}(\vec{\imath}) \in \mathtt{Own}(\mathtt{Y}, q)) \wedge (\mathcal{R}_{\mathcal{L}}(\vec{\imath}) \in \mathtt{Own}(\mathtt{X}, p)) \\
&\quad \wedge \neg(\exists \vec{\jmath}, \mathcal{R}_{\mathcal{L}}{}' \text{ s. t. } (\vec{i_l} \le \vec{\jmath} \le \vec{i_u}) \wedge (\vec{d} = \mathcal{R}_{\mathcal{L}}{}'(\vec{\jmath})) \\
&\quad \wedge (\vec{\imath} \prec_{\mathtt{level(i)}} \vec{\jmath}))\}.
\end{aligned}
$$

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.

Dataflow Equations In our framework, one-way communication calls are placed at the beginning of nodes in the CFG. Our dataflow analysis consists of a backward and a forward pass. In the backward pass, the compiler determines sets of data elements that can safely be communicated at specific points. The forward pass, 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 dataflow equations for the backward analysis are given by Equations (1) and (2) in Figure 5. Basically, they are used to combine and hoist communication. The sets $\texttt{Safe\_In}(\texttt{i}, p, q)$ and $\texttt{Safe\_Out}(\texttt{i}, p, q)$ consist of elements that *can* safely be communicated at the beginning and end of node i, respectively. Equation (1) says that an element should be communicated at a point if and only if it will be used in all of the following paths in the CFG. 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 the beginning of i if and only if it is either required by p in i or it reaches at the end of 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 i. $\mathcal{P}(\texttt{i})$=*false* implies aggressive communication combining and hoisting. An algorithm can also put a condition which tests the compatibility between $\texttt{Gen(i},p,q)$ and $\texttt{Safe\_Out(i},p,q)$ (e.g. two left-shift communications are compatible whereas a left-shift and a right-shift are not) [4].

The task of the forward analysis phase, which makes use of Equations (3), (4) and (5) in Figure 5, is to eliminate redundant communication by observing that (1) 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 (2) a successor should ignore what a predecessor has so far unless that predecessor dominates it. $\texttt{Put\_In}(\texttt{i}, p, q)$ and $\texttt{Put\_Out}(\texttt{i}, p, q)$ denote the set of elements that *have been* written so far (at the 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(i,p,q) set) (except the ones which have been killed in i) and the elements communicated in i and not killed subsequently (Post_Gen).

Interval Analysis Our approach starts by computing the Gen, Kill and Post_Gen sets for each node. Then the contraction phase reduces the intervals from the innermost loop to the outermost loop and annotates them with Gen, Kill and Post_Gen sets. When a reduced CFG with no cycles is reached, the expansion phase starts and 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 $\texttt{Put\_In}(\texttt{i}, \texttt{p}, \texttt{q})^{1^{st}pass}$ is set to the empty set. Note that the whole dataflow procedure operates on sets of equalities and inequalities which can be manipulated by the Omega library [11] or a similar tool.

Example Consider the synthetic benchmark given in Figure 6(a). In this example, communication occurs for three arrays: B, D and F. A communication optimization scheme based on message vectorization alone, can place communications and associated synchronizations as shown in Figure 6(b) before the loop bounds reduction. Note that a Synch message in that figure in fact represents a number of point-to-point synchronization messages. An application of our global communication optimization method generates the program shown in Figure 6(c). As compared with 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 the distribution directive is changed to CYCLIC(K) for any K. 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).

5 Optimizing Synchronization

In this section, we assume that the compiler has conducted the dataflow analysis described in Section 4 and determined the optimal communication points and communication sets. Assuming that these communications will be implemented by Put operations, we present a dataflow 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. For example, in Figure 6(c), the producer-consumer pattern for the communication occurring in 1 is identical for every repetition of time-step loop t.

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. We call the set of synchronizations (associated with a node i) that cannot be eliminated SynchFix(i). We refer the reader to [10] for the definition of SynchFix(i).

