Yale University
Department of Computer Science

Optimizing FORTRAN-90 Programs for Data Motion on Massively Parallel Systems

Marina C. Chen, Jan-Jan Wu

YALEU/DCS/TR-882
December 1991

Support for this work was provided by the Office of Naval Research under contract N00014-91-J-1559 and Defense Advanced Research Projects Agency monitored by Army Directorate of Contracting DABT 63-91-C-0031

Contents

Abstract

This paper describes a general compiler optimization technique that reduces communication overhead for FORTRAN-90 implementations on massively parallel machines. The main sources of communication, or data motion, for the parallel implementation of a FORTRAN-90 program are from array assignments (using the index triplet notation and vector indexing), array operators (e.g. `CSHIFT`, `TRANSPOSE`, etc.), and array parameter passing to and from subroutines. Coupled with the variety of ways arrays can be distributed, a FORTRAN-90 implementor faces a rich space in which data motion can be organized.

A model of data motion and an algebraic representation of data motion and data layout are presented. Yale Extension, a set of layout declarations for directing the compiler in distributing the data, is described. An array reference or an array operation extracted from the source FORTRAN-90 program, given a particular data layout specified in Yale Extension, is represented as a *communication expression*. A *communication algebra* and a set of *communication idioms* are described. Experimental results (on the Intel iPSC/2) demonstrating the effectiveness of the approach are provided.

1 Introduction

Cutting down communication overhead and optimizing code performance at the processor level are two main factors in achieving high performance on massively parallel machines. While development of optimizing compilers for superscalar architectures is becoming commonplace in the industry, work on optimization for data motion is mostly done in the context of specialized, handcrafted code written in assembly code, if not microcode, for specific target machines.

In this paper, we describe a general compiler optimization technique that reduces communication overhead for FORTRAN-90 implementations on massively parallel machines. In order to formulate the problem, a conceptual framework formalizing data motion is necessary. Based on this model, we came up with a set of layout directives and operators, called Yale Extension. Together with the array operators of FORTRAN-90, Yale Extension provides a layout specification language which serves several purposes: (1) It provides layout specification for array parameters of (scientific) library functions; (2) it is used by an automatic parallelizing compiler to annotate the data layout generated for the target code; and (3) it is used by the programmer to direct the compiler in distributing data.

The optimization engine that really makes this approach work is an algebraic one. Data motion and layout described in FORTRAN-90 + Yale Extension can be extracted as *communication expressions* which in turn can be simplified and pattern-matched algebraically to achieve minimum data motion. Initial results of this approach are very encouraging: for a benchmark computing the least-square approximation consisting of four data parallel subroutines, the total time, including computation (no optimization for the computation part), is shortened by half.

This work relates to other research in several different areas. Thinking Machine's CM/2 Convolution Compiler [4] optimizes data motion between processors as well as within a processor (register allocation) for a very specialized class of FORTRAN-90 expressions (stencil patterns) and for a specific target. Yale Extension grew out of the domain morphism construct in Crystal [7] and, in one way or another, is similar to layout extensions proposed in FORTRAN-D [9] and Vienna FORTRAN [6]. But there are important differences: (1) Alignment operators in Yale Extension are mostly from the repertoire of FORTRAN-90 array operators as opposed to general expressions used in Crystal or FORTRAN-D. (2) Layout directives are statically declared and lexically scoped to allow compile-time and link-time optimization. (3) Physical map operators give different ways to embed a logical machine into a target machine. (4) Finally, none of these other language extensions have support for optimization. In due time, the optimization module described in this paper will be integrated with the FORTRAN-90-Y compiler [8].

The organization of the rest of the paper is as follows: In Section 2 the model of data motion is introduced. In Section 3 the Yale extension for specifying data layout is presented. In Section 4 our method of optimization is described. Finally, in Section 5, the algebraic simplification and experimental

results for an example program (the least-square approximation) is presented.

2 Model of Data Motion

Data motion comes from the data dependences of the source program coupled with the layout of such data in the machine's distributed memory.

We use a π-block (a strongly connected component of the data-dependence graph [20]) as the unit for considering data layout and data motion.

2.1 Alignment, Partition, and Physical Map

Figure 1 illustrates three stages of mappings from arrays to the physical distributed memory with respect to a given π-block. The first stage, called *alignment*, is for fixing relative location of arrays $(D_1, D_2, \ldots, D_k)$ defined and used in the π-block. These arrays are mapped into a *common domain* E. The second stage, called *partition*, is to map the common domain E into the local memory M of a set of processors L of a logical machine. The third stage, called *physical map*, is responsible for embedding the logical machine (represented as $L \times M$) into the target machine with a set of processors P and their memory M.

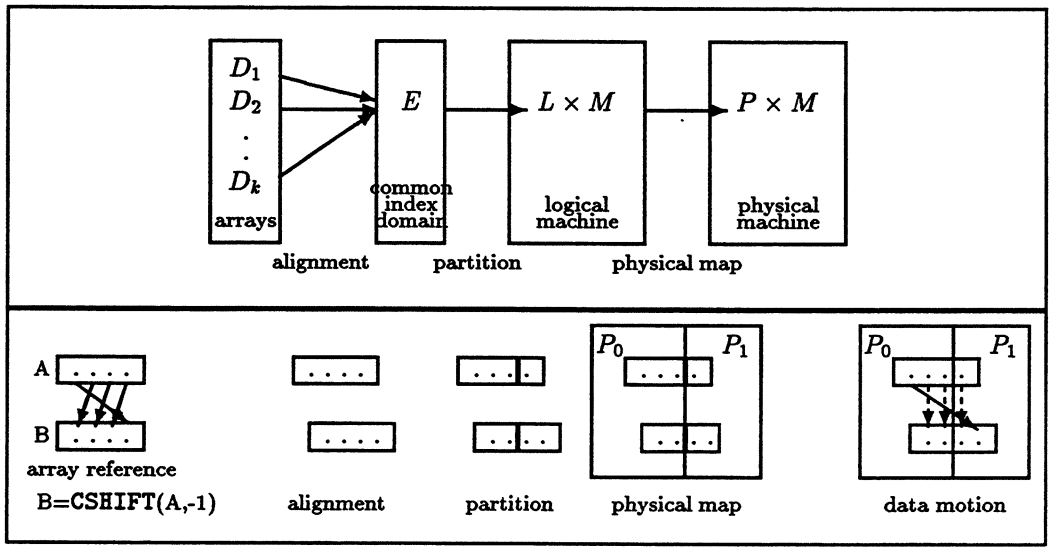


Figure 1: Mapping of Data to Distributed Memory

Example. Two arrays A and B related by the FORTRAN-90 array operator `CSHIFT` are shown in Figure 1. The alignment of the two arrays is responsible for the eventual reduction of the interprocessor communication (one element out of four references).

We assume the following relationship between the size and dimensionality of the arrays, the common domain, the logical machine, and the target machine:
(1) The dimensionality of the common domain is the highest of all arrays, i.e. let e be the dimensionality of the common domain E and d_i the dimensionality of arrays $D_1, D_2, \ldots, D_k$, then $e = \max_i\{d_i\}$.
(2) Sets P and L have the same number of processors.
(3) L is configured as an l-dimensional grid and P as a p-dimensional grid (e.g. a hypercube of p dimensions).

(4) The logical machine must not have higher dimensionality than the target machine, i.e., $l \leq p$; the motivation is that we force the mapping of higher-dimensional data structures to a possibly lower-dimensional machine into the partition stage, i.e.,
(5) $e \geq l$, the dimensionality of the common domain is greater than or equal to that of the logical machine. Note that the logical machine can be made to have a lower dimensionality than that of the target machine as well as the common domain.

2.2 Data Motion, Layout, and Layout Conversion

The three stages of mapping above are now formalized.

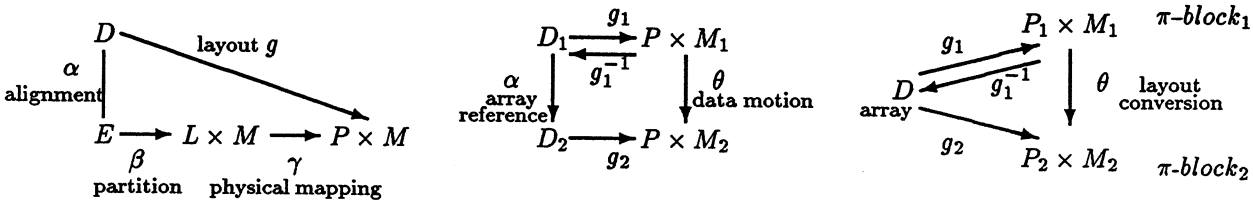
Domain Morphisms. Let the shape of an array, a common domain, a logical machine, or a target machine be captured as an *index domain*. We consider in this paper index domains which are Cartesian products of *interval domains*, denoted as $[l..u]$ where $l \leq u$ is a set of contiguous integers $\{l, l+1, \ldots, u\}$.

We model alignments, partitions, and physical maps as index domain morphisms. The type of *domain morphisms* [7] considered in this paper are *reshape morphisms*. A reshape morphism is a bijective function[1] $g : D \rightarrow E$ from one index domain to another. Since g is bijective, let g^{-1} denote the inverse of g.

Let α denote the alignment morphism from an array D to the common domain E, β denote the partition morphism from the common domain E to the logical machine[2] $L \times M$, and γ denote the physical map from the logical machine $L \times M$ to the target machine $P \times M$.

Communication Expressions. In the following, we are going to use expressions consisting of compositions of domain morphisms which are called *communication expressions*. Constructors other than composition for more complex domain morphisms (e.g. multidimensional) and communication expressions will be presented in Section 4.1.

Layout. With these definitions, the *layout* of an array D can be formally defined as a communication expression $g = \gamma \circ \beta \circ \alpha$, as shown in the commuting diagram of Figure 2(a).



(a) data layout (b) intra-block data motion (c) layout conversion

Figure 2: Commuting Diagrams for Layout, Data Motion, and Layout Conversion

Intra-Block Data Motion. *Intrablock data motion* refers to array references within a given π-block. Let an array reference from array D_1 (used) to array D_2 (defined) be denoted by α and let their layouts to the machine be g_1 and g_2, respectively. Then the data motion induced by the reference α is given by the communication expression $\theta = g_2 \circ \alpha \circ g_1^{-1}$ as shown in Figure 2(b).

