An Algebraic Machinery for Optimizing Data Motion for HPF

JAN-JAN WU AND MARINA C. CHEN

1 Institute of Information Science, Academia Sinica, Taipei, Taiwan 11529; e-mail: wuj@iis.sinica.edu.tw
2 Computer Science Department, Boston University, 111 Cummington Street, Boston, MA 02215, USA; e-mail: mcchen@cs.bu.edu

ABSTRACT

This paper describes a general compiler optimization technique that reduces communication overhead for FORTRAN-90 (and High Performance FORTRAN) implementations on massively parallel machines.

The main sources of communication, or data motion, for the parallel implementation of a FORTRAN-90 program are array assignments, array operators (e.g., CSHIFT, TRANSPOSE, etc.), and array parameters passing to and from subroutines. Coupled with the variety of ways arrays can be distributed, a FORTRAN-90 implementor faces a rich space of possibilities by which data motion can be organized.

We propose a unified framework for optimizing intra- and inter-procedural data motion. The central idea of this framework is algebraic analysis of data motion. We give an algebraic representation for each HPF’s array intrinsics and data distribution specifications. An array reference extracted from the source FORTRAN-90 program, given a particular data distribution specification, is represented as a communication expression, which in turn can be simplified according to a communication algebra. Fast communication is uncovered by pattern matching with a set of communication idioms. Experimental results on the Connection Machine CM-5 demonstrating the effectiveness of this approach are reported.

1 INTRODUCTION

Massively parallel machines offer the potential teraflop computing power. Such power cannot be fully utilized until such machines are made easy to program. A major difficulty of this class of machines is the need to distribute data and manage interprocessor communication explicitly. In recent years, much research effort has been devoted to providing suitable programming tools. One subject on which research focusses is the provision of appropriate high-level, data-parallel programming languages to easy parallel programming. In 1992, a coalition of researchers from academies, industry and governmental labs formed the High Performance Fortran Forum to develop a standard set of language extensions to Fortran 90. The forum has produced a proposal for a language, called High Performance Fortran (HPF) [24], which extends the Fortran 90 standard with data distribution directives for high performance target machines such as massively parallel machines and workstation clusters.

One of the main factors in achieving high performance for parallel programs on massively parallel machines is the reduction of communication overhead. While the development of optimizing compilers for super-scalar architectures is becoming commonplace in the industry, work on optimization for data movement and node code performance is mostly done in the context of specialized, hand-crafted code written in assembly code, if not mi-
crocode, for specific target machines (e.g, TMC’s Convolution Compiler for stencil computation [7]). Automatic transformations to reduce data movement have become an important issue for HPC architectures, where the time spent communicating can easily outweigh the time spent performing actual arithmetic.

In this paper, we describe a general compiler optimization technique that reduces communication overhead for Fortran-90/HPF implementations on massively parallel machines. Movement of distributed data in HPF can occur in two ways:

1. passing of distributed arrays in procedure calls. Note that the actual and dummy arguments may have different data distribution;
2. assignments on distributed arrays within a procedure body. Again, the LHS (left-hand side) and the RHS (right-hand side) of an assignment statement may have different data distribution.

So there are short-range (between LHS and RHS), medium-range (between a LHS or RHS and the alignment and distribution that are in force in the current scoping block), and long-range (actual to dummy across the procedure calling sequence) effects of data layout to consider. This makes compilation of HPF to efficient target code a complex task.

We propose an algebraic transformation and runtime support technique for reducing intra- and inter-procedural data movement in HPF programs. The theoretical framework within which we designed program transformations is algebraic analysis of data movement. We give each of HPF’s array operations and data distribution directives an algebraic representation. We then formalize data distribution, intra-procedural and inter-procedural data movement using communication expressions. We have developed a communication algebra and its associated heuristics to simplify communication expressions, and a hand-coded, optimized runtime communication library to carry out aggregate communication. Fast communication is uncovered by pattern-matching with a set of communication idioms. Calls to fast communication are generated if pattern matching is successful; otherwise, a general communication routine using direct send/receive is used.

The algebraic transformations are done abstractly in the logical, global space defined in the program. It does not require the notion of processor IDs and local memory offsets. As a result, although being demonstrated in the context of an HPF compiler for the Connection Machine CM-5, the optimization technique is applicable to other data-parallel languages and other massively parallel machines.

We have conducted experiments on the Connection Machine CM-5 to evaluate the effectiveness of this optimization. Our experimental results demonstrate significant performance improvement over benchmark codes.

The rest of the paper is organized as follows. Section 2 reviews the data mapping model of HPF and the associated data movement problem. Section 3 gives an overview of the algebraic transformation framework. Section 4 presents in more detail the communication algebra. Section 5 reports our experimental results on the Connection Machine CM-5. Finally, Section 6 reviews related work and Section 7 gives the conclusions.

2 HIGH PERFORMANCE FORTRAN

2.1 Data Mapping

There is a two-level mapping of array elements to logical processors. An HPF user can align array elements to a template, which is then partitioned and distributed onto an array of logical processors. The mapping of logical processors to physical processors is implementation dependent and may be specified by optional physical-mapping directives. We will use the term "data layout" as a generic term for the composition of the alignment and distribution (and physical mapping, if given explicitly).

Figure 1 shows an HPF program segment and a corresponding graphical representation of the effect of data mapping. Two arrays A and B are related by the FORTRAN-90 array intrinsic CSHIFT which shifts array A toward the negative direction by one element in wrap-around fashion. Assuming the number of processors is two and the logical processors are mapped in an obvious way to the physical processors (that is, the first block is assigned to processor P1 and the second block to processor P2), the alignment of the two arrays is intended for the eventual reduction of the interprocessor communication (only one element out of four references in the statement

$$B = \text{CSHIFT}(A, \text{dim}=1, \text{shift}=1)$$

requires communication).

2.2 Procedure Interfaces

HPF provides a rich set of procedure interface specifications for distributed array arguments. A dummy argument has a data layout that is either explicitly specified via user directives, or implicitly inherited from the caller’s actual argument. Similarly, an actual argument may have an explicitly prescriptive data layout, or inherits itself a data layout from yet another calling context. Consider the possibility of extended calling chains (‘A’ calls ‘B’ calls ‘C’, and so forth, each of which may add additional directives to the data layout specification and inherits the rest). This makes compilation of efficient data movement a complex task.
2.3 Optimization for Data Movement

Consider the case where both the actual and dummy arguments of a single-level procedure call have explicit data layouts (Figure 3). Array A is cyclically partitioned in procedure ALPHA, and should be redistributed using block partition within procedure BETA. Layout conversion for array B is a complex one due to change of both alignment (changing between offsetting two elements and offsetting one element) and distribution (changing between cyclic partition and block partition). In the procedure body of BETA, the assignment statement shifts array C toward the negative direction by one element and assigns the result to array D (i.e., C(i+1) is assigned to D(i)). We call this logical data movement defined by the program. Due to the effect of the alignment directives, actual data movement for executing the assignment statement may be different from the logical data movement as it appears. In order to satisfy the directives as given by the user, the compiler must combine all the layout requests that are in force.
Assuming "owner-compute" rule for the compilation of data movement, the compiler has two roles. First, the compiler should minimize time spent in moving A and B to C and D's preferred layouts, and moving them back when returning from the call. For instance, the compiler should optimize communication for BLOCK and CYCLIC data redistribution. It should also determine the order in moving data for complex data movement. For instance, layout conversion for passing array B to procedure BETA involves an offsetting alignment change and a conversion from CYCLIC distribution to BLOCK distribution. There are two alternatives in arranging data movement: offsetting realignment under CYCLIC distribution followed by CYCLIC-to-BLOCK redistribution, or CYCLIC-to-BLOCK redistribution followed by offsetting realignment under BLOCK distribution. The latter is more efficient because offsetting realignment requires much less communication under BLOCK distribution. Secondly, the compiler should minimize time spent in moving data array D for the execution of the assignment statement. For instance, with compiler optimization, the logical data movement specified by the EOSHIFT operation can be turned into local memory accesses as a result of the alignment directives for arrays C and D.

A simple but naive approach is to use general communication or array copying through temporary storage whenever non-canonical* data layouts is in force. This approach not only causes excessive data copying but also ignores many opportunities for fast communication. A better approach is to analyze data movement systematically. This can be achieved if we capture and manipulate data movement algebraically.

In this paper, we propose an algebraic analysis framework for optimizing data movement. Data layouts and data movement caused by passing distributed arrays between procedure boundaries, and array intrinsic traffic within procedure bodies can all be captured algebraically. Data movement can be reduced by simplifying corresponding algebraic expressions according to a set of rules. The manipulation of algebraic expressions can be carried out in the global, logical space defined in the program. Detailed, machine-dependent aspects of data movement can be postponed to runtime through a set of communication library routines.

### 3 ALGEBRAIC ANALYSIS FRAMEWORK

Figure 4 gives an overview of the framework. Data layouts and data movement in HPF codes are extracted and formalized as communication expressions. Reducing the terms in such an expression one by one from right to left does not eliminate unnecessary communication. Under the "owner compute" rule, the amount of communication is determined by the number of operators in a communication expression; i.e., a communication expression is simplified if its operator count is decreased. Our goal is to minimize the operator count. We design an algebraic engine which contains a set of rules for simplifying communication expressions and the associated heuristics which guide the engine in applying the rules. A communication expression is reduced to an expression that no further rules will be applied under the particular set of heuristics.

In the following, we describe the algebraic framework in more detail. We first describe the algebraic representations of data layout and array intrinsics. We then show what a communication expression might look like. Next, we give an overview of the communication algebra and outline the set of communication idioms we have collected.

#### 3.1 Algebraic Representations

General integer matrix notations are commonly used for affine alignment functions, which lead to a general transformation technique called affine transformation for optimizing data-parallel programs [3, 46]. General matrix notations and affine transformations are insufficient
for transforming non-linear alignment operators such as CSHIFT (cyclic shift), CSKEW (cyclic skew), and replication. A solution is to separate boundary array elements from interior elements using explicit loop structures followed by index function transformation within the loop bodies. However, this approach may increase program size rapidly when large number of non-linear operations occur in the source program.

Our approach is to associate each HPF alignment/distribution operation with a special algebraic representation. Each representation has a name and a matrix/vector/integer parameter as appropriate. The name characterizes the value of the parameter, and facilitates the design of the algebraic rules as well as efficient pattern matching for communication idioms. Formally speaking, these algebraic representations are functions with dependent types, which will become clear later.

We divided these algebraic representations into several classes according to the dimensionalities of their arguments and results. ALIGN1 operators capture alignments within a single dimension, including offsetting, strided, and reflection alignments. ALIGN2 and ALIGN3 operators capture alignments across multiple dimensions. The argument and result of an ALIGN2 operator have identical dimensionality. Typical examples are transposing an array (i.e. dimension permutation) and skewing an array dimension with respect to others. An ALIGN3 operator has different shapes for its argument and result. Array reshaping, replication, and embedding into higher-dimensional space all belong to ALIGN3. Figure 5 gives graphical representations of these operators. Table 1 outlines these classes, the HPF operators, and the corresponding algebraic representations and their definitions. In the following, we explain some of the definitions.

In ALIGN1, both the array intrinsic

EOSHIFT(A, dim, shift=-c)

and the alignment directive

ALIGN A(i) with T(i+c)

offsets array A at the positive direction by distance c, therefore both are denoted by EOSHIFT(c), which, when given an index domain D with lower bound lb(D) and upper bound ub(D), maps D to anew index domain with lower bound lb(D) + c and upper bound ub(D) + c. The CSHIFT operator performs offsetting operations in a wrap-around fashion.

In ALIGN2, both the TRANSPOSE array intrinsics (e.g., TRANSPOSE(A, (1, 3, 2)), which exchanges the second and the third dimensions of array A) and the “transpose” alignment directives (e.g.,

ALIGN A(i,j,k) with T(i,k,j))

are denoted by TRANS\(^{(d)}\)(M), which, when given d-dimensional index domain D, permutes dimensions of D according to M (which is the matrix representation for the permutation vector and the alignment index expression) and results in another index domain M(D). The SKEW\(^{(d)}\)(M) operator, when given d-dimensional index domain D, skews D (i.e., performs affine mapping on domain D) according to the coefficient matrix M. For instance, the operator

\[
\text{SKEW}^{(2)}\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}
\]

skews the second dimension of a two-dimensional domain (i.e. it maps index (i, j) to (i, i + j)). The CSKEW\(^{(d)}\)(M)

operator performs skewing operations in a wrap-around fashion.