The dataflow 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. The procedure works on an augmented CFG, where each communication loop is represented by a single node. In the following, 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) happens to be empty after our dataflow analysis.

If the compiler wants to eliminate a Synch message for communication i from p to q, it needs to find a message $\texttt{t}_j$ 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 $\texttt{t}_j$ ($1 \leq j \leq k$) 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 Pending(i) set for a specific i are eliminated. We can formalize this condition as

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

Backward Analysis:

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

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

Forward Analysis:

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

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

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

Figure 5: Dataflow equations for optimizing communication.

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

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

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

We note that the condition (7) implies the algorithms offered by [6] and [8].

Claim: The synchronizations eliminated by (7) are a subset of the synchronizations that can be eliminated by (6) (see [10] for the proof).

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`,

$$\begin{aligned} \texttt{SynchSet(i)} \quad &= \quad \texttt{Pending(i)} -_c \{(p,q) \mid \exists j \text{ s. t.} & (8)\\ & q \in \texttt{ConsumersFor(t}_j\texttt{,}p\texttt{)}\} \vee_c \texttt{SynchFix(i)} & (9)\end{aligned}$$

As an example, let us consider the program shown in Figure 6(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`. 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; that is, the communications occurring in `2` and `3` together kill the synchronization requirements 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`. By 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 communication in `1`.

Dataflow Analysis: Our data flow equations are shown in Figure 4(d). Our analysis consists of iterative forward passes on the CFG. We first concentrate on the second equation and explain the functionality of $\mathcal{F}_i$. Here `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 $\texttt{PENDING\_IN(i)} = \{P_1, ..., P_{i-1}, P_i, P_{i+1}, ..., P_m\}$, we can define $\mathcal{F}_i$ as

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

where $f_i(P_k) = P_k -_c \texttt{SendSet(i)}$. That is, 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, the `PENDING_OUT` set of the *last* node gives the synchronization requirements to be satisfied. The resulting `PENDING_OUT` set, which is in fact a set of sets, is then reduced to a single set and used to represent the synchronization requirements of this loop to the next upper level.

We now consider the first equation of Figure 4(d) and explain the $\bigsqcup$ operator appearing there. In the join nodes, the compiler takes a conservative approach by unioning the synchronization requirements for the same communication. Suppose that for each $j \in pred(i)$, $\texttt{PENDING\_OUT(j)} = \{P_{j1}, P_{j2}, ..., P_{jm}\}$, where P_{jk} is the synchronization requirement of communication k up to the end of `j`. Assuming then that the resulting $\texttt{PENDING\_IN(i)} = \{P_1, P_2, ..., P_m\}$, each $P_l \in \texttt{PENDING\_IN(i)}$ 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 in [8] and [6].

Claim: The dataflow procedure defined by equations given in Figure 4(d) can reach a steady state in at most after two iterations (see [10] for the proof).

Example The upper part of Table 1 shows the sets `SendSet` and `Pending` (in an open form rather than in terms of equalities and inequalities) for the program shown in Figure 6(c). Note that the sets `SendSet(1)` and `Pending(1)` are computed from the `SendSet` and `Pending` sets respectively given in Figure 4(a). The sets for `2` and `3` are computed similarly. Before the dataflow analysis starts, `PENDING_IN(1)` is initialized as follows:

```
PENDING_IN(1) = {{Pending(1)},{Pending(2)},{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)}
```

The lower part of Table 1 demonstrates application of our dataflow 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.

6 Preliminary Results

The applications used in our study and their characteristics are listed in Table 2. We experimented with `BLOCK`, `B-CYC` (block-cyclic) and `CYCLIC` distributions on 8 and 32 processors to measure the static improvements. We refer to a version of program which is optimized by message vectorization alone as `base`. Table 3 presents the communication volume and the number of data messages (that is also the number of `Synch` messages) across all processors in the `base` programs.

