An Algebraic Machinery for Optimizing Data Motion for HPF

JAN-JAN[1] WU AND MARINA C. CHEN[2]

[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 over-head 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 posibilities 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-

© 1997 IOS Press
ISSN 1058-9244/97/$8
Scientific Programming, Vol. 6, pp. 297–325 (1997) .

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 is 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

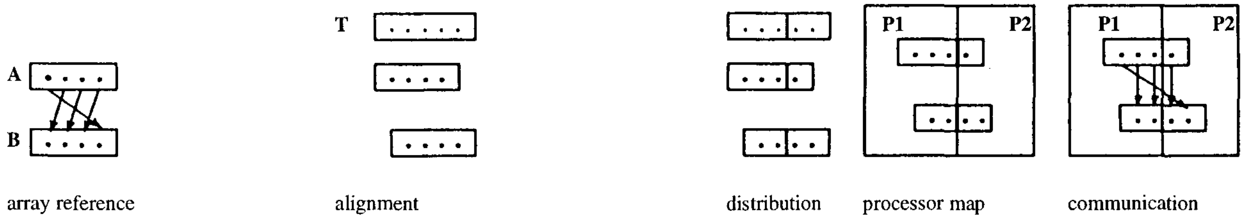
```
B = CSHIFT(A,dim=1,shift=1)
```

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 fo 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.

```
!!! HPF code
        REAL, DIMENSION(4)::A,B
!HPF$   TEMPLATE T(5)
!HPF$   DISTRIBUTE T BLOCK
!HPF$   ALIGN A(I) WITH T(I)
!HPF$   ALIGN B(I) WITH T(I+1)
        B = CSHIFT (A, dim=1, shift=1)
```



```
real, dimension(4)::A,B
B=CSHIFT(A,1)
HPF$ distribute T block
!HPF$ template T(5)
!HPF$ align A(i) with T(i)
!HPF$ align B(i) with T(i+1)
```

FIGURE 1 An example of data mapping and data movement in HPF. The alignment of the two arrays is intended for the eventual reduction of the communication in the CSHIFT array intrinsic.

Figure 2 shows three levels of procedure calls (ALPHA calls BETA calls FOO calls BAR). For the procedure call statement in ALPHA, both the actual (array A) and the dummy (array B) have explicit alignments. Array A is aligned to the template T by offseting one element. The template T, by default, is partitioned into contiguous blocks. In procedure BETA, array B (the dummy argument of BETA) is aligned to T by offseting two data elements. Therefore, data realignment is required for passing array A between ALPHA and BETA. In procedure FOO, the dummy argument (array C) inherits the alignment directive of the actual argument (array B). By aligning to the dummy argument C, the local array D defined in procedure FOO also inherits the actual argument's alignment specification, resulting in totally offseting five data elements with respective to template T. This layout effect will be propagated to the procedure interface when calling procedure BAR. In the procedure body of BAR, the dummy argument E inherits its actual argument's (array D) alignment, which differs from the local array L's alignment specification. When the calling chain is long, a systematic approach is desirable to automate the process of optimizing data movement for passing array arguments across procedure boundaries, as well as moving data elements in array assignment statements within a procedure body.

```
!!!  explicit  /         explicit
ALPHA (A)                BETA (B)
real A(100)              real B(100)
align A(i) with T(i+1)   align B(i) with T(i+2)
call BETA(A)             call FOO(B)
-----------------------------------------------
!!!  inherited  /        inherited
FOO(C)                   BAR(E)
real C(100)              real E(100)
align C *                align E *
real D(100)              real L(100)
align D(i) with C(i+3)   align L(i) with T(i)
call BAR(D)              D = L
```

FIGURE 2 Examples of HPF procedure interfaces.

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 offseting two elements and offseting 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.

```
!!! actual explicit / dummy explicit
ALPHA                  BETA(C,D)
real A(100),B(100)     real C(100),D(100)
distribute T cyclic    distribute T block
align A(i) with T(i)   align C(i) with T(i)
align B(i) with T(i+2) align D(i) with T(i+1)
call BETA(A,B)         D = EOSHIFT(C,dim=1,shift=1)
                       ...
```

FIGURE 3 An example for data movement optimization where both the actual and dummy arguments have explicit data layouts.

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 offseting 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 fraffic 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.

*By "canonical" we mean an array is only aligned to itself identically, and is distributed to the processors using the default distribution strategy. For instance, if the array size is 8 × 8 and the number of processors is four, CM-Fortran compiler partitions the array contiguously into four equal-sized (4 × 4) subarrays, and assign one subarray to one processor distinctly.

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 fransformation technique called affine transformation for optimizing data-parallel programs [3, 46]. General matrix notations and affine transformations are insufficient

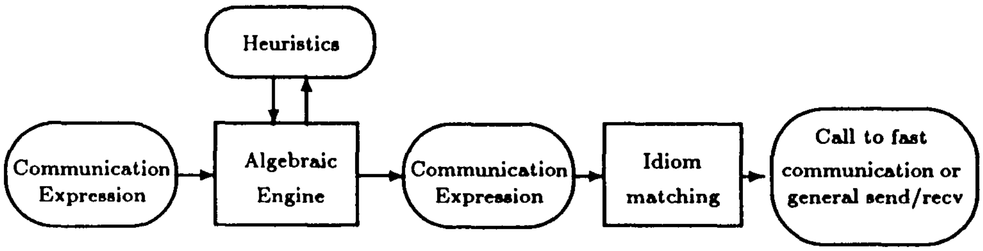


FIGURE 4 Overview of the algebraic analysis framework.

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 offseting 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 $\text{TRANS}^{(d)}(M)$, 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 $\text{SKEW}^{(d)}(M)$ 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

$$\text{CSKEW}^{(d)}(M)$$

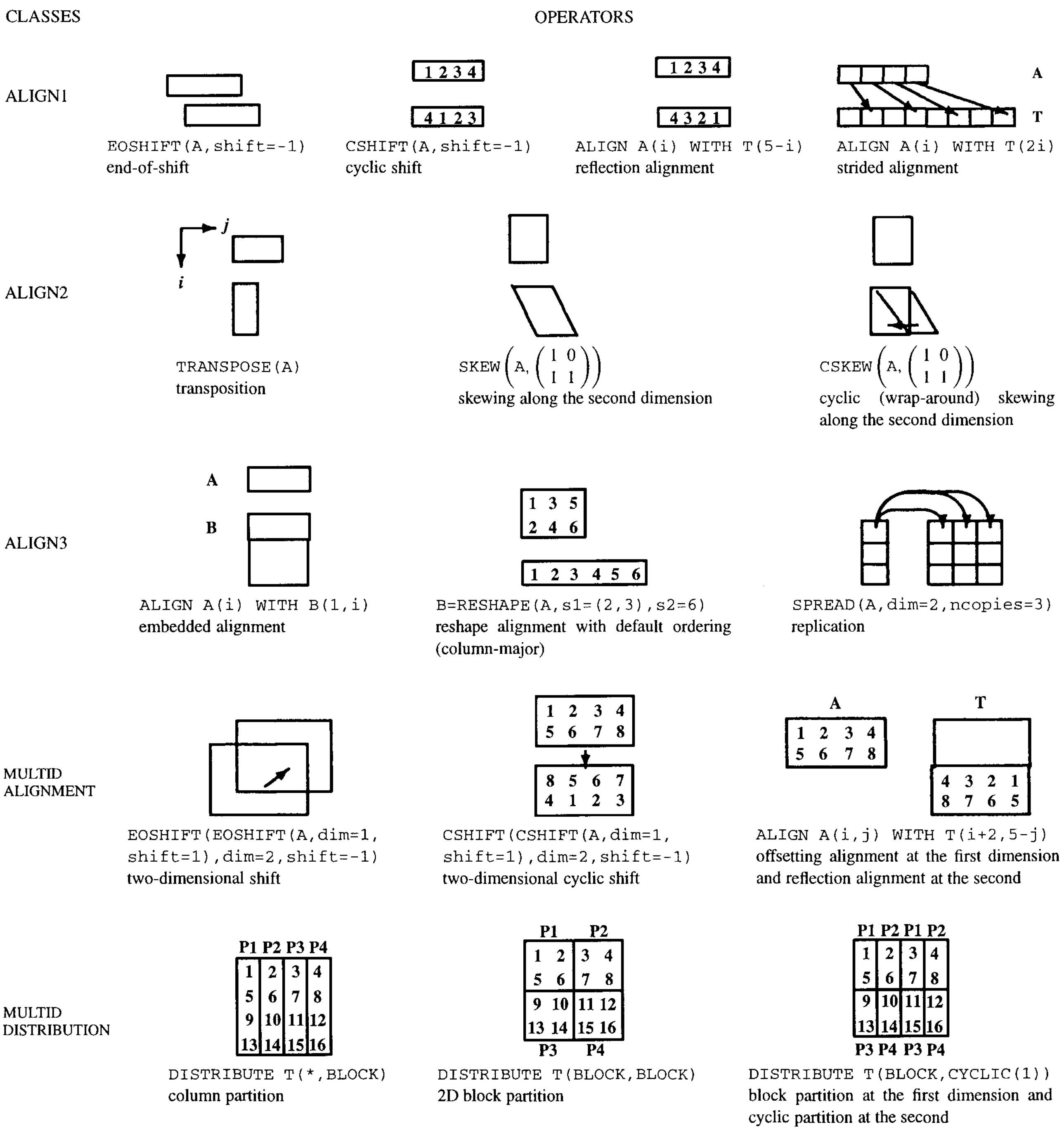
operator performs skewing operations in a wrap-around fashion.

