Effective Communication Coalescing for Data-Parallel Applications

Daniel Chavarria-Miranda
Dept. of Computer Science
Rice University
danich@cs.rice.edu

John Mellor-Crummey
Dept. of Computer Science
Rice University
johnmc@cs.rice.edu

ABSTRACT
Communication coalescing is a static optimization that can reduce both communication frequency and redundant data transfer in compiler-generated code for regular, data parallel applications. We present an algorithm for coalescing communication that arises when generating code for regular, data-parallel applications written in High-Performance Fortran (HPF). To handle sophisticated computation partitions, our algorithm normalizes communication before attempting coalescing. We experimentally evaluate our algorithm, which is implemented in the dHPF compiler, in the compilation of HPF versions of the NAS application benchmarks SP, BT and LU. Our normalized coalescing algorithm improves the performance and scalability of compiler-generated code for these benchmarks by reducing the communication volume up to 53% compared to a simpler coalescing strategy and enables us to match the communication volume and frequency in hand-optimized MPI implementations of these codes.

1. INTRODUCTION
Parallel systems built from off-the-shelf microprocessors use a distributed memory organization for scalability. On such systems, access to remote data is much more costly than access to local data. For a program to achieve scalability and good performance on such systems, one must minimize the number of communication events and the volume of data transferred between processors. Automatic techniques for minimizing communication are essential when compiling programs written in a high-level data-parallel language such as High Performance Fortran (HPF) [14] for such parallel systems.

*This research was supported in part by the Los Alamos Computer Science Institute under Department of Energy Contract Nos. 0891-001-09-13, 74837-001-03-49, and/or 86192-001-04-49 from the Los Alamos National Laboratory. The computations were performed on the National Science Foundation Terascale Computing System at the Pittsburgh Supercomputing Center.

HPF compilers transform single-threaded, implicitly parallel, global address space programs by distributing data among processors according to user-specified directives, partitioning the computation onto the processors in such a way that most of the computation can be done using local data on each processor, and inserting data movement and synchronization operations.

For regular applications—in which distributed arrays are indexed by affine functions of loop indices, global variables and constants—it is possible to statically determine closedform symbolic representations of communication sets for the partitioned data-parallel program. In an HPF program, communication is needed when a processor must read or write data resident on another processor. Each reference to a distributed array must be analyzed to determine whether it is always local to the computation manipulating it.

Analyzing and synthesizing communication for each array reference in isolation might yield too many communication events and redundant data movement; this could degrade the performance of generated code. To avoid this, data-parallel compilers employ a variety of communication optimizations, including vectorization [12] and coalescing [3].

Communication vectorization is a well understood transformation that reduces communication frequency by aggregating the communication for an array reference over a range of loop iterations into a single composite event. Communication coalescing reduces redundant data transfers by merging communication events for different references to a distributed array into a single event. Unlike vectorization, coalescing is significantly complicated by the presence of complex computation partitions.

This paper presents the design and evaluation of a communication coalescing algorithm that optimizes communication for regular, data parallel programs written in a language such as HPF. Our algorithm was designed to coalesce communication for arrays distributed using BLOCK-like data partitionings, such as symbolically-sized BLOCK partitionings and multipartitionings [10], in the presence of sophisticated computation partitionings that include partially-replicated computation and non-overlapping partitions.

We implemented our coalescing algorithm in the dHPF compiler for HPF. Experiments using HPF versions of the NAS application benchmarks SP, BT and LU show that our im-
proved coalescing algorithm reduces communication volume by as much as 55% compared to a simpler strategy that only coalesces communication resulting from references with equivalent computation partitionings. In combination with other optimizations, our coalescing algorithm enables us to match the communication volume and frequency in hand-optimized MPI implementations of these codes.

Section 2 briefly describes the dHFP compiler, along with its strategies for communication partitioning and computation partitioning. Section 3 describes our coalescing algorithm. Section 4 presents an evaluation of its effectiveness. Section 5 discusses related work. Section 6 presents our conclusions.

2. THE DHFP COMPILER

The dHFP compiler is described in detail elsewhere. It is worth mentioning here two novel aspects of dHFP that have a profound impact on communication optimization, which is the focus of this paper. First, dHFP uses an abstract equational framework that enables most of its communication analysis, optimization, and code generation to be expressed as operations on symbolic sets of integer tuples. Second, dHFP supports a more general computation partitioning model than other HPF compilers: it permits independent computation partitionings for each program statement, where the partitioning for a statement maps computation of its instances onto the processors that own data accessed by one (or more) of a set of references. This partitioning model enables dHFP to support sophisticated partially-replicated partitionings, which have proven essential for achieving high performance [7, 8].

In the next two sections, we briefly describe dHFP’s sophisticated computation partitioning model to provide a context for our discussion of communication coalescing.

2.1 Computation Partitioning

HPF programs contain data alignment and distribution directives that indicate how the compiler should partition arrays in its generated SPMD (Single Program Multiple Data) code. Prior to considering the computation partitionings for each statement, we use dataflow analysis to determine which layout directives each array reference.