[1] For domain morphism g which is injective but not bijective, a reshape morphism $g' : D \rightarrow \mathtt{image}(D, g)$ can be derived from g where $\mathtt{image}(D, g)$ is the image of D under g which is a subset of E.

[2] An abuse of notation here. Since β is mostlikely to be injective but not bijective, the correct morphism of interest should be $\beta' : E \rightarrow \mathtt{image}(E, \beta) \subset L \times M$.

Layout Conversion The second type of data motion, interblock data motion, also called *layout conversion*, refers to array copying due to change of array layout from one π-block to the next. Let the layout of array D in π-block B_1 and B_2 be g_1 and g_2, respectively. Then the layout conversion of D from B_1 to B_2 is given by the communication expression $\theta = g_2 \circ g_1^{-1}$, as shown in Figure 2(c).

3 Yale Extension for Specifying Data Layout

Yale Extension consists of a set of layout directives and operators for specifying layout of arrays for compiling to distributed memory machines. The semantics of a program are not affected by the presence of layout directives, which are for the sake of performance only. If layout strategies are to be automatically decided by a compiler, these layout directives and operators can be used internally to specify which layout strategy has been chosen. There are two major differences between Yale Extension and other layout specifications such as the extension in FORTRAN-D[9] and Vienna FORTRAN[6]. First, Yale Extension is statically declared, lexically scoped directives which achieve dynamic layout via user-defined layout operators while both FORTRAN-D and Vienna FORTRAN treat layout directives as executable statements (dynamically scoped). Consequently, Yale Extension can be optimized at compile-time for data motion and layout conversion, and for link-time optimizations for separately compiled modules. Second, alignment operators in Yale Extension are mostly from the repertoire of FORTRAN-90 array operators as opposed to general expressions used in FORTRAN-D. Consequently, Yale Extension makes explicit the algebraic properties of array operators and layout operators, which forms the theoretical foundation of compiler optimization.

3.1 Alignment Directives

Yale Extension uses a subset of FORTRAN-90 array operators (plus some additional operators) as alignment directives for specifying the relative placement of array elements. This is one of the major differences between Yale Extension and the FORTRAN-D Extension where general expressions are used in alignment.

ALIGN-WITH-BY Statement. The directive for specifying how arrays are aligned is of the form `ALIGN A1 WITH A2 BY G` where `A1` is the *source array* and `A2` the *target array* and `G` is an alignment operator. In the example below, the alignment directive states that element $i-1$ of array `B` should be placed in the same processor as element i of array `A`. As a result of the alignment directive, the execution of the assignment would require only in-processor memory access with no communication necessary. In this simple case it is clear how an alignment directive can be inserted by a compiler automatically. In general, when multiple arrays and multiple assignments are involved, the problem of finding the optimal alignment is NP-complete but heuristics are sufficiently effective [18].

```
REAL, DIMENSION(n) :: A, B
ALIGN B WITH A BY EOSHIFT(B,shift=1)

A = EOSHIFT(B,shift=1) + A
```

Let's say that the source array in an `ALIGN-WITH-BY` statement *precedes* the target array. The closure of multiple `ALIGN-WITH-BY` statements determines the final alignment of all arrays involved. The last array defined by the "precede" relation is referred to as the common domain in the model (Figure 1).

Alignment Operators. In the following, alignment operators will be given in three groups (classified by the shapes of the arrays to be aligned) in both FORTRAN-90 syntax and algebraic notation[3] (for the

[3] In the algebraic notation, the array parameter is not explicitly carried for the sake of brevity. Since only operators applied to the same array can appear in a given communication expression, this will not cause confusion.

purpose of deriving optimized data motion algebraically). Recall that alignment operators must be domain morphisms; their domains, codomains and definitions are given in Tables 10 to 12 of Appendix A. The legal shapes[4] of the arrays with respect to an alignment operator are checked by the compiler.

ALIGN-I operators are defined over one-dimensional arrays. For higher-dimensional arrays, an operator is repeatedly applied for different dimensions (specified by its second argument). In the algebraic notation, the *product* constructor is used to construct higher-dimensional operators from one-dimensional operators. Operators prefixed by * are those in Yale Extension but not in FORTRAN-90. Figure 3 illustrates examples of ALIGN-I operators.

ALIGN-I	FORTRAN-90 Operators	Algebraic Notations
	EOSHIFT(A, dim,shift, [,boundary])	EOSHIFT(shift, [,boundary])
	CSHIFT(A, dim, shift)	CSHIFT(shift)
	*REFLECT(A, dim)	REFLECT
	*STRIDE(A, dim, density, offset)	STRIDE(density, offset)

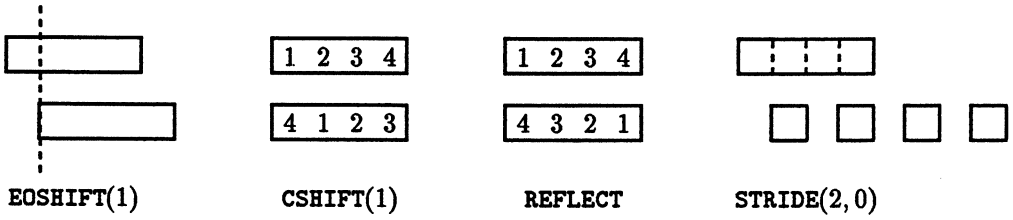


Figure 3: Examples of ALIGN-I Operators

ALIGN-II operators are multidimensional operators which take an array and a matrix $[a_{ij}]$ of the same dimensions as the arguments. FORTRAN-90's TRANSPOSE is specifically for two-dimensional arrays. A generalized TRANSPOSE takes a permutation matrix as its second argument. Operator SKEW takes an n-dimensional array as an argument, and returns a skewed array (which may be larger in size) where the second argument is an n-by-n coefficient matrix representing the n-variable (n-axes) linear function. See Table 11 of Appendix A for its definition. Operator CSKEW is similar except with the skewed array wrapped around in the shape of the argument array. Figure 4 illustrates some examples.

ALIGN-II	FORTRAN-90 Operators	Algebraic Notations
	TRANSPOSE(A)	TRANS $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
	TRANSPOSE(A, perm)	TRANS($[p_{ij}]$)
	*SKEW(A, matrix)	SKEW($[a_{ij}]$)
	*CSKEW(A, matrix)	CSKEW($[a_{ij}]$)

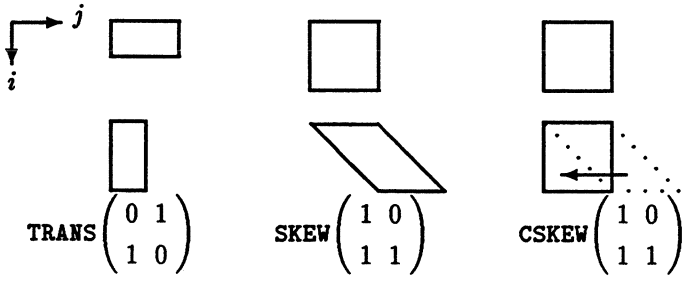


Figure 4: Examples of ALIGN-II Operators

ALIGN-III operators are for reshaping arrays and selecting regular sections of an array. It is characteristic for the operators in this group to have different shapes for its argument and the result arrays.

[4] In the above example, the shape of array B determines the domain of operator EOSHIFT, and array A must be of the same dimensionality as the codomain of EOSHIFT.

Operator **RESHAPE** of FORTRAN-90 turns a one-dimensional array into a multidimensional array. A generalized **RESHAPE** in the Yale Extension does reshaping between two arrays of arbitrary dimensionality. Taking a regular section of an array is achieved in FORTRAN-90 by the index triplet notation `lower:upper:stride`. Yale Extension has a new operator **EMBED** for embedding a lower-dimensional array into the selected dimensions of a higher-dimensional array. Figure 5 gives some examples, and Table 12 of Appendix A gives the definitions.

ALIGN-III	FORTRAN-90 Operators	Algebraic Notations
	RESHAPE(A, shape, [,pad] [,order])	**RESHAPE**(shape, order [,pad])
	RESHAPE(A, s1, s2 [,pad] [,o1] [,o2])	**RESHAPE**(shape1, order1, shape2, order2)
	A(lower:upper:stride)	**SECTION**(lower, upper, stride)
	*EMBED(source,target,dimensions)	**EMBED**($[a_{ij}]$, Domain(target))

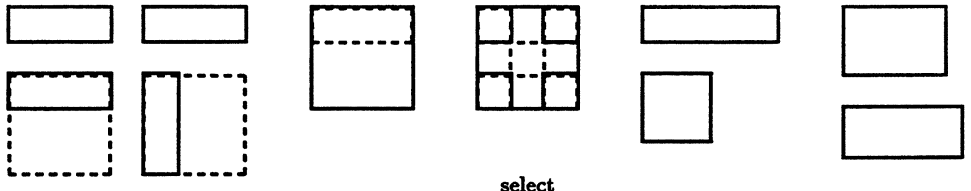


Figure 5: Examples of ALIGN-III Operators

COMMON-DOMAIN Statement. As described in the model, all arrays are mapped to a common array at the alignment stage. Most of the time, the common domain is implicitly defined by the closure of all alignment directives. But it can be explicitly defined by `COMMON-DOMAIN H(n1,n2,...)` where `n1, n2, ...` specify the dimension of array `H`. The form `COMMON-DOMAIN H(DEFAULT)` allows one to name a specific array `H` as the common domain whose dimensionality is implicitly defined by the alignment directive in which `H` is the target array. Note that the `COMMON-DOMAIN` statements are different from the `DECOMPOSITION` statement of FORTRAN-D which must be declared for subsequent `ALIGN` statements.

User-defined Alignment Operators. Though directives in Yale Extension are statically declared and lexically scoped (to be precisely defined later), the operators themselves can be user-defined, dynamically interacting with the program execution. A user-defined alignment operator is declared as a FORTRAN-90 subroutine taking the array to be aligned as its argument. It can then be called by the `ALIGN-WITH-BY` statement.

FORTRAN-90 Array Operators Excluded from Yale Extension. Before ending this section, we want to point out that the two very useful FORTRAN-90 operators **SPREAD** and **SUM** (and other reduction operations), for replication and reduction, are excluded from the repertoire of alignment operators because they are not domain morphisms. These two operators will play a part in the algebraic manipulation presented later when we show how data motion can be optimized. Their algebraic notations are given here, and their definitions are given in Table 12 of Appendix A.

