Static Analysis to Reduce Synchronization Costs in Data-Parallel Programs

Manish Gupta and Edith Schonberg
IBM T. J. Watson Research Center
P. O. Box 704, Yorktown Heights, NY 10598

Abstract

For a program with sufficient parallelism, reducing synchronization costs is one of the most important objectives for achieving efficient execution on any parallel machine. This paper presents a novel methodology for reducing synchronization costs of programs compiled for SPMD execution. This methodology combines data flow analysis with communication analysis to determine the ordering between production and consumption of data on different processors, which helps in identifying redundant synchronization. The resulting framework is more powerful than any that have been previously presented, as it provides the first algorithm that can eliminate synchronization messages even from computations that need communication. We show that several commonly occurring computation patterns such as reductions and stencil computations with reciprocal producer-consumer relationship between processors lend themselves well to this optimization, an observation that is confirmed by an examination of some HPF benchmark programs. Our framework also recognizes situations where the synchronization needs for multiple data transfers can be satisfied by a single synchronization message. This analysis, while applicable to all shared memory machines as well, is especially useful for those with a flexible cache-coherence protocol, as it identifies efficient ways of moving data directly from producers to consumers, often without any extra synchronization.[1]

1 Introduction

Traditionally, parallel machines have been available in two distinct flavors – shared memory and distributed memory, which have respectively given rise to the shared-memory and the message-passing programming models. The complementary nature of these architectures and programming models have led to many areas of convergence. While shared memory provides the benefit of ease of programming, message-passing styles of writing parallel programs usually lead to performance advantages [11, 3].

Languages like High Performance Fortran (HPF) [4] provide the abstraction of a global address space on distributed memory machines, the program is merely annotated with directives specifying the distribution of data across processors. The compilers for these languages generate parallel programs in *single-program multiple-data* (SPMD) form, and generate the communication necessary to fetch values of non-local data referenced by each processor [9, 19, 15, 10, 6]. The knowledge of data mapping and computation partitioning at compile time offers many opportunities for reducing synchronization costs. Tseng [18] and O'Boyle and Bodin [13] have recently presented techniques that exploit these opportunities on shared memory and distributed shared memory (DSM) systems. However, their work only deals with recognizing the absence of synchronization requirements for statements with no communication and replacing barriers with cheaper producer-consumer synchronization. These optimizations, while extremely useful, can be viewed as straightforward application of distributed memory compilation style to shared memory machines. They do not eliminate synchronization messages from a computation that requires interprocessor communication.

This paper presents a novel methodology for reducing synchronization costs of a compiler-generated data-parallel program. Our work combines data flow analysis with communication analysis to establish an ordering between relevant events in the production and consumption of data on different processors. That gives us the first compiler algorithm that can eliminate synchronization messages even from a computation that needs communication (our framework naturally eliminates synchronization not associated with interprocessor communication by simply excluding it from consideration). For computations from which synchronization messages cannot be completely eliminated, this

[1]

work provides a systematic framework for achieving synchronization for multiple data transfers with a single or fewer synchronization messages. Our use of mapping functions to characterize the relationship between producers and consumers enables the applicability of this framework to more general forms of computation than previous work. For instance, O'Boyle and Bodin's approach [13] to replace barrier synchronization with cheaper producer-consumer synchronization can only be applied to computations with constant synchronization directions. We show that several commonly occurring computation patterns such as reductions and stencil computations with reciprocal producer-consumer relationship between neighboring processors lend themselves well to the elimination of synchronization messages, as unavoidable data transfers themselves serve as synchronization mechanisms for other data transfers. The precise information about the ownership of data (that is available from HPF directives [4], but could also possibly be obtained from compiler analysis in preceding passes [5, 1, 2]) and about computation partitioning makes this analysis more accurate than that possible with earlier work on reducing synchronization [12, 17].

Our analysis is presented for distributed memory machines which support separate mechanisms for remote data access and synchronization, and is readily applicable to shared memory machines, which inherently support these mechanisms separately. Distributed memory machines have traditionally supported `send` and `receive` as the basic primitives that provide the capability for both interprocessor data transfer and producer-consumer synchronization. More recently, many commercial machines built with a distributed memory like Cray T3D, Meiko CS-2, and Fujitsu AP1000+ have begun to provide primitives like `get/put`, which allow a processor to directly read data from or write data into the remote memory of another processor, with separate primitives for synchronization. Stricker et el. [16] and Hayashi et al. [7] have argued for separation of data transfer and synchronization, this paper offers more reasons in favor of such a separation. However, these researchers exclusively use barriers for all synchronization accompanying the data transfers. While these programs have fewer barriers to begin with (which are harder to remove) as compared to programs with fork-join parallelism that are "SPMDized" [18, 13], our work shows that even these barriers can often be completely eliminated or replaced by cheaper synchronization.

Our work, while applicable to all shared memory machines, has a special significance for those machines, such as Stanford FLASH [8] and Wisconsin Typhoon [14], that have a flexible cache-coherence protocol. Falsafi et al. [3] have earlier reported results on improved performance of different application programs with *update protocols*, where data is directly copied from producers to consumers, rather than being read by consumers using a fixed coherence protocol. Our work is the first to show how the compiler may use such a protocol, with no separate synchronization, for many regular computations.

The rest of this paper is organized as follows. Section 2 describes an HPF compilation framework and the implications of using `get` or `put` as the basic communication primitive instead of `send-receive`. Section 3 presents the analysis to detect situations where repeated communication patterns may be implemented using `put` operations without any extra synchronization message. Section 4 extends this analysis to more general patterns, and presents additional techniques to reduce the cost of synchronization. Section 5 describes the results of a preliminary study on the effectiveness of our analysis for HPF programs. Finally, Section 6 presents conclusions.