Next, we consider partitionings for affine assignment statements, because computation partitionings for other types of statements are principally derived from the partitioning of the assignment statements. An affine assignment in Fortran can be defined in general terms as follows:

\[ A_f (f_i (i_1, \ldots, i_n)) = g (A_r (f_r (i_1, \ldots, i_n)), \ldots, A_r (f_r (i_1, \ldots, i_n))) \]

Each array \( A_f \) may be distributed (or not) and has a unique non-negative number of dimensions (\( \geq 0 \)). The \( i_r \) are induction variables defined by a loop nest enclosing the assignment or loop invariant variables or constants. The functions \( f_r \) are affine vector-valued functions of the \( i_r \) variables (as well as global variables and constants) and \( g \) is a function of the array expressions \( A_f (f_i (i_1, i_2, \ldots, i_n)) \), where \( n \) is the number of enclosing loops for the statement.

Distributing the computation of instances of such assignments can be done by analyzing which processors own the array elements involved in the expressions on the left and the right hand sides of the assignment. The ideal situation would be to execute the assignment on the processor that owns most of the array elements involved in it. This would minimize the communication needed. Selecting this optimal partitioning might be costly [4] and requires detailed information about the relative array sizes of \( A_f \) and the \( A_r \)’s, as well as knowledge about the computation defined by \( g \). In fact, this might not be statically determinable if \( g \) is an opaque function call.

A simple heuristic to partition the computation for an affine assignment statement is to assume that the computation will be performed by the processor that owns the left-hand side (or target) of the assignment \( (A_f) \). This partitioning strategy is known as the “owner-computes rule” [18].

Using the owner-computes partitioning strategy, communication must be synthesized by the compiler for array elements which do not reside on the same processor as the left hand side target. This “owner-computes” strategy is clearly not optimal: consider the statement \( A_f (i_1, i_2) = B_1 (i_1 + 1, i_2) + B_2 (i_1 + 1, i_2) + \ldots + B_2 (i_1 + 1, i_2) \). If the arrays are distributed and aligned equivalently, and if elements \( i \) and \( i+1 \) do not reside on the same processor, applying the “owner-computes” strategy to this assignment would require sections of 24 different arrays to be communicated. Obviously, in this case, it would be better to perform the computation of the expression on the processors that own the \( i+1 \) array elements and then communicate the result to the processor owning array element \( i \). The array reference chosen for computation partitioning, is called the 

HME reference (from the HPF directive 

\[ \text{on HME} \]).

The dHFP compiler uses a more sophisticated approach than the “owner-computes” strategy [1]. dHFP considers computing an affine assignment on processors that own any subset of the array expressions involved in the assignment. For example as described, dHFP would perform the computation on the processor that owns the array elements \( i+1 \) (\( \text{on HME} B_1 (i+1, j) \)). The flexibility of dHFP’s partitioning strategy enables the compiler to partially or totally replicate computation by computing the target of an assignment for partitioned array elements on multiple processors. Computation replication can reduce communication by making the target available on multiple processors, as long as it does not require communicating more right-hand side elements.

3. COMMUNICATION COALESCING

For best performance, an HPF compiler should minimize the number and volume of messages. Analyzing and generating messages for each non-local reference to a distributed array in isolation produces too many messages and the same values might be communicated multiple times.

In dHFP, a communication set is initially computed and placed separately for each individual non-local reference. Communication set is represented in terms of an 

\[ \text{on HME} \] reference (corresponding to the computation partitioning where the data is required) and a non-local reference. Data dependencies incident on a reference and its subscripts de-
CHPF$ distribute $a(\cdot, \text{block})$, $b(\cdot, \text{block})$ onto $P$

\[
\text{do } j = 2, n \\
\text{do } i = 1, n \\
\quad a(i, j) = b(i, j - 1) + c \text{ ! ON_HOME } a(i, j) \\
\quad a(i, j) = (a(i, j) + d + b(i, j - 1)) \text{ ! ON_HOME } a(i, j) \\
\text{end do} \\
\text{end do}
\]

Figure 1: Simple overlapping non-local data references.

CHPF$ distribute $a(\cdot, \text{block})$, $b(\cdot, \text{block})$ onto $P$

\[
\text{do } j = 2, n \\
\text{do } i = 1, n - 1, 2 \\
\quad a(i, j) = b(i, j - 1) + c \text{ ! ON_HOME } a(i, j) \\
\quad a(i + 1, j) = d + b(i, j) \text{ ! ON_HOME } a(i + 1, j) \\
\text{end do} \\
\text{end do}
\]

Figure 2: Complex overlapping non-local data references

To determine the loop level at which its communication should be placed, Communication for a reference is not needed at any loop level deeper than the innermost loop carrying a true or output data dependence incident on the reference or its subscripts. This transformation is known as communication vectorization [12]. At this point, communication sets have not been vectorized. When multiple communication events are scheduled at the same location in the code, dHPF will try to coalesce them to avoid reorganizing the same data multiple times. The resulting coalesced communication events will be vectorized subsequently.

In communication sets dHPF represents the ON_HOME reference and the non-local reference in terms of value numbers that appear in their subscripts. These value numbers are computed using a global value numbering scheme [19] operating at the source level of the subroutine being compiled. Each individual subscript position in an array reference is represented by a single value number. These value numbers are expressions involving possibly multiple variables and constants.

Consider the code in Figure 1, an HPF compiler should generate a single message to communicate a single copy of the off-processor data required by both references to $b(i, j - 1)$. However, detecting when sets of non-local data for multiple references overlap is not always so simple. References that are syntactically equivalent may require identical or overlapping non-local data. In Figure 2, references $b(i, j - 1)$ and $b(i, j)$ require identical non-local values and can be satisfied by a single message. To avoid communicating duplicate values, overlap between sets of non-local data required by different loop nests should be considered as well, as shown in Figure 3.