Non-ALIGN array operations only	FORTRAN-90 Operators	Algebraic Notations
	SPREAD(A, dim, ncopies)	**SPREAD**($[a_i], D$)
	SUM(A, dim)	**SUM**($[a_i]$)

3.2 Partition Directives

Partition Operators The partition operators in Yale Extension are no more than the standard strategies of decomposing arrays into blocks and (blocks of) columns or rows. This approach is well-known and used in various parallelizing compilers. Since there is no difference between the partition operators of Yale Extension and FORTRAN-D, we'll follow the FORTRAN-D syntax except that the block size or number of cyclic layers are optional parameters. When these parameters are given as compile-time constants, then the compiled code is specialized to a particular machine size. Otherwise, the machine size is kept as a runtime parameter. In the following, we simply list these operators along with their algebraic notations without any explanation. The definitions of the algebraic notations are given in Table 14 of Appendix A.

Partition	FORTRAN-D Specification	Algebraic Notations
	`BLOCK [(b)]`	`BLOCK`(b), where $b = n/p$
	`CYCLIC [(p)]`	`CYCLIC`(p)
	`BLOCK_CYCLIC`(b [,p])	`BCYCLIC`(b,p)
	*	`SEQ`

PARTITION-BY Statement The statement `PARTITION A BY G` says that array `A` will be decomposed into parts by partition operator `G`. An example of its use is given.

```
REAL, DIMENSION(n, m) :: A, B
ALIGN B WITH A BY EOSHIFT(B,dim=1,shift=1)
PARTITION A by (BLOCK,BLOCK)
```

The `PARTITION` statement says that array `A` should be decomposed into blocks and one block assigned to each logical processor. Since arrays `A` and `B` are aligned, both arrays will be partitioned as blocks. In general, partition operators need to be declared on only one array out of the group of arrays related by alignment. Inconsistent `PARTITION` statements[5] *declared* within the same scope (to be defined momentarily) for the related arrays will be signaled by the compiler and only the lexically first partition directive will be used and the rest will be ignored. What if a user is interested in partitioning in the *same* π-block two arrays differently, say one array by blocks and the other by blocks of rows? This effect can be achieved by alignment using composition of `RESHAPE`, which makes explicit the relative locations of the array elements instead of allowing this relation to be defined implicitly by two separate `PARTITION` statements unintentionally. It is important to note that Yale Extension relies on the three-stage data motion model and ensures strict adherence to it.

Similar to alignment, a partition operator `G` can be a user-defined FORTRAN-90 subroutine.

3.3 Physical Map Directives

The final stage is to map each logical processor (i.e. a partition of the common domain) to a processor of the target machine. Three different axis-encoding schemes [14], namely, binary encoding (`BINARY`), Gray (binary reflected) encoding (`GRAY`), and random assignment (`RANDOM`) are used for mapping multidimensional grids to hypercube networks. In the future, we expect new encoding schemes for networks such as the fat-tree network of CM-5 [1]. The directive is of the form `PMAP A BY G` where `A` is an array and `G` one of the axis-encoding schemes. Again, we restrict that those arrays that are related by alignment have the same axis-encoding. Inconsistent axis-encoding will be signaled and ignored.

[5] Let D_1 be the index domain of a given array, β_1 the partition operator for D_1, $\alpha : D_1 \rightarrow D_2$ the alignment operator, and β_2 the partition operator for D_2. We say that β_1 and β_2 are consistent if they are of the same kind under α. For instance, if α is a `EOSHIFT` operator, then the ith dimension of D_1 and ith dimension of D_2 must be partitioned using the operators of the same kind (e.g., both are `BLOCK`, `CYCLIC` or `BCYCLIC`).

3.4 Scope of Layout Directives

The layout directives in Yale Extension is statically declared within a *phase*, defined by `BEGIN_PHASE` and `END_PHASE`. Technically, a phase must be a π-block or composition of π-blocks. The compiler will signal a warning message if a phase is illegally defined, and the layout directives declared within it will be ignored. Phases can be nested and the scope of a layout directive is lexically determined to be analogous to the scope of variable binding in block-structured programming languages. For example, in the inner phase `P2`, the block-of-rows partition for `B` is inherited from `P1`, so no data movement for B will be needed. But array `A` needs to be remapped within `P2`, and, in theory, remapped again when leaving `P2`. But in practice, the algebraic simplification will eliminate unnecessary remapping.

```
   REAL, DIMENSION(n,n) :: A, B
P1 BEGIN_PHASE
      ALIGN A WITH B BY EOSHIFT(A,dim=1,shift=1)
      PARTITION A, B by (BLOCK,*)
      ...
P2    BEGIN_PHASE
         PARTITION A by (BLOCK,BLOCK)
         ...
      END_PHASE
      ...
   END_PHASE
```

The following code segment defines an iteration in an ADI method (example from [6]), which contains two subphases `P1` and `P2`; each encloses a loop. In the parent phase, arrays `U` and `F` are partitioned as blocks of columns. In the subphase `P1`, array `V` has the same blocks-of-columns partition. In subphase `P2`, array `V` is redeclared to have blocks-of-rows partition while arrays `U` and `F` inherit their layout from the parent phase P.

```
cc ADI iteration
   REAL, DIMENSION(nx,ny) :: U, F, V
P  BEGIN_PHASE
     PARTITION U, F by (*,BLOCK)
P1   BEGIN_PHASE
       PARTITION V by (*,BLOCK)
       CALL RESID(V,U,F,nx,ny)
       DO j=1,ny
          CALL TRIDIAG(V(:,j),nx)
       END DO
     END_PHASE
P2   BEGIN_PHASE
       PARTITION V by (BLOCK,*)
       DO i=1,nx
          CALL TRIDIAG(V(i,:),ny)
       END DO
       U = V
     END_PHASE
   END_PHASE
```

A phase can also be declared within a loop as shown in the example below.

```
   DO i=1, n
      BEGIN_PHASE
         PARTITION A, B by G
         ...
      END_PHASE
   END DO
```

If the partition operator `G` uses one of the intrinsic operators of Yale Extension, then the effect of the directive would be the same as if the phase is declared outside the loop because the layout does not change from one iteration to the next. This fact can be derived by simplifying the communication expression

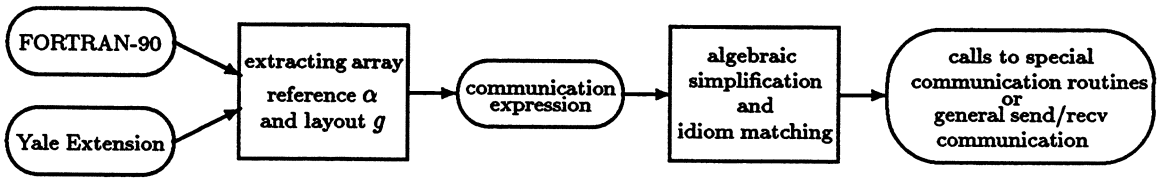


Figure 6: Overview of the Optimization Method

$\theta = \mathtt{G} \circ \mathtt{G}^{-1} = id$ (refer to Figure 2(c)). However, operator G can be a user-defined subroutine whose dynamic behavior depends on the loop, achieving the effect of dynamic remapping between iterations.

4 Method of Optimization

In this section, we outline the optimization method and describe communication algebra and communication idioms. Overview of the optimization method is shown in Figure 6.

The optimization method is based on a communication algebra. First, array references and layout directives in the source program are extracted and expressed as communication expressions for each array in a given scope. The communication expressions are then simplified and matched with a set of communication idioms for which specialized, fast communication routines are implemented. Calls to these special communication routines or to general send/receive commands — if no match exists — are generated. As a result of this optimization, any redundant layout conversion (using the intrinsic operators of Yale Extension) between program phases and subroutines will be eliminated. Data motion and local copying within a program phase and subroutine will be reduced to a minimum, and any opportunity for specialized fast communication will be uncovered by idiom matching.

In the following, we first describe very briefly the communication algebra (refer to Appendix B for a complete set of rules) and communication idioms. We then use the LSA example to illustrate the method.

4.1 Communication Algebra

The *communication algebra* is an algebraic system consisting of

- A set of array operators (FORTRAN-90) and layout operators (Yale Extension).
- An n-ary product operation "$\times$" (for multidimensional array reference and layout operators).
- A binary composition operation "$\circ$".
- A binary array-reference combination operation "$\dot{+}$" (for multiple instances of array references on the same array).
- A unary boundary operation "bd" (for extracting data for interprocessor communication).
- A binary boundary combination operation "+".

All primitive operators and operators constructed by various constructors $(\times, \circ, \dot{+}, \mathtt{bd}, +)$ (refer to Appendix B for the definitions) satisfy the basic properties shown in Figure 7, except that SPREAD and SUM do not have inverses (Prop 1). Let g and g_i be array operators or layout operators.

These basic properties allow the simplification of $g \circ g^{-1}$ at any place of a communication expression and partial simplification of the components in a compound operator constructed using product or combination.