In ALIGN3, both the array intrinsic

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

and the alignment directive

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



ALIGN1

`EOSHIFT(A,shift=-1)` end-of-shift

`CSHIFT(A,shift=-1)` cyclic shift

`ALIGN A(i) WITH T(5-i)` reflection alignment

`ALIGN A(i) WITH T(2i)` strided alignment

ALIGN2

`TRANSPOSE(A)` transposition

$\mathtt{SKEW}\left(\mathtt{A}, \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}\right)$ skewing along the second dimension

$\mathtt{CSKEW}\left(\mathtt{A}, \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}\right)$ cyclic (wrap-around) 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)

`SPREAD(A,dim=2,ncopies=3)` replication

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

`ALIGN A(i,j) WITH T(i+2,5-j)` offsetting alignment at the first dimension and reflection alignment at the second

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

Class	HPF Intrinsics/directives	Internal representations
ALIGN1	EOSHIFT(A,dim,shift=-c)	EOSHIFT(c)
	ALIGN A(i) WITH T(i+c)	
	CSHIFT(A,dim,shift=-c)	CSHIFT(c)
	ALIGN A(i) WITH T(n-i+1)	REFLECT
	ALIGN A(i) WITH T(a*i+c)	STRIDE(a, c)
ALIGN2	TRANSPOSE(A)	$\texttt{TRANS}^{(2)}\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$
	ALIGN A(i,j) WITH T(j,i)	
	TRANSPOSE(A, permute_vector)	$\texttt{TRANS}^{(d)}(M)$, where M is the matrix representation for permute_vector
	ALIGN A(i,j) WITH T(i,i+j)	$\texttt{SKEW}^{(2)}\begin{pmatrix}1 & 0\\ 1 & 1\end{pmatrix}$
	ALIGN A(I) WITH T(M * I)	$\texttt{SKEW}^{(d)}(M)$
	CSHIFT(A,dim=2,shift=1:n)	$\texttt{CSKEW}^{(2)}\begin{pmatrix}1 & 0\\ 1 & -1\end{pmatrix}$
ALIGN3	RESHAPE(A,s1,s2,[,o1][,o2])	RESHAPE(D_1, o_1, D_2, o_2), where D_i is the index domain representing s_i
	SPREAD(A,dim=k,ncopies=n)	$\texttt{REPLICATE}^{(\texttt{rank}(T))}(V, \texttt{Interval}(1, n))$, where $V(k) = 1$
	ALIGN A(I) WITH T(I,*)	$\texttt{REPLICATE}^{(2)}\left(\begin{pmatrix}0\\ 1\end{pmatrix}, \texttt{Domain}(T(1, :))\right)$
	ALIGN A(I) WITH T(I,1)	$\texttt{EMBED}^{(2,1)}\left(\begin{pmatrix}1\\ 0\end{pmatrix}, \texttt{Domain}(T)\right)$
DIST	DISTRIBUTE A BLOCK [(b)] onto P	BLOCK(b), where $b = n/p$
	DISTRIBUTE A CYCLIC [(b)] onto P	CYCLIC(b, p)
	*	SEQ
MULTID	multi-dimensional shift, reflection, etc., and distribution of multi-dimensional arrays.	Product of 1D Operators (ALIGN1, DIST)

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. D_1 and D_2 are the index domains representations for shape specifications s_1 and s_2, and o_1 and o_2 are permutation matrices representing the storage ordering in reshaping.

Class	Operator	Domain	Codomain	Definition
ALIGN1	EOSHIFT(c)	D	$D + c$	$(i) \longmapsto (i + c)$
	CSHIFT(c)	D	D	$(i) \longmapsto \texttt{lb}(D) + (i - \texttt{lb}(D) + c) \bmod \lvert D\rvert$
	REFLECT	D	D	$(i) \longmapsto \texttt{lb}(D) + \texttt{ub}(D) - i)$
	STRIDE(a, c)	D	$aD + c$	$(i) \longmapsto (a * i + c)$
ALIGN2	$\texttt{TRANS}^{(n)}(M)$	D	$M(D)$	$(i_1, \ldots, i_n) \longmapsto M(i_1, \ldots, i_n)$
	$\texttt{SKEW}^{(n)}(M)$	D	$M(D)$	$(i_1, \ldots, i_n) \longmapsto M(i_1, \ldots, i_n)$
	$\texttt{CSKEW}^{(n)}(M)$	D	$\texttt{Mod}(M(D), \lvert D\rvert)$	$(i_1, \ldots, i_n) \longmapsto ((M_1 \cdot I) \bmod m_1, \ldots, (M_1 \cdot I) \bmod m_n)$, where m_k are the sizes of D_k
ALIGN3	$\texttt{REPLICATE}^{(n)}(V, D_k)$	$D_1 \times \cdots \times D_{k-1} \times D_{k+1} \cdots \times D_n$	$D_1 \times \cdots \times D_{k-1} \times D_k \times D_{k+1} \cdots \times D_n$	$(i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_n) \longmapsto (i_1, \ldots, i_{k-1}, \texttt{lb}(D_k) : \texttt{ub}(D_k), i_{k+1}, \ldots, i_n)$, where $V(k) \neq 0$
	$\texttt{EMBED}^{(m,n)}(M, E)$	D	E	$(i_1, \ldots, i_n) \longmapsto M(i_1, \ldots, i_n)$
DIST	BLOCK(b)	D	$P \times L$	$(i) \longmapsto (i \operatorname{div} b, i \bmod b)$
	CYCLIC(b, p)	D	$P \times L$	$(i) \longmapsto ((i \operatorname{div} b) \bmod p, (i \operatorname{div} (p * b)) * b + i \bmod b)$
	SEQ	D	$[0] \times D$	$(i) \longmapsto (0, i)$

$\texttt{lb}(D)$ and $\texttt{ub}(D)$ denote the lower and upper bound of interval domain D, $aD + c$ denotes an interval domain of range $[(a * \texttt{lb}(D) + c) \ldots (a * \texttt{ub}(D) + c)]$, $|D|$ 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 $\texttt{Mod}(M(D), |D|)$ 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

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

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

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

which, when given index domain D, maps D to

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

according to the coefficient matrix $\binom{1}{0}$. The

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

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,

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

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

```
CSHIFT(CSHIFT(A,dim=1,shift=-c1),
       dim=2,shift=-c2)
```

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

[†]If template T is partitioned at the second dimension, the effect of the directive `ALIGN A(i) WITH T(i,*)` is to duplicate the same data of A onto all processors.

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 $f : D_1 \to E_1$ and $g : D_2 \to E_2$ is defined as

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

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

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

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 α denote the alignment operator which aligns array D to template E, β denote the partition operator which partitions template E into a pair of logical processors L and local index domain M, and γ 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 (α, β, and γ). 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.

$$\begin{tikzcd}
D \arrow[d, "\alpha"', "\text{alignment}"] \arrow[rrd, "\text{layout } g"] & & \\
E \arrow[r, "\beta\ \text{distribution}"'] & L \times M \arrow[r, "\gamma\ \text{physical mapping}"'] & P \times M
\end{tikzcd}$$

(a) data layout

$$\begin{tikzcd}
D_1 \arrow[r, shift left, "g_1"] \arrow[d, "\alpha\ \text{array reference}"'] & P \times M_1 \arrow[l, shift left, "g_1^{-1}"] \arrow[d, "\theta\ \text{data motion}"] \\
D_2 \arrow[r, "g_2"'] & P \times M_2
\end{tikzcd}$$

(b) intra-procedural data movement

$$\begin{tikzcd}
& P_1 \times M_1 \arrow[dl, shift left, "g_1^{-1}"] \arrow[dd, "\theta\ \text{layout conversion}"] & B_1 \\
D\ \text{array} \arrow[ur, "g_1"] \arrow[dr, "g_2"'] & & \\
& P_2 \times M_2 & B_2
\end{tikzcd}$$

(c) layout conversion

FIGURE 6 Commuting diagrams for data layout (a), intra-procedural data movement (b), and inter-procedural layout conversion (c).

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 α. Let the data layouts of D_1 and D_2 be g_1 and g_2, respectively. The data movement induced by the reference α can be formalized by the communication expression $\theta = g_2 \circ \alpha \circ g_1^{-1}$, 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 $\theta = g_2 \circ g_1^{-1}$, as shown in Figure 6c.

Most of the HPF alignment operators are *reshape morphisms* [13], which essentially are bijective functions defined over index domains. For an operator $f : D \to E$ which is injective but not bijective, a reshape morphism f': $D \to \texttt{image}(D, f)$ can be derived from f where $\texttt{image}(D, f)$ is the image of D under f which is a subset of E (i.e., $f' : D \to \texttt{image}(D, f) \subset E$). Since f is bijective, let f^{-1} denote the inverse of f. Let g be the composition of n bijective functions $f_i, i = 1, n$ (defined as $g = f_n \circ \cdots \circ f_1$). The inverse of g can be denoted by g^{-1}, and

$$g^{-1} = (f_n \circ \cdots \circ f_1)^{-1} = f_1^{-1} \circ \cdots \circ f_n^{-1}.$$

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

$$\texttt{SPREAD}^{(2)}\left(\begin{pmatrix} 0 \\ 1 \end{pmatrix}, D_2\right)_{D_1 \to D_1 \times D_2}$$

maps $(i) \in D_1$ to

$$\big((i, j), j = \texttt{lb}(D_2), \texttt{ub}(D_2)\big) \in D_1 \times D_2.$$

We abuse the notation

$$\texttt{SPREAD}^{(2)}\left(\begin{pmatrix} 0 \\ 1 \end{pmatrix}, D_2\right)^{-1}_{D_1 \to D_1 \times D_2}$$

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) is in C iff there exists a b such that (a, b) is in A and (b, c) is 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, is 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 $g_{D \to E}$) because now the operators are bound to the index domain of array A. The actual argument A is aligned with the template T1, which is partitioned into columns of blocks (denoted by

$$\texttt{SEQ}_{D_1+1 \to [0] \times (D_1+1)} \times \texttt{BLOCK}(v)_{D_2+2 \to P \times V_1}\ ),$$

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

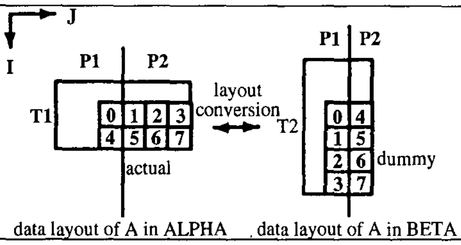
$$\big(\texttt{EOSHIFT}(1)_{D_1 \to D_1+1} \times \texttt{EOSHIFT}(2)_{D_2 \to D_2+2}\big)\ )$$

(refer to Figure 6a). The dummy argument B is aligned with template T2 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

$$\big(\texttt{EOSHIFT}(2)_{D_2 \to D_2+2} \times \texttt{EOSHIFT}(1)_{D_1 \to D_1+1}\big)$$
$$\circ\, \texttt{TRANS}^{(2)}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{D_1 \times D_2 \to D_2 \times D_1}\quad)$$