2 HPF Compilation Framework

We illustrate our ideas by discussing them in the context of `pHPF` [6], a prototype compiler for HPF, that currently generates communication on IBM SP1 and SP2 machines using `send-receive` as the basic primitive. We describe how communication generation could instead be done, in a future version, using `get` or `put`, which are being implemented in software on SP2 as part of a prototype run-time library. In this work, we shall view collective communication primitives such as broadcast and reduction also in terms of the basic primitives, since they can be implemented using `send-receive` or `get/put` for data transfer.

During the translation of an HPF program to the SPMD form, `pHPF` uses the *owner computes* rule [9, 19], which assigns a computation to the processor that owns the data being modified. In special cases like reductions, the computation is assigned to the owner of a reference other than the left hand side of the assignment statement. Given a right hand side reference corresponding to data not owned by the processor executing the statement, the compiler generates communication to send this processor the non-local values from the owner. `pHPF` performs optimizations like elimination of redundant communication [6] and message vectorization to move communication outside the loops [9, 19]. The analysis presented in this work is performed after the compiler has performed those optimizations and determined the placement of communication.

Using Get/Put instead of Send-Receive

We first note that using `send-receive` is inherently more expensive than `get` or `put` for just transferring data, because its underlying protocol is more complex. The implementation of `send-receive` has to guarantee the ordering of messages sent from one processor to another and/or provide mechanisms for matching `sends` and `receives` based on their tags. Furthermore, the receiving processor has to buffer any incoming messages that arrive before the `receive` has been posted, leading to buffer-copying overheads, or it

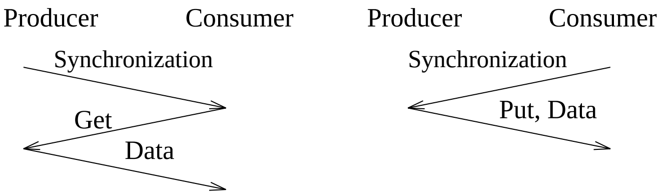

Figure 1: Communication using get or put with producer-consumer synchronization

has to engage in a hand-shake with the sender, leading to synchronization costs.

However, in contrast to send-receive, which takes care of both data transfer and synchronization between sending and receiving processors, a get or a put operation only realizes the data transfer, and synchronization has to be done separately. A get operation is performed by the consumer – there is synchronization needed to make sure that data has been produced. Similarly, a put operation into a location that has previously been used to hold a value being consumed requires synchronization so that the producer waits for that value to be consumed. Figure 1 illustrates a simple hand-shake protocol (omitting acknowledgement messages for simplicity) through which producer-consumer synchronization can be naively realized together with a get or put. The objective of our analysis is to eliminate the synchronization message whenever possible.

For regular computations where code generation is equally simple with get and put operations, we favor using put operations for two reasons. The use of put leads to one fewer exchange of messages between the producer and the consumer, as shown in Figure 1. Secondly, while the synchronization message in case of get is needed to satisfy the true dependence that data must be produced before being consumed, the synchronization message in case of put is needed for a storage-related dependence, and hence is easier to eliminate. Let us consider a hypothetical case where the HPF compiler uses a new, statically allocated buffer to hold non-local values on the consumer for each instance of a communication. There would be no need for a synchronization message from the consumer to the producer as it would always be legal for the producer to write into the buffer on the consumer, there would be no problems of overwriting a value not used yet. However, it is not usually practical to use a new buffer for every instance of communication in the program. Apart from using up excessive space, it could also lead to undesirable buffer-copying overhead. For example, for computations with nearest-neighbor communications, it is desirable to use *overlap regions* [19] to hold non-local data from neighboring processors, both for performance and ease of code generation. Using a new buffer for each instance of nearest-neighbor communication would require giving up on the convenience of using overlap regions or incurring the cost of local memory-to-memory copies to create multiple versions of overlap regions.

Our analysis for eliminating synchronization messages is explicitly presented only for put operations. On shared memory machines, this directly corresponds to using the *update protocol* [3], where data transfer is initiated by the producer. A similar analysis can also be applied to get operations, which correspond to regular, consumer-initiated read operations on shared memory systems. However, in that case, synchronization messages cannot be completely eliminated, since the first instance of at least one data transfer needs a synchronization message from the producer to the consumer. Our analysis can replace synchronization messages for multiple instances of data transfers and also those for multiple data transfers (with similar producer-consumer relationships) by a single synchronization message.

Overall Procedure Following the communication analysis which determines the placement of communication, the next step is to apply the data flow procedure described in the next section, to detect references for which no synchronization message is required. For the remaining references that need communication, the compiler examines introducing producer-consumer synchronization, one at a time. It checks if synchronization requirements of other references in that loop nest are met by the introduced synchronization, as described in Section 4. For a loop nest that needs more producer-consumer synchronization messages than a threshold value (performance estimation to determine that threshold is beyond the scope of this paper), it chooses a single barrier synchronization instead.

3 Elimination of Synchronization

In this section, we consider communication patterns which are identical for each repetition during the program. We describe a data flow procedure which analyzes each such communication to check if the put operation used to implement that communication is guaranteed to take place only after the value corresponding to the previous instance of that communication has been consumed. If so, each consumer can use the same buffer to hold non-local values during each instance of that communication, and there is still no synchronization message needed from the consumer to the producer.

3.1 Basic Idea

We illustrate a simplified version of this problem through Figure 2 in which a producer p repeatedly sends values corresponding to a variable reference r to a consumer c, which are received by c at the same memory location. We refer to the successive versions of the value corresponding to r as $v_1, v_2, \ldots, v_i, v_{i+1}, \ldots$. We can statically infer that the consumption of v_i at c precedes the put operation at p for the next version v_{i+1} if c sends a message to p (as part of

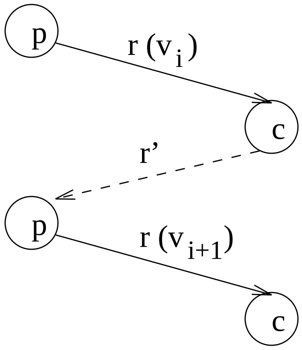

Figure 2: Precedence relationship between consumption and `put`

communication for some other reference r') after consuming v_i, and if p receives this message before sending v_{i+1}. The communication for r' is said to *kill* the producer-consumer synchronization requirements of repetitions of communication for r.