### 3.1 Normalization

dHPF’s value-number based representation for communication sets has the disadvantage that non-local references that arise in different loops are incomparable because their subscripts have different value number representations. To enable us to detect when such references may require overlap-

CHPF$ distribute $a(\text{block}, \cdot)$, $b(\text{block}, \cdot)$ onto $P$

\[
\text{do } \text{timestep} = 1, T \\
\text{do } j = 1, n \\
\quad a(i, j) = (a(i, j) + b(i, j)) \text{ ! ON_HOME } a(i, j) \\
\text{end do} \\
\text{end do}
\]

\[
\text{do } j = 1, n \\
\text{do } i = 1, n - 2 \\
\quad a(i, j) = (a(i, j) + b(i, j)) \text{ ! ON_HOME } a(i, j) \\
\text{end do} \\
\text{end do}
\]

\[
\text{do } j = 1, n \\
\text{do } i = 1, n - 1 \\
\quad a(i, j) = (a(i, j) + b(i, j)) \text{ ! ON_HOME } a(i, j) \\
\text{end do} \\
\text{end do}
\]

Figure 3: Coalescing non-local data across loops.

<table>
<thead>
<tr>
<th>Array Ref.</th>
<th>Index Var.</th>
<th>Val. Num.</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a(i, j)$</td>
<td>$v_1$</td>
<td>$v_2$</td>
<td>$3v_1$</td>
</tr>
<tr>
<td>$(\text{non-local}) a(i + 1, j)$</td>
<td>$v_1$</td>
<td>$v_2 + 1$</td>
<td>$4v_2 + 1$</td>
</tr>
<tr>
<td>$(\text{non-local}) b(i, j - 1)$</td>
<td>$v_3$</td>
<td>$v_2 - 1$</td>
<td>$2v_2 - 1$</td>
</tr>
<tr>
<td>$a(i + 2, j)$</td>
<td>$v_2$</td>
<td>$v_2 + 2$</td>
<td>$3v_2$</td>
</tr>
<tr>
<td>$(\text{non-local}) a(i + 3, j)$</td>
<td>$v_2$</td>
<td>$v_2 + 3$</td>
<td>$4v_2 + 1$</td>
</tr>
<tr>
<td>$(\text{non-local}) b(i + 1, j)$</td>
<td>$v_2$</td>
<td>$v_2 + 1$</td>
<td>$2v_2 - 1$</td>
</tr>
<tr>
<td>$a(i + 1, j)$</td>
<td>$v_3$</td>
<td>$v_3 + 1$</td>
<td>$2v_3$</td>
</tr>
<tr>
<td>$(\text{non-local}) a(i + 2, j)$</td>
<td>$v_3$</td>
<td>$v_3 + 2$</td>
<td>$3v_3 + 1$</td>
</tr>
<tr>
<td>$(\text{non-local}) b(i + 1, j)$</td>
<td>$v_3$</td>
<td>$v_3 + 1$</td>
<td>$2v_3$</td>
</tr>
</tbody>
</table>

Table 1: Value numbers used for the array references in figure 3

ping sets of non-local data, we convert all data and ON_HOME references to use a canonical set of value numbers for the loop induction variables involved.

Figure 3 illustrates this problem. In this case, the value numbers for each induction variable in different loop nests are going to be different, even though they are used to access the same distributed dimension of the arrays $a$ and $b$. In particular, the value numbers of the subscripts used to access the distributed dimension of the different array expressions in the figure are shown in Table 1. The first column in the table shows all array references from figure 3, the second column shows the induction variable used to index the distributed dimension of the array (a distinctly subscripted variable is used for each loop nest), the third column shows a symbolic form of the value number used for the subscript expression in the distributed (first) dimension of the arrays and the last column shows the resulting range of values accessed, according to the range of the induction variable and the offset applied to it.

In figure 3, some of the array references on the right hand side of the expressions will require communication on proces-
or partition boundaries, due to the fact that they differ from their ON_HME locations. For the first loop in figure 3, both right-hand side array references, \( a(\mathbf{i} + 1, j) \) and \( b(\mathbf{i} + 1, j) \) will require communication on block boundaries. For the second loop, also both RHS array references, \( a(\mathbf{i} + 3, j) \) and \( b(\mathbf{i} + 1, j) \) will require communication. For the last loop, only the \( a(\mathbf{i} + 2, j) \) reference requires communication because the \( b(\mathbf{i} + 1, j) \) reference coincides perfectly with the ON_HME location. Due to the dependence pattern in the example code and the ON_HME locations chosen for the computation partitioning, communication for all loop nests can be placed at a single point before all three nests.

The coalescing problem then becomes determining whether different communication events for the same distributed arrays, possibly derived from multiple loop nests, overlap with one another.

To solve this problem, we developed a normalization scheme as a basis for determining when communication sets for different references overlap. To compensate for differences in computation partitionings selected for different statements, normalization rewrites the value numbers representing a communication set into a form relative to its ON_HME reference expressed in a canonical form. In our discussion of normalization, we refer to the ON_HME reference for a communication set as the computation partitioning (CP) reference and the non-local reference as the data reference.