Prop 1 Inverse

$$g : D \to E,\ g^{-1} : E \to D$$
$$g \circ g^{-1} = id_E$$

Prop 2 Associativity of Composition

$$g_1 \circ (g_2 \circ g_3) = (g_1 \circ g_2) \circ g_3$$

Prop 3 Associativity of Combination

$$g_1 \dot{+} (g_2 \dot{+} g_3) = (g_1 \dot{+} g_2) \dot{+} g_3$$
$$\mathrm{bd}(g_1) + (\mathrm{bd}(g_2) + \mathrm{bd}(g_3)) = (\mathrm{bd}(g_1) + \mathrm{bd}(g_2)) + \mathrm{bd}(g_3)$$

Prop 4 Commutativity of Combination

$$g_1 \dot{+} g_2 = g_2 \dot{+} g_1$$
$$\mathrm{bd}(g_1) + \mathrm{bd}(g_2) = \mathrm{bd}(g_2) + \mathrm{bd}(g_1)$$

Prop 5 Product-Composition Exchange

$$(g_1 \times g_2) \circ (g_3 \times g_4) = (g_1 \circ g_3) \times (g_2 \circ g_4)$$

Prop 6 Product-Combination Exchange

$$(g_1 \times g_2) \dot{+} (g_3 \times g_4) = (g_1 \dot{+} g_3) \times (g_2 \dot{+} g_4)$$

Prop 7 Composition-Combination Exchange

$$(g_1 \circ g_2) \dot{+} (g_3 \circ g_4) = (g_1 \dot{+} g_3) \circ (g_2 \dot{+} g_4),$$
for $g_2 = g_4$ and g_1, g_3 combinable

Prop 8 Distribution of bd over Product

$$\mathrm{bd}(g_1 \circ g_2) + \mathrm{bd}(g_3 \circ g_4) = \{\mathrm{bd}(g_1 \dot{+} g_3), \mathrm{bd}(g_2 \dot{+} g_4)\}$$
$$g_1 : D_1 \to E_1, g_2 : D_2 \to E_2$$
$$\mathrm{bd}(g_1 \times g_2) = (\mathrm{bd}(g_1) \times D_2) + (D_1 \times \mathrm{bd}(g_2))$$

Figure 7: Basic Properties of Communication Algebra

Many additional rules are useful in the derivation procedure, such as those on the combination operator that allow coarse-grained operations on arrays, composition rules for alignment operators within each group and interactions of operators in different groups, and expansion rules for exposing communication idioms. A complete set of rules is given in Appendix B.

4.2 Communication Idioms

Since the advent of massively parallel machines, many researchers (e.g. [10]) have developed specialized communication routines to facilitate direct programming of distributed-memory machines. In building compilers, we might as well take advantage of these handcrafted, highly optimized routines which become part of the runtime system for the language. In [17], this approach is used to generate intraprocedure communication. Here we extend that work further to include those communication routines for converting layouts between program phases or subroutine calls.

All of the data motion in layout conversion can be formulated as so-called *personalized communication* and *dimension permutation* [11]. Optimal algorithms have been devised for one-to-all and all-to-all personalized communications [2, 3, 11, 16, 15, 19], and dimension permutations [11, 12, 13, 15] using nearest-neighbor communication on hypercubes.

We have collected a set of frequently occurring data motions based on the results of this work. We abstract the content of a communication routine as a *communication idiom* in terms of the FORTRAN-90 array operators and Yale Extension's layout operators, simplified into a *normal form.*

Due to the space limitation, we cannot go into these idioms except to list them here (see Table 1). The optimization procedure simply goes through this list of idioms and pattern matches on the normal form representation.

5 Example and Experimental Results

In this section, we illustrate the optimization method and demonstrate its effectiveness with experimental results using an example program computing the least-square approximation, or LSA. Gray code encoding, the default encoding scheme for hypercube networks, will be used for the entire program.

Idioms	Types of Conversion	Calling Convention for Communication Routines
$\gamma_1 \circ \gamma_2^{-1}$	change encoding	`code-conversion`(i, A, P, L)
$\gamma \circ \alpha \circ \gamma^{-1}$	coarse-grain alignment	`coarse-align`$(op, [a_{ij}], A, P, L, G)$
$\Pi^d(\gamma_1 \circ \beta_1 \circ \beta_2^{-1} \circ \gamma_2^{-1})$	change partition	`change-partition`$(A, A_t, P_1, P_2, L_1, L_2, B_1, B_2, G_1, G_2)$
$\gamma_i \circ \beta_i \circ$ `EOSHIFT`$(c, b) \circ \beta_i^{-1} \circ \gamma_i^{-1}$	shift	`shift`(i, c, A, A_t, P, L)
$\gamma_i \circ \beta_i \circ$ `CSHIFT`$(c) \circ \beta_i^{-1} \circ \gamma_i^{-1}$	cyclic shift	`cshift`(i, c, A, A_t, P, L)
$\gamma_i \circ \beta_i \circ$ `REFLECT` $\circ \beta_i^{-1} \circ \gamma_i^{-1}$	reversal permutation	`reflect`(i, A, P, L)
$\Pi^d(\gamma \circ \beta) \circ$ `EOSH`$([c_i]) \circ \Pi^d(\beta^{-1} \circ \gamma^{-1})$	Multi-D shift	`mshift`$([c_i], A, A_t, P, L)$
$\Pi^d(\gamma \circ \beta) \circ$ `CSH`$([c_i]) \circ \Pi^d(\beta^{-1} \circ \gamma^{-1})$	Multi-D cyclic shift	`mcshift`$([c_i], A, A_t, P, L)$
$\Pi^d(\gamma \circ \beta) \circ$ `REF` $\circ \Pi^d(\beta^{-1} \circ \gamma^{-1})$	Multi-D reflection	`mreflect`(A, P, L)
$\Pi^d(\gamma \circ \beta \circ (\bar{\alpha}(-c) \dot{+} \bar{\alpha}(c)) \circ \beta^{-1} \circ \gamma^{-1})$	n-port communication	`n-port`(c, A, A_t, P, L)
`TRANS`$([a_{ij}])$	transposition	`transpose`$([a_{ij}], A, A_t, P, L)$
$\Pi^d(\gamma \circ \beta) \circ$ `SKEW`$([a_{ij}]) \circ \Pi^d(\gamma \circ \beta)^{-1}$	skewing	`skew`$(cyclic?, [a_{ij}], A, P, L)$
$\Pi^d(\gamma_1 \circ \beta_1) \circ$ `RESHAPE`$([m], [], [l_i], [a_{ij}]) \circ \beta_2^{-1} \circ \gamma_2^{-1}$	axis spliting	`split`$([l_i], [a_{ij}], A, A_t, P_1, P_2, L)$
$\gamma_1 \circ \beta_1 \circ$ `RESHAPE`$([m_i], [a_{ij}], [l], []) \circ \Pi^d(\beta_2^{-1} \circ \gamma_2^{-1})$	axis combining	`merge`$([m_i], [a_{ij}], A, A_t, P_1, P_2, L)$
$\Pi^d(\gamma_1 \circ \beta_1) \circ$ `SECTION`$([a_i]) \circ \Pi^d(\beta_2^{-1} \circ \gamma_2^{-1})$	regular section re-mapping	`section`$([a_i], A, P_1, P_2, L_1, L_2, B_1, B_2, G_1, G_2)$
$\gamma \circ \beta \circ$ `EMBED`$([a_i]) \circ$ `EMBED`$([b_i])^{-1} \circ \beta^{-1} \circ \gamma^{-1}$	transposition of 1D array	`1d-transpose`$([a_i], [b_i], A, P_1, P_2, L)$

Table 1: Communication Idioms where α denotes alignment operators or array references, β denotes partition operators and γ denotes physical map operators; $\Pi^d(a \circ b)$ denotes $(a_1 \circ b_1) \times \ldots \times (a_d \circ b_d)$.

```
cc  Least-Square Approximation

M1    REAL, DIMENSION(p,q) :: A
 2    REAL, DIMENSION(q,p) :: B, D
 3    REAL, DIMENSION(q,q) :: C
 4    REAL, DIMENSION(p) :: b
 5    REAL, DIMENSION(q) :: x

cc  layout directives

 6    ALIGN B with D
 7    ALIGN b with D by EMBED(b,D,dim=1)
 8    ALIGN x with D by EMBED(x,D,dim=1)
 9    PARTITION A, C, D by (BLOCK,BLOCK)

      ... statements for input values of A and b

10    call transpose(A,B,p,q)
11    call mm(B,A,C)
12    call inverse(C)
13    call mm(C,B,D)
14    call mvm(D,b,x)
STOP
END


SUBROUTINE mm(A, B, C, p, q, r)

X1     REAL, DIMENSION(p,q) :: A, A_TEMP
 2     REAL, DIMENSION(q,r) :: B, B_TEMP
 3     REAL, DIMENSION(p,r) :: C
 4     INTEGER, DIMENSION(2,2) :: MA, MB

 5     DATA MA / 1, 1, 0, 1/
 6     DATA MB / 1, 0, 1, 1/

cc  layout directives

 7     ALIGN A with A_TEMP by CSKEW(A, MA)
 8     ALIGN B with B_TEMP by CSKEW(B, MB)
 9     PARTITION A_TEMP, B_TEMP, C by (BLOCK, BLOCK)

cc  executable statements

10     A_TEMP = CSKEW(A, MA)
11     B_TEMP = CSKEW(B, MB)

12     DO K = 1, q
13        C = C + A_TEMP * B_TEMP
14        A_TEMP = CSHIFT(A_TEMP, dim=2, shift=1)
15        B_TEMP = CSHIFT(B_TEMP, dim=1, shift=1)
       END DO

END SUBROUTINE MM


SUBROUTINE mvm(A,b,c,m,n)

V1     REAL, DIMENSION(m,n) :: A, B_TEMP, C_TEMP
 2     REAL, DIMENSION(n) :: b
 3     REAL, DIMENSION(m) :: c

cc  layout directives

 4     ALIGN C_TEMP with A
 5     ALIGN b with A by EMBED(b,A,dim=2)
 6     ALIGN c with A by EMBED(c,A,dim=1)
 7     PARTITION A by (BLOCK, BLOCK)

cc  executable statements

 8     C_TEMP = A * SPREAD(b, dim=1, ncopies=m)
 9     c = SUM(C_TEMP, dim=2)

END SUBROUTINE MVM


SUBROUTINE inverse(C,n)

I1     REAL, DIMENSION(n,n) :: C
 2     INTEGER, DIMENSION(1) :: MAX_LOC
 3     REAL, DIMENSION(n) :: TEMP_ROW
 4     INTEGER :: k

cc  layout directives

 5     ALIGN TEMP_ROW with C by EMBED(TEMP_ROW,C,dim=2)
 6     PARTITION C by (*,CYCLIC)

cc  executable statements

 7     DO k = 1, n-1
 8       MAX_LOC = MAXLOC( ABS( C(k:n,k)))
 9       TEMP_ROW(k:n) = C(k,k:n)
10       C(k,k:n) = c(k-1+MAX_LOC(1),k:n)
11       C(k-1+MAX_LOC(1), k:n) = TEMP_ROW(k:n)

12       C(k,k+1:n) = C(k,k+1:n) / C(k,k)
13       C(k,k) = 1
14       C(1:n, k+1:n) = C(1:n, k+1:n) -
15  &             SPREAD(C(k,k+1:n), dim=1, ncopies=n)
16  &             SPREAD(C(1:n,k), dim=2, ncopies=n-k)
       END DO
END SUBROUTINE INVERSE
```

Given a linear system $Ax = b$ which is inconsistent (the number of equations is more than the number of variables), the problem is to choose $\tilde{x}$ so as to minimize the error. The computation of $\tilde{x}$ can be decomposed into four parallel subroutines as shown in the FORTRAN-90 Program shown above.