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.

```
ALPHA
real A(2,4)
template T1(3,6)
distribute T1(*,block)
align A(i,j) with T1(i+1,j+2)
call BETA(A)
```



```
BETA(B)
real B(2,4)
template T2(6,3)
distribute T2(*,block)
align B(i,j) with T2(j+2,i+1)
```

Index domains:

$$D_1 = \texttt{Interval}(1,2), \qquad D_2 = \texttt{Interval}(1,4).$$

Actual's layout:

$$g_1 = \left(\mathrm{SEQ}_{D_1+1\to[0]\times(D_1+1)} \times \mathrm{BLOCK}(b)_{D_2+2\to P\times V_1}\right) \circ \left(\mathrm{EOSHIFT}(1)_{D_1\to D_1+1} \times \mathrm{EOSHIFT}(2)_{D_2\to D_2+2}\right).$$

Dummy's layout:

$$\begin{aligned} g_2 = {} & \left(\mathrm{SEQ}_{D_2+2\to[0]\times(D_2+2)} \times \mathrm{BLOCK}(b)_{D_1+1\to P\times V_2}\right) \\ & \circ \left(\mathrm{EOSHIFT}(2)_{D_2\to D_2+2} \times \mathrm{EOSHIFT}(1)_{D_1\to D_1+1}\right) \circ \mathrm{TRANS}^{(2)} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{D_1\times D_2\to D_2\times D_1}. \end{aligned}$$

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

$$\begin{aligned} \theta &= g_2 \circ g_1^{-1} \\ &= \left(\mathrm{SEQ}_{D_2+2\to[0]\times(D_2+2)} \times \mathrm{BLOCK}(b)_{D_1+1\to P\times V_2}\right) \\ &\quad \circ \left(\mathrm{EOSHIFT}(2)_{D_2\to D_2+2} \times \mathrm{EOSHIFT}(1)_{D_1\to D_1+1}\right) \circ \mathrm{TRANS}^{(2)} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{D_1\times D_2\to D_2\times D_1} \\ &\quad \circ \left(\mathrm{EOSHIFT}(1)^{-1}_{D_1\to D_1+1} \times \mathrm{EOSHIFT}(2)^{-1}_{D_2\to D_2+2}\right) \circ \left(\mathrm{SEQ}^{-1}_{D_1+1\to[0]\times(D_1+1)} \times \mathrm{BLOCK}(b)^{-1}_{D_2+2\to P\times V_1}\right). \end{aligned}$$

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.

Recal 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 containts 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 θ in Figure 7. By exchanging the TRANS operator with one of the two-dimensional EOSHIFT operators (using one of *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 show below:

$$\begin{aligned}
\theta = {} & \left(\mathtt{SEQ}_{D_2+2\to[0]\times(D_2+2)} \times \mathtt{BLOCK}(b)_{D_1+1\to P\times V_2}\right) \\
& \circ \left(\mathtt{EOSHIFT}(2)_{D_2\to D_2+2} \times \mathtt{EOSHIFT}(1)_{D_1\to D_1+1}\right) \\
& \circ \mathtt{TRANS}^{(2)} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{D_1\times D_2\to D_2\times D_1} \\
& \circ \left(\mathtt{EOSHIFT}(1)^{-1}_{D_1\to D_1+1} \times \mathtt{EOSHIFT}(2)^{-1}_{D_2\to D_2+2}\right) \\
& \circ \left(\mathtt{SEQ}^{-1}_{D_1+1\to[0]\times(D_1+1)} \times \mathtt{BLOCK}(b)^{-1}_{D_2+2\to P\times V_1}\right) \\
= {} & \left(\mathtt{SEQ}_{D_2+2\to[0]\times(D_2+2)} \times \mathtt{BLOCK}(b)_{D_1+1\to P\times V_2}\right) \\
& \quad \circ \mathtt{TRANS}^{(2)} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{(D_1+1)\times(D_2+2)\to(D_2+2)\times(D_1+1)} \\
& \quad \circ \left(\mathtt{SEQ}^{-1}_{D_1+1\to[0]\times(D_1+1)} \times \mathtt{BLOCK}(b)^{-1}_{D_2+2\to P\times V_1}\right).
\end{aligned}$$

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 θ 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 be 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 idioms*. 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 (*, BLOCK) and row partition (BLOCK, *). These idioms 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 $\times^d(\gamma_{i1} \circ \beta_{i1} \circ \beta_{i2}^{-1} \circ \gamma_{i2}^{-1})$ in the second row of the table indicates change of data partition. For instance, the idiom $\mathtt{CYCLIC}(1, p) \circ \mathtt{BLOCK}(b)^{-1}$ indicates conversion from BLOCK partition to CYCLIC partition. The idiom

$$\left(\mathtt{SEQ} \times \mathtt{BLOCK}(b_1)\right) \circ \left(\mathtt{BLOCK}(b_2) \times \mathtt{SEQ}\right)^{-1}$$

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

Idioms	Data Movement
$\gamma_1 \circ \gamma_2^{-1}$	change physical mapping
$\times^d\left(\gamma_{i1} \circ \beta_{i1} \circ \beta_{i2}^{-1} \circ \gamma_{i2}^{-1}\right)$	change partition
$\gamma_1 \circ \beta_1 \circ \mathtt{EOSHIFT}(c) \circ \beta_2^{-1} \circ \gamma_2^{-1}$	end-of-shift
$\gamma_1 \circ \beta_1 \circ \mathtt{CSHIFT}(c) \circ \beta_2^{-1} \circ \gamma_2^{-1}$	cyclic shift
$\gamma_1 \circ \beta_1 \circ \mathtt{REFLECT} \circ \beta_2^{-1} \circ \gamma_2^{-1}$	reversal permutation
$\times^d\left(\gamma_{i1} \circ \beta_{i1} \circ \mathtt{EOSHIFT}(c_i) \circ \beta_{i2}^{-1} \circ \gamma_{i2}^{-1}\right)$	multi-dimensional end-of-shift
$\times^d\left(\gamma_{i1} \circ \beta_{i1} \circ \mathtt{CSHIFT}(c_i) \circ \beta_{i2}^{-1} \circ \gamma_{i2}^{-1}\right)$	multi-dimensional cyclic shift
$\times^d\left(\gamma_{i1} \circ \beta_{i1} \circ \mathtt{REFLECT} \circ \beta_{i2}^{-1} \circ \gamma_{i2}^{-1}\right)$	multi-dimensional reflection
$\times^d(\gamma_{i1} \circ \beta_{i1}) \circ \mathtt{TRANS}^{(d)}(M) \circ \times^d\left(\beta_{i2}^{-1} \circ \gamma_{i2}^{-1}\right)$	matrix transpose
$\times^d(\gamma_{i1} \circ \beta_{i1}) \circ \mathtt{SKEW}^{(d)}(M) \circ \times^d\left(\beta_{i2}^{-1} \circ \gamma_{i2}^{-1}\right)$	skewing
$\times^d(\gamma_{i1} \circ \beta_{i1}) \circ \mathtt{CSKEW}^{(d)}(M) \circ \times^d\left(\beta_{i2}^{-1} \circ \gamma_{i2}^{-1}\right)$	cyclic skewing
$\times^d(\gamma_i \circ \beta_i) \circ \mathtt{REPLICATE}^{(d)}(V, D) \circ \times^m\left(\beta_j^{-1} \circ \gamma_j^{-1}\right)$	replication
$\gamma_1 \circ \beta_1 \circ \mathtt{RESHAPE}^{(d,1)}(D, M, \mathtt{interval}(1, n), I) \circ \times^d\left(\beta_{i2}^{-1} \circ \gamma_{i2}^{-1}\right)$	axis combining
$\times^d(\gamma_{i1} \circ \beta_{i1}) \circ \mathtt{RESHAPE}^{(1,d)}(\mathtt{interval}(1, n), I, D, M) \circ \beta_2^{-1} \circ \gamma_2^{-1}$	axis spliting

where α denotes alignment operators or array references, β denotes distribution operators, γ denotes physical mapping operators, and $\times^d(a_i \circ b_i)$ denotes $(a_1 \circ b_1) \times \cdots \times (a_d \circ b_d)$.

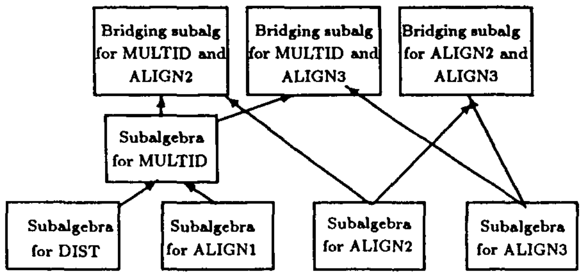

Bridging subalg for MULTID and ALIGN2
Bridging subalg for MULTID and ALIGN3
Bridging subalg for ALIGN2 and ALIGN3
Subalgebra for MULTID
Subalgebra for DIST
Subalgebra for ALIGN1
Subalgebra for ALIGN2
Subalgebra for ALIGN3

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, Ω, R), where S is the sorts, Ω 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 Dom. The operators include the four ALIGN1 operators and one composition operation ($\circ$). 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 convience, we use the notation $a * D + c$ for an index domain with lower bound $a * \mathtt{lb}(D) + c$ and upper bound $a * \mathtt{ub}(D) + c$. Rule 1 computers 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.

$$A_1 = \left(\{N, \mathrm{Dom}\}, \Omega_1, R_1\right)$$

Rule 1 Inverse of ALIGN1 Operators

(1) $\texttt{EOSHIFT}(c)^{-1}_{D\to D+c} = \texttt{EOSHIFT}(-c)_{D+c\to D}$

(2) $\texttt{CSHIFT}(c)^{-1}_{D\to D} = \texttt{CSHIFT}(-c)_{D\to D}$

(3) $\texttt{REFLECT}^{-1}_{D\to D} = \texttt{REFLECT}_{D\to D}$

(4) $\texttt{STRIDE}(a,c)^{-1}_{D\to a*D+c} = \texttt{STRIDE}\left(\frac{1}{a}, -\left\lfloor\frac{c}{a}\right\rfloor\right)_{a*D+c\to D}$

Rule 2 Reduction of Adjacent ALIGN1 Operators

(1) $\texttt{EOSHIFT}(c1)_{D+c2\to D+c2+c1} \circ \texttt{EOSHIFT}(c2)_{D\to D+c2} = \texttt{EOSHIFT}(c1+c2)_{D\to D+c2+c1}$