In ALIGN3, both the array intrinsic

SPREAD(A, dim=2, ncopies=n)

and the alignment directive

ALIGN A(i) WITH T(i, *)
**CLASSES**

**ALIGN1**

- EOShift($A$, shift=-1)
  - end-of-shift

- CShift($A$, shift=-1)
  - cyclic shift

**ALIGN2**

- Transpose($A$)
  - transposition

- Skew($A$, $\begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$)
  - skewing along the second dimension

**ALIGN3**

- Align $A(i)$ with $B(1,i)$
  - embedded alignment

- B = Reshape($A$, $s1 = (2,3), s2 = 6$)
  - reshape alignment with default ordering (column-major)

**MULTID ALIGNMENT**

- EOShift EOShift($A$, dim=1, shift=1), dim=2, shift=-1)
  - two-dimensional shift

- CShift(CShift($A$, dim=1, shift=1), dim=2, shift=-1)
  - two-dimensional cyclic shift

**MULTID DISTRIBUTION**

- Distribute $T(*,BLOCK)$
  - column partition

- Distribute $T(BLOCK,BLOCK)$
  - 2D block partition

- Distribute $T(BLOCK,CYCLIC(1))$
  - block partition at the first dimension and cyclic partition at the second

**FIGURE 5** Graphical illustration of some alignment operators and MULTID operators.
Table 1. Algebraic Notations and Definitions of Array Intrinsics and Layout Operators

<table>
<thead>
<tr>
<th>Class</th>
<th>HPF Intrinsics/directives</th>
<th>Internal representations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGN1</td>
<td>EOSHIFT(A,dim,shift=-c)</td>
<td>EOSHIFT(c)</td>
</tr>
<tr>
<td></td>
<td>ALIGN A(i) WITH T(i+c)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CSHIFT(A,dim,shift=-c)</td>
<td>CSHIFT(c)</td>
</tr>
<tr>
<td></td>
<td>ALIGN A(i) WITH T(n-i+1)</td>
<td>REFLECT</td>
</tr>
<tr>
<td></td>
<td>ALIGN A(i) WITH T(a*i+c)</td>
<td>STRIDE(a,c)</td>
</tr>
<tr>
<td>ALIGN2</td>
<td>TRANSPOSE(A)</td>
<td>TRANS^2(0 1 1 0)</td>
</tr>
<tr>
<td></td>
<td>ALIGN A(i,j) WITH T(j,i)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TRANSPOSE(A, permute_vector)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ALIGN A(i,j) WITH T(i,i+j)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ALIGN A(I) WITH T(M*I)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CSHIFT(A,dim=2,shift=1:n)</td>
<td>CSK 2(1 0 1 -1)</td>
</tr>
<tr>
<td>ALIGN3</td>
<td>RESHAPE(A,s1,s2,[[0],[2]])</td>
<td>RESHAPE(D1,o1,D2,o2), where D1 is the index domain representing s1</td>
</tr>
<tr>
<td></td>
<td>SPREAD(A,dim=k,ncopies=n)</td>
<td>REPLICATE^2(rank(T))(V, Interval(I,n)), where V(k) = 1</td>
</tr>
<tr>
<td></td>
<td>ALIGN A(I) WITH T(I,*</td>
<td>REPLICATE^2(M)</td>
</tr>
<tr>
<td></td>
<td>ALIGN A(I) WITH T(I,l)</td>
<td>EMBED^2,l(M)</td>
</tr>
<tr>
<td>DIST</td>
<td>BLOCK(b)</td>
<td>BLOCK(b), where b = n/p</td>
</tr>
<tr>
<td></td>
<td>CYCLIC(b, p)</td>
<td>CYCLIC(b, p)</td>
</tr>
<tr>
<td>MULTID</td>
<td>multi-dimensional shift, reflection, etc., and distribution of multi-dimensional arrays</td>
<td>Product of 1D Operators (ALIGN1, DIST)</td>
</tr>
</tbody>
</table>

T is the template that array A is aligned to. V denotes an integer vector of length rank (T) and V(k) denotes the kth element of V. \( g^d(M) \) denotes an operator which takes a d-dimensional integer matrix M as argument, and M * I denotes the multiplication of matrix M with the index vector I. D1 and D2 are the index domains representations for shape specifications s1 and s2, and o1 and o2 are permutation matrices representing the storage ordering in reshaping.

<table>
<thead>
<tr>
<th>Class</th>
<th>Operator</th>
<th>Domain</th>
<th>Codomain</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGN1</td>
<td>EOSHIFT(c)</td>
<td>D</td>
<td>D + c</td>
<td>(i) ( \rightarrow ) (i + c)</td>
</tr>
<tr>
<td></td>
<td>CSHIFT(c)</td>
<td>D</td>
<td>D</td>
<td>(i) ( \rightarrow ) 1b(D) + (i - 1b(D) + c) mod</td>
</tr>
<tr>
<td></td>
<td>REFLECT</td>
<td>D</td>
<td>D</td>
<td>(i) ( \rightarrow ) 1b(D) + ub(D) - i</td>
</tr>
<tr>
<td></td>
<td>STRIDE(a,c)</td>
<td>D</td>
<td>aD + c</td>
<td>(i) ( \rightarrow ) (a * i + c)</td>
</tr>
<tr>
<td>ALIGN2</td>
<td>TRANS^d(M)</td>
<td>D</td>
<td>M(D)</td>
<td>(i_1,..,i_n) ( \rightarrow ) M(i_1,..,i_n)</td>
</tr>
<tr>
<td></td>
<td>SKEW^d(M)</td>
<td>D</td>
<td>M(D)</td>
<td>(i_1,..,i_n) ( \rightarrow ) M(i_1,..,i_n)</td>
</tr>
<tr>
<td></td>
<td>CSK EW^d(M)</td>
<td>D</td>
<td>Mod(M(D),</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>where m_k are the sizes of D_k</td>
</tr>
<tr>
<td>ALIGN3</td>
<td>REPLICATE^d(V,D_k)</td>
<td>D_1 x ... x D_{k-1} \times D_{k+1} x ... x D_n</td>
<td>D_1 x ... x D_{k-1} \times D_{k+1} x ... x D_n</td>
<td>(i_1,..,i_{k-1},i_{k+1},..,i_n) ( \rightarrow ) (i_1,..,i_{k-1},\frac{1}{1b(D_k) : ub(D_k),i_{k+1},..,i_n}, where V(k) ( \neq ) 0</td>
</tr>
<tr>
<td></td>
<td>EMBED^m,n(M,E)</td>
<td>D</td>
<td>E</td>
<td>(i_1,..,i_n) ( \rightarrow ) M(i_1,..,i_n)</td>
</tr>
<tr>
<td>DIST</td>
<td>BLOCK(b)</td>
<td>D</td>
<td>P \times L</td>
<td>(i) ( \rightarrow ) (i div b, i mod b)</td>
</tr>
<tr>
<td></td>
<td>CYCLIC(b, p)</td>
<td>D</td>
<td>P \times L</td>
<td>(i) ( \rightarrow ) (i div b) mod p, (i div (p * b)) * b + i mod b</td>
</tr>
<tr>
<td></td>
<td>SEQ</td>
<td>D</td>
<td>[0] × D</td>
<td>(i) ( \rightarrow ) (0, i)</td>
</tr>
</tbody>
</table>

1b(D) and ub(D) denote the lower and upper bound of interval domain D, aD + c denotes an interval domain of range \([a * 1b(D) + c) ... (a * ub(D) + c)]\), |D| denotes the size of interval domain D, M(D) constructs a multi-dimensional domain by permuting or skewing domain D according to the integer matrix M, and Mod(M(D),|D|) constructs a multi-dimensional domain where each dimension of M(D) is modulus by the size of D at that dimension.
(assuming the size of template T at the second dimension is $n$) duplicate $n$ copies of A at the second dimension. Both are denoted by

$$\text{REPLICATE}^{(2)}\left(\begin{pmatrix} 0 \\ 1 \end{pmatrix}, \text{Interval}(1,n)\right).$$

where the nonzero element (the second element) in the vector argument indicates that the replication takes place at the second dimension. The directive $\text{ALIGN A} (i)$ with $T(i,1)$ aligns A to the first column of T, denoted by

$$\text{EMBED}^{(2,1)}\left(\begin{pmatrix} 0 \\ 1 \end{pmatrix}, \text{Domain}(T)\right),$$

which, when given index domain $D$, maps $D$ to

$$\text{Domain}(T)$$

according to the coefficient matrix $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$. The

$$\text{RESHAPE}(D_1, o_1, D_2, o_2)$$

operator maps index domain $D_1$ to $D_2$, according to the mapping orderings given in permutation matrices $o_1$ and $o_2$, respectively. For instance,

$$\text{RESHAPE} \left(\text{Interval}(1,2) \times \text{Interval}(1,6), \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \text{Interval}(1,3) \times \text{Interval}(1,4), \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\right)$$

reshapes a 2 by 6 domain into a 3 by 4 domain, where the argument is enumerated in a column-major fashion (denoted by the first matrix) and the result is enumerated in a row-major fashion (denoted by the second matrix).

The standard distribution directives in HPF are BLOCK, CYCLIC and generic BLOCK_CYCLIC partitioning strategies. We collect them in the class DIST. MULTID operations capture multi-dimensional array intrinsics and data layouts that can be formulated as “product” of one-dimensional operators. For instance, a two-dimensional shift operation

$$\text{CSHIFT} (\text{CSHIFT} (A, \text{dim}=1, \text{shift}=-c_1), \text{dim}=2, \text{shift}=-c_2)$$

is denoted by the product of the two $\text{ALIGN1}$ operators $\text{CSHIFT}(c_1)$ and $\text{CSHIFT}(c_2)$.

The five classes of algebraic representations capture most of HPF's array intrinsics and alignment/distribution directives which can be formalized as index domain morphisms (functions that map an index domain to another). General array references using index expressions are transformed to corresponding algebraic representations (or composition of algebraic representations) whenever possible, according to a set of simple transformation rules.

### 3.2 Communication Expressions

We can formalize data layout and data movement as communication expressions using “product” and “composition” operators. The product of $f: D_1 \rightarrow E_1$ and $g: D_2 \rightarrow E_2$ is defined as

$$f \times g = \lambda(i, j): D_1 \times D_2 \rightarrow E_1 \times E_2. (f(i), g(i)).$$

The composition of two functions $g: D_2 \rightarrow D_3$ and $f: D_1 \rightarrow D_2$ is defined as

$$g \circ f = \lambda(i): D_1 \rightarrow D_3. g(f(i)).$$

In order to formalize to relationship between a data element and a concrete store within a processor, it is necessary to formalize the three stages of data mapping (alignment to template, partitioning template to logical processors, and the mapping logical processors to physical processors). Let $\alpha$ denote the alignment operator which aligns array $D$ to template $E$, $\beta$ denote the partition operator which partitions template $E$ into a pair of logical processors $L$ and local index domain $M$, and $\gamma$ denote the operator which maps logical processor-memory pairs ($L \times M$) to physical processor-memory pairs ($P \times M$).

### Data Layout

Data layout is simply the composition of the three stages of data mapping ($\alpha$, $\beta$, and $\gamma$). The layout of an array $D$ can be defined as a communication expression $g = \gamma \circ \beta \circ \alpha$, as shown in the commuting diagram of Figure 6a.

The formalization of data layout is used to derive communication expression for data movement. Intra-procedural data movement refers to array references within a procedure body. Inter-procedural data movement, also called layout conversion, refers to array copying due to change of data layouts between the actual and dummy arguments in procedure calls, which is a special case of intra-procedural data movement with array reference being identity.
Intra-Procedural Data Movement

Consider an assignment statement where the left-hand-side array \( D_2 \) and the right-hand-side array \( D_1 \) are related by an array reference \( \alpha \). Let the data layouts of \( D_1 \) and \( D_2 \) be \( g_1 \) and \( g_2 \), respectively. The data movement induced by the reference \( \alpha \) can be formalized by the composition \( g_2 \circ \alpha \circ g_1^{-1} \), as shown in the commuting diagram of Figure 6b.

Inter-Procedural Data Movement

Let the data layout of array \( D \) in procedure \( B_1 \) and procedure \( B_2 \) be \( g_1 \) and \( g_2 \), respectively. The data movement required to move array \( D \) from \( B_1 \) to \( B_2 \) is given by the communication expression \( g_2 \circ g_1^{-1} \), as shown in Figure 6c.

One exception is replication operations, which are relations. For instance, the operator

\[
\text{SPREAD}^{(2)} \left( \begin{array}{c} 0 \\ 1 \end{array} \right), D_2 \right)_{D_1 \rightarrow D_1 \times D_2}
\]

maps \((i) \in D_1 \) to

\[
((i, j), j = \text{lb}(D_2), \text{ub}(D_2)) \in D_1 \times D_2.
\]

We abuse the notation

\[
\text{SPREAD}^{(2)} \left( \begin{array}{c} 0 \\ 1 \end{array} \right), D_2 \right)^{-1}_{D_1 \rightarrow D_1 \times D_2}
\]

for its inverse, which is a relation that maps \((i, j) \in D_1 \times D_2 \) to \((i) \in D_1 \). The composition of two relations \( A \) and \( B \) is denoted as \( C = A \circ B \) such that \((a, c) \in C \) iff there exists a \( b \) such that \((a, b) \in A \) and \((b, c) \in B \). If both \( A \) and \( B \) are functions, this definition is the same as function composition.

Examples

Figure 7 shows a communication expression for inter-procedural layout conversion. In order to formalize the data movement between procedures ALPHA and BETA, it is necessary to construct the layouts of the actual and dummy arguments and to use these constructions in the application of the Inter-Procedural Rule (Figure 6c). We explicitly indicate the domain \( D \) and codomain \( E \) of each operator \( g \) (written as \( g_{D \rightarrow E} \)) because now the operators are bound to the index domain of array \( A \). The actual argument \( A \) is aligned with the template \( T_1 \), which is partitioned into columns of blocks (denoted by

\[
\text{SEQ}_{D_1+1 \rightarrow [0 \times (D_1+1)]} \times \text{BLOCK}(v)_{D_2+2 \rightarrow p \times v_1}
\]

by offsetting one element at the first dimension and two elements at the second dimension (denoted by

\[
\text{EOSHIFT}(1)_{D_1 \rightarrow D_1+1} \times \text{EOSHIFT}(2)_{D_2 \rightarrow D_2+2}
\]

(refer to Figure 6a). The dummy argument \( B \) is aligned with template \( T_2 \) by a transposition followed by an offsetting alignment with distance two at the first dimension and with distance 1 at the second dimension (denoted by

\[
\text{EOSHIFT}(2)_{D_2 \rightarrow D_2+2} \times \text{EOSHIFT}(1)_{D_1 \rightarrow D_1+1}
\]

\circ \text{TRANS}^{(2)} \left( \begin{array}{c} 0 \\ 1 \end{array} \right)_{D_1 \times D_2 \rightarrow D_2 \times D_1}
\]

The communication expression for changing from \( A \)'s layout to \( B \)'s can be constructed by composing \( B \)'s layout and the inverse of \( A \)'s layout. A communication expression for intra-procedural data movement is shown in Figure 11 of Appendix A.
Index domains:

\[ D_1 = \text{Interval}(1, 2), \quad D_2 = \text{Interval}(1, 4). \]

Actual’s layout:

\[ g_1 = (\text{SEQ}D_{1+1} \rightarrow [0] \times (D_{1+1}) \times \text{BLOCK}(b)D_{2+2} \rightarrow P \times V_1) \circ (\text{EOSHIFT}(1)D_1 \rightarrow D_1+1 \times \text{EOSHIFT}(2)D_2 \rightarrow D_2+2). \]

Dummy’s layout:

\[ g_2 = (\text{SEQ}D_{2+2} \rightarrow [0] \times (D_{2+2}) \times \text{BLOCK}(b)D_{1+1} \rightarrow P \times V_2) \]

\[ \circ (\text{EOSHIFT}(2)D_2 \rightarrow D_2+2 \times \text{EOSHIFT}(1)D_1 \rightarrow D_1+1) \circ \text{TRANS}^{(2)} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} D_1 \times D_2 \rightarrow D_2 \times D_1. \]

Communication expression for changing from actual’s layout to dummy’s:

\[ \theta = g_2 \circ g_1^{-1} \]

\[ = (\text{SEQ}D_{2+2} \rightarrow [0] \times (D_{2+2}) \times \text{BLOCK}(b)D_{1+1} \rightarrow P \times V_2) \]

\[ \circ (\text{EOSHIFT}(2)D_2 \rightarrow D_2+2 \times \text{EOSHIFT}(1)D_1 \rightarrow D_1+1) \circ \text{TRANS}^{(2)} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} D_1 \times D_2 \rightarrow D_2 \times D_1 \]

\[ \circ (\text{EOSHIFT}(1)^{-1}D_{1+1} \rightarrow D_{1+1} \times \text{EOSHIFT}(2)^{-1}D_{2+2} \rightarrow D_{2+2}) \circ \left(\text{SEQ}^{-1}D_{1+1} \rightarrow [0] \times (D_{1+1}) \times \text{BLOCK}(b)^{-1}D_{2+2} \rightarrow P \times V_1\right). \]

FIGURE 7 A communication expression for inter-procedural layout conversion. In order to formalize the data movement between procedures ALPHA and BETA, it is necessary to construct the layouts of the actual and dummy arguments and to use these constructions in the application of the inter-procedural rule.