In actual practice, however, communication for a single reference may involve multiple producers and/or consumers and varied relationships between producers and consumers. Hence, a compiler can draw inferences of the type described above only by a *collective* reasoning about communications for each reference. We introduce mapping functions describing producer-consumer relationships to enable such a reasoning. We refer to the communication for a reference r as C_r, and refer to the statement with the computation using (consuming) the value of r as S_r. For each communication C_r, we define the following functions:

1. *Producers*$[C_r]$: The set of processors that send data.
2. *Consumers*$[C_r]$: The set of processors that receive data.
3. *ReceiveFrom*$[C_r](p)$: Given a processor p, the set of processors that receive data from p.

We use the terms "send" and "receive" in the above definitions merely to convey the direction of data transfer, not to suggest that the communication primitives used are `send` and `receive`.

Given that an HPF compiler usually tries to move communication outside loops, S_r often appears at a deeper nesting level than C_r. We define `CommOver`(C_r) (CO_r) as the statement before which communication C_r must finish, and `UseOver`(C_r) (UO_r) as the statement after which there is no more reference to r corresponding to a given instance of C_r. When communication C_r is placed at the same nesting level as S_r, both CO_r and UO_r are set to S_r. However, when communication is hoisted outside loops to a nesting level l, CO_r and UO_r are respectively set to the beginning and the end of the loop at nesting level $l+1$ surrounding statement S_r. For example, CO_r is set to S_1 and UO_r is set to S_3 for the communication shown in Figure 3.

The following result forms the basis of our analysis.

Theorem 1 *Let x and y be two references with their communications C_x and C_y being implemented as* `puts`. *Communication C_y kills the producer-consumer synchronization requirements of identical repetitions of C_x if:*

1. *C_y and CO_y always execute after UO_x and before the next repetition of C_x,*
2. *Consumers$[C_x] \subseteq$ Producers$[C_y]$, and*
3. *$\forall p \in$ Producers$[C_x], \forall q \in$ ReceiveFrom$[C_x](p), p \in$ ReceiveFrom$[C_y](q)$*

Proof : Consider an arbitrary processor p that performs a `put` operation on a value v_i as part of an arbitrary, ith instance of C_x, referred to as C_x^i. We show that p cannot perform a put operation as part of C_x^{i+1} until each receiver q has consumed v_i (and does not refer to v_i any more) in its computation.

Consider an arbitrary receiver q that receives data from p. By Condition 2, $q \in$ *Producers*$[C_y]$, i.e., q sends data under C_y. Furthermore, since $q \in$ *ReceiveFrom*$[C_x](p)$, by Condition 3, $p \in$ *ReceiveFrom*$[C_y](q)$, i.e., p receives data from q as part of C_y. Condition 1 implies that under C_y, q sends data to p only after executing UO_x, i.e., after consuming v_i, and p receives that data at CO_y before participating in C_x^{i+1}. Since p cannot possibly receive data before it is sent by q, the consumption of v_i at q takes place before participation of p in C_x^{i+1}.
□

3.2 Computation of Processor Functions

We first describe the analysis pertaining to the relationship between producers and consumers for different communications, which forms the basis for testing the last two conditions of Theorem 1. In order to simplify the analysis, we deal with *virtual* processors, which correspond to template positions over which different array elements are aligned, as specified by HPF alignment directives. During the rest of this paper, we shall refer to HPF template positions as processors (we only consider the distributed dimensions of a template). An alignment function maps an array subscript i to a processor grid position $c * i + o$, where c and o are integers, and when $c = 0$, o may take a special value $*$, which represents all processors along that grid dimension. We shall refer to this special value as $\mathcal{U}$ in this paper, to avoid confusing it with the sign for multiplication.

We now show how to determine the *Producers*, *Consumers*, and *ReceiveFrom* functions for communication involving array references in one dimension, where the arrays are mapped to a one-dimensional processor grid. We will later show the extension to the multi-dimensional case. We assume that the subscripts in array references are constants or affine functions of loop indices. In Figure 3, $B(\alpha_2 * i + \beta_2)$

```
      !HPF$ Align $A(i)$ with $VPROCS(c_A * i + o_A)$
      !HPF$ Align $B(i)$ with $VPROCS(c_B * i + o_B)$
            ⟨ communication for $B$ ⟩
      $S_1$    do $i = low, high$
      $S_2$        $A(\alpha_1 * i + \beta_1) = B(\alpha_2 * i + \beta_2)$
      $S_3$    end do
```

Figure 3: Example of regular communication in one dimension

is sent to owner of $A(\alpha_1 * i + \beta_1)$. Hence, using the alignment information, we infer that as part of this communication C_r, $VPROCS(c_B\alpha_2 * i + c_B\beta_2 + o_B)$ sends a value to $VPROCS(c_A\alpha_1 * i + c_A\beta_1 + o_A)$. The sets of producers and consumers are obtained as triples by "expanding" these processor positions with respect to the loop bounds:

$$\begin{aligned}
Producers[C_r] &= (c_B\alpha_2 low + c_B\beta_2 + o_B : \\
&\qquad c_B\alpha_2 high + c_B\beta_2 + o_B : c_B\alpha_2) \\
Consumers[C_r] &= (c_A\alpha_1 low + c_A\beta_1 + o_A : \\
&\qquad c_A\alpha_1 high + c_A\beta_1 + o_A : c_A\alpha_1)
\end{aligned}$$