(2) $\texttt{CSHIFT}(c1)_{D\to D} \circ \texttt{CSHIFT}(c2)_{D\to D} = \texttt{CSHIFT}(c1+c2)_{D\to D}$

(3) $\texttt{REFLECT}_{D\to D} \circ \texttt{REFLECT}_{D\to D} = \texttt{id}_D$

(4) $\texttt{STRIDE}(a1,c1)_{a2*D+c2\to a1*a2*D+a1*c2+c1} \circ \texttt{STRIDE}(a2,c2)_{D\to a2*D+c2}$
$= \texttt{STRIDE}(a1*a2, a1*c2+c1)_{D\to a1*a2*D+a1*c2+c1}$

(5) $\texttt{STRIDE}(a,b)_{D+c\to a*D+a*c+b} \circ \texttt{EOSHIFT}(c)_{D\to D+c} = \texttt{STRIDE}(a, a*c+b)_{D\to a*D+a*c+b}$

(6) $\texttt{EOSHIFT}(c)_{a*D+b\to a*D+b+c} \circ \texttt{STRIDE}(a,b)_{D\to a*D+b} = \texttt{STRIDE}(a, b+c)_{D\to a*D+b+c}$

Rule 3 Exchange of ALIGN1 Operators

(1) $\texttt{CSHIFT}(c2)_{D+c1\to D+c1} \circ \texttt{EOSHIFT}(c1)_{D\to D+c1} = \texttt{EOSHIFT}(c1)_{D\to D+c1} \circ \texttt{CSHIFT}(c2)_{D\to D}$

(2) $\texttt{CSHIFT}(c)_{D\to D} \circ \texttt{REFLECT}_{D\to D} = \texttt{REFLECT}_{D\to D} \circ \texttt{CSHIFT}(c)_{D\to D}$

(3) $\texttt{EOSHIFT}(c)_{D\to D+c} \circ \texttt{REFLECT}_{D\to D} = \texttt{REFLECT}_{D+c\to D+c} \circ \texttt{EOSHIFT}(c)_{D\to D+c}$

(4) $\texttt{STRIDE}(a,c)_{D\to a*D+c} \circ \texttt{REFLECT}_{D\to D} = \texttt{REFLECT}_{a*D+c\to a*D+c} \circ \texttt{STRIDE}(a,c)_{D\to a*D+c}$

(5) $\texttt{STRIDE}(a,c2)_{D\to a*D+c2} \circ \texttt{EOSHIFT}(c1)_{D\to D} = \texttt{CSHIFT}(a,c1)_{a*D+c2\to a*D+c2} \circ \texttt{STRIDE}(a,c2)_{D\to a*D+c2}$

$$\begin{aligned}
\Omega_1 = \big\{&\texttt{EOSHIFT}(c\in N)(D\in \text{Dom}): D\to D+c,\\
&\texttt{CSHIFT}(c\in N)(D\in \text{Dom}): D\to D,\\
&\texttt{REFLECT}(D\in \text{Dom}): D\to D,\\
&\texttt{STRIDE}(a\in N, c\in N)(D\in \text{Dom}):\\
&\quad D\to a*D+c,\\
&\circ: (D_1\to D_2)\to(D_2\to D_3)\to(D_1\to D_3),\\
&\text{where } D_1, D_2, D_3\in \text{Dom}\big\}\\
R_1 = \{&\text{Rule 1, Rule 2, Rule 3}\}
\end{aligned}$$

Example

$$\begin{aligned}
&\texttt{CSHIFT}(2)_{D+1\to D+1} \circ \texttt{EOSHIFT}(1)_{D\to D+1}\\
&\circ\, \texttt{CSHIFT}(-2)_{D\to D}\\
=\ &\text{By Rule 3(1)}\\
&\texttt{EOSHIFT}(1)_{D\to D+1} \circ \texttt{CSHIFT}(2)_{D\to D}\\
&\circ\, \texttt{CSHIFT}(-2)_{D\to D}\\
=\ &\text{By Rule 2(2)}\\
&\texttt{EOSHIFT}(1)_{D\to D+1}
\end{aligned}$$

4.2 Subalgebra for ALIGN2

The `sorts` of subalgebra for ALIGN2 contains the integer set N, the set of $d\times d$ integer matrices $\texttt{Mat}^d$ and the set of interval domains Dom. Let 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 ($D_1\to D_2$, $D_2\to D_3$) 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.

$$\begin{aligned}
A_2 &= \left(\{N, \texttt{Mat}^d, \text{Dom}\}, \Omega_2, R_2\right)\\
R_2 &= \{\text{Rule 4, Rule 5, Rule 6, Rule 7}\}
\end{aligned}$$

Example

$$\texttt{CSKEW}^{(2)}\begin{pmatrix}1&0\\1&1\end{pmatrix}_{D\to D} \circ \texttt{CSKEW}^{(2)}\begin{pmatrix}1&1\\0&1\end{pmatrix}_{D\to D}$$

Rule 4 Inverse of ALIGN2 Operators

(1) $\texttt{TRANS}^{(d)}(M)^{-1}_{D\to E} = \texttt{TRANS}^{(d)}\left(M^{-1}\right)_{E\to D}$

(2) $\texttt{SKEW}^{(d)}(M)^{-1}_{D\to E} = \texttt{SKEW}^{(d)}\left(M^{-1}\right)_{E\to D}$

(3) $\texttt{CSKEW}^{(d)}(M)^{-1}_{D\to E} = \texttt{CSKEW}^{(d)}\left(M^{-1}\right)_{E\to D}$

Rule 5 Reduction of ALIGN2 Operators

(1) $\texttt{TRANS}^{(d)}(M2)_{D_2\to D_3} \circ \texttt{TRANS}^{(d)}(M1)_{D_1\to D_2} = \texttt{TRANS}^{(d)}(M2 * M1)_{D_1\to D_3}$

(2) $\texttt{SKEW}^{(d)}(M2)_{D_2\to D_3} \circ \texttt{SKEW}^{(d)}(M1)_{D_1\to D_2} = \texttt{SKEW}^{(d)}(M2 * M1)_{D_1\to D_3}$

(3) $\texttt{CSKEW}^{(d)}(M2)_{D\to D} \circ \texttt{CSKEW}^{(d)}(M1)_{D\to D} = \texttt{CSKEW}^{(d)}(M2 * M1)_{D\to D}$

Rule 6 Left-Trans Reduction

(1) $\texttt{TRANS}^{(d)}(M2)_{D_2\to D_3} \circ \texttt{SKEW}^{(d)}(M1)_{D_1\to D_2} = \texttt{SKEW}^{(d)}(M2 * M1)_{D_1\to D_3}$

(2) $\texttt{TRANS}^{(d)}(M2)_{D_1\to D_2} \circ \texttt{CSKEW}^{(d)}(M1)_{D_1\to D_1} = \texttt{CSKEW}^{(d)}(M2 * M1)_{D_1\to D_2}$

Rule 7 Right-Trans Reduction

(1) $\texttt{SKEW}^{(d)}(M2)_{D_2\to D_3} \circ \texttt{TRANS}^{(d)}(M1)_{D_1\to D_2} = \texttt{SKEW}^{(d)}(M2 * M1)_{D_1\to D_3}$

(2) $\texttt{CSKEW}^{(d)}(M2)_{D_2\to D_2} \circ \texttt{TRANS}^{(d)}(M1)_{D_1\to D_2} = \texttt{CSKEW}^{(d)}(M2 * M1)_{D_1\to D_2}$

$$\begin{aligned}\Omega_2 = \{&\texttt{TRANS}(d \in N)(M \in \texttt{Mat}^d)(D \in \texttt{Dom}^d) : D \to M(D),\\ &\texttt{SKEW}(d \in N)(M \in \texttt{Mat}^d)(D \in \texttt{Dom}^d) : D \to M(D),\\ &\texttt{CSKEW}(d \in N)(M \in \texttt{Mat}^d)(D \in \texttt{Dom}^d) : D \to D,\\ &\circ : (D_1 \to D_2) \to (D_2 \to D_3) \to (D_1 \to D_3),\\ &\text{where } D_1, D_2, D_3 \in \texttt{Dom}^d\}\end{aligned}$$

$$\begin{aligned}&= \text{By Rule 5(3)}\\ &\quad \texttt{CSKEW}^{(2)}\left(\begin{pmatrix}1&0\\1&1\end{pmatrix} * \begin{pmatrix}1&1\\0&1\end{pmatrix}\right)_{D\to D}\\ &= \texttt{CSKEW}^{(2)}\begin{pmatrix}2&1\\1&1\end{pmatrix}_{D\to D}\end{aligned}$$

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

4.3 Subalgebra for ALIGN3

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

$$\begin{aligned}&V(D_1 \times \cdots \times D_{k-1} \times D_{k+1} \times \cdots \times D_d, D_k)\\ &\quad = D_1 \times \cdots \times D_{k-1} \times D_k \times D_{k+1} \times \cdots \times D_d,\end{aligned}$$

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 α^{-1} to denote the inverse of an ALIGN3 operator. The inverse-

Rule 8 Inverse Cancellation of ALIGN3 Operators

For all $\alpha_{D \to E}$ in ALIGN3, $\quad \alpha_{D \to E} \circ \alpha^{-1}_{D \to E} = \mathtt{id}_E$.