### 3.3 Algebraic Simplification

A generic method for simplifying a communication expression is using functional transformation [6, 47]. Since we only deal with the set of standard HPF data distribution directives (instead of general functions) in the context of optimizing data movement, we look for a simpler and more efficient solution. We have designed a communication algebra to serve this purpose. The communication algebra manipulates communication expressions at the algebraic representation level only.

Recall that we divide HPF’s array operators and layout operators into several classes according to the dimensionality of their domains and ranges. A communication expression may be the composition of operators from any or all of the classes. Based on the classification, we design sub-algebras for each class to manipulate the interaction of operators within that class, as well as bridging sub-algebras for manipulating the interaction of operators from different classes.

Each sub-algebra contains three kinds of rules:

1. the inverse rules that computes the inverse of an operator;
2. the reduction rules that reduce two adjacent operators to one or zero new operator; and
3. the exchange rules that make two operators adjacent to each other by exchanging with other operators between them, so that the reduction rules may be applied later to simplify them.

The communication algebra will be presented in more detail in Section 4.
Example

Consider the communication expression \( \theta \) in Figure 7. By exchanging the TRANS operator with one of the two-dimensional EOSHIFT operators (using one of the exchange rules in the bridging sub-algebra for ALIGN2 and MULTID), the two EOSHIFT operators become adjacent and can be canceled with each other (using one of the reduction rules in ALIGN1 sub-algebra). This results in a TRANSPOSE operation on a two-dimensional index domain which is partitioned at the second dimension as shown below:

\[
\theta = \left( \text{SEQ}_{D_2+2} \times [0] \times (D_2+2) \right) \times \text{BLOCK}(b)D_{1+1} \rightarrow P \times V_2 \\
\circ \left( \text{EOSHIFT}(2) D_2 \rightarrow D_2+2 \times \text{EOSHIFT}(1) D_1 \rightarrow D_1+1 \right) \\
\circ \text{TRANS}^{(2)} \left( \begin{array}{cc} 0 & 1 \\ 1 & 0 \end{array} \right) D_1 \times D_2 \rightarrow D_2 \times D_1 \\
\circ \left( \text{EOSHIFT}(1) D_1 \rightarrow D_1+1 \times \text{EOSHIFT}(2) D_2 \rightarrow D_2+2 \right) \\
\circ \left( \text{SEQ}^{-1} D_1 \rightarrow [0] \times (D_1+1) \times \text{BLOCK}(b) D_2 \rightarrow 2 \rightarrow P \times V_1 \right) \\
= \left( \text{SEQ}_{D_2+2} \times [0] \times (D_2+2) \right) \times \text{BLOCK}(b) D_{1+1} \rightarrow P \times V_2 \\
\circ \text{TRANS}^{(2)} \left( \begin{array}{cc} 0 & 1 \\ 1 & 0 \end{array} \right) (D_{1+1} \times (D_2+2) \times (D_2+2) \times (D_1+1)) \\
\circ \left( \text{SEQ}^{-1} D_1 \rightarrow [0] \times (D_1+1) \times \text{BLOCK}(b) D_2 \rightarrow 2 \rightarrow P \times V_1 \right).
\]

3.4 Communication Idiom Matching

A simplified communication expression contains the actual data movement that needs be performed. A naive approach is to use general communication for all cases. This approach ignores any opportunity for fast communication. A better approach is to uncover frequently occurring data movement and use specialized, fast communication whenever possible. For instance, the simplified communication expression \( \theta \) shown in the example in Section 3.3 is a transposition of a two-dimensional matrix which is partitioned one-dimensionally, resulting in so-called all-to-all personalized communication [21], in which every processor exchanges distinct data with every other processor. Due to the uniform communication patterns, communication overhead may be reduced by carefully scheduling messages to avoid contention in the network.

Since the advent of massively parallel machines, many researchers have developed specialized communication routines to facilitate direct programming of distributed-memory machines (e.g. [5, 21–23, 26, 36, 37, 39, 42]). In building compilers, we might take advantage of these hand-crafted, highly optimized routines which become part of the runtime system for the language. In the Crystal compiler developed at Yale University [30–32], this approach is used to generate intra-procedural communication. We extend that work further to include those communication routines for converting data layouts between subprograms.

We have collected a set of frequently occurring communication patterns, and extracted the contents of their communication expressions into communication idoms. They include most of the array intrinsics and frequently occurring layout conversions such as conversion between BLOCK and CYCLIC partitioning and conversion between column partition \((\ast, \text{BLOCK})\) and row partition \((\text{BLOCK}, \ast)\). These idoms may or may not have specialized, fast communication, perhaps microcoded or otherwise hand-optimized, depending on the target machine. A list of communication idioms is shown in Table 2. The optimization procedure simply goes through this list of idioms and pattern matches with the communication expression. If a simplified communication expression contains more than one alignment operators, the compiler will either expand the expression to match multiple idioms or unfold and collapse the expression into a general communication, depending on the characteristics of the target machine.

Example

The expression \( \gamma_1 \circ \gamma_2^{-1} \) in the first row of Table 2 indicates change of physical mapping strategy (e.g. change from Gray code encoding to Binary code encoding on hypercube architectures) because alignment and partition operators have all been reduced away. The expression \( x^d (\gamma_1 \circ \beta_1 \circ \beta_2^{-1} \circ \gamma_2^{-1}) \) in the second row of the table indicates change of data partition. For instance, the idiom CYCLIC(1, \( p \)) \circ BLOCK(b)^{-1} indicates conversion from BLOCK partition to CYCLIC partition. The idiom

\[
\left( \text{SEQ} \times \text{BLOCK}(b_1) \right) \circ \left( \text{BLOCK}(b_2) \times \text{SEQ} \right)^{-1}
\]

indicates conversion from column partition to row partition.

4 COMMUNICATION ALGEBRA

In this section we present in more detail the communication algebra. Recall that the alignment/distribution operators are functions with matrix/vector/integer parameters. Although the simplification structure is not closed under composition, we abuse the name “algebra” to reflect that the algebraic properties are preserved in combining the matrix/vector/integer parameters between composition of operators.
Table 2. Communication Idioms

<table>
<thead>
<tr>
<th>Idioms</th>
<th>Data Movement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_1 \circ y_2^{-1}$</td>
<td>change physical mapping</td>
</tr>
<tr>
<td>$x^d \left( y_{i1} \circ b_{i1} \circ b_{j2}^{-1} \circ y_{i2}^{-1} \right)$</td>
<td>change partition</td>
</tr>
<tr>
<td>$y_1 \circ b_{i1} \circ \text{EOSHIFT}(c) \circ b_{j2}^{-1} \circ y_{i2}^{-1}$</td>
<td>end-of-shift</td>
</tr>
<tr>
<td>$y_1 \circ b_{i1} \circ \text{CSHIFT}(c) \circ b_{j2}^{-1} \circ y_{i2}^{-1}$</td>
<td>cyclic shift</td>
</tr>
<tr>
<td>$y_1 \circ b_{i1} \circ \text{REFLECT} \circ b_{j2}^{-1} \circ y_{i2}^{-1}$</td>
<td>reversal permutation</td>
</tr>
<tr>
<td>$x^d \left( y_{i1} \circ b_{i1} \circ \text{EOSHIFT}(c) \circ b_{j2}^{-1} \circ y_{i2}^{-1} \right)$</td>
<td>multi-dimensional end-of-shift</td>
</tr>
<tr>
<td>$x^d \left( y_{i1} \circ b_{i1} \circ \text{CSHIFT}(c) \circ b_{j2}^{-1} \circ y_{i2}^{-1} \right)$</td>
<td>multi-dimensional cyclic shift</td>
</tr>
<tr>
<td>$x^d \left( y_{i1} \circ b_{i1} \circ \text{REFLECT} \circ b_{j2}^{-1} \circ y_{i2}^{-1} \right)$</td>
<td>multi-dimensional reflection</td>
</tr>
<tr>
<td>$x^d \left( y_{i1} \circ b_{i1} \circ \text{TRANS}^d(M) \circ b_{j2}^{-1} \circ y_{i2}^{-1} \right)$</td>
<td>matrix transpose</td>
</tr>
<tr>
<td>$x^d \left( y_{i1} \circ b_{i1} \circ \text{SKEW}^d(M) \circ b_{j2}^{-1} \circ y_{i2}^{-1} \right)$</td>
<td>skewing</td>
</tr>
<tr>
<td>$x^d \left( y_{i1} \circ b_{i1} \circ \text{CSKew}^d(M) \circ b_{j2}^{-1} \circ y_{i2}^{-1} \right)$</td>
<td>cyclic skewing</td>
</tr>
<tr>
<td>$x^d \left( y_{i1} \circ b_{i1} \circ \text{REPLICATE}^d(V, D) \circ b_{j2}^{-1} \circ y_{i2}^{-1} \right)$</td>
<td>replication</td>
</tr>
<tr>
<td>$y_1 \circ b_{i1} \circ \text{RESHAPe}^{(d, 1)}(D, M, \text{interval}(1, n), I) \circ x^d \left( b_{j2}^{-1} \circ y_{i2}^{-1} \right)$</td>
<td>axis combining</td>
</tr>
<tr>
<td>$x^d \left( y_{i1} \circ b_{i1} \circ \text{RESHAPe}^{(d, 1)}(\text{interval}(1, n), I, D, M) \circ b_{j2}^{-1} \circ y_{i2}^{-1} \right)$</td>
<td>axis splitting</td>
</tr>
</tbody>
</table>

where $a$ denotes alignment operators or array references, $b$ denotes distribution operators, $y$ denotes physical mapping operators, and $x^d(a_1 \circ b_1)$ denotes $(a_1 \circ b_1) \times \cdots \times (a_d \circ b_d)$.

FIGURE 8 Organization of the communication algebra.

Based on our classification of array operators and layout operators, we design subalgebras for each class to manipulate the interaction of operators within that class, as well as bridging subalgebras for manipulating the interaction of operators from different classes. Figure 8 shows the organization of these subalgebras. Each subalgebra contains three kinds of rules:

1. the **inverse** rules that compute the inverse of an operator;
2. the **reduction** rules that reduce two adjacent operators to one or zero new operator; and
3. the **exchange** rules that make two operators adjacent to each other by exchanging positions with the operators in between.

Given a communication expression, the algebraic engine applies these rules according to a simple heuristic until no rules can be applied.

In the following, we present the subalgebras, the bridging subalgebras and the simplification procedure for communication expressions. Each subalgebra is denoted by a triple $(S, \Omega, R)$, where $S$ is the sorts, $\Omega$ is the operators and $R$ is the set of rules. The sorts of each subalgebra define the appropriate set of index domains in which the operators in the particular classes are defined. The set $R$ outlines the rules that either directly or indirectly reduce the length for a communication expression. Proofs of the rules are included in Appendix B.