If $c_B, \alpha_2, c_A, \alpha_1$ all take non-zero values, the i-loop is said to be a *producer-consumer traversal loop* of the processor grid dimension for the given communication. If there is no producer-consumer traversal loop, $ReceiveFrom[C_r](p)$ becomes a constant function that does not depend upon the value of p, and is obtained as $Consumers[C_r]$. Otherwise, we obtain $ReceiveFrom[C_r](p)$ by setting $p = c_B\alpha_2 * i + c_B\beta_2 + o_B$ and substituting for i in terms of p. Thus, we get:

$$\text{ReceiveFrom}[C_r](p) = \begin{cases} [(c_A\alpha_1)/(c_B\alpha_2)] * p + [c_A\beta_1 + o_A - (c_B\beta_2 + o_B)/(c_B\alpha_2)] \\ \qquad \text{if } c_B * \alpha_2 \neq 0 \\ (c_A\alpha_1 low + c_A\beta_1 + o_A : c_A\alpha_1 high + c_A\beta_1 + o_A : c_A\alpha_1) \\ \qquad \text{otherwise} \end{cases}$$

We can represent *ReceiveFrom* in the form of a coefficient C and a displacement D, such that it maps p to $p' = C * p + D$. The displacement term may be represented as a range (in the form of a triple), or may include the special value $\mathcal{U}$, which signifies all processor positions. It is more convenient to represent the *ReceiveFrom* function as a single transformation matrix, which is done by augmenting the representation of a k-dimensional processor grid with an extra dummy element. Thus, for a 1-dimensional processor grid, we use the vector $[p, 1]$ to represent processor p, and get a 2 x 2 transformation matrix which maps $[p, 1]$ to $[p', 1]$ as shown below:

$$\begin{pmatrix} p' \\ 1 \end{pmatrix} = \begin{pmatrix} C & D \\ 0 & 1 \end{pmatrix} \begin{pmatrix} p \\ 1 \end{pmatrix}$$

In general, for a k-dimensional processor grid, we obtain the *ReceiveFrom* function as a $(k+1)$ x $(k+1)$ transformation matrix. The algorithm for determining the transformation matrix RF representing *ReceiveFrom* is sketched in Figure 4. The entries in the last column of the transformation matrix correspond to the "offset" terms, represented as triples. Any loop index appearing in an offset term, which corresponds to a loop from which communication has been moved out, is expanded to its loop bounds.

```
Compute-ReceiveFrom($l$, $r$, $RF$) /* inputs: rhs reference $r$,
                                               lhs reference $l$,
                                         output: matrix $RF$ */
{
  1. Apply alignment function to each subscript in distributed dimension to
     obtain information of the form $VPROCS(p_1, \ldots, p_k)$
     sends data to $VPROCS(p'_1, \ldots, p'_k)$, where $p_i$ and $p'_i$
     are expressions.

  2. Identify a producer-consumer traversal loop index $L_i$,
     if possible, for each dimension $i$ of $VPROCS$. If there
     is more than one candidate for any dimension (due
     to the subscript being an affine function of more than
     one loop index), choose a loop so that over all $k$ dimensions,
     there are as many loops chosen as possible.

  3. For each grid dimension $i$, if $L_i$ is a valid loop index
     identified in the previous step, then $p_i$ would be expressible
     as $p_i = \gamma_i * L_i + \delta_i$. Regard $p_i$ as a variable
     now and $L_i$ as a function of $p_i$ (swapping the roles of
     $p_i$ and $L_i$), and obtain $L_i = (p_i - \delta_i)/\gamma_i$

  4. For each grid dimension $i$, substitute any occurrence
     in $p'_i$ of $L_j, 1 \leq j \leq k$, by $(p_j - \delta_j)/\gamma_j$. Let the final
     expression for $p'_i$ be of the form $p'_i = t_i^1 * p_1 + \ldots + t_i^k *$
     $p_k + t_i^{k+1}$. The elements $RF(i,j), 1 \leq j \leq k+1$, are
     given by the terms $t_i^j, 1 \leq j \leq k+1$ in the expression
     for $p'_i$. The $(k+1)$th row of $RF$ is given by $RF(k+1, j) = 0, 1 \leq j \leq k$,
     and $RF(k+1, k+1) = 1$.
}
```

Figure 4: Algorithm to compute *ReceiveFrom* function

Example Consider the example shown in Figure 5. The communication for B involves $VPROCS(j, i+1)$ sending data to $VPROCS(i, \mathcal{U})$. We obtain the *ReceiveFrom* transformation matrix as:

$$RF = \begin{pmatrix} 0 & 1 & -1 \\ 0 & 0 & \mathcal{U} \\ 0 & 0 & 1 \end{pmatrix}$$

```
      !HPF$ Align $A(i,j)$ with $VPROCS(i,*)$
      !HPF$ Align $B(i,j)$ with $VPROCS(i,j)$
            ⟨ communication for $B$ ⟩
            do $j = 1, n$
               do $i = 1, n$
                  $A(i,j) = B(j, i+1)$
               end do
            end do
```

Figure 5: Example of communication in multiple dimensions

```
!HPF$ Distribute A(block)
      do i = 1, n
         s = s + A(i)
      end do
```

Figure 6: Example of reduction

Check for Condition 2 of Theorem 1 Given two communications C_x and C_y in a k-dimensional processor grid, let $Consumers[C_x]_i$ and $Producers[C_y]_i$ represent the ith positions in the corresponding k-dimensional vectors. Let $Consumers[C_x]_i$ be the triple $(l_1^i : u_1^i : s_1^i)$, and let $Producers[C_y]_i$ be the triple $(l_2^i : u_2^i : s_2^i)$. The triples are represented (if necessary, by a trivial transformation) such that the strides s_1^i and s_2^i are positive. We can infer that $Consumers[C_x] \subseteq Producers[C_y]$ if for each i in the range $1 \leq i \leq k$, we have (i) $l_1^i \geq l_2^i$, (ii) $u_1^i \leq u_2^i$, (iii) $(l_1^i - l_2^i) \bmod s_2^i = 0$, and (iv) $s_1^i \bmod s_2^i = 0$.