Rule 9 Reduction of Adjacent ALIGN3 Operators

(1) $\mathtt{EMBED}^{(m,k)}(M2)_{D_2 \to D_3} \circ \mathtt{EMBED}^{(k,l)}(M1)_{D_1 \to D_2} = \mathtt{EMBED}^{(m,l)}(M2 * M1)_{D_1 \to D_3}$

(2) $\mathtt{REPLICATE}^{(k)}(V2, E_2)_{D_2 \to D_3} \circ \mathtt{REPLICATE}^{(k)}(V1, E_1)_{D_1 \to D_2} = \mathtt{REPLICATE}^{(k)}(V2 + V1, E_1 \times E_2)_{D_1 \to D_3}$

(3) $\mathtt{RESHAPE}^{(l,m)}(D_2, M3, D_3, M4)_{D_2 \to D_3} \circ \mathtt{RESHAPE}^{(k,l)}(D_1, M1, D_2, M2)_{D_1 \to D_2}$
$= \mathtt{RESHAPE}^{(k,m)}(D_1, M1, D_3, M4)_{D_1 \to D_3}, \quad$ if $M2 = M3$

$$\begin{aligned}
\Omega_3 = \big\{&\mathtt{EMBED}(m \in N)(n \in N)\big(m \in \mathtt{Mat}^{(m,n)}\big)(D \in \mathtt{Dom}^n) : D \to M(D),\\
&\mathtt{REPLICATE}(n \in N)\big(V \in \mathtt{Vec}^n\big)(E \in \mathtt{Dom})(D \in \mathtt{Dom}) : D \to V(D, E),\\
&\mathtt{RESHAPE}(m \in N)(n \in N)(D_1 \in \mathtt{Dom}^n)\big(M1 \in \mathtt{Mat}^{(m,m)}\big)(D_2 \in \mathtt{Dom}^n)\big(M2 \in \mathtt{Mat}^{(n,n)}\big) : D_1 \to D_2,\\
&\circ : (D_1 \to D_2) \to (D_2 \to D_3) \to (D_1 \to D_3)\big\},\\
\Omega_5 = \big\{&\times^d : (D_1 \to E_2) \to \cdots \to (D_d \to E_d) \to (D_1 \times \cdots \times D_d \to E_1 \times \cdots \times E_d),\\
&\text{where } D_1, \ldots, D_d \in \mathtt{Dom},\ E_1, \ldots, E_d \in \mathtt{Dom},\\
&\circ : (D_1 \to D_2) \to (D_2 \to D_3) \to (D_1 \to D_3),\\
&\text{where } D_1, D_2, D_3 \in \mathtt{Dom}^d\big\}
\end{aligned}$$

cancellation rule (Rule 8) cancels out an ALIGN3 operator with its inverse. Rule 9 reduces two adjacent ALIGN3 operators into one simply by standard matrix-vector algebra. The purpose of this rule is to simplify a chain of compositions of ALIGN3 operators, which may be a result of propagating inherited alignments across multiple levels of procedure calls.

$$\begin{aligned}
A_3 &= \big(\{N, \mathtt{Mat}^{(m,n)}, \mathtt{Vec}^d, \mathtt{Dom}\}, \Omega_3, R_3\big)\\
R_3 &= \{\text{Rule 8, Rule 9}\}
\end{aligned}$$

Example

$$\begin{aligned}
&\mathtt{REPLICATE}^{(3)}\left(\begin{pmatrix}0\\1\\0\end{pmatrix}, E_2\right)_{E_1 \to E_1 \times E_2}\\
&\circ\,\mathtt{REPLICATE}^{(3)}\left(\begin{pmatrix}0\\0\\1\end{pmatrix}, E_3\right)_{E_1 \times E_2 \to E_1 \times E_2 \times E_3}\\
= &\quad \text{By Rule 9(2)}\\
&\mathtt{REPLICATE}^{(3)}\left(\begin{pmatrix}0\\1\\1\end{pmatrix}, E_2 \times E_3\right)_{E_1 \to E_1 \times E_2 \times E_3}
\end{aligned}$$

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.

$$\begin{aligned}
A_5 &= \big(\{N, \mathtt{Dom}\}, \Omega_1 \cup \Omega_5, R_5\big)\\
R_5 &= \{\text{Rule 10}\}
\end{aligned}$$

Example

$$\begin{aligned}
&\big(\mathtt{EOSHIFT}(1) \times \mathtt{CSHIFT}(2)\big)\\
&\circ\big(\mathtt{EOSHIFT}(3) \times \mathtt{CSHIFT}(4)\big)\\
= &\quad \text{By Rule 10}\\
&\big(\mathtt{EOSHIFT}(1) \circ \mathtt{EOSHIFT}(3)\big)\\
&\times\big(\mathtt{CSHIFT}(2) \circ \mathtt{CSHIFT}(4)\big)\\
= &\quad \text{By Rules 2(1) and 2(2)}\\
&\mathtt{EOSHIFT}(4) \times \mathtt{CSHIFT}(6)
\end{aligned}$$

Rule 10 Product-Composition Exchange

For all $\alpha 1_i(c_i, d_i)_{E_i \to F_i}$ and $\alpha 2_i(a_i, b_i)_{D_j \to E_j}$ in ALIGN1,

$$\times^d \alpha 1_i(c_i, d_i)_{E_i \to F_i} \circ \times^d \alpha 2_i(a_i, b_i)_{D_i \to E_i} = \times^d\Big(\alpha 1_i(c_i, d_i)_{E_i \to F_i} \circ \alpha 2_i(a_i, b_i)_{D_i \to E_i}\Big).$$

Rule 11 Exchange of ALIGN2 and MULTID Operators

(1) Let $\kappa^{(n)}(V)_{D\to E} = \alpha_{V(1)}{}_{D_{V(1)}\to E_{V(1)}} \times \cdots \times \alpha_{V(n)}{}_{D_{V(n)}\to E_{V(n)}}$ where $\alpha_i,\ i = 1, n$ are ALIGN1 operators,

$$\mathtt{TRANS}^{(n)}(M)_{E\to M(E)} \circ \kappa^{(n)}(V)_{D\to E} = \kappa^{(n)}(M * V)_{M(D)-M(E)} \circ \mathtt{TRANS}^{(n)}(M)_{D\to M(D)}.$$

(2) Let $\mathtt{SH}^{(n)}(V)_{D_1\times\cdots\times D_n\to(D_1+c_1)\times\cdots\times(D_n+c_n)} = (\mathtt{EOSHIFT}(V(1)))_{D_1\to D_1+c_1} \times \cdots \times \mathtt{EOSHIFT}(V(n)))_{D_n\to D_n+c_n}),$

$$\begin{aligned}&\mathtt{SKEW}^{(n)}(M)_{(D_1+c_1)\times\cdots\times(D_n+c_n)\to E_1\times\cdots\times E_n} \circ \mathtt{SH}^{(n)}(V)_{(D_1\times\cdots\times D_n\to(D_1+c_1)\times\cdots\times(D_n+c_n))}\\ &\quad = \mathtt{SH}^{(n)}(M * V)_{M(D_1\times\cdots\times D_n)\to E_1\times\cdots\times E_n} \circ \mathtt{SKEW}^{(n)}(M)_{D_1\times\cdots\times D_n\to M(D_1\times\cdots\times D_n)}.\end{aligned}$$

(3) Let $\mathtt{REF}^{(n)}(V)_{D\to D} = (\mathtt{REFLECT}_{D_{V(1)}\to D_{V(1)}} \times \cdots \times \mathtt{REFLECT}_{D_{V(n)}\to D_{V(n)}}),$

$$\mathtt{SKEW}^{(n)}(M)_{D\to M(D)} \circ \mathtt{REF}^{(n)}(V)_{D\to D} = \mathtt{REF}^{(n)}(M * V)_{M(D)\to M(D)} \circ \mathtt{SKEW}^{(n)}(M)_{D\to M(D)}.$$

(4) Let $\mathtt{CSH}^{(n)}(V)_{D_1\times\cdots\times D_n\to D_1\times\cdots\times D_n} = (\mathtt{CSHIFT}(V(1)))_{D_1\to D_1} \times \cdots \times \mathtt{CSHIFT}(V(n)))_{D_n\to D_n})$
where $|D_1| = D_2| = \cdots = |D_n|$,

$$\begin{aligned}&\mathtt{CSKEW}^{(n)}(M)_{D_1\times\cdots\times D_n\to D_1\times\cdots\times D_n} \circ \mathtt{CSH}^{(n)}(V)_{D_1\times\cdots\times D_n\to D_1\times\cdots\times D_n}\\ &\quad = \mathtt{CSH}^{(n)}(M * V)_{D_1\times\cdots\times D_n\to D_1\times\cdots\times D_n} \circ \mathtt{CSKEW}^{(n)}(M)_{D_1\times\cdots\times D_n\to D_1\times\cdots\times D_n}.\end{aligned}$$

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.

$$\begin{aligned}A_6 &= \big(\{N, \mathtt{Mat}^d, \mathtt{Dom}\}, \Omega_2 \cup \Omega_5, R_2 \cup R_5, \cup R_6\big)\\ R_6 &= \{\text{Rule } 11\}\end{aligned}$$

Example

The expression

$$\begin{aligned}&\mathtt{TRANS}^{(2)}\left(\begin{pmatrix}0&1\\1&0\end{pmatrix}\right)_{D_2\times D_1\to D_1\times D_2}\\ &\circ \big(\mathtt{CSHIFT}(1) \times \mathtt{REFLECT}\big)_{D_2\times D_1\to D_2\times D_1}\\ &\circ \mathtt{TRANS}^{(2)}\left(\begin{pmatrix}0&1\\1&0\end{pmatrix}\right)_{D_1\times D_2\to D_2\times D_1}\end{aligned}$$

can be reduced to a MULTID operation

$$\big(\mathtt{REFLECT} \times \mathtt{CSHIFT}(1)\big)_{D_1\times D_2\to D_1\times D_2}$$

by first exchanging the position of the left-most TRANS operator with the MULTID operator using Rule 11(1), and then cancelling out the two TRANS operators by applying Rule 5(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.

$$\begin{aligned}A_7 &= \big(\{N, \mathtt{Mat}^d, \mathtt{Dom}\}, \Omega_3 \cup \Omega_5, R_3 \cup R_5, \cup R_7\big)\\ R_7 &= \{\text{Rule } 12\}\end{aligned}$$

Rule 12 Exchange of Adjacent ALIGN3 and MULTID Operators

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

(1) $\mathrm{EMBED}^{(m,n)}(M)_{D_2\to D_3}\circ\mathrm{SH}^{(m)}(V)_{D_1\to D_2} = \mathrm{SH}^{(n)}(M*V)_{M(D_1)\to D_3}\circ\mathrm{EMBED}^{(m,n)}(M)_{D_1\to M(D_1)}$

(2) $\mathrm{EMBED}^{(m,n)}(M)_{D_2\to D_3}\circ\mathrm{CSH}^{(m)}(V)_{D_1\to D_2} = \mathrm{CSH}^{(n)}(M*V)_{M(D_1)\to D_3}\circ\mathrm{EMBED}^{(m,n)}(M)_{D_1\to M(D_1)}$

(3) $\mathrm{REPLICATE}^{(m)}(V_2,E)_{D_2\to D_3}\circ\mathrm{SH}^{(n)}(V_1)_{D_1\to D_2}$
$= \mathrm{SH}^{(m)}(V_1\parallel 0^{m-n})_{V_2(D_1,E)\to D_3}\circ\mathrm{REPLICATE}^{(m)}(V_2,E)_{D_1\to V_2(D_1,E)}$

(4) $\mathrm{REPLICATE}^{(n)}(V_2,E)_{D_2\to D_3}\circ\mathrm{CSH}^{(m)}(V_1)_{D_1\to D_2}$
$= \mathrm{CSH}^{(m)}(V_1\parallel 0^{m-n})_{V_2(D_1,E)\to D_3}\circ\mathrm{REPLICATE}^{(m)}(V_2,E)_{D_1\to V_2(D_1,E)}$.