4.1 Subalgebra for ALIGN1

The sorts of subalgebra for ALIGN1 contains the integer set $N$ and the set of interval domains $\text{Dom}$. The operators include the four ALIGN1 operators and one composition operation $(o)$. The composition operator is a higher-order function that takes two ALIGN1 operators $(D_1 \rightarrow D_2, D_2 \rightarrow D_3)$ as arguments. For convenience, we use the notation $a \ast D + c$ for an index domain with lower bound $a \ast \text{lb}(D) + c$ and upper bound $a \ast \text{ub}(D) + c$. Rule 1 computes the inverse of an ALIGN1 operator, Rule 2 reduces two adjacent ALIGN1 operators into one simply by integer addition and multiplication on their arguments. Rule 3 exchanges the positions of two adjacent ALIGN1 operators that cannot be directly reduced. Non-adjacent operators are reduced by applying a sequence of Rule 3 and Rule 2.

$$A_1 = (\{N, \text{Dom}\}, \Omega_1, R_1)$$
Rule 1 Inverse of ALIGN1 Operators

1. \( \text{EOSHIFT}(c)^{-1}_{D \rightarrow D+c} = \text{EOSHIFT}(-c)_{D+c \rightarrow D} \)
2. \( \text{CSHIFT}(c)^{-1}_{D \rightarrow D} = \text{CSHIFT}(-c)_{D \rightarrow D} \)
3. \( \text{REFLECT}^{-1}_{D \rightarrow D} = \text{REFLECT}_{D \rightarrow D} \)
4. \( \text{STRIDE}(a, c)^{-1}_{D \rightarrow a*D+c} = \text{STRIDE}\left(\frac{1}{a}, -\frac{c}{a}\right)_{a*D+c \rightarrow D} \)

Rule 2 Reduction of Adjacent ALIGN1 Operators

1. \( \text{EOSHIFT}(c_1)_{D \rightarrow D+c_1} \circ \text{EOSHIFT}(c_2)_{D+c_1 \rightarrow D+c_2} = \text{EOSHIFT}(c_1 + c_2)_{D \rightarrow D+c_2} \)
2. \( \text{CSHIFT}(c_1)_{D \rightarrow D} \circ \text{CSHIFT}(c_2)_{D \rightarrow D} = \text{CSHIFT}(c_1 + c_2)_{D \rightarrow D} \)
3. \( \text{REFLECT}_{D \rightarrow D} \circ \text{REFLECT}_{D \rightarrow D} = \text{id}_D \)
4. \( \text{STRIDE}(a_1, c_1)_{D \rightarrow a_1*D+c_1} \circ \text{STRIDE}(a_2, c_2)_{D \rightarrow a_2*D+c_2} = \text{STRIDE}(a_1*a_2, a_1*c_1)_{D \rightarrow a_1*a_2*D+a_1*c_1} \)
5. \( \text{STRIDE}(a, b)_{D \rightarrow a*D+b} \circ \text{EOSHIFT}(c)_{D \rightarrow D+c} = \text{STRIDE}(a, a*c+b)_{D \rightarrow a*D+a*c+b} \)
6. \( \text{EOSHIFT}(c)_{a*D+c \rightarrow D} \circ \text{STRIDE}(a, c)_{D \rightarrow D} = \text{STRIDE}(a, a*c+b)_{D \rightarrow a*D+a*c+b} \)

Rule 3 Exchange of ALIGN1 Operators

1. \( \text{CSHIFT}(c_1)_{D \rightarrow D+c_1} \circ \text{EOSHIFT}(c_2)_{D \rightarrow D+c_2} \circ \text{CSHIFT}(c_2)_{D \rightarrow D+c_2} = \text{EOSHIFT}(c_1)_{D \rightarrow D+c_1} \circ \text{CSHIFT}(c_2)_{D \rightarrow D} \)
2. \( \text{CSHIFT}(c_1)_{D \rightarrow D} \circ \text{REFLECT}_{D \rightarrow D} \circ \text{CSHIFT}(c_2)_{D \rightarrow D} = \text{REFLECT}_{D \rightarrow D} \circ \text{CSHIFT}(c_2)_{D \rightarrow D} \)
3. \( \text{EOSHIFT}(c_1)_{D \rightarrow D+c_1} \circ \text{REFLECT}_{D \rightarrow D} = \text{REFLECT}_{D \rightarrow D} \circ \text{EOSHIFT}(c_1)_{D \rightarrow D+c_1} \)
4. \( \text{STRIDE}(a, c)_{D \rightarrow a*D+c} \circ \text{REFLECT}_{D \rightarrow D} = \text{REFLECT}_{a*D+c \rightarrow a*D+c} \circ \text{STRIDE}(a, c)_{D \rightarrow a*D+c} \)
5. \( \text{STRIDE}(a, c)_{D \rightarrow a*D+c} \circ \text{EOSHIFT}(c)_{D \rightarrow D+c} = \text{STRIDE}(a, a*D+c)_{D \rightarrow a*D+c} \circ \text{STRIDE}(a, c)_{D \rightarrow a*D+c} \)

\[ \Omega_1 = \{ \text{EOSHIFT}(c \in N)(D \in \text{Dom}) : D \rightarrow D + c, \]
\[ \text{CSHIFT}(c \in N)(D \in \text{Dom}) : D \rightarrow D, \]
\[ \text{REFLECT}(D \in \text{Dom}) : D \rightarrow D, \]
\[ \text{STRIDE}(a \in N, c \in N)(D \in \text{Dom}) : D \rightarrow a*D+c, \]
\[ \circ : (D_1 \rightarrow D_2) \rightarrow (D_2 \rightarrow D_3) \rightarrow (D_1 \rightarrow D_3), \]
\[ \text{where } D_1, D_2, D_3 \in \text{Dom} \] \]
\[ R_1 = \{ \text{Rule 1, Rule 2, Rule 3} \} \]

Example

\[ \text{CSHIFT}(2)_{D+1 \rightarrow D+1} \circ \text{EOSHIFT}(1)_{D \rightarrow D+1} \circ \text{CSHIFT}(-2)_{D \rightarrow D} = \text{By Rule 3(1)} \]
\[ \text{EOSHIFT}(1)_{D \rightarrow D+1} \circ \text{CSHIFT}(2)_{D \rightarrow D} \circ \text{CSHIFT}(-2)_{D \rightarrow D} = \text{By Rule 2(2)} \]
\[ \text{EOSHIFT}(1)_{D \rightarrow D+1} \]

4.2 Subalgebra for ALIGN2

The sorts of subalgebra for ALIGN2 contains the integer set \( N \), the set of \( d \times d \) integer matrices \( \text{Mat}^d \) and the set of interval domains \( \text{Dom} \). Let \( \text{Dom}^d \) denote the set of \( d \)-dimensional index domains, and \( M(D) \) denote an affine transformation on index domain \( D \) with coefficient matrix \( M \). The operators include the three ALIGN2 operators and one composition operation \( (\circ) \). The composition operator is a higher-order function that takes two ALIGN2 operators \( (D_1 \rightarrow D_2, D_2 \rightarrow D_3) \) as arguments. Rule 1 computes the inverse of an ALIGN2 operator by deriving the inverse of its coefficient matrix. Rules 5–7 reduce two adjacent ALIGN2 operators into one by multiplying their coefficient matrices.

\[ A_2 = \{ [N, \text{Mat}^d, \text{Dom}], \Omega_2, R_2 \} \]
\[ R_2 = \{ \text{Rule 4, Rule 5, Rule 6, Rule 7} \} \]

Example

\[ \text{CSKEW}(2) \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}_{D \rightarrow D} \circ \text{CSKEW}(2) \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}_{D \rightarrow D} \]
Rule 4 Inverse of ALIGN2 Operators

1. \( \text{TRANS}^{(d)}(M)_{D \rightarrow E}^{-1} = \text{TRANS}^{(d)}(M^{-1})_{E \rightarrow D} \)
2. \( \text{SKEW}^{(d)}(M)_{D \rightarrow E}^{-1} = \text{SKEW}^{(d)}(M^{-1})_{E \rightarrow D} \)
3. \( \text{CSKEW}^{(d)}(M)_{D \rightarrow E}^{-1} = \text{CSKEW}^{(d)}(M^{-1})_{E \rightarrow D} \)

Rule 5 Reduction of ALIGN2 Operators

1. \( \text{TRANS}^{(d)}(M_2)_{D_2 \rightarrow D_3} \circ \text{TRANS}^{(d)}(M_1)_{D_1 \rightarrow D_2} = \text{TRANS}^{(d)}(M_2 \ast M_1)_{D_1 \rightarrow D_3} \)
2. \( \text{SKEW}^{(d)}(M_2)_{D_2 \rightarrow D_3} \circ \text{SKEW}^{(d)}(M_1)_{D_1 \rightarrow D_2} = \text{SKEW}^{(d)}(M_2 \ast M_1)_{D_1 \rightarrow D_3} \)
3. \( \text{CSKEW}^{(d)}(M_2)_{D \rightarrow D} \circ \text{CSKEW}^{(d)}(M_1)_{D \rightarrow D} = \text{CSKEW}^{(d)}(M_2 \ast M_1)_{D \rightarrow D} \)

Rule 6 Left-Trans Reduction

1. \( \text{TRANS}^{(d)}(M_2)_{D_2 \rightarrow D_3} \circ \text{SKEW}^{(d)}(M_1)_{D_1 \rightarrow D_2} = \text{SKEW}^{(d)}(M_2 \ast M_1)_{D_1 \rightarrow D_3} \)
2. \( \text{TRANS}^{(d)}(M_2)_{D_1 \rightarrow D_2} \circ \text{CSKEW}^{(d)}(M_1)_{D_1 \rightarrow D_1} = \text{CSKEW}^{(d)}(M_2 \ast M_1)_{D_1 \rightarrow D_2} \)

Rule 7 Right-Trans Reduction

1. \( \text{SKEW}^{(d)}(M_2)_{D_2 \rightarrow D_3} \circ \text{TRANS}^{(d)}(M_1)_{D_1 \rightarrow D_2} = \text{TRANS}^{(d)}(M_2 \ast M_1)_{D_1 \rightarrow D_3} \)
2. \( \text{CSKEW}^{(d)}(M_2)_{D_2 \rightarrow D_2} \circ \text{TRANS}^{(d)}(M_1)_{D_1 \rightarrow D_1} = \text{CSKEW}^{(d)}(M_2 \ast M_1)_{D_1 \rightarrow D_2} \)

\[ \Omega_2 = \left\{ \text{TRANS}(d \in N)(M \in \text{Mat}^d)(D \in \text{Dom}^d) : D \rightarrow M(D), \right. \\
\text{SKEW}(d \in N)(M \in \text{Mat}^d)(D \in \text{Dom}^d) : D \rightarrow M(D), \\
\text{CSKEW}(d \in N)(M \in \text{Mat}^d)(D \in \text{Dom}^d) : D \rightarrow D, \\
(\circ : (D_1 \rightarrow D_2) \rightarrow (D_2 \rightarrow D_3) \rightarrow (D_1 \rightarrow D_3), \\
\text{where } D_1, D_2, D_3 \in \text{Dom}^d \right\} \]

= By Rule 5(3)

\[ \text{CSKEW}^{(2)} \left( \begin{array}{cc} 1 & 0 \\ 1 & 1 \end{array} \right) \ast \left( \begin{array}{c} 1 \\ 1 \\ 0 \end{array} \right) \right)_{D \rightarrow D} \]

\[ = \text{CSKEW}^{(2)} \left( \begin{array}{c} 2 \\ 1 \\ 1 \end{array} \right)_{D \rightarrow D} \]

The result is to skew index domain \( D \) at both dimensions in wrap-around fashion according to the coefficient matrix \( \left( \begin{array}{cc} 2 & 1 \\ 1 & 1 \end{array} \right) \).

4.3 Subalgebra for ALIGN3

The sorts of subalgebra for ALIGN3 contains the integer set \( N \), the set of interval domains \( \text{Dom} \), the set of \( m \times n \) integer matrices \( \text{Mat}^{m,n} \), the set of length-\( d \) integer vectors \( \text{Vec}^d \) (we define \( V \in \text{Vec}^d \),

\[ V(D_1 \times \cdots \times D_{k-1} \times D_{k+1} \times \cdots \times D_d, D_k) \]

\[ = D_1 \times \cdots \times D_{k-1} \times D_k \times D_{k+1} \times \cdots \times D_d, \]

where \( V(k) \neq 0 \) (kth element of \( V \) is nonzero), to be a length-\( d \) vector). The operators include the three ALIGN3 operators and one composition operation (\( \circ \)). The composition operator is a higher-order function that takes two ALIGN3 operators as arguments.

Inverses of ALIGN3 operators can be devised without appeal to standard matrix/vector/integer algebra (for discussion on inverses of ALIGN3 operators, please refer to Section 3.2). Therefore, no algebraic rules are needed for finding their inverses. We abuse the notation \( a^{-1} \) to denote the inverse of an ALIGN3 operator. The inverse-
Rule 8  Inverse Cancellation of ALIGN3 Operators

For all \( a_{D \to E} \) in ALIGN3,
\[ a_{D \to E} \circ a_{D \to E}^{-1} = \text{id}_E. \]

Rule 9  Reduction of Adjacent ALIGN3 Operators

1. \( \text{EMBED}^{(m,k)}(M_2)_{D_2 \to D_3} \circ \text{EMBED}^{(k,l)}(M_1)_{D_1 \to D_2} = \text{EMBED}^{(m,l)}(M_2 \times M_1)_{D_1 \to D_3} \)
2. \( \text{REPLICATE}^{(k)}(V_2, E_2)_{D_2 \to D_3} \circ \text{REPLICATE}^{(k)}(V_1, E_1)_{D_1 \to D_2} = \text{REPLICATE}^{(k)}(V_2 + V_1, E_1 \times E_2)_{D_1 \to D_3} \)
3. \( \text{RESHAPE}^{(l,m)}(D_2, M_3, D_3, M_4)_{D_2 \to D_3} \circ \text{RESHAPE}^{(k,l)}(D_1, M_1, D_2, M_2)_{D_1 \to D_2} = \text{RESHAPE}^{(k,m)}(D_1, M_1, D_3, M_4)_{D_1 \to D_3}, \) if \( M_2 = M_3 \)

\[ \Omega_3 = \left\{ \text{EMBED}(m \in N)(n \in N)(m \in \text{Mat}^{(m,n)})(D \in \text{Dom}^n) : D \to M(D), \right. \]
\[ \left. \text{REPLICATE}(n \in N)(V \in \text{Vec}^n)(E \in \text{Dom})(D \in \text{Dom}) : D \to V(D, E), \right. \]
\[ \left. \text{RESHAPE}(m \in N)(n \in N)(D_1 \in \text{Dom}^n)(M_1 \in \text{Mat}^{(m,m)})(D_2 \in \text{Dom}^n)(M_2 \in \text{Mat}^{(n,n)}): D_1 \to D_2, \right. \]
\[ \left. o: (D_1 \to D_2) \to (D_2 \to D_3) \to (D_1 \to D_3), \right. \]
\[ \Omega_5 = \left\{ x^d : (D_1 \to E_2) \to \cdots \to (D_d \to E_d) \to (D_1 \times \cdots \times D_d \to E_1 \times \cdots \times E_d), \right. \]
where \( D_1, \ldots, D_d \in \text{Dom}, E_1, \ldots, E_d \in \text{Dom}, \)
\[ o: (D_1 \to D_2) \to (D_2 \to D_3) \to (D_1 \to D_3), \]
where \( D_1, D_2, D_3 \in \text{Dom}^d \}

A replication along the second dimension followed by a replication of the new index domain along the third dimension can be combined into a multi-dimensional replication of the original domain.