Check for Condition 3 of Theorem 1 Let RF_x and RF_y represent the *ReceiveFrom* transformation matrices for communications C_x and C_y respectively. We infer that Condition 3 of Theorem 1 is satisfied if the product matrix $T = RF_y * RF_x$ is a "superset" of the identity matrix I. Given a k-dimensional processor grid, we conclude that $T \supseteq I$ (when applied to the producers of C_x) if for each i in the range $1 \leq i \leq k$, we have one of the following conditions being satisfied:

1. $T(i,i) = 1$; $T(i,j) = 0, 1 \leq j \leq k, j \neq i$; and $0 \in T(i, k+1)$, or
2. $T(i, k+1) = \mathcal{U}$, or
3. $T(i,j) = 0, 1 \leq j \leq k$, and $Producers[C_x]_i \subseteq T(i, k+1)$.

The above test ensures that for each processor p that plays the part of a producer in C_x, the application of the mapping functions $ReceiveFrom[C_x]$ followed by $ReceiveFrom[C_y]$ maps it back to a set of processors that includes p.

Reductions Reduction operations across processors are handled in a special manner during compilation, and have a special significance with respect to our analysis. Consider the example shown in Figure 6. In the SPMD program, the computations of partial sums are performed over the local portion of A on each processor, followed by a global reduction operation requiring communication, that takes place after the loop. Let us view the sequence of operations involving the global reduction as a communication from the participating processors to the root of the reduction fan-in tree (C_x), followed by the reduction computation (UO_x and CO_x), followed by a broadcast of the final value to all the processors (C_y), which in turn is followed by consumption of that value (CO_y). If the global reduction operation over this group of processors is repeated in a loop, we observe that C_x and C_y mutually kill the synchronization requirements of repetitions of each other. Even though the actual implementation of the reduction primitive may deviate from this simplified view, it can still be done so as to preserve the property that no synchronization messages are needed.

```
SynchApply($SC\_IN_i$, $LOC_i$)
{
   $SC\_OUT_i = SC\_IN_i$
   if ($GetComm(LOC_i) = r$) then        /* comm node */
      if ($SC\_IN_i[r] = U$) then
         $SC\_OUT_i[r] = CS$
      else if ($SC\_IN_i[r] = CS$ or $SC\_IN_i[r] = Cons$ or
              $SC\_IN_i[r] = NoSyncP$) then
         $SC\_OUT_i[r] = Sync$
      endif
      for $k = 1$ to #comm do
         if ($SC\_IN_i[k] = Cons$ and repetitions of $C_k$ are
            identical and $C_r$ kills producer-consumer
            synchronization requirements of $C_k$ by
            satisfying Conditions 2, 3 of Theorem 1) then
               $SC\_OUT_i[k] = NoSyncP$
               Add $\langle k, r \rangle$ to $SC\_OUT_i$.Tag
         endif
      endfor
   else      /* computation node */
      if ($GetUO(LOC_i) = r$ and $SC\_IN_i[r] = CS$) then
         $SC\_OUT_i[r] = Cons$
      endif
      if ($GetCO(LOC_i) = r$) then
         for each $k$ such that $SC\_IN_i$.Tag includes $\langle k, r \rangle$
            if ($SC\_IN_i[k] = NoSyncP$) then
               $SC\_OUT_i[k] = NoSync$
            endif
         endfor
      endif
   endif
   return $SC\_OUT_i$
}
```

Figure 7: Pseudocode for function *SynchApply*

3.3 Data Flow Analysis

We now describe the data flow procedure which determines for each communication if its repetitions require any synchronization, given that the same buffer is used to hold non-local values for each instance of that communication, and that the communication is implemented using puts. Essentially, the data flow procedure provides a method for testing Condition 1 of Theorem 1 for the relevant pairs of communications, while the remaining conditions are tested separately where needed. The analysis is performed on the control flow graph representation of the program, after the placement of communication has been determined by the

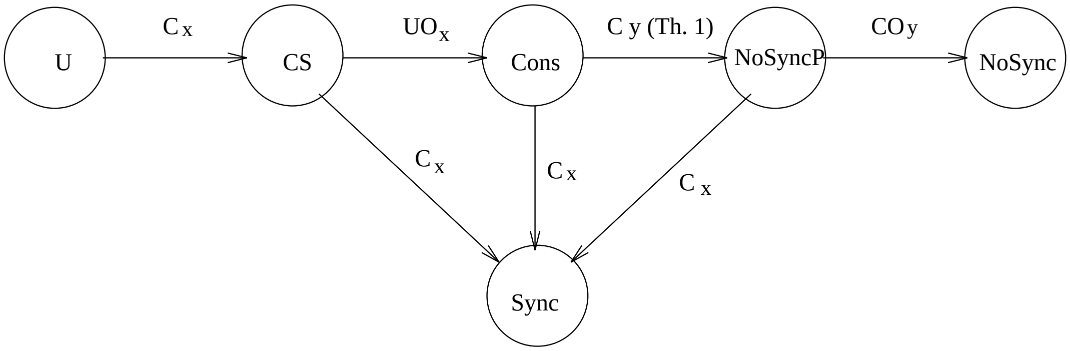

Figure 8: State transitions for communication C_x

SynchMeet	*U*	*CS*	*NoSync*	*NoSyncP*	*Cons*	*Sync*
U	*U*	*CS*	*NoSync*	*NoSyncP*	*Cons*	*Sync*
CS	*CS*	*CS*	*NoSync*	*NoSyncP*	*Cons*	*Sync*
NoSync	*NoSync*	*NoSync*	*NoSync*	*NoSyncP*	*Cons*	*Sync*
NoSyncP	*NoSyncP*	*NoSyncP*	*NoSyncP*	*NoSyncP*	*Cons*	*Sync*
Cons	*Cons*	*Cons*	*Cons*	*Cons*	*Cons*	*Sync*
Sync	*Sync*	*Sync*	*Sync*	*Sync*	*Sync*	*Sync*

Table 1: The definition of *SynchMeet* operation over data flow values

compiler. The augmented control flow graph has two kinds of nodes – computation nodes and communication nodes, and edges that denote the flow of control. Each statement in the source program has an associated computation node. Similarly, each communication for a reference is represented by a single node.