The matrix multiplication subroutine `mm` uses Cannon's algorithm [5] which requires that the two operand matrices be cyclically skewed by using the array operator `CSKEW` (line 10 and 11 of `mm`). Due to performance concerns, the array parameters require different layouts in each subroutine as shown in Figure 8. For example, array `B` must be changed from the standard representation by `CSKEW` when it is copied into `B_TEMP` in the first call to `mm`, then again changed by `CSKEW` but in a different direction when copied into `B_TEMP` in the 2nd call to `mm`. Array `C` would change from a two-dimensional block partition in the first call to `mm` to a cyclic column partition in `inverse` and then change back to a two-dimensional block partition in the second call to `mm` and be copied by a `CSKEW`.

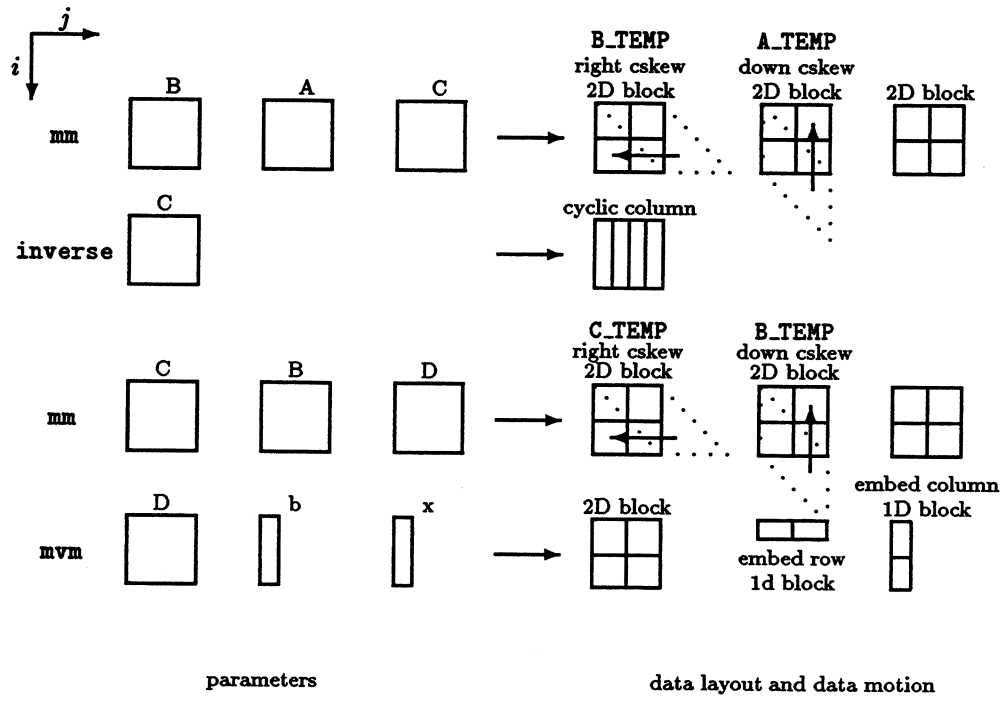


Figure 8: Data Layout Strategies for the Least Square Approximation

Our problem now is how to minimize the communication overhead within each subroutine as well as reduce the overhead of layout conversion. The first step is to extract communication expressions from the source program and then apply the algebraic simplification procedure. The initial communication expressions for intrablock data motion are shown in Table 2. For simplicity, throughout this section we will ignore physical map operators in the communication expressions, since the Gray code encoding scheme is used for the entire program and there will be no code conversion in the program.

The communication expression for `A_TEMP` is extracted from the two nested loops strip-mined from line 12 in subroutine `mm` with the outer loop iterating from 1 to p_2 and the inner loop iterating from 1 to b_2 where p_2 is the number of processors at the second dimension and b_2 the block size at the second dimension. The extraction of the communication expression for `B_TEMP` is similar. The extraction of other communication expressions is straightforward.

Subroutine	Line	Array	Communication Expression
mm	10	A	$(\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2)) \circ \texttt{CSKEW}\begin{pmatrix}1&0\\1&1\end{pmatrix} \circ \texttt{CSKEW}\begin{pmatrix}1&0\\1&1\end{pmatrix}^{-1} \circ (\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2))^{-1}$
	11	B	$(\texttt{BLOCK}(b_2) \times \texttt{BLOCK}(b_3)) \circ \texttt{CSKEW}\begin{pmatrix}1&1\\0&1\end{pmatrix} \circ \texttt{CSKEW}\begin{pmatrix}1&1\\0&1\end{pmatrix}^{-1} \circ (\texttt{BLOCK}(b_2) \times \texttt{BLOCK}(b_3))^{-1}$
	14	A_TEMP	$(\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2)) \circ ((id \times \texttt{CSHIFT}(1))^{b_2})^{p_2} \circ (\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2))^{-1}$
	15	B_TEMP	$(\texttt{BLOCK}(b_2) \times \texttt{BLOCK}(b_3)) \circ ((\texttt{CSHIFT}(1) \times id)^{b_2})^{p_2} \circ (\texttt{BLOCK}(b_2) \times \texttt{BLOCK}(b_3))^{-1}$
mvm	8	b	$(\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2)) \circ \texttt{SPREAD}([1,0],[1..m]) \circ (\texttt{BLOCK}(b_2))^{-1}$
	9	C_TEMP	$(\texttt{BLOCK}(b_1)) \circ \texttt{SUM}([0,1]) \circ (\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2))^{-1}$
inverse	15	C	$(\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2)) \circ \texttt{SPREAD}([1,0],[1..n]) \circ (\texttt{BLOCK}(b_2))^{-1}$
	16	C	$(\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2)) \circ \texttt{SPREAD}([0,1],[k..n]) \circ (\texttt{BLOCK}(b_1))^{-1}$

Table 2: Communication Expressions for Intra-Block Data Motion in LSA

Array	Communication Expression
θ_B	$(\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2)) \circ \texttt{CSKEW}\begin{pmatrix}1&1\\0&1\end{pmatrix} \circ \texttt{CSKEW}\begin{pmatrix}1&0\\1&1\end{pmatrix}^{-1} \circ (\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2))^{-1}$
θ_{C_1}	$(\texttt{SEQ} \times \texttt{CYCLIC}(p)) \circ (\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2))^{-1}$
θ_{C_2}	$(\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2)) \circ \texttt{CSKEW}\begin{pmatrix}1&0\\1&1\end{pmatrix} \circ (\texttt{SEQ} \times \texttt{CYCLIC}(p))^{-1}$
θ_D	$(\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2)) \circ (\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2))^{-1}$

Table 3: Communication Expressions for Inter-Block Data Motion in LSA

The communication expressions for interblock data motion are shown in Table 3.

Appendix C contains the derivation steps for generating calls to communication primitives. The simplifications for array A and B to identity functions are straightforward (simply by Prop 1). The first set of derivations (Figure 9) is for optimizing the data motion for array A_TEMP within the matrix multiplication subroutine mm. The derivations for array B_TEMP are similar. The performance effects of these optimizations are shown in Table 4. Optimization of data motion for array C_TEMP within the matrix-vector multiplication subroutine mvm is shown in Figure 10. The performance effects of these are shown in Table 5. Next is the simplification of layout conversion for array B between the first and the second calls to subroutine mm (Derivation 6 of Appendix C). The performance effects of these are shown in Table 6. The layout conversion for array C between the calls to subroutine inverse and mm and its performance effects are shown in Derivation 7 of Appendix C and Table 7 respectively. The last derivation (Derivation 8 of Appendix C) is for optimizing the layout conversion for array C between the calls to subroutine mm and inverse. Its performance effects are shown in Table 8.

The experiment was conducted using handcompiled code on the Intel iPSC/2 hypercube located at Yale, which has 64 Intel 80386 processors, runs Unix V/386 3.2 as its host operating system, NX 3.2 for the nodes, and provides the Intel 3.2 C compiler. The time units shown in the tables are milliseconds.

Finally, Table 9 shows the overall effect of the data motion optimization on the total computation and communication time between the unoptimized version and the fully optimized version. The average improvement of the overall performance is about 127% for a small problem size and 95% for a large problem size. The degradation of improvement for the large problem size may be caused by the increasing overhead for boundary checking in the computation part.

Problem Size	Fine-Grained Unoptimized	Fine-Grained Boundary Extraction Derivation 1	Coarse-Grained Boundary Extraction Derivation 2,3,4	Total Improvement
32 x 32	64	48 (33%)	16 (200%)	300%
64 x 64	240	132 (82%)	75 (76%)	220%
128 x 128	1354	525 (158%)	365 (44%)	272%
256 x 256	9914	3452 (187%)	2752 (25%)	260%

Table 4: Matrix Multiplication with Different Schemes of Shifting

Problem Size	Fine-Grained Unoptimized	Coarse-Grained Derivation 5	Improvement
32 x 32	3	1	200%
64 x 64	14	3	366%
128 x 128	28	7	300%
256 x 256	58	16	263%

Table 5: Matrix-vector Multiplication with Different Schemes of Reduction

Problem Size	No Simplification	With Simplification Derivation 6	Improvement
32 x 32	2	1	100%
64 x 64	4	2	100%
128 x 128	10	5	100%
256 x 256	24	13	84%

Table 6: Layout Conversion for Array `B` Between the First Call and the Second Call to Subroutine `mm`

Problem Size	General Comm. Unoptimized	Naive send/recv Derivation 7	Conversion Idiom Derivation 7	Total Improvement
32 x 32	55	54 (2%)	32 (69%)	72%
64 x 64	73	68 (7%)	38 (79%)	92%
128 x 128	78	72 (8%)	39 (80%)	97%
256 x 256	102	92 (11%)	51 (80%)	100%

Table 7: Layout Conversion for Array `C` Between the Call to Subroutine `mm` and the Call to Subroutine `inverse`

Problem Size	General Comm. Unoptimized	Naive send/recv Derivation 7	Conversion Idiom Derivation 7	Total Improvement
32 x 32	56	55 (2%)	33 (67%)	70%
64 x 64	75	70 (7%)	39 (79%)	92%
128 x 128	81	74 (9%)	42 (76%)	93%
256 x 256	108	97 (11%)	56 (73%)	93%

Table 8: Layout Conversion for Array `C` from Subroutine `inverse` to Subroutine `mm`

Problem Size	No Optimization	With Optimization	Improvement
32 x 32	440	193	127%
64 x 64	1541	707	118%
128 x 128	5410	2516	115%
256 x 256	29610	15173	95%

Table 9: Total Elapsed Time for LSA

6 Conclusion

In this paper, we have presented a general compiler optimization technique that reduces communication overhead for FORTRAN-90 implementations on massively parallel machines. We have also demonstrated the effectiveness of the optimization method on an example program. We expect such effectiveness will become even more significant for larger application programs which usually contain many program modules and may involve abundant use of array operations. We are now integrating the optimization module described in this paper with the FORTRAN-90-Y compiler at Yale [8].