4.4 Subalgebra for MULTID

The purpose of this subalgebra is to distribute composition operations through product operations so that composition of MULTID operators can be simplified.

\[ A_5 = \{\{N, \text{Dom}\}, \Omega_1 \cup \Omega_5, R_5\} \]
\[ R_5 = \{\text{Rule 10}\} \]

\[ \text{Example} \]

\[ \text{REPLICATE}^{(3)} \left( \begin{array}{c} 0 \\ 1 \\ 0 \end{array} \right), E_2 \]
\[ E_1 \to E_1 \times E_2 \]
\[ \circ \text{REPLICATE}^{(3)} \left( \begin{array}{c} 0 \\ 0 \\ 1 \end{array} \right), E_3 \]
\[ E_1 \times E_2 \to E_1 \times E_2 \times E_3 \]
\[ = \text{By Rule 9(2)} \]
\[ \text{REPLICATE}^{(3)} \left( \begin{array}{c} 0 \\ 1 \\ 1 \end{array} \right), E_2 \times E_3 \]
\[ E_1 \to E_1 \times E_2 \times E_3 \]
Rule 10 Product-Composition Exchange

For all $\alpha_1 i (c_i, d_i) E_i \rightarrow F_i$ and $\alpha_2 i (a_i, b_i) D_i \rightarrow E_i$ in ALIGN1,

$$x^d \alpha_1 i (c_i, d_i) E_i \rightarrow F_i \circ x^d \alpha_2 i (a_i, b_i) D_i \rightarrow E_i = x^d \left( \alpha_1 i (c_i, d_i) E_i \rightarrow F_i \circ \alpha_2 i (a_i, b_i) D_i \rightarrow E_i \right).$$

Rule 11 Exchange of ALIGN2 and MULTID Operators

1. Let $k^{(n)}(V) D \rightarrow E = \alpha V(1)_{D V(n)} \rightarrow E_{V(n)} \times \cdots \times \alpha V(n)_{D V(n)} \rightarrow E_{V(n)}$ where $\alpha_i, i = 1, n$ are ALIGN1 operators,

$$\text{TRANS}^{(n)}(M) E \rightarrow M(E) \circ k^{(n)}(V) D \rightarrow E = k^{(n)}(M \ast V) M(D) \rightarrow M(E) \circ \text{TRANS}^{(n)}(M) D \rightarrow M(D).$$

2. Let $S^{(n)}(V) D_1 \times \cdots \times D_n \rightarrow (D_1 + c_1) \times \cdots \times (D_n + c_n) = (\text{EOShift}(V(1)) D_1 \rightarrow D_1 + c_1 \times \cdots \times \text{EOShift}(V(n)) D_n \rightarrow D_n + c_n),$

$$S^{(n)}(M) (D_1 + c_1) \times \cdots \times (D_n + c_n) \rightarrow E_1 \times \cdots \times E_n = S^{(n)}(M \ast V) M(D_1 \times \cdots \times D_n) \rightarrow E_1 \times \cdots \times E_n \circ S^{(n)}(M) D_1 \times \cdots \times D_n \rightarrow M(D_1 \times \cdots \times D_n).$$

3. Let $\text{REF}^{(n)}(V) D \rightarrow D = (\text{REFL}ect \ D V(1) \rightarrow D V(n)) \times \cdots \times \text{REFL}ect \ D V(n) \rightarrow D V(n)$,

$$S^{(n)}(M) D \rightarrow M(D) \circ \text{REF}^{(n)}(V) D \rightarrow D = \text{REF}^{(n)}(M \ast V) M(D) \rightarrow M(D) \circ S^{(n)}(M) D \rightarrow M(D).$$

4. Let $C^{(n)}(V) D_1 \times \cdots \times D_n \rightarrow D_1 \times \cdots \times D_n = (C^{(n)}(V(1)) D_1 \rightarrow D_1 \times \cdots \times C^{(n)}(V(n)) D_n \rightarrow D_n)$

where $|D_1| = |D_2| = \cdots = |D_n|$, then cancelling out the two TRANS operators by applying Rule 5(1).

4.5 Bridging Subalgebra for ALIGN2 and MULTID

This subalgebra exchanges the positions of adjacent ALIGN1 and MULTID operators, so that either one of them may be reduced with other operators in the communication expression.

$$A_6 = \{\{N, \text{Mat}^d, \text{Dom}\}, \Omega_2 \cup \Omega_5, R_2 \cup R_5, UR_6\}$$

$$R_6 = \{\text{Rule 11}\}$$

**Example**

The expression

$$\text{TRANS}^{(2)} \left( \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \right)_{D_2 \times D_1 \rightarrow D_1 \times D_2} \circ \left( \text{CSHIFT}(1) \times \text{REFL}ect \right)_{D_2 \times D_1 \rightarrow D_2 \times D_1} \circ \text{TRANS}^{(2)} \left( \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \right)_{D_1 \times D_2 \rightarrow D_2 \times D_1}$$

4.6 Bridging Subalgebra for ALIGN3 and MULTID

This subalgebra exchanges the positions of adjacent ALIGN3 and MULTID operators, so that either one of them may be reduced with other operators in the communication expression. Let $V_1 \parallel V_2$ denote the concatenation of two vectors, $0^d$ denote a vector of $d$ zeros.

$$A_7 = \{\{N, \text{Mat}^d, \text{Dom}\}, \Omega_3 \cup \Omega_5, R_3 \cup R_5, UR_7\}$$

$$R_7 = \{\text{Rule 12}\}$$
Rule 12  Exchange of Adjacent ALIGN3 and MULTID Operators

Let $\text{SH}^{(n)}(V)_{D_1 \times \cdots \times D_n} \rightarrow (D_1 + c_1) \times \cdots \times (D_n + c_n) = (\text{EOSHIFT}(V(1))_{D_1} \rightarrow D_1 + c_1 \times \cdots \times \text{EOSHIFT}(V(n))_{D_n} \rightarrow D_n + c_n)$ and $\text{CSH}^{(n)}(V)_{D_1 \times \cdots \times D_n} \rightarrow (D_1 + c_1) \times \cdots \times (D_n + c_n) = (\text{SHIFT}(V(1))_{D_1} \rightarrow D_1 \times \cdots \times \text{SHIFT}(V(n))_{D_n} \rightarrow D_n)$.

1. $\text{EMBED}^{(m,n)}(M)_{V2, D_2} \rightarrow D_1 \circ \text{SH}^{(m)}(M)_{V1, D_1} \rightarrow D_2 = \text{EMBED}^{(m,n)}(M \ast V)_{D_1 \rightarrow D_3} \circ \text{SH}^{(m)}(M)_{V1, D_1} \rightarrow D_2$
2. $\text{EMBED}^{(m,n)}(M)_{V2, D_2} \rightarrow D_1 \circ \text{CSH}^{(m)}(V)_{V1, D_1} \rightarrow D_2 = \text{CSH}^{(m)}(M \ast V)_{M(D_1) \rightarrow D_3} \circ \text{EMBED}^{(m,n)}(M)_{V1, D_1} \rightarrow D_2$
3. $\text{REPLICATE}^{(n)}(V2, E)_{D_2, V1} \rightarrow D_1 \circ \text{EMBED}^{(m,n)}(M)_{V1, D_1} \rightarrow D_2 = \text{EMBED}^{(m,n)}(M \ast V)_{D_1 \rightarrow D_3} \circ \text{REPLICATE}^{(n)}(V2, E)_{V2, D_2} \rightarrow D_1 \circ \text{EMBED}^{(m,n)}(M)_{V1, D_1} \rightarrow D_2$
4. $\text{REPLICATE}^{(n)}(V2, E)_{D_2, V1} \rightarrow D_1 \circ \text{CSH}^{(m)}(V)_{V1, D_1} \rightarrow D_2 = \text{CSH}^{(m)}(M \ast V)_{M(D_1) \rightarrow D_3} \circ \text{REPLICATE}^{(n)}(V2, E)_{V2, D_2} \rightarrow D_1 \circ \text{CSH}^{(m)}(V)_{V1, D_1} \rightarrow D_2$

Rule 13  Reduction of ALIGN2 and ALIGN3 Operators

1. $\text{TRANS}^{(m)}(M)_{D_2, D_3} \circ \text{EMBED}^{(m,n)}(M_2)_{D_1 \rightarrow D_2} = \text{EMBED}^{(m,n)}(M_1 \ast M_2)_{D_1 \rightarrow D_3}$
2. $\text{SKEW}^{(m)}(M)_{D_2, D_3} \circ \text{EMBED}^{(m,n)}(M_2)_{D_1 \rightarrow D_2} = \text{EMBED}^{(m,n)}(M_1 \ast M_2)_{D_1 \rightarrow D_3}$
3. $\text{TRANS}^{(n)}(M_3)_{D_2, D_3} \circ \text{RESHAPE}^{(m,n)}(D_1, M_1, M_2, D_2)_{D_1 \rightarrow D_2} = \text{RESHAPE}^{(m,n)}(D_1, M_1, M_3(D_1), M_3 \ast M_2)_{D_1 \rightarrow D_3}$
4. $\text{RESHAPE}^{(m,n)}(D_2, M_1, D_3, M_2)_{D_2 \rightarrow D_3} \circ \text{TRANS}^{(m)}(M_3)_{D_1 \rightarrow D_2} = \text{RESHAPE}^{(m,n)}(M_2(D_2), M_3 \ast M_1, D_3, M_2)_{D_1 \rightarrow D_3}$

4.7 Bridging Subalgebra for ALIGN2 and ALIGN3

The reduction rules in this subalgebra collapse adjacent ALIGN2 and ALIGN3 operators into an ALIGN3 operator.

$A_8 = \{\{N, \text{Mat}^d, \text{Dom}\}, \Omega_2 \cup \Omega_3, R_2 \cup R_3, \cup R_8\}$

$R_8 = \{\text{Rule 13}\}$

4.8 Subalgebras for Distribution

Since change of data distribution and physical mapping strategies is a task that is handed over to the runtime system, the compiler's job is simply to detect whether data movement is required and to identify the appropriate communication routine for it. Applying the Product-composition-exchange rule (in the subalgebra for MULTID) followed by inverse-cancellation ($g$ canceled out with $g^{-1}$) for each product component is sufficient for this purpose.

4.9 Simplification Procedure

The compiler simplifies a communication expression by applying a sequence of the algebraic rules. Although the length of a communication expression (and therefore the complexity of the algebraic simplification) only depends on the number of levels in nested procedure calls, which usually is a small constant (less than four or five in most application programs). For efficiency of the compiler, it is desirable to minimize the execution time of the simplification procedure. We use a simple heuristic to solve this problem. Before describing the heuristic, we first define some relevant terminology.

Definition. The composition of two operators $\theta = g_1 \circ g_2$ is immediately reducible if $\theta$ can be reduced to a basic operator by the reduction rules. A basic operator is either an identity function or a single alignment operator.

For example, the composition

$\text{CSHIFT}(c_2) \circ \text{CSHIFT}(c_1)$

is immediately reducible (it can be reduced to a basic operator $\text{CSHIFT}(c_2 + c_1)$).

Definition. A redcom is a subexpression $\theta = g_1 \circ [\cdots]g_2$ in a communication expression such that either $\theta$ is immediately reducible or the two operators $g_1$ and $g_2$ can be reduced to a basic operator by applying a sequence of exchange rules and reduction rules.

For example, the subexpression

$\text{CSHIFT}(c_1) \circ \text{CSHIFT}(c_2)$
is a redcom, and the subexpression

\[(\text{CSHIFT}(c_1) \times \text{CSHIFT}(c_2)) \circ \text{TRANS}(2) \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}
\circ (\text{CSHIFT}(c_3) \times \text{CSHIFT}(c_4)) \circ \text{CSKEW}(2) \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}\]

contains two redcoms: one is

\[(\text{CSHIFT}(c_1) \times \text{CSHIFT}(c_2)) \circ \text{TRANS}(2) \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}
\circ (\text{CSHIFT}(c_3) \times \text{CSHIFT}(c_4))\]

and the other is

\[\text{TRANS}(2) \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \circ (\text{CSHIFT}(c_3) \times \text{CSHIFT}(c_4)) \circ \text{CSKEW}(2) \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} .\]

Definition. If a communication expression contains no redcoms then the communication expression is said to be in its final form.

The compiler simplifies a communication expression by repeatedly reducing the redcoms in the expression until the expression reaches its final form. A communication expression may contain multiple redcoms, therefore there exist multiple choices in simplification order.

Compiler execution time may vary depending on the order of simplification. We use a simple heuristic to minimize simplification time. Under “owner-compute” model, the amount of communication is determined by the number of operators in a communication expression; i.e., a communication expression is simplified if its operator count is decreased. Our current heuristic employs a greedy algorithm which reduces immediately reducible operators as early as possible because that always reduces operator count. The algorithm also avoids infinite looping by adjusting the starting pointer after application of each exchange rule. Note that MULTID operators are denoted by the product of one-dimensional operators (e.g. product of ALIGN1, DIST, etc.). The simplification of the composition of MULTID operators proceeds by simplifying each product terms independently.

Recall that a communication expression is in the form

\[g_1 \circ \alpha \circ g_2^{-1} = (\gamma_1 \circ \beta_1 \circ \alpha_1) \circ \alpha \circ (\alpha_2^{-1} \circ \beta_2^{-1} \circ \gamma_2^{-1}) ,\]

where \(\alpha_i\) are alignment operators, \(\beta_i\) are distribution operators, and \(\gamma_i\) physical mapping operators (please refer to Figure 6). Following elaboration, we simplify a communication expression according to the order: alignment \(\alpha_1 \circ \alpha \circ \gamma_2^{-1}\) first (which is machine independent), then distribution \(\beta_1 \circ \beta_2^{-1}\) (which is somewhat machine dependent), and then physical mapping \(\gamma_1 \circ \gamma_2\) (which is completely machine dependent), as shown in Procedure simplify-communication (Figure 9). Since in most cases, the length of a communication expression is dominated by the number of alignment operators and array intrinsics in the expression, our heuristic is mainly applied to the simplification of alignment subexpressions, as shown in Procedure simplify-alignment (Figure 9). The procedure for simplifying distribution and physical mapping is just to cancel out \(g\) with \(g^{-1}\) in each dimension.