The only computation nodes that influence any data flow value are the *join* nodes in the CFG and the nodes that correspond to UO_r or CO_r for some communication C_r. We shall refer to the local information associated with a node i as LOC_i. The functions $GetUO(LOC_i)$ and $GetCO(LOC_i)$ respectively return r when node i represents UO_r or CO_r, and return `null` otherwise. The function $GetComm(LOC_i)$ returns r for a communication node i representing C_r, and returns `null` for a computation node. The basic data flow variables computed by our analysis are the following:
SC_IN_i : synchronization requirements of communications, as inferred at entry to node i.
SC_OUT_i : synchronization requirements of communications, as inferred at exit from node i.

For the purpose of data flow analysis, the information on synchronization requirements of communications is maintained as a vector of a 6-valued enumeration type, with an entry for each reference that needs communication. The six possible values are: U (communication not seen yet), CS (communication seen), *Cons* (consumption of communicated value over), *NoSyncP* (preliminary indication that no synchronization message is needed for repetitions), *NoSync* (confirmation that no synchronization message is needed for repetitions), and *Sync* (synchronization message needed for repetitions). Associated with each reference, there is also a pointer to the relevant information about its communication. For the entry node, SC_IN is initialized to all U's. The value of LOC for each node is set by the compiler based on the placement of communication. The following data flow equations are applied in a traversal along the direction of edges.

$$SC\_OUT_i = SynchApply(SC\_IN_i, LOC_i) \quad (1)$$

$$SC\_IN_i = SynchMeet_{p \in pred(i)}(SC\_OUT_p) \quad (2)$$

The function $SynchApply(SC\_IN_i, LOC_i)$ is described in Figure 7. The transitions effected by this function are pictorially shown in Figure 8. At each join node, the function *SynchMeet* is applied elementwise over all the SC_OUT values of its predecessors. Table 1 defines the *SynchMeet* operation at the element level. Following the *SynchMeet* operation, a data flow value can only move lower in the semi-lattice consisting of the values *Sync* ($-$), *Cons*, *NoSyncP*, *NoSync*, *CS*, and U ($\top$). Our data flow procedure handles loops by applying the Equations 1 and 2 iteratively over the loop body until the solution converges. We can show that the solution will always converge within three iterations, which makes our data flow procedure quite efficient. Let us consider the data flow value with respect to communication C_r and a loop L. There are two cases:
Case 1 : C_r appears inside the loop L. Let C_r correspond to node i. At the end of the first iteration, $SC\_OUT_i[r]$ can possibly take only a value other than U. Hence, during the second iteration, along the back-edge of the loop, $SC\_IN_i[r]$ cannot take the value U. If $SC\_IN_i[r]$ has been

set to CS, $NoSyncP$, or $Sync$, then $SC\_OUT_i[r]$ gets the value $Sync$, which cannot change any further. If $SC\_IN_i[r]$ had the value $NoSync$, either it does not change during the next and future iterations, or it changes to CS, $NoSyncP$, or $Sync$, in which case $SC\_OUT_i[r]$ gets the value $Sync$ during the third iteration, which does not change during future iterations as well.

Case 2 : C_r does not appear inside the loop L. Due to the nature of communication code generation, UO_r also cannot appear inside L. Hence, the only transitions possible in the data flow value are due to some C_y or CO_y, which respectively change the value from $Cons$ to $NoSyncP$ or $NoSyncP$ to $NoSync$. Since these transitions can happen only if the value on entry to C_y is $Cons$, and the value on entry to CO_y is $NoSyncP$, any change in these values at entry to those respective nodes stops further transitions.

Finally, after the data flow solution is obtained, the value of $SC\_OUT_i[r]$ at exit from the node i containing C_r can only be CS (if there is no repetition possible of C_r in the procedure), $NoSync$, or $Sync$. A value of CS or $NoSync$ implies that no synchronization message is needed for C_r, if implemented using **put** operation(s) into a fixed buffer. In this work, we have assumed a static allocation of buffers holding non-local data. If the compiler uses dynamic allocation instead, there would be a need for synchronization from the consumers to the producers after the allocation of the buffer. In that case, a producer would specify the remote address on consumer using indirect addressing mode in the **put** operation.

4 Extensions and Additional Techniques

We first describe extensions to our analysis that allow synchronization messages associated with **put** operations to be eliminated in more general conditions than those covered in Section 3. We then describe additional techniques that help reduce the cost of synchronization messages, when they cannot be eliminated.

Different Producers The procedure described in the previous section eliminates synchronization messages only for repetitions of identical communication patterns associated with a reference. We now show how to deal with different instances of communication that involve different producers. In the absence of data redistributions, this situation arises in the context of communication for an array reference with a subscript in a distributed dimension that is not invariant with respect to the repetitions of communication. For example, if in Figure 5 communication for B were to be placed inside the j-loop, different instances of that communication would involve different producers. Given such a communication C_x, we define the following function:
$NextProducers[C_x](p)$: Given a processor p which produces a value for the current instance of C_x, the set of processors which may produce the value for the next instance of C_x.

For the modified example in Figure 5 with communication for B placed inside the j-loop, we would get the transformation matrix representing $NextProducers$ function as:

$$NP = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Let us consider references C_x and C_y as defined in Theorem 1, with the generalization that different instances of C_x may have different producers. The result stated in Theorem 1 can be generalized to the following:

Theorem 2 *Communication C_y kills the producer-consumer synchronization requirements of repetitions of different instances of C_x if:*

1. *C_y and CO_y always execute after UO_x and before the next repetition of C_x,*
2. *$Consumers[C_x] \subseteq Producers[C_y]$, and*
3. *$\forall p \in Producers[C_x], \quad \forall q \in ReceiveFrom[C_x](p)$, $NextProducers[C_x](p) \subseteq ReceiveFrom[C_y](q)$*

Proof: Omitted, similar to the proof of Theorem 1.