We apply our normalization algorithm to communication sets represented in terms of value numbers based on affine subscript expressions of the form \( a_i + b \), where \( i \) is an induction variable, \( a \) is a known integer constant and \( b \) is a (possibly symbolic) constant. This restriction comes from the need to compute a symbolic inverse function for such expressions. If this restriction is not met, the communication set is left in its original form.

In our further discussion, whenever we mention a reference or the subscripts of a reference, we are referring to the value number representation of the reference or subscript.

A communication set is in normal form if:

- The CP reference is of the form \( A(i_1, i_2, i_3, ..., i_n) \)
- The data reference is of the form \( A(a_1 i'_1 + b_1, a_2 i'_2 + b_2, a_3 i'_3 + b_3, ..., a_n i'_n + b_n) \)

where \( A \) is an \( n \)-dimensional array, each \( i_k \) a value number for an induction variable or a constant, and each \( i'_k \) corresponds to a unique \( i_k \). We say that a particular subscript expression in the CP reference is normalized if it is of the form \( i'_k \), where \( i'_k \) is the value number of an induction variable or a constant.

In our example in figure 3 and table 1, the communication sets induced by the non-local array references for the first loop, are in normal form: \( a(\mathbf{i} + 1, j) \) and \( b(\mathbf{i} + 1, j) \) with respect to ON_HME \( a(\mathbf{i}, j) \). The communication sets induced by the non-local array references for the other two loops are not in normal form, because of the offsets in their CP references.

<table>
<thead>
<tr>
<th>Array Ref.</th>
<th>Norm. Subscript</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a(i, j) )</td>
<td>( v_{i_1} )</td>
<td>( 3n )</td>
</tr>
<tr>
<td>( \text{r-local} ) ( a(i + 1, j) )</td>
<td>( v_{i_1} + 1 )</td>
<td>( 4n + 1 )</td>
</tr>
<tr>
<td>( \text{r-local} ) ( b(i + 1, j) )</td>
<td>( v_{i_1} - 1 )</td>
<td>( 2n + 1 )</td>
</tr>
<tr>
<td>( a(i + 2, j) )</td>
<td>( v_{i_2} )</td>
<td>( 3n )</td>
</tr>
<tr>
<td>( \text{r-local} ) ( a(i + 3, j) )</td>
<td>( v_{i_2} + 1 )</td>
<td>( 4n + 1 )</td>
</tr>
<tr>
<td>( \text{r-local} ) ( b(i + 1, j) )</td>
<td>( v_{i_2} - 1 )</td>
<td>( 2n + 1 )</td>
</tr>
<tr>
<td>( a(i + 1, j) )</td>
<td>( v_{i_3} )</td>
<td>( 2n )</td>
</tr>
<tr>
<td>( \text{r-local} ) ( a(i + 2, j) )</td>
<td>( v_{i_3} + 1 )</td>
<td>( 3n + 1 )</td>
</tr>
<tr>
<td>( \text{r-local} ) ( b(i + 1, j) )</td>
<td>( v_{i_3} )</td>
<td>( 2n )</td>
</tr>
</tbody>
</table>

Table 2: Normalized value numbers used for the array references in figure 3

If a communication set is not in normal form but meets our restriction of affine subscript expressions, we normalize it by computing symbolic inverse functions for each non-normalized CP subscript position. That is, if a CP subscript position has the non-normal form \( f(i_k) = a_k i_k + b_k \), we compute a symbolic inverse function \( f^{-1}(j) = \frac{j - b_k}{a_k} \). We then apply this function to each subscript position in both the CP and data references. This function application is performed in terms of the symbolic composite value numbers of each subscript. After this step, only the data reference has subscripts of the form \( a_i j_i + b_i \). Each subscript position gets a canonical value number represented by a synthetic induction variable that has the range and stride of the original \( a_k i_k + b_k \) subscript expression. Table 2 illustrates the normalization process for our example code in figure 3, the first column shows the original array reference, the second column shows the transformed subscripts for the reference, and the third column shows the ranges for the synthetic value numbers. As can be observed in the third column, the ranges of the synthetic value numbers correspond to the ranges of the value number expressions derived from the original loop nests. The ranges of the value numbers will become extremely important in the coalescing step of our algorithm.

Figure 4 shows pseudocode for our normalization algorithm for affine subscripts.

### 3.2 Coalescing Normalized Communication

Following normalization, we coalesce communication events to eliminate redundant data transfers.

There are two types of coalescing operations that can be applied to a pair of communication events:

- **Subsumption**: Identify and eliminate a communication event (nearly) completely covered by another.
- **Union**: Identify and fuse partially overlapping communication events, which do not cover one another.

Both operations eliminate communication redundancy, however these two cases are handled separately at compile time.