Acknowledgement

We thank Young-il Choo and Allan Yang for much helpful advice and inspiring discussion on the communication algebra. We also thank Lennart Johnsson, Ching-Tien Ho and Yu Hu for very helpful information and discussion on interprocessor communication and scientific computing applications for massively parallel systems. We also thank Ken Kennedy for pointing us to the regular section analysis. Our thanks also to Chris Hatchell for his careful proofreading.

References

[1] The Connection Machine CM-5 Technical Summary. Technical report, Thinking Machines Corporation, 1991.

[2] Shahid H. Bokhari. Complete Exchange on The iPSC-860. Technical report, ICASE, NASA Langley Research Center, 1991.

[3] Shahid H. Bokhari. Multiphase Complete Exchange on A Circuit Switched Hypercube. Technical report, ICASE, NASA Langley Research Center, 1991.

[4] M. Bromley, S. Heller, T. McNerney, and G. L. Steele Jr. Fortran at Ten Gigaflops: The Connection Machine Convolution Compiler. In *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 145–156, June 1991.

[5] L. E. Cannon. A Cellular Computer to Implement the Kalman Filter Algorithm. Technical report, Montana State University, 1969.

[6] Barbara Chapman, Piyush Mehrotra, and Hans Zima. Vienna FORTRAN – A Fortran Language Extension for Distributed Memory Multiprocessors. Technical report, ICASE, NASA Langley Research Center, 1991.

[7] Marina Chen, Young-il Choo, and Jingke Li. Theory and pragmatics of generating efficient parallel code. In *Parallel Functional Languages and Compilers*, chapter 7. ACM Press and Addison-Wesley, 1991.

[8] Marina Chen and James Cowie. Prototyping Fortran-90 Compilers for Massively Parallel Machines. Technical report, Department of Computer Science, Yale University, 1991.

[9] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. Technical report, Dept. of Computer Science, Rice University, 1990.

[10] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lysenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors.* Prentice Hall, 1988.

[11] Ching-Tien Ho. *Optimal Communication Primitives and Graph Embeddings on Hypercubes.* PhD thesis, Dept. of Computer Science, Yale University, 1990.

[12] Ching-Tien Ho and S. Lennart Johnsson. Optimal Algorithms for Stable Dimension Permutations on Boolean Cubes. In *The Third Conference on Hypercube Concurrent Computers and Applications*, pages 725–736. ACM, 1988.

[13] Ching-Tien Ho and S. Lennart Johnsson. The Complexity of Reshaping Arrays on Boolean Cubes. In *proceedings of 5th Distributed Memory Computing Conference*, 1990.

[14] S. Lennart Johnsson. The FFT and Fast Poisson Solvers on Parallel Architectures. Technical report, Department of Computer Science, Yale University, 1987.

[15] S. Lennart Johnsson and Ching-Tien Ho. Matrix Transposition on Boolean n-cube Configured Ensemble Architectures. *SIAM J. Matrix Anal. Appl.*, 9(3):419–454, July 1988.

[16] S. Lennart Johnsson and Ching-Tien Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, 38(9):1249–1268, September 1989.

[17] Jingke Li and Marina Chen. Compiling Communication-Efficient Programs for Masssively Parallell Machines. *IEEE Transactions on Parallel and Distributed System*, (3), July 1991.

[18] Jingke Li and Marina Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 1991.

[19] T. Schmiermund and S. R. Seidel. A Communication Model for the Intel iPSC/2. Technical report, Michigan Technological University, 1990.

[20] M.J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, 1989.

Appendix A: Algebraic Notation and Definition of Layout Operators

Let $D + c$ denote the interval domain $\texttt{interval}(\texttt{lb}(D) + c, \texttt{ub}(D) + c)$ and $aD + c$ denote the interval domain $\texttt{interval}(a * \texttt{lb}(D) + c, a * \texttt{ub}(D) + c)$. Let $U \cdot V$ denote the inner product of two vectors. The definitions of the algebraic notations for array operators and layout operators are given in Tables 10 to 15.

Operator	Domain	Codomain	Definition
EOSHIFT(c,b)	D	$D+c$	$(i) \mapsto (i+c)$
CSHIFT(c)	D	D	$(i) \mapsto \texttt{lb}(D) + (i - \texttt{lb}(D) + c) \bmod (\texttt{ub}(D) - \texttt{lb}(D) + 1)$
REFLECT	D	D	$(i) \mapsto \texttt{lb}(D) + \texttt{ub}(D) - i$
STRIDE(a,c)	D	$aD+c$	$(i) \mapsto (a*i+c)$

Table 10: Array Operators in Class ALIGN-I with Their Algebraic Notations and Definitions

Operator	Domain	Codomain	Definition
TRANS($[a_{ij}]$)	$D_1 \times \cdots \times D_n$	$D_{d_1} \times \cdots \times D_{d_n}$	$(i_1, \ldots, i_n) \mapsto ([a_{1j}] \cdot I, \ldots, [a_{nj}] \cdot I)$
SKEW($[a_{ij}]$)	$D_1 \times \cdots \times D_n$	$E_1 \times \cdots \times E_n$	$(i_1, \ldots, i_n) \mapsto ([a_{1j}] \cdot I, \ldots, [a_{nj}] \cdot I)$
CSKEW($[a_{ij}]$)	$D_1 \times \cdots \times D_n$	$D_1 \times \cdots \times D_n$	$(i_1, \ldots, i_n) \mapsto (([a_{1j}] \cdot I) \bmod m_1, \ldots, ([a_{nj}] \cdot I) \bmod m_n)$ where m_k are the sizes of D_k

Table 11: Array Operators in Class ALIGN-II with Their Algebraic Notations and Definitions

Operator	Domain	Codomain	Definition
RESHAPE($[m_i], [a_{ij}], [l_i], [b_{ij}]$)	$D_1 \times \ldots \times D_p$	$D_1 \times \ldots \times D_q$	$(i_1, \ldots, i_p) \mapsto (j_{d_1}, \ldots, j_{d_q})$, $(d_1, \ldots, d_q)$ is the permutation of $(1, \ldots, q)$ defined by matrix $[a_{ij}]$, $(h_1, \ldots, h_p)$ is the permutation of $(1, \ldots, p)$ defined by matrix $[b_{ij}]$, $j_k = (j \bmod r_{k-1}) \operatorname{div} r_k$, $r_k = m_{d_{k+1}} \times \ldots \times m_{d_q}$ $j = (s_1 i_{h_1} + \ldots + s_p i_{h_p})$ $s_k = l_{h_{k+1}} \times \ldots \times l_{h_p}$
SECTION(l, u, d)	D	E	$(i) \mapsto \begin{cases} l \le i \le u \text{ and } (i-l) \bmod d = 0 & \rightarrow i \\ \text{else} & \rightarrow \perp \end{cases}$
EMBED($[a_{ij}], E_1 \times \ldots \times E_m$)	$D_1 \times \ldots \times D_n$	$E_1 \times \ldots \times E_m$	$(i_1, \ldots, i_n) \mapsto ([a_{1j}] \cdot I, \ldots, [a_{mj}] \cdot I)$

Table 12: Array Operators in Class ALIGN-III with Their Algebraic Notations and Definitions

Operator	Domain	Codomain	Definition
SPREAD($[a_i], D_k$)	$D_1 \times \ldots D_{k-1}$ $\times D_{k+1} \ldots \times D_n$	$D_1 \times \ldots \times D_{k-1}$ $\times D_k \times D_{k+1} \ldots \times D_n$	$(i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_n) \mapsto$ $(i_1, \ldots, i_{k-1}, \mathrm{lb}(D_k) : \mathrm{ub}(D_k), i_{k+1}, \ldots, i_n)$ where $a_k = 1$
SUM($[a_i]$)	$D_1 \times \ldots \times D_n$	$D_1 \times \ldots \times D_{k-1}$	$(i_1, \ldots, i_n) \mapsto (i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_n)$ where $a_k = 1$

Table 13: Non-Align Array Operators with Their Algebraic Notations and Definitions

Operator	Domain	Codomain	Definition
BLOCK(b)	D	$L \times M$	$(i) \mapsto (i \text{ div } b, i \bmod b)$
CYCLIC(p)	D	$L \times M$	$(i) \mapsto ((i \bmod p, i \text{ div } p)$
BCYCLIC(b, p)	D	$L \times M$	$(i) \mapsto (i \text{ div } b) \bmod p, (i \text{ div } (p * b)) * b + i \bmod b)$
SEQ	D	$[0] \times D$	$(i) \mapsto (0, i)$

Table 14: Partition Operators with Their Algebraic Notations and Definitions

Operator	Domain	Codomain	Definition
BINARY	$[0 \ldots 2^n - 1]$	$H(n)$	$(i) \mapsto \begin{cases} i = 0 \rightarrow 0 \\ i = 1 \rightarrow 1 \\ else \rightarrow \text{BINARY}(i \text{ div } 2) \mid\mid (i \bmod 2) \end{cases}$
GRAY	$[0 \ldots 2^n - 1]$	$H(n)$	$(i) \mapsto b_n \mid\mid b_n \oplus b_{n-1} \mid\mid b_{n-1} \oplus b_{n-2} \mid\mid \ldots \mid\mid b_1 \oplus b_0$ where $(b_n, \ldots, b_0) = \text{BINARY}(i)$
RANDOM	$[0 \ldots 2^n - 1]$	$H(n)$	$(i) \mapsto \text{BINARY}(\text{randp}(i))$ where randp is a random permutation function

Table 15: Physical Map Operators and Their Definitions

Appendix B: Algebraic Rules

Data Motion Constructors

The five constructors $(\times, \circ, \dot{+}, \mathtt{bd}, +)$ are used internally for constructing the algebraic notations for complex array operators, layout operators and communication expressions.