Rule 13 Reduction of ALIGN2 and ALIGN3 Operators

(1) $\mathrm{TRANS}^{(m)}(M_1)_{D_2\to D_3}\circ\mathrm{EMBED}^{(m,n)}(M_2)_{D_1\to D_2} = \mathrm{EMBED}^{(m,n)}(M_1*M_2)_{D_1\to D_3}$

(2) $\mathrm{SKEW}^{(m)}(M_1)_{D_2\to D_3}\circ\mathrm{EMBED}^{(m,n)}(M_2)_{D_1\to D_2} = \mathrm{EMBED}^{(m,n)}(M_1*M_2)_{D_1\to D_3}$

(3) $\mathrm{TRANS}^{(n)}(M_3)_{D_2\to D_3}\circ\mathrm{RESHAPE}^{(m,n)}(D_1,M_1,D_2,M_2)_{D_1\to D_2} = \mathrm{RESHAPE}^{(m,n)}(D_1,M_1,M_3(D_1),M_3*M_2)_{D_1\to D_3}$

(4) $\mathrm{RESHAPE}^{(m,n)}(D_2,M_1,D_3,M_2)_{D_2\to D_3}\circ\mathrm{TRANS}^{(m)}(M_3)_{D_1\to D_2} = \mathrm{RESHAPE}^{(m,n)}(M_2(D_2),M_3*M_1,D_3,M_2)_{D_1\to D_3}$

4.7 Bridging Subalgebra for ALIGN2 and ALIGN3

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

$$\begin{aligned} A_8 &= \left(\{N,\mathtt{Mat}^d,\mathtt{Dom}\},\Omega_2\cup\Omega_3,R_2\cup R_3,\cup R_8\right)\\ R_8 &= \{\text{Rule 13}\}\end{aligned}$$

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 θ can be reduced to a basic operator by the reduction rules. A basic operator is either an identify function or a single alignment operator.

For example, the composition

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

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

Definition. A *redcom* is a subexpression $\theta = g_1\circ[\cdots]g_2$ in a communication expression such that either θ 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

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

is a *redcom*, and the subexpression

$$\left(\texttt{CSHIFT}(c_1) \times \texttt{CSHIFT}(c_2)\right) \circ \texttt{TRANS}^{(2)}\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$$
$$\circ \left(\texttt{CSHIFT}(c_3) \times \texttt{CSHIFT}(c_4)\right) \circ \texttt{CSKEW}^{(2)}\begin{pmatrix}1 & 1\\ 0 & 1\end{pmatrix}$$

contains two *redcoms*: one is

$$\left(\texttt{CSHIFT}(c_1) \times \texttt{CSHIFT}(c_2)\right) \circ \texttt{TRANS}^{(2)}\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$$
$$\circ \left(\texttt{CSHIFT}(c_3) \times \texttt{CSHIFT}(c_4)\right)$$

and the other is

$$\texttt{TRANS}^{(2)}\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} \circ \left(\texttt{CSHIFT}(c_3) \times \texttt{CSHIFT}(c_4)\right)$$
$$\circ \texttt{CSKEW}^{(2)}\begin{pmatrix}1 & 1\\ 0 & 1\end{pmatrix}.$$

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

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

where α_i are alignment operators, β_i are distribution operators, and γ_i physical mapping operators (please refer to Figure 6). Following elaboration, we simplify a communication expression according to the order: alignment $\alpha_1 \circ \alpha \circ \alpha_2^{-1}$ 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:

$$(1)\ \left(\texttt{EOSHIFT}(1) \times \texttt{id}\right)_{D_1\times D_2 \to (D_1+1)\times D_2} \circ \texttt{SPREAD}\left(\begin{pmatrix}0\\1\end{pmatrix}, \texttt{ub}(D_2)\right)_{D_1 \to D_1\times D_2};$$

$$(2)\ \texttt{SPREAD}\left(\begin{pmatrix}0\\1\end{pmatrix}, \texttt{ub}(D_2)\right)_{D_1+1 \to (D_1+1)\times D_2} \circ \texttt{EOSHIFT}(1)_{D_1 \to D_1+1}.$$

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 ith 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 β_{1_i} and $\beta_{2_i}^{-1}$ 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 γ_{1_i}, and $\gamma_{2_i}^{-1}$ 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 tat 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.

FIGURE 9 Algebraic simplification procedure.

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 `sizeof`(D_1) · `sizeof`(D_2) elements in the `EOSHIFT` operation, while expression (2) only requires communicating `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 operaotors (e.g. spread).

Example

Figure 10 shows the transformation result for the inter-procedural 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 disprocedure ALPHA in Figure 7

```
source program                    transformedprogram
real A(2,4)                       real A'(2 : 3, 3 : 6) TEMP(3 : 6, 2 : 3)
template T1(3,6)
align A(i,j) with T1(i+1,j+2)
                                  call matrix-transpose-1d-partition(TEMP,A')
call BETA(A)                      call BETA(TEMP)
                                  call matrix-transpose-1d-partition(A',TEMP)
```

Simplification procedure

$$
\begin{aligned}
\theta = {} & (\mathtt{SEQ}_{D_2+2\to[0]\times(D_2+2)} \times \mathtt{BLOCK}(v)_{D_1+1\to P\times V_1}) \circ (\mathtt{EOSHIFT}(2)_{D_2\to D_2+2} \times \mathtt{EOSHIFT}(1)_{D_1\to D_1+1}) \\
& \circ \mathtt{TRANS}^{(2)} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{D_1\times D_2\to D_2\times D_1} \\
& \circ (\mathtt{EOSHIFT}(-1)_{D_1+1\to D_1} \times \mathtt{EOSHIFT}(-2)_{D_2+2\to D_2}) \\
& \circ \left(\mathtt{SEQ}^{-1}_{D_1+1\to[0]\times(D_1+1)} \times \mathtt{BLOCK}(v)^{-1}_{D_2+2\to P\times V_2}\right)
\end{aligned}
$$

= By Rule 11: exchange of MULTID and ALIGN2

$$
\begin{aligned}
& (\mathtt{SEQ}_{D_2+2\to[0]\times(D_2+2)} \times \mathtt{BLOCK}(v)_{D_1+1\to P\times V_1}) \circ \mathtt{TRANS}^{(2)} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{D_1+1\times D_2+2\to D_2+2\times D_1+1} \\
& \circ (\mathtt{EOSHIFT}(1)_{D_1\to D_1+1} \times \mathtt{EOSHIFT}(2)_{D_2\to D_2+2}) \circ (\mathtt{EOSHIFT}(-1)_{D_1+1\to D_1} \times \mathtt{EOSHIFT}(-2)_{D_2+2\to D_2}) \\
& \circ \left(\left(\mathtt{SEQ}^{-1}_{D_1+1\to[0]\times(D_1+1)} \times \mathtt{BLOCK}(v)^{-1}_{D_2+2\to P\times V_2}\right)\right.
\end{aligned}
$$

= By Rule 10: product composition exchange

$$
\begin{aligned}
& (\mathtt{SEQ}_{D_2+2\to[0]\times(D_2+2)} \times \mathtt{BLOCK}(v)_{D_1+1\to P\times V_1}) \circ \mathtt{TRANS}^{(2)} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{D_1+1\times D_2+2\to D_2+2\times D_1+1} \\
& \circ (\mathtt{EOSHIFT}(1)_{D_1\to D_1+1} \circ_1 \mathtt{EOSHIFT}(-1)_{D_1+1\to D_1}) \times (\mathtt{EOSHIFT}(2)_{D_2\to D_2+2} \circ_1 \mathtt{EOSHIFT}(-2)_{D_2+2\to D_2}) \\
& \circ \left(\mathtt{SEQ}^{-1}_{D_1+1\to[0]\times(D_1+1)} \times \mathtt{BLOCK}(v)^{-1}_{D_2+2\to P\times V_2}\right)
\end{aligned}
$$

= By Rule 2: reduction of ALIGN1 operators

$$
\begin{aligned}
& (\mathtt{SEQ}_{D_2+2\to[0]\times(D_2+2)} \times \mathtt{BLOCK}(v)_{D_1+1\to P\times V_1}) \circ \mathtt{TRANS}^{(2)} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{D_1+1\times D_2+2\to D_2+2\times D_1+1} \\
& \circ \left(\mathtt{SEQ}^{-1}_{D_1+1\to[0]\times(D_1+1)} \times \mathtt{BLOCK}(v)^{-1}_{D_2+2\to P\times V_2}\right)
\end{aligned}
$$

Match Idiom: matrix transposition

FIGURE 10 Algebraic simplification for inter-procedural layout conversion.

tribution) were used to demonstrate the impact of idiom matching for fast data layout conversion. The benchmark codes were listed in Appendix C.

We report our experimental results on the Connection Machines CM-5 located at AHPCRC of Minnesota University. The CM-5 has totally 896 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

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

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)

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

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)

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

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 xlHPF 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 [1] 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 transformative 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

[1] Reeuwijk et al., "An implementation framework for HDF distributed arrays on message-passing parallel computer systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 9, 1996.

[2] A. P. Amarasinghe and M. S. Lam, "Communication optimization and code generation for distributed memory machines," in *Proc. ACM SIGPLAN'93 Programming Language Design and Implementation*, Albuquergue, New Mexico, June 1993.

[3] J. M. Anderson and M. S. Lam, "Global optimization for parallelism and locality on scalable parallel machines," in *Proc. ACM SIGPLAN'93 Programming Language Design and Implementation*, Albuquergue, New Mexico, June 1993.