4.10 Discussion on Algebraic Simplification

In the communication algebra, each reduction rule results in a basic operator, decreasing the length of the communication expression. The inverse rules and the exchange rules do not increase the number of operators in the expression. Consequently, the algebraic simplification procedure converges.

The simplification procedure can be carried out efficiently. The length of a communication expression only depends on the number of levels in nested procedure calls, which usually is a small constant (less than 4 in most application programs). Let the length of a communication expression be \(L\) and the number of array references in the program be \(N\). It requires \(\frac{1}{2}L(L-1)N\) time steps to simplify the communication expressions for the entire program.

4.11 Optimization of Composition Order

The simplification procedure reduces a communication expression to its final form. The compiler then pattern matches the final form with the set of communication idioms. If the final form contains more than one alignment operators (i.e., which cannot be further simplified by current set of rules), it will either be pattern matched with multiple communication idioms or be unfolded and collapsed into a general communication function, depending on the characteristics of the target machine.

Note that the ordering of the operators in the composition in the final form does not affect the correctness of the target program. However, it does affect the cost of communication. For example, suppose a communication expression is reduced to a final form which contains a shift operation and a replication operation. There are two alternative order of composition:

\[(1) \ (\text{EOSHIFT}(1) \times \text{id})_{D_1 \times D_2 \rightarrow (D_1+1) \times D_2} \circ \text{SPREAD} \left( \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \text{ub}(D_2) \right)_{D_1 \rightarrow D_1 \times D_2} ; \]

\[(2) \ \text{SPREAD} \left( \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \text{ub}(D_2) \right)_{D_1+1 \rightarrow (D_1+1) \times D_2} \circ \text{EOSHIFT}(1)_{D_1 \rightarrow D_1+1} .\]
Procedure simplify-communication

Input: a communication expression $\gamma_1 \circ \beta_1 \circ \alpha_1 \circ \alpha_2^{-1} \circ \beta_2^{-1} \circ \gamma_2^{-1}$, where $\beta_1, \beta_2, \gamma_1, \gamma_2$ are $d$-dimensional operators denoted by the product of $d$ one-dimensional operators. We use $g_i$ to refer to the $i$th product component of multi-dimensional operator $g$ (i.e. $g_i$ is the one-dimensional operator in dimension $i$).

Output: a simplified communication expression

1. Call Procedure simplify-alignment to simplify the alignment subexpression $\alpha_1 \circ \alpha_2^{-1}$.
2. If the alignment subexpression is reduced to an identity operator, then simplify the distribution subexpression $\beta_1 \circ \beta_2^{-1}$ by reducing the composition of $\beta_1_i$ and $\beta_2_i$ in each dimension $i$. Otherwise, return the simplified expression (It will then be pattern matched with the communication idioms).
3. If the distribution subexpression is reduced to an identity operator, then simplify the physical mapping subexpression $\gamma_1 \circ \gamma_2^{-1}$ by reducing the composition of $\gamma_1_i$, and $\gamma_2_i$ in each dimension $i$. Otherwise, return the simplified expression (It will then be pattern matched with the communication idioms for layout conversion).
4. If the physical mapping subexpression is reduced to an identity operator, then no data movement is required.
   Otherwise, return the simplified expression for pattern matching with the communication idioms for changing physical mapping.

Procedure simplify-alignment

Input: an alignment subexpression

Output: a simplified alignment subexpression

Repeat step 1 to step 4 until no rules can be applied.

1. Reduce immediately reducible operators, except adjacent MULTID operators, in the expression by the inverse rules and reduction rules.
2. Apply the appropriate exchange rule to the first pair of operators in the expression that cannot be reduced immediately. Then move the starting pointer to right by one operator.
3. Perform Product-composition-exchange on adjacent MULTID operators in the expression, so that composition of MULTID operators becomes a product form whose components are composition of ALIGN1 operators.
4. Simplify each of the product components produced from step 3 using the rules in ALIGN1 subalgebra.

Expression (1), which contains replication in dimension two followed by end-of-shift in dimension one, costs more than expression (2), which contains shift in dimension one followed by replication in dimension two, since the communication volume for replication is the same in both expression, but expression (1) requires communicating $\text{sizeof}(D_1) \cdot \text{sizeof}(D_2)$ elements in the EOSHIFT operation, while expression (2) only requires communicating $\text{sizeof}(D_1)$ data elements.

To further optimize communication volume, we have communication idioms in the final form appear in the following order from right to left: message-reducing operators (e.g. reduction), then message-preserving operators (e.g. shift, transpose, affine transform), and then finally message-broadcasting operators (e.g. spread).

Example

Figure 10 shows the transformation result for the interprocedural layout conversion given in Figure 7. The simplified communication expression matches with the idiom for matrix transposition where the matrix is partitioned one-dimensionally. In the transformed program, the size of array $A$ is expanded to match template size according to the alignment directives. Calls to a communication routine matrix-transpose-1d-partition are inserted to move array $A$ to the proper layout before calling BETA and restore array $A$'s layout after returning from BETA.

Figure 12 of Appendix A illustrates the transformation for intra-procedural data movement.

5 EXPERIMENTAL STUDIES

Experiments were conducted to evaluate the effectiveness of the optimizations. Three synthetic codes (ALIGN1, ALIGN2, ALIGN3) were used to evaluate the benefit of algebraic simplification, and two benchmark codes (ADI: a PDE solver using Alternate Direction Method, and FFT: Fast Fourier Transform using hybrid block and cyclic dis-
procedure ALPHA in Figure 7

source program

real A(2,4)
template T1(3,6)
align A(i,j) with T1(i+1,j+2)
call BETA(A)

transformed program

real A'(2:3,3:6) TEMP(3:6,2:3)
call matrix-transpose-ld-partition(TEMP,A')
call BETA(TEMP)
call matrix-transpose-ld-partition(A',TEMP)

Simplification procedure

\[
\begin{align*}
\theta &= (SEQ_{D_2+2\rightarrow[0]\times(D_2+2) \times BLOCK(v)D_1+1\rightarrow P \times V_1}) \circ (EOSHIFT(2)_{D_2\rightarrow D_2+2 \times EOSHIFT(1)_{D_1\rightarrow D_1+1}}) \\
&\circ TRANS(2) \left( \begin{array}{cc} 0 & 1 \\ 1 & 0 \end{array} \right)_{D_1 \times D_2 \rightarrow D_2 \times D_1} \\
&\circ (EOSHIFT(-1)_{D_1+1\rightarrow D_1 \times EOSHIFT(-2)_{D_2\rightarrow D_2}}) \\
&\circ (SEQ^{-1}_{D_1+1\rightarrow[0]\times(D_1+1) \times BLOCK(v)D_2+2\rightarrow P \times V_2}) \\
&= \text{By Rule 11: exchange of MULTID and ALIGN2} \\
&\circ (SEQ_{D_1+2\rightarrow[0]\times(D_1+2) \times BLOCK(v)D_1+1\rightarrow P \times V_1}) \circ TRANS(2) \left( \begin{array}{cc} 0 & 1 \\ 1 & 0 \end{array} \right)_{D_1+1 \times D_2+2 \rightarrow D_2+2 \times D_1+1} \\
&\circ (EOSHIFT(1)_{D_1\rightarrow D_1+1 \times EOSHIFT(2)_{D_2\rightarrow D_2+2}) \circ (EOSHIFT(-1)_{D_1+1\rightarrow D_1 \times EOSHIFT(-2)_{D_2\rightarrow D_2}}) \\
&\circ (SEQ^{-1}_{D_1+1\rightarrow[0]\times(D_1+1) \times BLOCK(v)D_2+2\rightarrow P \times V_2}) \\
&= \text{By Rule 10: product composition exchange} \\
&\circ (SEQ_{D_2+2\rightarrow[0]\times(D_2+2) \times BLOCK(v)D_1+1\rightarrow P \times V_1}) \circ TRANS(2) \left( \begin{array}{cc} 0 & 1 \\ 1 & 0 \end{array} \right)_{D_1+1 \times D_2+2 \rightarrow D_2+2 \times D_1+1} \\
&\circ (EOSHIFT(1)_{D_1\rightarrow D_1+1 \times EOSHIFT(-1)_{D_1+1\rightarrow D_1}) \times (EOSHIFT(2)_{D_2\rightarrow D_2+2}) \circ (EOSHIFT(-1)_{D_2\rightarrow D_2+2}) \circ (EOSHIFT(-2)_{D_2\rightarrow D_2}) \\
&\circ (SEQ^{-1}_{D_1+1\rightarrow[0]\times(D_1+1) \times BLOCK(v)D_2+2\rightarrow P \times V_2}) \\
&= \text{By Rule 2: reduction of ALIGN1 operators} \\
&\circ (SEQ_{D_2+2\rightarrow[0]\times(D_2+2) \times BLOCK(v)D_1+1\rightarrow P \times V_1}) \circ TRANS(2) \left( \begin{array}{cc} 0 & 1 \\ 1 & 0 \end{array} \right)_{D_1+1 \times D_2+2 \rightarrow D_2+2 \times D_1+1} \\
&\circ (SEQ^{-1}_{D_1+1\rightarrow[0]\times(D_1+1) \times BLOCK(v)D_2+2\rightarrow P \times V_2}) \\
&\text{Match Idiom: matrix transposition}
\end{align*}
\]

Figure 10: Algebraic simplification for inter-procedural layout conversion.

We report our experimental results on the Connection Machines CM-5 located at AHPCRC of Minnesota University. The CM-5 has totally 986 processing nodes (PN), configured as various-sized partitions. Each processing node is a SPARC with four optional vector units that totally can deliver peak rate of 128MFlops [14].

In our experiments, we wrote the five benchmark codes in CM-Fortran syntax. Two versions of codes were generated for each benchmark code: a CM-Fortran version without any of the algebraic optimizations (called unoptimized version), and a CM-Fortran version which was the transformed result from algebraic optimization. We then compiled the CM-Fortran codes with vector-unit option on.

5.1 Algebraic Simplification

Table 3 shows the performance of the three synthetic codes on 64 processors. In ALIGN1, without optimization, array a was copied to a canonical heap temporary using shift communication, and then the two-dimensional EOSHIFT was carried out on the temporary. With algebraic simplification, the EOSHIFT intrinsic traffic in the assignment statement was cancelled out with the data alignment and therefore all data movement became local memory accesses. This optimization improves
Table 3. Execution Time in Seconds on 64 Processors

<table>
<thead>
<tr>
<th>Program</th>
<th>Problem size</th>
<th>unoptimized</th>
<th>optimized</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGN1</td>
<td>256 × 256</td>
<td>0.005</td>
<td>0.002</td>
<td>2.50</td>
</tr>
<tr>
<td></td>
<td>512 × 512</td>
<td>0.010</td>
<td>0.006</td>
<td>1.67</td>
</tr>
<tr>
<td></td>
<td>1k × 1k</td>
<td>0.027</td>
<td>0.021</td>
<td>1.29</td>
</tr>
<tr>
<td></td>
<td>2k × 2k</td>
<td>1.105</td>
<td>0.078</td>
<td>1.35</td>
</tr>
<tr>
<td></td>
<td>4k × 4k</td>
<td>0.477</td>
<td>0.305</td>
<td>1.56</td>
</tr>
<tr>
<td>ALIGN2</td>
<td>256 × 256</td>
<td>0.019</td>
<td>0.002</td>
<td>9.5</td>
</tr>
<tr>
<td></td>
<td>512 × 512</td>
<td>0.062</td>
<td>0.005</td>
<td>12.4</td>
</tr>
<tr>
<td></td>
<td>1k × 1k</td>
<td>0.208</td>
<td>0.016</td>
<td>13.0</td>
</tr>
<tr>
<td></td>
<td>2k × 2k</td>
<td>0.959</td>
<td>0.054</td>
<td>17.8</td>
</tr>
<tr>
<td></td>
<td>4k × 4k</td>
<td>4.061</td>
<td>0.203</td>
<td>20.0</td>
</tr>
<tr>
<td>ALIGN3</td>
<td>32 × 32 × 32</td>
<td>0.009</td>
<td>0.002</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>64 × 64 × 64</td>
<td>0.017</td>
<td>0.004</td>
<td>4.3</td>
</tr>
<tr>
<td></td>
<td>128 × 128 × 128</td>
<td>0.056</td>
<td>0.013</td>
<td>4.3</td>
</tr>
<tr>
<td></td>
<td>256 × 256 × 256</td>
<td>0.277</td>
<td>0.063</td>
<td>4.4</td>
</tr>
<tr>
<td></td>
<td>512 × 512 × 512</td>
<td>1.270</td>
<td>0.261</td>
<td>4.9</td>
</tr>
</tbody>
</table>

Table 4. Execution Time in Seconds on 64 Processors (ADI: 10 iterations, double precision floating points, FFT: double precision complex, with block-cyclic layout conversion)