For both cases, the coalescing algorithm first tests to see if communication events are compatible. To be compatible, both communication events must correspond to the same distributed array, be reached by the same HPF alignment and distribution directives, have the same communication
for each communication event
  if the CP reference is not in normal form and
  loop induction variables appear at most once
  in all subscripts of the CP and data references then
    for each subscript \( f(i) = \alpha i + \beta \) in the CP reference
      let \( f^{-1}(i) = \alpha^{-1} i - \beta \) (symbolic inverse of the
      \( \alpha i + \beta \) subscript)
      apply \( f^{-1}(i) \) to the unique position in the
      CP reference where \( i \) is used
      this subscript now becomes \( f^{-1}(f(i)) = i \)
    let \( g(i) = \gamma i + \delta \) be the subscript of the data
    reference where \( i \) is used
    apply \( f^{-1}(i) \) to the unique position in the
    data reference where \( i \) is used
    this subscript now becomes \( f^{-1}(g(i)) \)
  end do
end do

Figure 4: Normalization Algorithm

for each location in the code where communication
events have been placed
  \( osets = \emptyset \) // array of processed comm. event unions
for each communication event \( c \)
  normalize communication event \( c \)
  if \( c \) is subsumed by any event in \( osets \) then
    discard \( c \); continue with next event
  else if \( c \) partially overlaps with any event in union \( u \)
    in \( osets \) then
    add \( c \) into union \( u \)
    record the ranges of the value numbers of \( c \)
  else
    construct a new union \( u \) with \( c \) as its only element
    add \( u \) to \( osets \)
  end if
for each union \( u \) in \( osets \)
  postprocess \( u \)

Figure 5: Coalescing Algorithm

type (read or write) and be placed at the same location in
the intermediate code.

Figure 5 presents pseudocode for the static coalescing algo-
rithm used in dHPF to reduce overlap between communi-
cation events. The subsumption, union and postprocessing

CHPF$ distribute a(*, block), b(*, block) onto P
  do j = 2, n
    do i = 1, n
      a(i, j) = b(i, j - 2) + c ! ON_HMEM a(i, j)
      a(i, j) = d + b(i, j - 1) ! ON_HMEM a(i, j)
    end do
  end do

Figure 6: Different shift widths for the same array.

operations appearing in the algorithm are described in the
following sections.

3.2.1 Subsumption

Previous analysis during dHPF’s initial communication place-
ment phase, identified two very important components of
each communication event:

- Non-local dimensions: a non-local dimension is a dis-
  tributed dimension in which the data reference has a
different value number for the corresponding subscript
  than the CP reference.

- Shift widths: shift widths are computed for non-local
dimensions as the constant (possibly symbolic) differ-
  ence between the CP reference and the data reference
  in terms of their value numbers.

The subsumption of one communication event by another
requires that the non-local data of the subsuming event be
a superset of that of the subsumed one. This implies that
the non-local dimensions of the subsuming event should be a
superset of the non-local dimensions of the subsumed event.

Shift widths represent the width of the communication sets
in the communication dimension for communication events
that have been vectorized out of all loops traversing dis-
tributed dimensions of the arrays. Figure 6 shows a loop
nest where two non-local references to array \( b \) have differ-
ent shift widths. The first non-local reference, \( b(i, j - 2) \)
has a shift width of 2, while the second non-local refer-
ence, \( b(i, j - 1) \) has a shift width of 1. These correspond
to the constant differences between the respective CP and
data references.

Figure 7 shows non-local data being accessed for two array
references with different shift widths. The original source
loop nest is shown on the top of the figure. Below the loop,
a diagram shows the data partition of both arrays \( a \) and \( b \) for
two adjacent processors. After this, a table shows the non-
local data access for the array references \( b(i, j + 1) \) and
\( b(i, j + 2) \) for the index values \( k - 1 \) and \( k \). In this case,
it is possible to coalesce the communication for both non-
local array references into a single event that covers their
needs. If communication is fully vectorized, then a single
coaesced communication event will send two full columns
of data from processor \( p + 1 \) to processor \( p \).

Following our example from figure 3 and table 2, the only
non-local dimension is the first one, indexed by the \( i \) vari-
able in the three loop nests. The shift widths for the dif-
ferent non-local array references are shown in table 3. The
widths represent the constant differences, in the distributed
CHPF$ distribute a(*, block), b(*, block)
do \ j = 1, n - 2
  do \ i = 1, n
       a(i, j) = b(i, j + 1) + b(i, j + 2)
  end do
end do

k + 1

processor \ p

k + 1

processor \ p + 1

Non-local data accessed for b(i, j + 1) & b(i, j + 2) for proc. \ p:

\[
\begin{align*}
b(i, j + 2) &\quad \{ 1 \leq i \leq n \quad \text{and} \quad k + 1 \leq j \leq k + 2 \} \\
b(i, j + 1) &\quad \{ 1 \leq i \leq n \quad \text{and} \quad j = k + 1 \}
\end{align*}
\]

Figure 7: Non-local data accessed for multiple references.

<table>
<thead>
<tr>
<th>Array Ref.</th>
<th>Shift Width (first dim.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>a(i + 1, j)</td>
</tr>
<tr>
<td>Original</td>
<td>b(i + 1, j)</td>
</tr>
<tr>
<td>Original</td>
<td>a(i + 3, j)</td>
</tr>
<tr>
<td>Original</td>
<td>b(i + 3, j)</td>
</tr>
<tr>
<td>Original</td>
<td>a(i + 2, j)</td>
</tr>
<tr>
<td>Original</td>
<td>b(i + 1, j)</td>
</tr>
</tbody>
</table>

Table 3: Shift widths for the array references in figure 3

dimension, between the data reference and the respective CP reference for each possible non-local access.