Product "×" A *product operator* can be used to define array operators or layout operators for multidimensional arrays. The product of $f : D_1 \to E_1$ and $g : D_2 \to E_2$ is defined as $f \times g = \lambda(i,j) : D_1 \times D_2 \to E_1 \times E_2 \ . \ (f(i), g(j))$. The product of operators will be called a Multi-D operator. Some examples are given in Table 16.

Composition "∘" A *composition operator* can be used to define complex operators in the form of compositions of simpler operators. The composition of two functions $g : D_2 \to D_3$ and $f : D_1 \to D_2$ is defined as $g \circ f = \lambda(i) : D_1 \to D_3 . g(f(i))$. Some examples are given in Table 17.

Combination "$\dot{+}$" A *combination operator* is used to extract multiple[6] array references or operations in a given scope for later optimization. The combination of two operators $f : D \to E_1$ and $g : D \to E_2$ is defined as $f \dot{+} g = \lambda(i) : D \ . \ (f^{-1}(i); g^{-1}(i))$ where $(a; b)$ denotes a list of two elements a and b. Furthermore, the product of combination $(a; b) \times (c; d)$ is defined to be $(a \times c; b \times d)$. Some examples are given in Table 18.

Boundary "bd" An array operator α, when composed with a partition operator β, denoted as $\beta \circ \alpha \circ \beta^{-1}$ is called a *coarse-grained* array operator. A *boundary operator* bd extracts the data elements for interprocessor communication. Definition of bd is given in BD 1 to 2. The interaction of bd with the other constructors is shown in BD 3 to 5.

Compound Operator	Functionality
$(\mathtt{BLOCK}(b_1) \times \mathtt{BLOCK}(b_2))_{D_1 \times D_2 \to (L_1 \times M_1) \times (L_2 \times M_2)}$	2-D block partition
$(\mathtt{EOSHIFT}(c_1, b) \times \mathtt{EOSHIFT}(c_2, b))_{D_1 \times D_2 \to (D_1 + c_1) \times (D_2 + c_2)}$	2-D shifting
$(\mathtt{G} \times \mathtt{B})_{L_1 \times L_2 \to H(n_1) \times H(n_2)}$	compound encoding

Table 16: Examples of Multi-D Operators

Compound Operator	Resulting Alignment
$\mathtt{EOSHIFT}(c, b)_{D \to D+1} \circ \mathtt{REFLECT}_{D \to D}$	$(i) \mapsto (n - i + 1)$
$(\mathtt{EOSHIFT}(c_1, b) \times \mathtt{EOSHIFT}(c_2, b))_{D_2 \times D_1 \to (D_2 + c_1) \times (D_1 + c_2)} \circ \mathtt{TRANS} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}_{D_1 \times D_2 \to D_2 \times D_1}$	$(i,j) \mapsto (j + c_1, i + c_2)$
$(\mathtt{EOSHIFT}(c_1, b) \times \mathtt{EOSHIFT}(c_2, b))_{D_1 \times D_2 \to (D_1 + c_1) \times (D_2 + c_2)} \circ \mathtt{EMBED}(\begin{bmatrix} 1 \\ 0 \end{bmatrix}, D_1 \times D_2)_{D_1 \to D_1 \times D_2}$	$(i) \mapsto (i + c_1, c_2)$

Table 17: Examples of Compound Alignment Operators

In the following the algebraic rules are given. To avoid possible confusion, in some of the rules we will use the notation $g(\mathtt{parameters})_{domain \to codomain}$ for a domain morphism.

[6] Only combinable array operations will be extracted into the same communication expression. Two array operators are said to be combinable if (1) they are of the same kind (e.g., both are shifts), (2) they operate on the same array and there is no dependence betweem them, and (3) the defined arrays of them have consistent layout.

Compound Operator	Functionality
$\texttt{CSHIFT}(c_1)\dot{+}\texttt{EOSHIFT}(c_2, b)$	two instances of 1-D shifting
$(\texttt{CSHIFT}(c_1) \times \texttt{CSHIFT}(c_2))\dot{+}(\texttt{CSHIFT}(c_3) \times \texttt{CSHIFT}(c_4))$	two instances of 2-D shifting
$(\texttt{CSHIFT}(c_1) \circ \texttt{REFLECT})\dot{+}(\texttt{CSHIFT}(c_2) \circ \texttt{REFLECT})$	two reflection–followed–by–shifting

Table 18: Examples of Combinations of Multiple Array Operators

BD 1 Boundary for ALIGN-I Operators

Let M be the local index domain

(1) Let $\bar{\alpha}$ be an ALIGN-I operator, $\texttt{bd}(\texttt{SEQ} \circ \bar{\alpha} \circ \texttt{SEQ}^{-1}) = \{\}$

(2) Let $\bar{\alpha}$ be an ALIGN-I operator, $\texttt{bd}(\texttt{CYCLIC}(p) \circ \bar{\alpha} \circ \texttt{CYCLIC}(p)^{-1}) = M$

(3) $\texttt{bd}(\texttt{BLOCK}(b) \circ \bar{\alpha} \circ \texttt{BLOCK}(b)^{-1})$

$$= \left\{ \begin{array}{ll} \bar{\alpha} = \texttt{REFLECT} & \rightarrow M \\ (\bar{\alpha} = \texttt{CSHIFT}(c) \ or \ \texttt{EOSHIFT}(c)) \ and \ (c \geq 0) & \rightarrow \texttt{interval}(\texttt{ub}(M) - c, \texttt{ub}(M)) \\ (\bar{\alpha} = \texttt{CSHIFT}(c) \ or \ \texttt{EOSHIFT}(c)) \ and \ (c < 0) & \rightarrow \texttt{interval}(\texttt{lb}(M), \texttt{lb}(M) + c) \end{array} \right\}$$

(4) $\texttt{bd}(\texttt{BCYCLIC}(b, p) \circ \bar{\alpha} \circ \texttt{BCYCLIC}(b, p)^{-1})$

$$= \left\{ \begin{array}{ll} \bar{\alpha} = \texttt{REFLECT} & \rightarrow M \\ (\bar{\alpha} = \texttt{CSHIFT}(c) \ or \ \texttt{EOSHIFT}(c)) \ and \ (c \geq 0) & \rightarrow \{i \in M, ((i + c) \bmod b) < c\} \\ (\bar{\alpha} = \texttt{CSHIFT}(c) \ or \ \texttt{EOSHIFT}(c)) \ and \ (c < 0) & \rightarrow \{i \in M, ((i - c) \bmod b) \geq (b - c)\} \end{array} \right\}$$

BD 2 Boundary for Replication Operators and Reduction Operators

(1) $\texttt{bd}((\beta_1 \times \ldots \times \beta_n) \circ \texttt{SPREAD}([a_i], D_k) \circ (\beta_1 \times \ldots \times \beta_{k-1} \times \beta_{k+1} \times \ldots \times \beta_n)^{-1})$

$= M_1 \times \ldots \times M_{k-1} \times M_{k+1} \times \ldots \times M_n$

(2) $\texttt{bd}((\beta_1 \times \ldots \times \beta_{k-1} \times \beta_{k+1} \times \ldots \times \beta_n) \circ \texttt{SUM}([a_i]) \circ (\beta_1 \times \ldots \times \beta_n)^{-1})$

$= M_1 \times \ldots \times M_{k-1} \times M_{k+1} \times \ldots \times M_n$

BD 3 Boundary for Product-Compound Operators

$\texttt{bd}(f \times g) = \{(i, j) | i \in \texttt{bd}(f) \ or \ j \in \texttt{bd}(g)\}$

BD 4 Boundary for Composition-Compound Operators

$\texttt{bd} \ (f \circ g) = \{\texttt{bd}(f), \texttt{bd}(g)\}$, *where* $\{a, b\}$ *denotes a set containing* a *and* b.

BD 5 Boundary for Combination-Compound Operators

$\texttt{bd} \ (f\dot{+}g) = \texttt{bd}(f) + \texttt{bd}(g)$, where $+$ is defined as

$$a + b = \left\{ \begin{array}{ll} a \subset S \ and \ b \subset S & \rightarrow a \bigcup b \\ (a = \{a_1, \ldots, a_n\}, a_i \subset S) \ and \ (b = \{b_1, \ldots, b_n\}, b_i \subset S) & \\ & \rightarrow \{a_1 \bigcup b_1, \ldots, a_n \bigcup b_n\} \end{array} \right\}$$

where S *denotes the integer set.*

Table 19: Definition of Boundary Operator

Combining Rules

A combining rule simplifies a communication expression of the form $f \dot{+} g$ into h where data motion in h satisfies the data motions for both f and g. The theoretical foundation for the combining rules are unions of domain boundaries (subsets of Integer). The union of boundaries $\mathtt{bd}(f)$ and $\mathtt{bd}(g)$ are denoted by $\mathtt{bd}(f) + \mathtt{bd}(g)$.