<table>
<thead>
<tr>
<th>Program</th>
<th>Problem size</th>
<th>Unoptimized total(comm)</th>
<th>Optimized total(comm)</th>
<th>Speedup total(comm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADI</td>
<td>128 × 128</td>
<td>0.159 (0.127)</td>
<td>0.112 (0.081)</td>
<td>1.42 (1.57)</td>
</tr>
<tr>
<td></td>
<td>256 × 256</td>
<td>0.331 (0.270)</td>
<td>0.219 (0.161)</td>
<td>1.51 (1.68)</td>
</tr>
<tr>
<td></td>
<td>512 × 512</td>
<td>0.678 (0.562)</td>
<td>0.432 (0.319)</td>
<td>1.57 (1.76)</td>
</tr>
<tr>
<td></td>
<td>1k × 1k</td>
<td>1.511 (1.279)</td>
<td>0.863 (0.641)</td>
<td>1.75 (1.99)</td>
</tr>
<tr>
<td></td>
<td>2k × 2k</td>
<td>3.658 (3.191)</td>
<td>1.723 (1.275)</td>
<td>2.12 (2.50)</td>
</tr>
<tr>
<td>FFT</td>
<td>128k</td>
<td>0.137 (0.107)</td>
<td>0.074 (0.043)</td>
<td>1.85 (2.48)</td>
</tr>
<tr>
<td></td>
<td>256k</td>
<td>0.201 (0.143)</td>
<td>0.109 (0.055)</td>
<td>1.84 (2.66)</td>
</tr>
<tr>
<td></td>
<td>512k</td>
<td>0.337 (0.231)</td>
<td>0.186 (0.083)</td>
<td>1.80 (2.78)</td>
</tr>
<tr>
<td></td>
<td>1M</td>
<td>0.635 (0.428)</td>
<td>0.357 (0.152)</td>
<td>1.78 (2.82)</td>
</tr>
<tr>
<td></td>
<td>2M</td>
<td>1.262 (1.835)</td>
<td>0.701 (0.278)</td>
<td>1.80 (3.01)</td>
</tr>
<tr>
<td></td>
<td>4M</td>
<td>2.692 (1.746)</td>
<td>1.369 (0.499)</td>
<td>1.92 (3.50)</td>
</tr>
</tbody>
</table>

program performance of ALIGN1 by a factor of 1.29 to 2.50. In ALIGN2, without optimization, array a was moved to array b using general communication. With algebraic simplification, the TRANSPOSE intrinsic traffic was cancelled out with the effect of data alignment and therefore actual data movement becomes a two-dimensional EOSHIFT operation, which involves only nearest-neighbor communication. This optimization improved program performance of ALIGN2 by a factor of 9.5 to 20.0. Speedup factors increase with problem sizes. In ALIGN3, again, without optimization, the assignment statement was carried out via copying through a canonical heap temporary. With algebraic simplification, the actual data movement become local copy on array section. Program performance was improved by a factor of 4 to 5.

5.2 Data Layout Conversion

Table 4 shows performance of ADI and FFT on 64 processors. The computation time in different versions is almost identical. The unoptimized versions use general communication for layout conversion. With optimized transposition operations, communication time of ADI was improved by a factor of 1.6 to 2.5 and the total execution time of ADI was improved by a factor of 1.4 to 2.1. Speedup factors increase with problem sized. A possible explanation is that on CM-5 a long message is sent in patches; larger problem sizes produce longer messages, and therefore heavier traffic in the network and higher speedup factors due to optimized communication. With optimized block-cyclic layout conversions, communication time of FFT was improved by a factor of 1.4 to 3.5 and the total execution time was improved by a factor of
Table 5. Execution Time in Seconds on Different Number of Processors with Fixed Per-Processor Problem Size (ADI: double precision, 128 × 128 per-processor problem size, FFT: double precision complex, 8k per-processor problem size)

<table>
<thead>
<tr>
<th>Program</th>
<th>nproc</th>
<th>Problem size</th>
<th>Unoptimized total(comm)</th>
<th>Optimized total(comm)</th>
<th>Speedup total(comm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADI</td>
<td>32</td>
<td>1k × 2k</td>
<td>3.07 (2.73)</td>
<td>1.61 (1.26)</td>
<td>1.90 (2.15)</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>2k × 2k</td>
<td>3.66 (3.19)</td>
<td>1.72 (1.27)</td>
<td>2.12 (2.50)</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>2k × 4k</td>
<td>5.28 (4.78)</td>
<td>2.33 (1.83)</td>
<td>2.27 (2.61)</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>4k × 4k</td>
<td>8.17 (7.23)</td>
<td>3.52 (2.61)</td>
<td>2.32 (2.76)</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>4k × 8k</td>
<td>11.59 (10.63)</td>
<td>4.65 (3.72)</td>
<td>2.49 (2.86)</td>
</tr>
<tr>
<td>FFT</td>
<td>32</td>
<td>256k</td>
<td>0.313 (0.208)</td>
<td>0.178 (0.073)</td>
<td>1.75 (2.85)</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>512k</td>
<td>0.336 (0.231)</td>
<td>0.185 (0.083)</td>
<td>1.80 (2.78)</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>1M</td>
<td>0.416 (0.311)</td>
<td>0.221 (0.114)</td>
<td>1.88 (2.73)</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>2M</td>
<td>0.648 (0.538)</td>
<td>0.282 (0.173)</td>
<td>2.29 (3.10)</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>4M</td>
<td>1.027 (0.924)</td>
<td>0.394 (0.284)</td>
<td>2.61 (3.25)</td>
</tr>
</tbody>
</table>

1.3 to 1.9. The speedup factors in FFT are consistent with the results of ADI.

Table 5 shows the scaled speedups (by fixing per-processor problem size and changing number of processors) of ADI and FFT. When per-processor size is fixed, speedup factors for ADI increase with number of processors (by a factor of 2.15 on 32 processors to a factor of 2.86 on 515 processors), because number of messages increases linearly with machine size and message contention becomes a more serious problem on larger machines. For larger machine sizes, speedup factors for FFT also increase with machine sizes.

5.3 Summary

The results from the three synthetic codes all show positive impact of algebraic simplification on program performance, because it is always beneficial to reduce away redundant layout conversions between procedure calls and unnecessary local copying through canonical temporary storage.

The results from the two benchmark codes ADI and FFT show that optimized layout conversion (compile-time idiom matching + specialized runtime communication routine) can reduce communication time significantly. The reason is that even on a regular communication architecture like CM-5, message contention may cause inefficiency. Both of the two benchmark codes involve all-to-all communication, which produces heavy message traffic in the network. By carefully scheduling these messages, contention can be reduced greatly. Research has shown that message contention problem is present on many massively parallel machines [4, 5, 36, 37, 39]. Consequently, those machines will also profit from this optimization (perhaps with different implementation of the runtime communication routines).

6 RELATED WORK

A number of prototype compilers for Fortran 90/HPF have been developed in the past few years. We first briefly review the communication optimization techniques used in some of these compilers. The Fortran D compiler [20, 44] performs various optimizations (message vectorization, message pipelining) to reduce communication overhead. However, the Fortran D compiler currently only handle a small subset of HPF’s data layouts: canonical alignment and one-dimensional data partitioning, while our framework is applicable to more general cases. The Fortran 90D compiler [6] optimizes data movement for subscripted array references in parallel loops using linear index-function transformation and pattern matching for collective communication. By formulating data movement using linear transformations, optimization for non-linear alignment, such as CSHIFT and replication, and data redistribution are not possible. Vienna Fortran and Vienna Fortran-90, based upon the parallelizing system SUPERB, extends Fortran/Fortran-90 by providing alignment and distribution specifications. The Vienna Fortran compiler [8, 48] currently only supports arbitrary rectilinear block distributions. The ADAPT system [35], developed at University of Southampton, compiles Fortran 90 for execution on MIMD distributed-memory machines. The ADAPT system also simplifies data movement problem by restricting itself to a universal communication model. The SUIF compilation system [3, 46] developed at Stanford University uses integer matrix notations and affine transformation for optimizing data movement in data-parallel programs. This approach is insufficient for handling HPF’s non-linear alignment operations (e.g. CSHIFT (cyclic shift), CSKEW (cyclic skew), and SPREAD (replication)) and data redistribution.

Of the part of industry, the CM Fortran compiler [15] uses simple but naive copy-in, copy-out strategy for interprocedural data movement, and copying via canonical
temporary for intra-procedural data movement. To our knowledge, many commercial compilers either only support a subset of HPF standard alignment and distribution specifications or, although they support full HPF data distributions, do not tackle complex data movement optimization issues like we do (e.g. APR’s Forge90 compiler and the IBM xHPF compiler).

Our algebraic transformative framework also relates to other more specific research efforts. The technique for generating collective communication, pioneered by the Crystal compiler [32, 33], has great influence on our work. The major differences are:

1. the Crystal compiler finds optimal (or near optimal) data alignment automatically, while our framework optimizes data movement in the presence of user-provided data layout specifications; and
2. the Crystal compiler does not optimize inter-procedural data movement as we do.

The array synthesis scheme proposed by Hwang [25] employs index function transformation for HPF’s alignment directives and array intrinsics. Array operations are translated to nested loops with explicit index subscript expressions. Non-linear array operations are handled by duplicating multiple copies of the loop nest, each corresponding to a particular boundary condition. Our concern with this approach is that the program size may increase rapidly with the number of non-linear operations.

Another important research area is generating communication sets for array section movement. Several approaches have addressed the efficient execution of array statements involving block-cyclically distributed array sections (e.g. Gupta et al.’s virtual processor approach [28, 38], Chatterjee et al.’s finite-state machine approach [9], Stichnoth’s [40, 41] array slice analysis, Kenedy et al.’s [29] and Thirumalai and Ramanujam’s [43] integer lattice approach, Venkatachar et al.’s [45] row-column padding techniques, and Reeuwijk et al.’s [11] address generation framework). We would like to point out that the algebraic transformative framework presented in this paper does not compete with that work. The main focus of this paper is a framework for high-level optimization of data movement (i.e. the optimization, including algebraic simplification and idiom matching for collective communication, is performed purely in the global, logical space defined in the program, working on the algebraic representation level only). A lower level framework, which is not presented in this paper, handles all the details of generating send/receive pairs. Internally, the lower level framework employs similar techniques borrowed from existing literatures listed above.

There are also work on global optimization for data movement. Gilbert and Schreiber [10, 17] designed a dynamic programming algorithm for optimizing temporary storage use for Fortran 90 array statements. Chatterjee et al. [11, 12] extended that work to allow loop nests. Ju et al. [27] (and later Hwang et al. [25]) proposed a synthesis scheme for combining consecutive data reference patterns to reduce communication. Another line of work on global optimization of communication is based on dataflow analysis, e.g. Amarasinghe and Lam’s [2], Gong et al.’s [18], and Gupta et al.’s [19] work. The current implementation of our optimization framework assumes owner-computer rule for compilation of data movement. We plan to incorporate global optimization techniques in the near future.

7 CONCLUSIONS

In this paper, we have described the theoretical aspect and experimental results of the algebraic transformation framework. We expect the effectiveness of this optimization technique to be even more significant for larger application programs which usually contain many program modules and may involve abundant use of array operations.

Two major optimization primitives in our framework are algebraic simplification of communication expressions and idiom matching for fast communication. Most of the communication idioms we have collected are not architecture-specific. They may or may not have specialized, fast communication, depending on the target machine. Specialized implementation of communication routines may or may not have significant impact on more regular communication architectures. As a result, idiom matching may not be crucial to achieving high performance on this kind of machines. On the other hand, algebraic simplification is a high-level, abstract transformation technique that carries out data movement reduction within the purely logical, global space defined in the program. Any redundant layout conversions between procedure calls and any unnecessary local copying through canonical temporary storage will be reduced away abstractly by algebraic simplification. Consequently, even on a very balanced, regular communication architecture, communication overhead can still be reduced by high-level pattern matching and algebraically simplifying them.

The algebraic transformation framework (including algebraic representation of data movement, alignment and distribution, an algebraic engine and associated heuristics, and runtime layout conversion and communication services) allows a compiler to reduce data movement at the abstract level and leave machine-dependent details to
the runtime system. Although presented in the context of an optimizing compiler, the algebraic transformation framework can also be used as part of a runtime system for optimizing data movement that is dynamic or dependent on runtime computed values. By modeling different stages of data mapping (alignment, distribution, physical mapping) and data movement using communication expressions and providing algebraic rules to simplify each stage of data movement, the algebraic framework is conceptually clean and portable to different target architectures.

Recently, there has been some progress on the part of industry toward applying some simple kinds of layout optimizations in commercial HPF compilers. For example, TMC's CM-Fortran compiler version 2.2 has included some similar optimization techniques for a very restricted subset of layout directives (shift/shift combination). We hope that eventually most commercial compiler groups will adopt our algebraic transformation strategy, or something similar to it, to make sure that users can write HPF code without worrying about compiler blind spots (like "copy in – copy out" calling sequences, or redundant sequences of copies through heap temporaries whenever non-trivial alignments are in force).

ACKNOWLEDGEMENTS

The authors thank Young-il Choo and James Cowie for helpful discussions on the presentation of this work. The authors also thank the anonymous referees for many helpful comments on earlier versions of the manuscript.

Support for this work was provided by Cooperating Systems Corporation and Syracuse University under ARPA contract DABT63-91-C-0031, the Army High Performance Computing Research Center at Minnesota University, and the Institute of Information Science at Academia Sinica.

REFERENCES


Appendix A

Figure 11 shows some examples of communication expressions for intra-procedural data movement. In order to formalize data movement for the assignment statements, it is necessary to construct the layouts of the local data and to use these constructions in the application of the Intra-Procedural Rule (Figure 6b).
REAL A(100,200), B(100,200), C(100), D(100)

!HPF$ TEMPLATE, DISTRIBUTED (BLOCK,BLOCK) :: T(202,101)
!HPF$ ALIGN A(I,J) WITH T(J+2,I+1)
!HPF$ ALIGN B(I,J) WITH T(I,J)
!HPF$ ALIGN C(I) WITH T(1,J)
!HPF$ ALIGN D(I) WITH T(1,101-I)

S1  B = EOSHIFT(EOSHIFT(TRANSPOSE(A),dim=1,shift=-1),dim=2,shift=-1)
S2  C = CSHIFT(D,dim=1,shift=-1)

Index domains:
D1 = Interval(1,100),    D2 = Interval(1,200).