For two compatible shift communications, where the shift width of one is larger than that of the other, the volume of data transferred can be reduced by completely eliminating the smaller shift since its data values will be provided by the larger one. For dimensions not involved in a shift, the ranges of the data reference subscript value numbers in the subsumed communication event must be supersets of the corresponding ranges in the subsumed event. For data dimensions involved in a shift, subsumption does not require that their range be a strict superset of the corresponding ranges in the subsumed event. If the subsuming communication has a wider shift width, but doesn’t have a large enough range for its loop induction variable, the coalescing algorithm synthesizes a new induction variable with extended range to cover the induction variable in the subsumed event as well. This range-extension technique is very effective for avoiding partially-redundant messages and generating simple communication code. Subsumption in this form is only applicable when the shift width is smaller than the block size of the data owned by each processor.

In our example in figure 3 and table 2, normalization and subsumption will produce two communication events with the following characteristics:

- Communication event of shift width 1 for array a (data sent from the right neighbor), with value numbers \( v_0 + 1 \), where \( i' \) is a synthetic value number with a range of \( (3 : n + 1) \). This event provides the necessary non-local data for the original references \( a(i + 1, j), a(i + 3, j) \) and \( a(i + 2, j) \).

- Communication event of shift width -1 for array b (data sent from the left neighbor), with value numbers \( v_0 - 1 \), where \( i' \) is a synthetic value number with a range of \( (2 : n - 1) \). This event provides the necessary non-local data for the original references \( b(i + 1, j) \) and \( b(i + 1, j) \).

### 3.2.2 Union

Coalescing partially overlapping communication events requires less strict conditions than subsumption. Given normalized data and CP references, the coalescing algorithm will only try to union communication events that have common shift dimensions and directions.

The CHPF compiler uses the Omega library [17] to implement operations on integer tuples for its communication analysis and code generation. We apply integer set-based analysis to determine the profitability of unioning two communication events. We construct sets that represent the global, non-partitioned data accessed by the non-local references of each communication event. If these sets intersect, this implies the communication sets for some pair of processors may intersect. In this case, the algorithm will coalesce the two communication events by unioning them. If the sets do not intersect, then the communication sets for any pair of processors also do not intersect, which implies there is no redundancy to eliminate, so coalescing is not performed. Keeping non-overlapping sets separate leads to simpler generated code.

Figure 8 shows two loop nests with a non-local array reference each. The shift widths of both references are the same (2), but the global ranges of the loop nests are different. In particular, the ranges for the non-distributed dimension are different. Figure 9 shows the communication sets for both loop nests and how they partially overlap. Constructing a single set that covers the non-local data needs of both loop nests, eliminates the communication of redundant data elements.

As described in the previous section, generation of a coalesced communication set can require that new induction
variables be synthesized with extended range. When unioning communication sets, this transformation is applied to an induction variable appearing in any dimension as necessary.

3.3 Postprocessing

After constructing the communication sets that reduce redundancy by applying subsumption and union, as described previously, these sets have to be postprocessed to convert them into a form suitable for code generation.

The most important postprocessing step is to create a value number-based synthesized CP reference for unions of communication events. This reference has to use synthetic value numbers for all of its subscript components that have a large enough range to cover the needs of each data reference in the union of communication events.

The construction of these value numbers is critical for further processing of the communication sets, since the Omega integer sets generated for each communication event are based on the synthesized value numbers and their ranges.

4. EXPERIMENTAL EVALUATION

We present a detailed experimental evaluation of our coalescing algorithm, using the NAS SP, BT, and LU application benchmarks [2]. We compare the benefit of using our coalescing algorithm versus not coalescing communication at all, and also against a simpler coalescing algorithm which does not perform normalization.

To compare the effectiveness of our compiler-based approach, we also include a comparison of dHPF's generated code against the parallel hand-coded versions of these benchmarks.

All experiments were run on the Lumieux Alpha cluster at the Pittsburgh Supercomputing Center. The cluster uses a Quadrics interconnection network and runs HP's Tru64 UNIX version. All codes were compiled using HP's native compilers for this platform, and linked against a Quadrics-aware MPI library.

4.1 NAS SP and BT Application Benchmarks

The NAS SP and BT application benchmarks [2] are tightly-coupled computational fluid dynamics codes that use line-sweep computations to perform Alternating Direction Implicit (ADI) integration to solve discretized versions of the Navier-Stokes equation in three dimensions. SP solves scalar penta-diagonal systems, while BT solves block-tridiagonal systems. Both codes perform an iterative computation. In each time step, the codes have a loosely synchronous phase followed by tightly-coupled bidirectional sweeps along each of the three spatial dimensions. These codes have been widely used to evaluate the performance of parallel systems. Sophisticated hand-coded parallelizations of these codes developed by NASA provide a yardstick for evaluating the quality of code produced by parallelizing compilers.

The NAS 2.3-serial versions of the SP and BT benchmarks each consists of more than 3500 lines of Fortran 77 sequential code (including comments). To these, HPF directives were added, which account for 2% of the line count. The main arrays were distributed using a 3D multipartitioning [9]. More details on how these benchmarks were compiled using dHPF can be found in [8] and [5].

Figures 10 and 11 compare the parallel efficiency of different versions of dHPF-generated code for the SP class 'C' (162\textsuperscript{2} problem size) and BT class 'B' (102\textsuperscript{3} problem size) benchmarks, respectively (BT class 'B' was chosen instead of class 'C' due to memory limitations). Parallel efficiency

1\textsuperscript{d}dHPF is a source-to-source compiler, generating SPMD Fortran + MPI code from HPF source.