For these applications, we first applied our global communication optimization algorithm. The percentage improvements are listed in Table 4. It should be noted that in block-cyclic distributions where most of the previous approaches fail, we have, on the average, a 29% reduction in communication volume and 40% reduction in number of messages across all processors.

We then applied our synchronization elimination scheme to the `base` version as well as the globally optimized version (`C-opt`) for each program. The results shown in Table 5 reveal that the algorithm is surprisingly successful in eliminating the redundant `Synch` messages. Except for two programs, the algorithm eliminates all synchronization messages from the `base` programs. When we look at the results for the programs that are optimized for communication prior to synchronization analysis, however, the picture somewhat changes. In `C-Opt` versions of `Jacobi` and `stfrg`, since the communication loop is reduced to one, no synchronization messages can be eliminated.

To sum up, our communication optimization algorithm eliminates 37.3% of the data messages and synchronization messages and reduces communication volume across all processors by 26%. Our synchronization elimination algorithm eliminates 96.8% of the synchronization messages in the message vectorized programs and 74% of the synchronization messages in the globally optimized programs.

Table 1: Dataflow sets and PENDING_OUT sets for the example in Figure 6(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)},{∅},{∅}}	{{∅},{∅},{∅}}
2	{{(3,4),(4,5),(5,6),(6,7)},{∅},{∅}}	{{∅},{∅},{∅}}
3	{{∅},{∅},{∅}}	{{∅},{∅},{∅}}

7 Related Work

Previously several methods have been presented to optimize the communication on distributed-memory message-passing machines. Most of the efforts considered communication optimization at 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 [9, 3]. Recently some researchers have proposed techniques for optimizing communication across multiple loop nests. The works in [5], [7], [12], [4], and [14] present similar frameworks to optimize `send/recv` communication globally and use variants of Regular Section Descriptors (RSD). Although this representation is convenient for simple array sections such as those found in block distributions, it is hard to embed alignment and general distribution information into it. Apart from this, working with section descriptors may result in overestimation of the communication sets. Instead, our approach is based on a linear algebra framework, and can represent all HPF-like alignment and distribution information accurately.

The approaches given in [6] and [8] examine the problem of eliminating redundant synchronization operations by piggy-backing them on data messages. Our approach is superior to both of these in eliminating synchronization as explained in the paper. Tseng [13] focuses on synchronization elimination problem. There is an important difference between our work and his. He eliminates synchronizations which are introduced by the insufficient communication analysis performed by the shared memory compilers. A compiler approach based on a 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 anyway. We rather focus on elimination of synchronizations that are caused by mandatory data communications. Such types of synchronization are not eliminated by Tseng's solution [13].

8 Summary

We presented dataflow algorithms to reduce number of data messages, communication volume and number of synchronization messages. Our experimental results revealed that our approach is quite successful in practice reducing on average 37.3% of data messages and 96.8% of synchronization messages in the message vectorized programs. We are working on compiling data-parallel programs with `Get` primitives, and elimination of synchronizations and possible deadlocks from programs compiled under hybrid approaches which employ both `Put/Get` and `send/recv` primitives.

References

[1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, techniques, and tools.* Addison-Wesley, 1986.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy. A combined communication and synchronization optimization algorithm for one-way communication. Technical Report, CPDC-TR-97-03, Northwestern University, October 1997.

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

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

[13] 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.

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

Table 2: Programs in our experiment set and their characteristics (+ = block, block-cyclic(4) or cyclic; * = not distributed).

PROGRAM	SUITE	LINES	ARRAYS	SIZE	DISTRIBUTION	DESCRIPTION
Jacobi	–	25	$2 \times 2D$	1024×1024	(+,*)	Jacobi iteration
2D hydro	Livermore 18	38	$9 \times 2D$	512×512	(+,*)	2D hydrodynamics
ADI	Livermore 8	30	$3 \times 3D,\ 3 \times 1D$	$256 \times 256 \times 2$	(+,*,*)	ADI fragment
vpenta	Spec92/NAS	147	$2 \times 3D,\ 7 \times 2D$	$128 \times 128 \times 3$	(*,+,*)	pentadiagonal inversion
SOR	–	25	$2 \times 2D$	256×256	(+,*)	successive over-relaxation
stfrg	Livermore 7	17	$1 \times 1D$	1024	(+)	state fragment equation
tomcatv	Spec95	190	$7 \times 2D,\ 2 \times 1D$	512×512	(+,*)	2D mesh generation
swim256	Spec95	428	$14 \times 2D$	512×512	(+,*)	shallow water eqn. solver