Data movement for statement S1:
RHS layout (array A):

\[ g_{rA} = (EOSHIFT(2)_{D2\rightarrow D2+2} \times EOSHIFT(1)_{D1\rightarrow D1+1}) \circ \text{TRANS}^2 \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{D1 \times D2 \rightarrow D2 \times D1} \]

Array reference:

\[ \alpha_A = (EOSHIFT(1)_{D2\rightarrow D2+1} \times EOSHIFT(1)_{D1\rightarrow D1+1}) \circ \text{TRANS}^2 \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{D1 \times D2 \rightarrow D2 \times D1} \]

LHS layout (array A):

\[ g_{lA} = \text{id} \circ \alpha_A \circ g_{rA}^{-1} \]

(see Figure 6b)

Data movement for statement S2:
RHS layout (array D):

\[ g_{rD} = (\text{id} \times \text{REFLECT}_{D1\rightarrow D1}) \circ \text{EMBED}^{(1,2)} \begin{pmatrix} 0 \\ 1 \end{pmatrix}_{D1 \rightarrow ([1] \times D1)} \]

array reference:

\[ \alpha_D = \text{CSHIFT}(1)_{D1\rightarrow D1} ; \]

LHS layout (array D):

\[ g_{lD} = \text{EMBED}^{(1,2)} \begin{pmatrix} 0 \\ 1 \end{pmatrix}_{D1 \rightarrow ([1] \times D1)} \]

actual data movement:

\[ \theta_D = g_{lD} \circ \alpha_D \circ g_{rD}^{-1} \]

(see Figure 6b)

\[ = \text{EMBED}^{(1,2)} \begin{pmatrix} 0 \\ 1 \end{pmatrix}_{D1 \rightarrow ([1] \times D1)} \circ \text{CSHIFT}(1)_{D1\rightarrow D1} \circ \text{EMBED}^{(1,2)} \begin{pmatrix} 0 \\ 1 \end{pmatrix}_{D1 \rightarrow ([1] \times D1)} \circ (\text{id} \times \text{REFLECT}^{-1}_{D1\rightarrow D}) \]

FIGURE 11 Communication expressions for intra-procedural data movement. For simplicity, we only show the alignment subexpressions.
procedure ALPHA in Figure 11

source program

real A(100,200), B(100,200), C(100), D(100)
template T(202,101)
align A(i,j) with T(j+2,i+1)
align C(i) with T(1,i)
align D(i) with T(1,101-i)

transformed program

real A'(3:202, 2:101), B'(100,200)
real C'(1,100), D'(1,100)

S1 B = EOSHIFT(EOSHIFT(TRANSPOSE(A),
    dim=1,shift=-1),dim=2,shift=-1)
S2 C = CSHIFT(D,dim=1,shift=-1)

Simplification procedure:

Data movement for statement S1:

\[ \theta_A = (\text{EOSHIFT}(1)D_2 \rightarrow D_2 + 1 \times \text{CSHIFT}(1)D_1 \rightarrow D_1 + 1) \circ \text{TRANS} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \circ \text{TRANS} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \]

\[ \circ (\text{EOSHIFT}(-2)D_2 + 2 \times \text{EOSHIFT}(-1)D_1 + 1 \rightarrow D_1) \]

By Rule 5: Reduction of ALIGN2 operators

\[ (\text{EOSHIFT}(1)D_2 \rightarrow D_2 + 1 \times \text{EOSHIFT}(1)D_1 \rightarrow D_1 + 1) \circ (\text{EOSHIFT}(-2)D_2 + 2 \rightarrow D_2 \times \text{EOSHIFT}(-1)D_1 + 1 \rightarrow D_1) \]

By Rule 10: Product-Composition Exchange

\[ (\text{EOSHIFT}(1)D_2 \rightarrow D_2 + 1 \circ \text{EOSHIFT}(-2)D_2 + 2 \rightarrow D_2) \times (\text{EOSHIFT}(1)D_1 \rightarrow D_1 + 1 \circ \text{EOSHIFT}(-1)D_1 + 1 \rightarrow D_1) \]

By Rule 2: Reduction of ALIGN1 operators

\[ \text{EOSHIFT}(-1)D_2 + 2 \times D_2 + 1 \times \text{id}D_1 + 1 \rightarrow D_1 + 1 \]

Data movement for statement S2:

\[ \theta_D = \text{EMBED}^2 \begin{pmatrix} 0 \\ 1 \end{pmatrix}_{D_1 \rightarrow ([1] \times D_1)} \circ \text{CSH}(1)D_1 \rightarrow D_1 \circ \text{EMBED}^2 \begin{pmatrix} 0 \\ 1 \end{pmatrix}_{D_1 \rightarrow ([1] \times D_1)} \circ \text{id} \times \text{REFLECT}^{-1}_{D_1 \rightarrow D_1} \]

By Rule 12: Exchange of ALIGN1 and ALIGN3

\[ \text{CSH} \begin{pmatrix} 0 \\ 1 \end{pmatrix}_{[1] \times D_1 \rightarrow [1] \times D_1} \circ \text{EMBED}^2 \begin{pmatrix} 0 \\ 1 \end{pmatrix}_{D_1 \rightarrow ([1] \times D_1)} \circ \text{EMBED}^2 \begin{pmatrix} 0 \\ 1 \end{pmatrix}_{D_1 \rightarrow ([1] \times D_1)} \circ \text{id} \times \text{REFLECT}^{-1}_{D_1 \rightarrow D_1} \]

By Rule 8: Inverse cancellation of ALIGN3

\[ \text{CSH} \begin{pmatrix} 0 \\ 1 \end{pmatrix}_{[1] \times D_1 \rightarrow [1] \times D_1} \circ \text{id}[1] \times \text{REFLECT}_{D_1 \rightarrow D_1} \]

By definition of CSH

\[ \circ \text{id}[1] \times \text{CSHIFT}_{D_1 \rightarrow D_1} \circ \text{id}[1] \times \text{REFLECT}_{D_1 \rightarrow D_1} \]

By Rule 10: Product-composition exchange

\[ \text{id}[1] \times (\text{CSHIFT}_{D_1 \rightarrow D_1} \circ \text{REFLECT}_{D_1 \rightarrow D_1}). \]

FIGURE 12 Algebraic simplification for intra-procedural data movement.

Figure 12 illustrates the algebraic transformations for these expressions. Data movement in the first statement is reduced to an one-dimensional CSHIFT operation, while the second statement requires an additional reflection operation (REFLECT) due to the effect of the data alignment directives.

Appendix B

This appendix gives proofs of some of the rules in each communication subalgebra. The inverse rules (Rules 1, 4, and 8) are obvious. Proofs of Rules 2(1), 5(3), and 9(2) are give below. Proofs of other rules are analogous.
Rule 2(1) $\text{CSHIFT}(c_2)_{D+c_1 \rightarrow D+c_1} \circ \text{EOSHIFT}(c_1)_{D \rightarrow D+c_1} = \text{EOSHIFT}(c_1)_{D \rightarrow D+c_1} \circ \text{CSHIFT}(c_2)_{D \rightarrow D}$. 

Proof.

$$\left(\text{CSHIFT}(c_2)_{D+c_1 \rightarrow D+c_1} \circ \text{EOSHIFT}(c_1)_{D \rightarrow D+c_1}\right)(i \in D)$$

- By definition of EOSHIFT

$$\left(\text{CSHIFT}(c_2)_{D+c_1 \rightarrow D+c_1}\right)(i + c_1) \in D + c_1$$

- By definition of CSHIFT

$$1b(D + c_1) + \left(i + c_1 - 1b(D + c_1) + c_2 \mod \left(ub(D + c_1) - 1b(D + c_1)\right)\right)$$

$$= 1b(D) + c_1 + \left(i + c_1 - 1b(D) - c_1 + c_2 \mod \left(ub(D) - 1b(D)\right)\right)$$

$$= 1b(D) + \left(i - 1b(D) + c_2 \mod |D| + c_1\right).$$

$$\left(\text{EOSHIFT}(c_1)_{D \rightarrow D+c_1} \circ \text{CSHIFT}(c_2)_{D \rightarrow D}\right)(i \in D)$$

- By definition of CSHIFT

$$\text{EOSHIFT}(c_1)_{D \rightarrow D+c_1}(1b(D) + \left(i - 1b(D) + c_2 \mod |D|\right)$$

$$= 1b(D) + \left(i - 1b(D) + c_2 \mod |D| + c_1\right). \quad \square$$

Rule 5(3) $\text{CSKEW}^{(d)}(M_2)_{D \rightarrow D} \circ \text{CSKEW}^{(d)}(M_1)_{D \rightarrow D} = \text{CSKEW}^{(d)}(M_2 * M_1)_{D \rightarrow D}$. 

Proof. Let $I = (i_1, \ldots, i_d)$, $D = D_1 \times \cdots \times D_d$, $\text{Mod}(I, |D|) = (i_1 \mod |D_1|, \ldots, i_d \mod |D_d|)$. 

$$\left(\text{CSKEW}^{(d)}(M_2)_{D \rightarrow D} \circ \text{CSKEW}^{(d)}(M_1)_{D \rightarrow D}\right)(I \in D)$$

- By definition of CSKEW

$$\text{CSKEW}^{(d)}(M_2)_{D \rightarrow D}(\text{Mod}(M_1 * I, |D|))$$

- By definition of CSKEW

$$\text{Mod}(M_2 * (\text{Mod}(M_1 * I, |D|)))$$

- By property of modulus

$$\text{Mod}((M_2 * M_1) * I, |D|).$$

$$\text{CSKEW}^{(d)}(M_2 * M_1)_{D \rightarrow D}(I \in D)$$

- By definition of CSKEW

$$\text{Mod}((M_2 * M_1) * I, |D|). \quad \square$$

Rule 9(2)

$$\text{REPLICATE}^{(n)}(V_2, E_2)_{D_2 \rightarrow D_3} \circ \text{REPLICATE}^{(n)}(V_1, E_1)_{D_1 \rightarrow D_2} = \text{REPLICATE}^{(n)}(V_2 + V_1, E_1 \times E_2)_{D_1 \rightarrow D_3}.$$ 

Proof. Let $V_1(k) \neq 0$, $V_2(l) \neq 0$. 

$$\left(\text{REPLICATE}^{(n)}(V_2, E_2)_{D_2 \rightarrow D_3} \circ \text{REPLICATE}^{(n)}(V_1, E_1)_{D_1 \rightarrow D_2}\right)(i_1, \ldots, i_k-1, i_k+1, \ldots, i_{l-1}, i_{l+1}, \ldots, i_n) \in D_1$$

- By definition of REPLICATE

$$\left(\text{REPLICATE}^{(n)}(V_2, E_2)_{D_2 \rightarrow D_3}\right)(i_1, \ldots, i_k-1, i_k, i_k+1, \ldots, i_{l-1}, i_{l+1}, \ldots, i_n)$$

- By definition of REPLICATE

$$(i_1, \ldots, i_k-1, i_k, i_k+1, \ldots, i_{l-1}, i_{l+1}, \ldots, i_n).$$

$$\left(\text{REPLICATE}^{(n)}(V_2 + V_1, E_1 \times E_2)_{D_1 \rightarrow D_3}\right)(i_1, \ldots, i_k-1, i_k+1, \ldots, i_{l-1}, i_{l+1}, \ldots, i_n) \in D_1$$

- for $V(k) \neq 0$, $V(l) \neq 0$

$$\left(\text{REPLICATE}^{(n)}(V, E_1 \times E_2)_{D_1 \rightarrow D_3}\right)(i_1, \ldots, i_k-1, i_k+1, \ldots, i_{l-1}, i_{l+1}, \ldots, i_n) \in D_1$$

- By definition of REPLICATE

$$(i_1, \ldots, i_k-1, i_k, i_k+1, \ldots, i_{l-1}, i_{l+1}, \ldots, i_n). \quad \square$$
Appendix C: benchmark codes

program ALIGN1
cc test program for ALIGN1
subalgebra
integer m,n,ntimes
!HPF$ template T(m,n)
!HPF$ distribute T (block,block)
real a(m-1,n-1), b(m-1,n-1)
!HPF$ align a(i,j) with T(i+1,j+1)
!HPF$ align b(i,j) with T(i,j)

   do n = 1, ntimes
   b = eoshift(eoshift(a,dim=1,
      shift=1,dim=2,shift=1)
   enddo

program ALIGN2
cc test program for ALIGN1
and ALIGN2 subalgebras
integer n,ntimes
!HPF$ template T(n,n)
!HPF$ distribute T (block,block)
real a(n-1,n-1), b(n-1,n-1)
!HPF$ align a(i,j) with T(j+1,i+1)
!HPF$ align b(i,j) with T(i,j)

   do n = 1, ntimes
   b = transpose (a)
   enddo

program ALIGN3
cc test program for ALIGN1
and ALIGN3 subalgebras
integer m,n,k,ntimes
!HPF$ template T(m,n,k)
!HPF$ distribute (block,block,block)
real a(m,n-2), b(m,n-2)
!HPF$ align a(i,j) with T(i,j+2,1)
!HPF$ align b(i,j) with T(i,j,1)

   do n = 1, ntimes
   a = eoshift (b,2,2)
   enddo

program ADI
cc Alternate Direction Implicit
method
cc for PDE solver

real U(n,n), V(n,n)
!HPF$ distribute U (*,block)
!HPF$ align V(i,j) with U(i,j)

   do i = 1, ntimes
      ...
      call tridiag_solver(V)
      !HPF$ realign V(i,j) with U(j,i)
      call tridiag_solver(V)
      !HPF$ realign V(i,j) with U(i,j)
      U=V
   enddo

program FFT
cc Fast Fourier Transform
cc using hybrid data distribution
complex x(n), w
logical ones(n)
integer p

!hpf$ distribute cyclic :: x

   p = number_0f_processors()
   ...

cc butterfly stages using cyclic
distribution
   do k = log2(n)-1, log2(p), -1
      where (ones)
         x = x + w * cshift(x,dim=1,
            shift=2**k)
      elsewhere
         x = x * (-w) + cshift(x,dim=1,
            shift=-2**k)
      end where
   end do

cc butterfly stages using
block distribution
   do k = log2{p)-1, 0, -1
      where (ones)
         x = x + w * cshift(x,dim=1,
            shift=2**k)
      elsewhere
         x = x * (-w) + cshift(x,dim=1,
            shift=-2**k)
      end where
   end do