2\textsuperscript{2}The dHPF-generated code for the four principal routines in SP and BT (computerhs, x_solve, y_solve, and z_solve) was postprocessed by hand to change the use of global pointer variables for the distributed arrays into subroutine arguments, to improve aliasing information for the Alpha backend compiler. This results in a performance improvement of 20%.

Figure 10: Parallel Efficiency Comparison for SP class 'C' (162\textsuperscript{2}).
is defined as speedup divided by the number of processors. Perfect speedup corresponds to an efficiency value of 1.0. The first column in each chart shows the parallel efficiency of the version using the normalized coalescing algorithm presented in the previous sections (‘dHPF’); the second column shows the efficiency of a dHPF-generated version that uses a simple coalescing algorithm which does not perform normalization (‘un-dHPF’); the last column shows the efficiency of a dHPF-generated version which does not use coalescing at all (‘nc-dHPF’).

For both benchmarks, the efficiency of the ‘un-dHPF’ and the non-coalesced (‘nc-dHPF’) versions drops as the ratio of communication to computation becomes larger when using more processors. The efficiency of the version using fully normalized coalescing (‘dHPF’) remains relatively constant across the range of processors used (1-100). We were not able to collect data for the 49 processor SP ‘un-dHPF’ version, due to an unexpected error produced by interaction between the simple coalescing algorithm and other optimizations in dHPF.

For SP class ‘C’, the average efficiency is 87% for the fully coalesced ‘dHPF’ version, 74% for the coalesced without normalization ‘un-dHPF’ version, and 70% for the non-coalesced ‘nc-dHPF’ version. For BT class ‘B’, the average efficiency is 77% for the fully coalesced ‘dHPF’ version, 73% for the coalesced without normalization ‘un-dHPF’ version, and 61% for the non-coalesced ‘nc-dHPF’ version.

Table 4 illustrates the effect of coalescing on total communication volume and frequency for the SP and BT benchmarks running on 16 processors. The MPI hand-coded version of the benchmarks is used as a base line and the communication volume and frequency of the three dHPF-generated versions are compared against it. The communication frequency (number of messages) for SP does not increase, even for the non-coalesced version, because dHPF was able to apply communication aggregation which fuses together messages going between the same pair of processes without any overlap analysis of their contents. This optimization does not reduce the communication volume. For BT, dHPF was not able to apply this optimization (for both ‘un-dHPF’ and ‘nc-dHPF’ versions), thus resulting in an increased message frequency.

Communication volume and frequency were measured using the `srun` and `cperf` tools from SGI's SpeedShop on a 16 processor Origin 2000, executing 10% of the total iterations of each code (40 for SP, 20 for BT). We collected these non-time sensitive measurements on a different platform due to the availability of automatic MPI measuring tools.

Table 4 demonstrates that the good parallel efficiency exhibited by the fully coalesced dHPF versions is due to significant reductions in overall communication volume and frequency, coming from our improved normalized coalescing algorithm.
Figures 12 and 13 show comparisons between the parallel efficiency of the MPI hand-coded and fully coalesced dHPF versions of the SP and BT benchmarks, respectively. The average efficiencies of the MPI and dHPF versions of SP are 86% and 87% respectively, while scaling well from 1 up to 100 processors. The average efficiencies of the MPI and dHPF versions of BT are 105% and 77%, also scaling well from 1 up to 100 processors. The performance difference observed for the BT benchmark is that the MPI hand-coded version scales superlinearly, starting at 25 processors, while the dHPF-generated version continues to scale only linearly. We believe this effect is due to the backend compiler handling the constant array sizes and common block-based MPI implementation of the benchmarks for each number of processors, better than how the backend compiler handles the code dHPF generates (single source for any number of processors, symbolic array sizes, Cray pointer-based memory allocation).

4.2 NAS LU Application Benchmark

LU solves the same 3D Navier-Stokes equation as SP and BT. LU implements the solution by using a Successive Over-Relaxation (SSOR) algorithm which splits the operator of the Navier-Stokes equation into a product of lower-triangular and upper-triangular matrices (see [2] and [11]).

The algorithm solves five coupled nonlinear partial differential equations, on a 3D logically structured grid, using an implicit pseudo-time marching scheme. It is a challenging application to parallelize effectively due to the potential for generating many small messages between processors.

The application starts by computing the elements of the triangular matrices in the subroutines jac3d and jacu respectively. The next step is to solve the lower and upper triangular systems, using subroutines bits and buts. After these steps, the variables are updated, a new right-hand side is computed and the process repeats inside a time-step loop.

The NAS 23-serial version of this benchmark consists of approximately 3700 lines of Fortran 77 source code (including comments). We added HPF data layout and other directives, for a total of 136 lines. This application did not require restructuring for compilation with the dHPF compiler. The addition of the HPF directives was enough.  

We use a 2D BLOCK distribution with a symbolic partition onto powers-of-two numbers of processors (1, 2, 4, 8, 16, etc.). Multipartitioning [9] cannot be applied to this problem because the parallelism is not by-line, instead parallel execution happens along a waveform determined by a diagonal hyperplane on the 3D domain.