Table 3: Communication volume and number of data messages in the `base` (message vectorized) programs for a single iteration of the time-step loop.

PROGRAM	comm. volume (in thousand elements)						no. of data messages (Synch messages)					
	No. of Processors = 8			No. of Processors = 32			No. of Processors = 8			No. of Processors = 32		
	BLOCK	B-CYC	CYCLIC	BLOCK	B-CYC	CYCLIC	BLOCK	B-CYC	CYCLIC	BLOCK	B-CYC	CYCLIC
Jacobi	14.4	522.2	2088.9	63.3	522.2	2088.9	14	16	16	62	64	64
2D hydro	57.1	1036.3	4169.8	252.9	1036.3	4169.8	112	128	128	496	512	512
ADI	10.7	96.8	387.1	47.2	96.8	387.1	42	48	48	186	192	192
vpenta	343.2	343.2	343.2	1519.7	1519.7	1519.7	784	784	784	13888	13888	13888
SOR	21.3	193.5	774.2	94.5	193.5	774.2	84	96	96	372	384	384
stfrg	0.2	4.6	6.1	0.7	4.6	6.1	42	64	48	186	256	192
tomcatv	57.1	1040	8323.2	252.9	2072.6	8323.2	112	128	128	496	1024	1024
swim256	57.6	914.4	3649.1	229	914.4	3649.1	128	142	142	464	478	478

Table 4: Percentage (%) improvement over `base` (message vectorized) in communication volume and number of data messages.

PROGRAM	% improvement in comm. volume						% improvement in no. of data messages					
	No. of Processors = 8			No. of Processors = 32			No. of Processors = 8			No. of Processors = 32		
	BLOCK	B-CYC	CYCLIC	BLOCK	B-CYC	CYCLIC	BLOCK	B-CYC	CYCLIC	BLOCK	B-CYC	CYCLIC
Jacobi	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
2D hydro	50%	50%	50%	50%	50%	50%	50%	51%	50%	50%	50%	51%
ADI	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
vpenta	1%	1%	1%	1%	1%	1%	79%	78%	78%	78%	78%	78%
SOR	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
stfrg	71%	64%	0%	71%	64%	0%	83%	74%	0%	81%	73%	0%
tomcatv	74%	74%	75%	75%	75%	75%	74%	74%	75%	75%	75%	74%
swim256	37%	41%	43%	41%	41%	43%	44%	42%	42%	43%	41%	43%
average	29%	29%	21%	30%	29%	21%	41%	40%	31%	41%	40%	31%

Table 5: Percentage (%) improvement in number of synchronization messages obtained by our approach over the message vectorized (`base`) and globally optimized (`C-Opt`) versions.

PROGRAM	No. of processors = 8						No. of Processors = 32					
	BLOCK		B-CYC		CYCLIC		BLOCK		B-CYC		CYCLIC	
	Base	C-Opt	Base	C-Opt	Base	C-Opt	Base	C-Opt	Base	C-Opt	Base	C-Opt
Jacobi	100%	0%	100%	0%	100%	0%	100%	0%	100%	0%	100%	0%
2D hydro	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
ADI	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
vpenta	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
SOR	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
stfrg	83%	0%	75%	0%	83%	0%	83%	0%	75%	0%	83%	0%
tomcatv	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
swim256	94%	89%	93%	89%	93%	89%	93%	88%	93%	88%	93%	88%
average	97%	74%	96%	74%	97%	74%	97%	74%	96%	74%	98%	74%