[4] S. H. Bokhari, "Complete Exchange on The iPSC-860," ICASE, NASA Langley Research Center, Tech. Rep., 1991.

[5] S. H. Bokhari, "Multiphase Complete Exchange on A Circuit Switched Hypercube," ICASE, NASA Langley Research Center, Tech. Rep., 1991.

[6] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M. Wu, "Compiling Fortran 90/HPF for distributed memory MIMD computers," *Journal of Parallel and Distributed Computing*, 1994.

[7] M. Bromley, S. Heller, T. McNerney, and G. L. Steele Jr., "Fortran and Ten Gigaflops: The Connection Machine Convolution Compiler," in *ACM SIGPLAN'91 Conf. Programming Language Design and Implementation*, June 1991, pp. 145–156.

[8] B. Chapman, H. Herbeck, and H. P. Zima, "Automatic Support for Data Distribution," in *Proc. 6th Distributed Memory Computing Conference*, April 1991.

[9] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng, "Generating local addresses and communication sets for data-parallel programs," in *Proc. Principles and Practice of Parallel Programming*, San Diego, CA, May 1993, pp. 149–158.

[10] S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng, "Optimal evaluation of array expressions on massively parallel machines," XEROX Palo Alto Research Center, Tech. Rep. CSL-92-11, December 1992.

[11] S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng, "Automatic array alignment in data-parallel programs," in *Proc. 20th Annual ACM Symp. Principles of Programming Languages*, 1993.

[12] S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng, "Optimal Evaluation of Array Expressions on Massively Parallel Machines," XEROX Corporation, Palo Alto Research Center, Tech. Rep., December 1992.

[13] M. Chen, Y. Choo, and J. Li, "Theory and pragmatics of generating efficient parallel code," in *Parallel Functional Languages and Compilers*, chapter 7. ACM Press and Addison-Wesley, 1991.

[14] *The Connection Machine CM-5 Technical Summary*, Thinking Machines Corporation, Tech. Rep., 1991.

[15] *CM Fortran Reference Manual*. Cambridge, MA: Thinking Machines Corporation, July 1991.

[16] *DECmpp 12000 Sx – High Performance Fortran Reference Manual*. Maynard, MA: Digital Equipment Corporation, 1993.

[17] J. Gilbert and R. Schreiber, "Optimal Expression Evaluation for Data Parallel Architectures," *Journal of Parallel and Distributed Computing*, vol. 13, no. 1, September 1991.

[18] C. Gong, R. Gupta, and R. Melhem, "Compilation techniques for optimizing communication in distributed memory systems," in *Proc. Int. Conf. Parallel Processing*, St. Charles, IL, August 1993.

[19] M. Gupta, E. Schonberg, and H. Srinivasan, "An unified data-flow framework for optimizing communication," in *Proc. 7th Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[20] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiling Fortran D for MIMD distributed-memory machines," *Comm. ACM*, vol. 35, no. 8, pp. 66–80, August 1992.

[21] C. T. Ho, *Optimal Communication Primitives and Graph Embeddings on Hypercubes.* PhD thesis, Department of Computer Science, Yale University, 1990.

[22] C. T. Ho and S. L. Johnsson, "Optimal Algorithms for Stable Dimension Permutations on Boolean Cubes," in *The Third Conference on Hypercube Concurrent Computers and Applications*, ACM, 1988, pp. 725–736.

[23] C. T. Ho and S. L. Johnsson, "The Complexity of Reshaping Arrays on Boolean Cubes," in *Proc. 5th Distributed Computing Conference*, 1990.

[24] *High Performance Fortran Language Specification*, Rice University, Houston Texas, Tech. Rep., May 1993.

[25] G. H. Hwang, J. K. Lee, and D. C. Ju, "An array synthesis scheme to optimize Fortran 90 programs," in *Proc. Principles and Practice of Parallel Programming*, April 1995.

[26] S. L. Johnsson and C. T. Ho, "Matrix Transposition on Boolean n-cube Configured Ensemble Architectures," *SIAM J. Matrix Anal. Appl.*, vol. 9, no. 3, pp. 419–454, July 1988.

[27] D. C. Ju, C. L. Wu, and P. Carin, "The synthesis of array functions and its use in parallel computation," in *Proc. Int. Conf. Parallel Processing*, 1992.

[28] S. D. Kaushik, C. H. Huang, and P. Saydayappan, "Compiling array statements for efficient execution on distributed memory machines: two-level mappings," in *Proc. 8th Workshop on Languages and Compilers for Parallel Computing*, August 1995.

[29] K. Kennedy, N. Nedeljkovic, and A. Sethi, "A linear-time algorithm for computing the memory access sequence in data-parallel programs," in *Proc. 5th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[30] J. Li, *Compiling Crystal for Distributed Memory Machines.* PhD thesis, Dept. of Computer Science, Yale University, 1991.

[31] J. Li and M. Chen, "Generating Explicit Communication from Shared-Memory Program References," in *Supercomputing*, pp. 865–876, 1990.

[32] J. Li and M. Chen, *Proc. the Workshop on Programming Languages and Compilers for Parallel Computing*, chapter "Automating the Coordination of Interprocessor Communication." MIT Press, 1990.

[33] J. Li and M. Chen, "The data alignment phase in compiling programs for distributed-memory machines," *Journal of Parallel and Distributed Computing*, 1991.

[34] D. Loveman, "High performance Fortran: Proposal," in *High Performance FORTRAN Forum*, Houston, Texas, January 1992.

[35] J. Merlin, "Techniques for the automatic parallelisation of distributed Fortran 90," Department of Electronics and Computer Science, University of Southampton, Tech. Rep. SNARC 92-02, November 1991.

[36] R. Ponnusamy, R. Thakur, A. Choudhary, and G. Fox, "Scheduling Regular and Irregular Communication Patterns on the CM-5," in *Proc. Supercomputing'92*, 1992.

[37] R. Ponnusamy, A. Choudhary, and G. Fox, "Communication Overhead on CM5: An Experimental Performance Evaluation," in *Proc. Frontiers'92*, 1992.

[38] K. S. Gupta, S. D. Kaushik, C. H. Huang, and P. Sadayappan, "Compiling array statements for efficient execution on distributed memory machines: two-level mappings," *Journal of Parallel and Distributed Computing*, vol. 32, pp. 155–172, 1996.

[39] T. Schmiermund and S. R. Seidel, "A Communication Model for the Intel iPSC/2," Michigan Technological University, Tech. Rep. , 1990.

[40] J. M. Stichnoth, "Efficient compilation of array statements for private memory multicomputers," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-93-109, February 1993.

[41] J. M. Stichnoth, D. O'Hallaron, and T. R. Gross, "Generating communication for array statements: Design, implementation, and evaluation," *Journal of Parallel and Distributed Computing*, vol. 21, no. 1, pp. 150–158, April 1994.

[42] R. Thakur, A. Choudhary, and J. Ramanujam, "Efficient algorithms for array redistribution," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, no. 7, pp. 587–594, June 1996.

[43] A. Thirumalai and J. Ramanujam, "Fast address sequence generation for data-parallel programs using integer lattices," in *Proc. International Parallel Processing Symposium*, 1995.

[44] C. W. Tseng, *An Optimizing Fortran D Compilers for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, 1993.

[45] A. Venkatachar, J. Ramanujam, and A. Thirumalai, "Generalized overlap regions for communication optimization in data-parallel programs," in *Proc. 6th Int. Workshop on Languages and Compilers for Parallel Computing*, 1997.

[46] M. Wolf and M. Lam, "A data locality optimizing algorithm," in *Proc. ACM SIGPLAN'91 Conf. Programming Language Design and Implementation*, Toronto, Ontario, Canada, June 1991, pp. 30–44.

[47] J. A. Yang and Y. Choo, "Parallel-program transformation using a metalanguage," in *Proc. 3rd ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, Williamsburg, Virginia, April 1991, pp. 11-20.

[48] H. Zima and B. Chapman, "Compiling for distributed memory systems," in *Proc. IEEE Special Section on Languages and Compilers for Parallel Machines*, February 1993, pp. 264–287.

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 :

$D_1 = \texttt{Interval}(1,100), \qquad D_2 = \texttt{Interval}(1,200).$

Data movement for statement S1 :

RHS layout (array A) :

$$g_{rA} = \left(\text{EOSHIFT}(2)_{D_2\to D_2+2} \times \text{EOSHIFT}(1)_{D_1\to D_1+1}\right) \circ \text{TRANS}^{(2)}\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}_{D_1\times D_2\to D_2\times D_1};$$

Array reference :

$$\alpha_A = \left(\text{EOSHIFT}(1)_{D_2\to D_2+1} \times \text{EOSHIFT}(1)_{D_1\to D_1+1}\right) \circ \text{TRANS}^{(2)}\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}_{D_1\times D_2\to D_2\times D_1}.$$

LHS layout (array A) :

$$g_{lA} = \texttt{id}_{(D_2+1)\times(D_1+1)\to(D_2+1)\times(D_1+1)}.$$

Actual data movement :

$$\theta_A = g_{lA} \circ \alpha_A \circ g_{rA}^{-1} \qquad \text{(see Figure 6b)}$$

$$= \left(\text{EOSHIFT}(1)_{D_2\to D_2+1} \times \text{EOSHIFT}(1)_{D_1\to D_1+1}\right) \circ \text{TRANS}^{(2)}\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}_{D_1\times D_2\to D_2\times D_1}$$

$$\circ\, \text{TRANS}^{(2)}\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}^{-1}_{D_1\times D_2\to D_2\times D_1} \circ \left(\text{EOSHIFT}(2)^{-1}_{D_2\to D_2+1} \times \text{EOSHIFT}(1)^{-1}_{D_1\to D_1+1}\right).$$

Data movement for statement S2 :