The routines bits and buts exhibit carried communication on 2D-BLOCK partitioned data. We have implemented a complete-time option in dHPF that enables the programmer to specify that she wants aggressive communication hoisting

Table 5: Communication volume and frequency for LU

<table>
<thead>
<tr>
<th></th>
<th>MPI</th>
<th>dHPF</th>
<th>un-dHPF</th>
<th>no-dHPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 proc. vol.</td>
<td>100.00%</td>
<td>101.56%</td>
<td>101.26%</td>
<td>101.78%</td>
</tr>
<tr>
<td>16 proc. freq.</td>
<td>100.00%</td>
<td>99.91%</td>
<td>99.91%</td>
<td>99.91%</td>
</tr>
</tbody>
</table>

Figure 14: Parallel Efficiency Comparison for LU class 'C' (1623).

Figure 14 compares the parallel efficiency of different versions of dHPF-generated code for the LU class 'C' (1623 problem size) benchmark. The first column in the chart shows the parallel efficiency of the version using the normalized coalescing algorithm presented in the previous sections (dHPF); the second column shows the efficiency of a dHPF-generated version that uses a simple coalescing algorithm which does not perform normalization (un-dHPF); the last column shows the efficiency of a dHPF-generated version which does not use coalescing at all (no-dHPF).

For LU the efficiency of the coalesced without normalization (un-dHPF) and the non-coalesced (no-dHPF) versions drops slightly as the ratio of communication to computation becomes larger when using more processors, compared to the fully coalesced version (dHPF). This happens because in this benchmark there are fewer opportunities to apply communication coalescing, in comparison to BT and SP. Also, LU exhibits an important degradation in efficiency due to algorithmic factors in the application itself: the code uses coarse-grained communication pipelining to exploit waveform parallelism in the code.

Table 5 illustrates the effect of coalescing on total communication volume and frequency for the LU benchmark running on 16 processors. The MPI hand-coded version of the benchmark is used as a base line and the communication volume and frequency of the three dHPF-generated versions are compared against it. This set of data was also collected.
using SGI's tools, executing 10% of the total iterations (25 for LU).

In LU, the opportunities for coalescing are few and produce small reductions in communication volume, leading to small differences in execution time.

![Efficiency LU class 'C'](image)

Figure 15: Parallel Efficiency for MPI & dHPF LU class 'C' (16^2).

Figure 15 shows a comparison between the parallel efficiency of the MPI hand-coded and fully coalesced dHPF versions of LU. The average efficiencies of the MPI and dHPF versions of LU are 98% and 77% respectively. The MPI version exhibits better efficiency than the dHPF-generated version due to similar factors as mentioned previously for BT: better blocked compiler handling of the constant array sizes and compile-time block-based MPI implementation of LU for each number of processors, compared to the parameterized symbolic code dHPF generates.

5. RELATED WORK

Gupta et al. [13] discuss the design and implementation of the pHPF compiler done at IBM Research. pHPF can handle symbolic process numbers and by extension, symbolic block sizes. Their compiler also supports message vectorization and message coalescing but in a more restricted format than what we have implemented in dHPF, due to their system being restricted to the use of the owner-computes rule for partitioning computation.

Kandemir et al. [16] present the techniques implemented inside their HPF compiler to globally optimize communication. They use a data-flow approach combined with linear algebra techniques to accurately reduce communication volume and frequency across whole HPF programs. Their framework supports message vectorization, message coalescing and redundancy elimination across multiple loop nests. Their approach is limited because they require compile-time knowledge of the exact number processors under which the program will be executed. In comparison to dHPF's techniques, their approach has the advantage of global communication scheduling and movement, while dHPF is loop nest-based. dHPF also supports symbolic communication set analysis and code generation for statically unknown processor counts. More importantly, their framework does not support partially replicated computation partitionings, which previous work on dHPF has shown to be of critical importance to reduce communication requirements.

Iwahata, Sueyasu and Kaniya [15] and Schmeel et al. [20] discuss the extensions proposed in Japan to the HPF language (HPF/JA). These extensions focus on improving the quality of the generated code through the use of lower level directives, than those in the core HPF language. These directives specify explicit communication and locality properties for distributed arrays, as well as a form of partial computation replication based on computing on halo regions surrounding processor tiles.

Our work in [7] and [8] presents and quantitatively evaluates the impact of several partitioning, communication and memory hierarchy optimizations in the dHPF compiler. These optimizations enable performance comparable to hand-coded applications. We presented normalized coalescing and its impact on application performance, but did not provide a detailed, algorithmic description of how dHPF performs communication coalescing on codes with complex computation partitionings.

6. CONCLUSIONS

We have presented an algorithm for coalescing communication that arises when generating code for regular, data-parallel applications written in HPF. This algorithm, which has been implemented in the dHPF compiler, is effective in reducing the overlap between messages in the presence of complex computation and data partitionings. We evaluated the effectiveness of our normalized coalescing algorithm for optimizing communication that arises when compiling HPF versions of the NAS application benchmarks SP, BT and LU. For SP, our normalized coalescing algorithm yielded 55% lower communication volume than coalescing without normalization. For BT, our normalized coalescing algorithm reduced communication volume by 26% and communication frequency by 51% compared to coalescing without normalization. For LU, there were few opportunities for coalescing communication with or without normalization. For all three applications, our coalescing algorithm was able to achieve communication volume and frequency within 1-2% of those found in hand-optimized MPI implementations of these applications. For SP, normalized coalescing significantly boosted scalability and performance. This provides strong evidence for the effectiveness of our approach.

7. REFERENCES