The first two conditions are identical to those of Theorem 1. Condition 3 can be verified by checking if $RF_y * RF_x \supseteq NP_x$, where NP_x is the transformation matrix representing $NextProducers[C_x]$, and the test for the superset relationship is done in the same manner as defined in Section 3.2. When $NextProducers[C_x]$ is an identity function, the above result reduces to Theorem 1. The $NextProducers$ function can usually be specified in a simple form only for a single loop outside the point where communication is placed. Beyond that, it can always be conservatively estimated as mapping p to $\mathcal{U}$, which signifies all processors. However, if elimination of synchronization is possible only with the precise information for $NextProducers$, synchronization can at least be moved outside the immediately surrounding loop in which communication is repeated.

Killing Synchronization Requirement with Multiple References Instead of a single reference y, let us consider a sequence of references $y_1, y_2, \ldots, y_n$, such that communication for y_i starts after communication for y_{i-1}, and completes before communication for $y_{i+1}, 1 < i < n$. Using the above notation, we obtain the following generalization to Theorem 2:

Theorem 3 *The communication for a sequence of references $y_1, \ldots, y_n$, as specified above, kills the producer-consumer synchronization requirements of repetitions of different instances of C_x if:*

1. *C_{y_i} and S_{y_i} always execute after UO_x and before the next repetition of C_x, $\quad 1 \leq i \leq n$,*

2. *Consumers*$[C_x] \subseteq$ *Producers*$[C_{y_1}]$; *Consumers*$[C_{y_i}] \subseteq$ *Producers*$[C_{y_{i+1}}]$, $1 \leq i < n$, *and*

3. $RF_{y_n} * RF_{y_{n-1}} * \ldots * RF_{y_1} * RF_x \supseteq NP_x$

Using Distribution information for Templates For simplicity, our analysis is performed at the level of virtual processors, which correspond to template positions in HPF, and therefore does not take advantage of information about different virtual processors being mapped to the same physical processor. The *ReceiveFrom* and the *NextProducers* mappings require a significantly more complex representation in the space of physical processors. However, it is relatively easy to perform the test for Condition 2 of Theorems 1, 2 and 3 by computing the set of producers and consumers in physical processor space, when the number of processors is known statically. In fact, this computation is any way done by the pHPF compiler during code generation.

Synchronizing Multiple Communications with a Single Message When the synchronization message for a given communication C_r cannot be eliminated, the compiler needs to generate synchronization from the consumer to the producer. Viewing this synchronization as a new communication C_y[2] and treating C_r as the statement (CO_y in our notation) which executes only after communication C_y has completed, we can now apply our analysis to see if C_y kills the synchronization requirements of repetitions of any other communication C_x. Clearly, this technique will succeed in cases where the message aggregation optimization [9] (combining communication for multiple references with the same communication pattern) is applicable.

This approach can also be applied to other forms of synchronization, such as barriers, that will service multiple communications. For a barrier synchronization, each of the functions, *Producers*, *Consumers*, and *ReceiveFrom*(p) evaluates to the set of all processors. Thus, a barrier outside a parallel loop covers the synchronization requirements of all references inside that loop, as their communications are moved outside the loop by message vectorization. Unlike a program with fork-join parallelism, no synchronization is needed after the loop if computations following the loop body use put operations into separate buffers for interprocessor data transfer.

Using Extra Storage As mentioned earlier, the compiler can always use extra storage in conjunction with put to eliminate the need for synchronization from the consumer(s) to the producer(s). In general, the compiler has the flexibility of trading off memory with synchronization costs, and choosing the appropriate loop level upto which a buffer used to hold non-local values would be expanded. Unfortunately, as we noted in Section 2, using this technique can lead to buffer-copying overhead, when different instances of an array reference in the program correspond to both local and non-local data. For computations with nearest-neighbor communication, it is usually better to use fixed *overlap regions* [19] to hold non-local data.

```
do $j' = 1, n, s$
    ⟨ communication for $A$ ⟩
    do $j = j', \min(j' + s - 1, n)$
        do $i = 1, n$
            $\ldots = A(i, j)$
        end do
    end do
end do
```

Figure 9: Loop stripmining to control message size

Managing Storage with Loop Stripmining A well-known technique employed by compilers to control the size of messages is the stripmining transformation illustrated in Figure 9. This example shows the j-loop split into two levels and communication placed between those loops even though it could be legally moved beyond the outer loop. If synchronization of producers and consumers is needed for this communication, the compiler can reduce the cost by moving synchronization outside the j' loop. The initial synchronization is needed from the producers to the consumers indicating that data is ready, and the actual data transfer can be realized using gets inside the j'-loop without any further synchronization. In fact, even the initial synchronization can be avoided by using put for the data transfer (into a statically allocated buffer) during the first iteration of j'-loop and using get during subsequent iterations.

5 Preliminary Experiments

In order to examine the applicability of the analysis presented in this paper to HPF programs, we conducted a study with two programs representing different kinds of regular computations. We started with the information on placement of communication generated by the pHPF compiler, and applied the steps of the data flow analysis procedure by hand.