RHS layout (array D) :

$$g_{rD} = \left(\texttt{id} \times \text{REFLECT}_{D_1\to D_1}\right) \circ \text{EMBED}^{(1,2)}\begin{pmatrix}0\\ 1\end{pmatrix}_{D_1\to([1]\times D_1)};$$

array reference :

$$\alpha_D = \text{CSHIFT}(1)_{D_1\to D_1};$$

LHS layout (array D) :

$$g_{lD} = \text{EMBED}^{(1,2)}\begin{pmatrix}0\\ 1\end{pmatrix}_{D_1\to([1]\times D_1)},$$

actual data movement :

$$\theta_D = g_{lD} \circ \alpha_D \circ g_{rD}^{-1} \qquad \text{(see Figure 6b)}$$

$$= \text{EMBED}^{(1,2)}\begin{pmatrix}0\\ 1\end{pmatrix}_{D_1\to([1]\times D_1)} \circ \text{CSHIFT}(1)_{D_1\to D_1} \circ \text{EMBED}^{(1,2)}\begin{pmatrix}0\\ 1\end{pmatrix}^{-1}_{D_1\to([1]\times D_1)} \circ \left(\texttt{id} \times \text{REFLECT}^{-1}_{D_1\to D}\right).$$

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

```
procedure ALPHA in Figure 11
   source program                                 transformed program
   real A(100,200), B(100,200), C(100), D(100)    real A'(3:202, 2:101), B'(100,200)
   template T(202,101)                            real C'(1,100), D'(1,100)
   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)
S1 B = EOSHIFT(EOSHIFT(TRANSPOSE(A),           S1 B' = EOSHIFT(A',dim=1,shift=1)
     dim=1,shift=-1),dim=2,shift=-1)
S2 C = CSHIFT(D,dim=1,shift=-1)                S2 C' = CSHIFT(REFLECT(D',dim=2)dim=2,shift=1)
```

Simplification procedure:

Data movement for statement S1 :

$$
\begin{aligned}
\theta_A &= (\mathtt{EOSHIFT}(1)_{D_2\to D_2+1} \times \mathtt{CSHIFT}(1)_{D_1\to D_1+1}) \circ \mathrm{TRANS}\begin{pmatrix}0&1\\1&0\end{pmatrix}_{D_1\times D_2\to D_2\times D_1} \circ \mathrm{TRANS}\begin{pmatrix}0&1\\1&0\end{pmatrix}_{D_2\times D_1\to D_1\times D_2}\\
&\quad \circ (\mathtt{EOSHIFT}(-2)_{D_2+2\to D_2} \times \mathtt{EOSHIFT}(-1)_{D_1+1\to D_1})\\
&= \text{By Rule 5: Reduction of ALIGN2 operators}\\
&\quad (\mathtt{EOSHIFT}(1)_{D_2\to D_2+1} \times \mathtt{EOSHIFT}(1)_{D_1\to D_1+1}) \circ (\mathtt{EOSHIFT}(-2)_{D_2+2\to D_2} \times \mathtt{EOSHIFT}(-1)_{D_1+1\to D_1})\\
&= \text{By Rule 10: Product-Composition Exchange}\\
&\quad (\mathtt{EOSHIFT}(1)_{D_2\to D_2+1} \circ \mathtt{EOSHIFT}(-2)_{D_2+2\to D_2}) \times (\mathtt{EOSHIFT}(1)_{D_1\to D_1+1} \circ \mathtt{EOSHIFT}(-1)_{D_1+1\to D_1})\\
&= \text{By Rule 2: Reduction of ALIGN1 operators}\\
&\quad \mathtt{EOSHIFT}(-1)_{D_2+2\to D_2+1} \times \mathtt{id}_{D_1+1\to D_1+1}
\end{aligned}
$$

Data movement for statement S2 :

$$
\begin{aligned}
\theta_D &= \mathrm{EMBED}^2\begin{pmatrix}0\\1\end{pmatrix}_{D_1\to([1]\times D_1)} \circ \mathtt{CSH}(1)_{D_1\to D_1} \circ \mathrm{EMBED}^2\begin{pmatrix}0\\1\end{pmatrix}^{-1}_{D_1\to([1]\times D_1)} \circ \left(\mathtt{id}\times\mathtt{REFLECT}^{-1}_{D_1\to D_1}\right)\\
&= \text{By Rule 12: Exchange of ALIGN1 and ALIGN3}\\
&\quad \mathtt{CSH}\begin{pmatrix}0\\1\end{pmatrix}_{[1]\times D_1\to[1]\times D_1} \circ \mathrm{EMBED}^2\begin{pmatrix}0\\1\end{pmatrix}_{D_1\to([1]\times D_1)} \circ \mathrm{EMBED}^2\begin{pmatrix}0\\1\end{pmatrix}^{-1}_{D_1\to([1]\times D_1)} \circ \left(\mathtt{id}\times\mathtt{REFLECT}^{-1}_{D_1\to D_1}\right)\\
&= \text{By Rule 8: Inverse cancellation of ALIGN3}\\
&\quad \mathtt{CSH}\begin{pmatrix}0\\1\end{pmatrix}_{[1]\times D_1\to[1]\times D_1} \circ (\mathtt{id}_{[1]\to[1]}\times\mathtt{REFLECT}_{D_1\to D_1})\\
&= \text{By definition of } \mathtt{CSH}\\
&\quad (\mathtt{id}_{[1]\to[1]}\times\mathtt{CSHIFT}_{D_1\to D_1}) \circ (\mathtt{id}_{[1]\to[1]}\times\mathtt{REFLECT}_{D_1\to D_1})\\
&= \text{By Rule 10: Product-composition exchange}\\
&\quad \mathtt{id}_{[1]\to[1]} \times (\mathtt{CSHIFT}_{D_1\to D_1} \circ \mathtt{REFLECT}_{D_1\to D_1}).
\end{aligned}
$$

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) $\mathtt{CSHIFT}(c_2)_{D+c_1\to D+c_1} \circ \mathtt{EOSHIFT}(c_1)_{D\to D+c_1} = \mathtt{EOSHIFT}(c_1)_{D\to D+c_1} \circ \mathtt{CSHIFT}(c_2)_{D\to D}$.

Proof.

$$\begin{aligned}
&\big(\mathtt{CSHIFT}(c_2)_{D+c_1\to D+c_1} \circ \mathtt{EOSHIFT}(c_1)_{D\to D+c_1}\big)(i \in D)\\
=&\quad \text{By definition of } \mathtt{EOSHIFT}\\
&\big(\mathtt{CSHIFT}(c_2)_{D+c_1\to D+c_1}\big)\big((i+c_1) \in D+c_1\big)\\
=&\quad \text{By definition of } \mathtt{CSHIFT}\\
&\mathtt{lb}(D+c_1) + \big(i + c_1 - \mathtt{lb}(D+c_1) + c_2\big) \bmod \big(\mathrm{ub}(D+c_1) - \mathtt{lb}(D+c_1)\big)\\
=&\ \mathtt{lb}(D) + c_1 + \big(i + c_1 - \mathtt{lb}(D) - c_1 + c_2\big) \bmod \big(\mathrm{ub}(D) - \mathtt{lb}(D)\big)\\
=&\ \mathtt{lb}(D) + \big(i - \mathtt{lb}(D) + c_2\big) \bmod |D| + c_1.\\
&\big(\mathtt{EOSHIFT}(c_1)_{D\to D+c_1} \circ \mathtt{CSHIFT}(c_2)_{D\to D}\big)(i \in D)\\
=&\quad \text{By definition of } \mathtt{CSHIFT}\\
&\mathtt{EOSHIFT}(c_1)_{D\to D+c_1}\big(\mathtt{lb}(D) + \big(i - \mathtt{lb}(D) + c_2\big) \bmod |D|\\
=&\ \mathtt{lb}(D) + \big(i - \mathtt{lb}(D) + c_2\big) \bmod |D| + c_1. \qquad \square
\end{aligned}$$

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

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

$$\begin{aligned}
&\big(\mathtt{CSKEW}^{(d)}(M_2)_{D\to D} \circ \mathtt{CKEW}^{(d)}(M_1)_{D\to D}\big)(I \in D)\\
=&\quad \text{By definition of } \mathtt{CSKEW}\\
&\big(\mathtt{CSKEW}^{(d)}(M2)_{D\to D}\big)\big(\mathtt{Mod}(M_1 * I, |D_1|)\big)\\
=&\quad \text{By definition of } \mathtt{CSKEW}\\
&\mathtt{Mod}\big(M_2 * \big(\mathtt{Mod}(M_1 * I, |D|)\big)\big)\\
=&\quad \text{By property of modulus}\\
&\mathtt{Mod}\big((M_2 * M_1) * I, |D|\big).\\
&\mathtt{CSKEW}^{(d)}(M_2 * M_1)_{D\to D}\big)(I \in D)\\
=&\quad \text{By definition of } \mathtt{CSKEW}\\
&\mathtt{Mod}\big((M_2 * M_1) * I, |D|\big). \qquad \square
\end{aligned}$$

Rule 9(2)

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

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

$$\begin{aligned}
&\big(\mathtt{REPLICATE}^{(n)}(V_2, E_2)_{D_2\to D_3}\\
&\circ \mathtt{REPLICATE}^{(n)}(V_1, E_1)_{D_1\to D_2}\big)\big((i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_{l-1}, i_{l+1}, \ldots, i_n) \in D_1\big)\\
=&\quad \text{By definition of } \mathtt{REPLICATE}\\
&\big(\mathtt{REPLICATE}^{(n)}(V_2, E_2)_{D_2\to D_3}\big)\big(i_1, \ldots, i_{k-1}, i_k, i_{k+1}, \ldots, i_{l-1}, i_{l+1}, \ldots, i_n\big)\\
=&\quad \text{By definition of } \mathtt{REPLICATE}\\
&\big(i_1, \ldots, i_{k-1}, i_k, i_{k+1}, \ldots, i_{l-1}, i_{l+1}, \ldots, i_n\big).\\
&\big(\mathtt{REPLICATE}^{(n)}(V_2 + V_1, E_1 \times E_2)_{D_1\to D_3}\big)\big((i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_{l-1}, i_{l+1}, \ldots, i_n) \in D_1\big)\\
=&\quad \text{for } V(k) \neq 0,\ V(l) \neq 0\\
&\big(\mathtt{REPLICATE}^{(n)}(V, E_1 \times E_2)_{D_1\to D_3}\big)\big((i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_{l-1}, i_{l+1}, \ldots, i_n) \in D_1\big)\\
=&\quad \text{By definition of } \mathtt{REPLICATE}\\
&(i_1, \ldots, i_{k-1}, i_k, i_{k+1}, \ldots, i_{l-1}, i_l, i_{l+1}, \ldots, i_n). \qquad \square
\end{aligned}$$

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

!hpf$ redistribute block :: x
     ...
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
```