Tred2 The program tred2 is from the EISPACK library. It reduces a real symmetric matrix to a symmetric tridiagonal matrix, and has a number of sum reductions in its computation. We added HPF directives to align the 1-D arrays in the program with the rows of the 2-D arrays and to distribute those arrays by rows [5]. Figure 10 shows a representative segment from the program. We have marked the references which require communication in bold letters. The communications for the three references in the first j-loop are vectorized and moved before the j-loop. The synchro-

[2] The processor functions for C_y are computed in exactly the same manner as those for C_r except that the roles of the rhs and the lhs references are reversed.

```
do $ii = 2, n$
    ...
    do $j = 1, l$
        $F = \mathbf{D(j)}$
        $G = \mathbf{E(j)} + \mathbf{Z(j,j)} * F$
        ...
    end do
    ...
    do $j = 1, l$
        $E(j) = E(j)/H$
        $F = F + \mathbf{E(j)} * D(j)$
    end do
    ...
end do
```

Figure 10: Program segment from `tred2`

```
!HPF$ Distribute grid(block,block,*)
do $it = 1, ncycles$
  do $j = 2, n+1$
    do $i = 2, n+1$
      $grid(i,j,inew) = exp((alog(grid(i,j-1,iold))+$
                        $alog(grid(i,j,iold))+$
                        $alog(grid(i,j+1,iold))+$
                        $alog(grid(i-1,j-1,iold))+$
                        $alog(grid(i-1,j,iold))+$
                        $alog(grid(i-1,j+1,iold))+$
                        $alog(grid(i+1,j-1,iold))+$
                        $alog(grid(i+1,j,iold))+$
                        $alog(grid(i+1,j+1,iold)))/9.0)$
    end do
  end do
  ...
end do
```

Figure 11: Program segment from `grid`

nization requirements for their repetitions inside the *ii*-loop are killed by the communication due to the global reduction taking place after the next j loop. The reference to $E(j)$ in the statement with reduction represents the array reference whose ownership determines the assignment of computation for the partial sum.

For the entire program, the pHPF compiler identifies 19 references to array and scalar variables as requiring communication. Our analysis identifies 10 of those communications that are repeated inside loops (with potentially different producers) as not requiring any synchronization message. The synchronization requirements are killed by references involved in the sum reductions. Another 2 of the remaining communications are executed only once. Therefore, only 36.8% of the 19 references with communication require synchronization messages after this analysis.

Grid The `grid` program is taken from the benchmark suite developed by Applied Parallel Research, Inc. This program performs a 9-point stencil computation followed by global reductions, and is representative of computations with nearest-neighbor communication. The kernel of the computation is shown in Figure 11. The i and the j loops are marked as parallel using the HPF *independent* directive. In this kernel, pHPF identifies 4 references as needing communication (communication for 4 other references is eliminated by the optimization for message coalescing and eliminating redundant communication [6]). Our analysis is unable to eliminate the synchronization need for any of these references. However, we observed that the analysis would work much better if the program is rewritten by unrolling the outermost loop once to make it clear that the variables *iold, inew* alternatively take the values of [1,2] and [2,1] in different iterations of the outermost loop. In the resulting program, our analysis would eliminate synchronization message for every reference in this program segment. For example, the references to $A(i-1, j-1, 1)$ in the first i,j-loop nest and to $A(i+1, j+1, 2)$ in the second i,j-loop nest would mutually kill the synchronization requirements of each other's communication.

This study suggests that it may be possible to eliminate synchronization messages to a considerable extent by using `put` operations for data transfer, and using the analysis presented in this paper. Other researchers [18, 13] have shown impressive improvements in program performance with smaller reductions of synchronization than we are able to accomplish. Intuitively, our analysis is likely to work well on regular applications which are loosely synchronous or have reciprocal producer-consumer relationships of data in their basic computations. However, some cases may require intervention from the programmer or sophisticated capabilities for program analysis in the underlying compiler and transformations like loop-unrolling to enable our techniques to work effectively.

6 Conclusions

We have presented a framework, based on data flow analysis and communication analysis, for reducing synchronization costs of programs compiled for SPMD execution. This framework is more powerful than any previously presented work. It can eliminate synchronization messages even for computations that need communication. A preliminary examination of some HPF benchmark programs confirms that this optimization works extremely well with commonly occurring computation patterns like reductions and stencil computations. Our framework also recognizes situations where the synchronization needs for multiple data transfers can be satisfied by a single synchronization message. We expect the kind of analysis presented in this paper to play an increasingly important role in future high-performance compilers for parallel machines, as synchronization costs become more dominant with improvements in processor speeds.

We plan to implement these ideas in the pHPF compiler and conduct a more detailed study of the performance benefits of using this analysis. In the future, we will also investigate integrating this analysis with program transformations like limited loop-unrolling and with performance estimation for optimal insertion of synchronization.

Acknowledgements

We would like to thank Bob Cypher, M. Raghunath, and Marc Snir for valuable discussions on the use of get and put primitives.

References

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

[2] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Automatic array alignment in data-parallel programs. In *Proc. Twentieth Annual ACM Symposium on Principles of Programming Languages*, Charleston, SC, January 1993.

[3] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood. Application-specific protocols for user-level shared memory. In *Proc. Supercomputing '94*, Washington D.C., November 1994.

[4] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice University, May 1993.

[5] 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.

[6] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, K.Y. Wang, D. Shields, W.-M. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Proc. Supercomputing '95*, San Diego, CA, December 1995.

[7] 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, October 1994.

[8] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of 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.

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

[10] C. Koelbel and P. Mehrotra. Compiling global namespace parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[11] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.-H. Lim. Integrating message passing and shared-memory: Early experience. In *Proc. 4th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, May 1993.

[12] S. Midkiff and D. Padua. Compiler generated synchronization for do loops. *IEEE Transactions on Computers*, 36:1485–1495, December 1987.

[13] M. O'Boyle and F. Bodin. Compiler reduction of synchronization in shared virtual memory systems. In *Proc. 9th ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[14] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *Proc. 21st International Symposium on Computer Architecture*, April 1994.

[15] 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, June 1989.

[16] 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, July 1995.

[17] P. Tang, P. Yew, and C. Zhu. Compiler techniques for data synchronization in nested parallel loops. In *Proc. 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[18] C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proc. 5th ACM Symposium on Principles and Practices of Parallel Programming*, Santa Barbara, CA, July 1995.

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