Rule 1 Combining Multiple Shifts

Let $\beta = \mathtt{BLOCK}(b)$ or $\mathtt{BCYCLIC}(b,p)$, $\bar{\alpha}_1$ and $\bar{\alpha}_2$ be $\mathtt{EOSHIFT}$ or $\mathtt{CSHIFT}$, and $\mathtt{sign}(c_1) = \mathtt{sign}(c_2)$

(1) $\mathtt{bd}(\beta \circ \bar{\alpha}_1(c_1) \circ \beta^{-1}) + \mathtt{bd}(\beta \circ \bar{\alpha}_2(c_2) \circ \beta^{-1})$

$= \mathtt{bd}(\beta \circ \bar{\alpha}_3(\mathtt{sign}(c_1) * \mathtt{max}(\mathtt{abs}(c_1), \mathtt{abs}(c_2))) \circ \beta^{-1})$

(2) $(\beta \circ \bar{\alpha}_1(c_1) \circ \beta^{-1}) \dot{+} (\beta \circ \bar{\alpha}_2(c_2) \circ \beta^{-1})$

$= (\beta \circ \bar{\alpha}_3(\mathtt{sign}(c_1) * \mathtt{max}(\mathtt{abs}(c_1), \mathtt{abs}(c_2))) \circ \beta^{-1})$

where $\bar{\alpha}_3 = \mathtt{CSHIFT}$ if $\bar{\alpha}_1 = \mathtt{CSHIFT}$ or $\bar{\alpha}_2 = \mathtt{CSHIFT}$, otherwise $\bar{\alpha}_3 = \mathtt{EOSHIFT}$

Rule 2 Combining Multiple REFLECT

(1) $\mathtt{bd}(\beta \circ \mathtt{REFLECT} \circ \beta^{-1}) + \mathtt{bd}(\beta \circ \mathtt{REFLECT} \circ \beta^{-1}) = \mathtt{bd}(\beta \circ \mathtt{REFLECT} \circ \beta^{-1})$

(2) $(\beta \circ \mathtt{REFLECT} \circ \beta^{-1}) \dot{+} (\beta \circ \mathtt{REFLECT} \circ \beta^{-1}) = (\beta \circ \mathtt{REFLECT} \circ \beta^{-1})$

Rule 3 Combining Multiple STRIDE

Let $\beta = \mathtt{BLOCK}(b)$ or $\mathtt{BCYCLIC}(b,p)$, $\mathtt{sign}(c_1) = \mathtt{sign}(c_2)$, and $\mathtt{abs}(c_1 - c_2) \leq \mathtt{abs}(\frac{b}{a})$

(1) $\mathtt{bd}(\beta \circ \mathtt{STRIDE}(a, c_1) \circ \beta^{-1}) + \mathtt{bd}(\beta \circ \mathtt{STRIDE}(a, c_2) \circ \beta^{-1})$

$= \mathtt{bd}(\beta \circ \mathtt{STRIDE}(a, c_1) \circ \beta^{-1})$

(2) $(\beta \circ \mathtt{STRIDE}(a, c_1) \circ \beta^{-1}) \dot{+} (\beta \circ \mathtt{STRIDE}(a, c_2) \circ \beta^{-1})$

$= (\beta \circ \mathtt{STRIDE}(a, c_1) \circ \beta^{-1})$

Unification Rule for n-port Communications

Applying a sequence of the combining rule for shifting operators simplifies a communication expression to the following form:

$$\prod_{i=1}^{d} ((\beta_i \circ \bar{\alpha}(-c_{i_1}) \circ \beta_i^{-1}) \dot{+} (\beta_i \circ \bar{\alpha}(c_{i_2}) \circ \beta_i^{-1}))$$

where $\prod_{i=1}^{d}$ denotes the product of d terms. For the machines that support only one-port communications, the data motion implied in the form will be carried out one dimension at a time. For the parallel machines such as the Connection Machines that support n-port communications, each processor can exchange data with its neighbors concurrently. A particular implementation for concurrent exchange is to unify the shifting offsets at each dimension [4]. This is shown in the following unification rule.

Rule 4 Boundary Unification

Let $\bar{\alpha} = \mathtt{CSHIFT}$ or $\mathtt{EOSHIFT}$

(1) $((\mathtt{bd}(\beta \circ \bar{\alpha}(-c_{1_1}) \circ \beta^{-1}) + \mathtt{bd}(\beta \circ \bar{\alpha}(c_{1_2}) \circ \beta^{-1})) \times D_2)$

$+ (D_1 \times (\mathtt{bd}(\beta \circ \bar{\alpha}(-c_{2_1}) \circ \beta^{-1}) + \mathtt{bd}(\beta \circ \bar{\alpha}(c_{2_2}) \circ \beta^{-1})))$

$\subseteq \bigcup_{i=1}^{2} (D_1 \times \ldots \times D_{i-1} \times (\mathtt{bd}(\beta \circ \bar{\alpha}(-m) \circ \beta^{-1}) + \mathtt{bd}(\beta \circ \bar{\alpha}(m) \circ \beta^{-1})) \times D_{i+1} \times \ldots \times D_2)$

(2) $((\beta \circ \bar{\alpha}(-c_{1_1}) \circ \beta^{-1}) \dot{+} (\beta \circ \bar{\alpha}(c_{1_2}) \circ \beta^{-1})) \times ((\beta \circ \bar{\alpha}(-c_{2_1}) \circ \beta^{-1}) \dot{+} (\beta \circ \bar{\alpha}(c_{2_2}) \circ \beta^{-1}))$

$= \prod_{i=1}^{2} (\beta \circ \bar{\alpha}(-m) \circ \beta^{-1} \dot{+} (\beta \circ \bar{\alpha}(m) \circ \beta^{-1}))$

where $m = \mathtt{max}(c_{1_1}, c_{1_2}, c_{2_1}, c_{2_2})$

Aggregation Rule for Array Operators in Loops

The aggregation rule packs a large number of small messages into a smaller number of large messages.

Rule 5 Aggregation

Let $\bar{\alpha}$ = CSHIFT or EOSHIFT, β = BLOCK(b) or BCYCLIC(b, p), and b be divisible by c

$\beta \circ (\bar{\alpha}(c))^b) \circ \beta^{-1} = \beta \circ (\bar{\alpha}(b))^c) \circ \beta^{-1} = (\beta \circ \bar{\alpha}(b) \circ \beta^{-1})^c$

Rule for Exposing Coarse-Grained Communication

The result of applying Rule 5 is an expression in the form $(\beta \circ \bar{\alpha}(b) \circ \beta^{-1})^c$, which implies c shifts each with shifting offset b, the block size of BLOCK partition or the grain size of BCYCLIC partition. The following rule makes explicit such coarse-grained shifting. The resulting expression $(\bar{\alpha}(1) \times id)_{L \times M \to L \times M}$ implies each processor shift the whole block of data to its neighbor.

Rule 6 Coarse-Grained Communication

Let L be the logical processor domain and M the local index domain

Let $\bar{\alpha}$ = CSHIFT or EOSHIFT, and β = BLOCK(b) or BCYCLIC(b, p)

$\beta_{E \to L \times M} \circ \bar{\alpha}(b)_{D \to E} \circ \beta^{-1}_{L \times M \to D} = (\bar{\alpha}(1) \times id)_{L \times M \to L \times M}$

Composition of ALIGN-I Operators

Operators in this class are all linear functions with single variables. This property allows the inverse of an operator and the composition of two operators to be derived from linear algebra on the parameters ($y = x + c \Rightarrow x = y - c$, and $((x + a) + b) = (x + (a + b))$). For STRIDE alignment, we consider the image of the interval domain after alignment. The *exchange* rule says $g \circ f$ can be transformed to $f' \circ g'$, where f' and f are instances of the same operator except with different domains and codomains, and similarly for g and g'. The exchange rule also implies simplification of nonadjacent operators.

Rule 7 Inverse of ALIGN-I Operators

$\texttt{EOSHIFT}(c)^{-1}_{D_1 \to D_2} = \texttt{EOSHIFT}(-c)_{D_1 \to D_2}$

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

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

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

Rule 8 Reduction of Adjacent ALIGN-I Operators

$\texttt{EOSHIFT}(c_1)_{D_2 \to D_3} \circ \texttt{EOSHIFT}(c_2)_{D_1 \to D_2} = \texttt{EOSHIFT}(c_1 + c_2)_{D_1 \to D_3}$

$\texttt{CSHIFT}(c)_{D \to D} \circ \texttt{CSHIFT}(c_2)_{D \to D} = \texttt{CSHIFT}(c_1 + c_2)_{D \to D}$

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

$\texttt{STRIDE}(a_1, c_1)_{a_2 D + c_2 \to a_1 a_2 D + a_1 c_2 + c_1} \circ \texttt{STRIDE}(a_2, c_2)_{D \to a_2 D + c_2}$

$= \texttt{STRIDE}(a_1 a_2, a_1 c_2 + c_1)_{D \to a_1 a_2 D + a_1 c_2 + c_1}$

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

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

Rule 9 Exchange of ALIGN-I Operators

For all $\bar{\alpha}_{1(D_1 \to D_2)}$ and $\bar{\alpha}_{2(D_2 \to D_3)}$ in ALIGN-I

$\bar{\alpha}_{2(D_2 \to D_3)} \circ \bar{\alpha}_{1(D_1 \to D_2)} = \bar{\alpha}'_{1(E_1 \to D_3)} \circ \bar{\alpha}'_{2(D_1 \to E_1)}$

where $\bar{\alpha}'_1$ is an ALIGN-I operator that is of the same kind as α_1,

$\bar{\alpha}'_2$ is an ALIGN-I operator that is of the same kind as α_2

Rule 10 Reduction of Non-Adjacent ALIGN-I Operators

For all $\bar{\alpha}'_1, \bar{\alpha}_2$ and $\bar{\alpha}_1$ in ALIGN-I

$\bar{\alpha}'_{1(D_3 \to D_4)} \circ \bar{\alpha}_{2(D_2 \to D_3)} \circ \bar{\alpha}_{1(D_1 \to D_2)}$

$= \bar{\alpha}'_{2(E_1 \to D_4)} \circ \bar{\alpha}''_{1(D_1 \to E_1)}$

where $\bar{\alpha}_1, \bar{\alpha}'_1, \bar{\alpha}''_1$ are of the same kind and $\bar{\alpha}_2, \bar{\alpha}'_2$ are of the same kind

Composition of ALIGN-II Operators

Operators in this class are all linear functions with multiple variables. If the coefficient matrices $[a_{ij}]$ for the operators are nonsingular, the inverse of the operator can be derived by matrix inverse ($Y = AX \Rightarrow X = A^{-1}Y$). Furthermore, due to the linearity of **TRANS** and **SKEW** and the special algebraic property of the mod operation in **CSKEW**, composition of intercomponent alignment operators can be reduced to a single one with the new matrix derived from the multiplication of the matrices in the components (i.e. $A(BX) = (AB)X$).

Rule 11 Inverse of ALIGN-II Operators

For all $\tilde{\alpha}$ in ALIGN-II

$$\tilde{\alpha}([a_{ij}])^{-1}_{D\to E} = \tilde{\alpha}([a_{ij}]^{-1})_{D\to E}$$

Rule 12 Reduction of Adjacent ALIGN-II Operators

For all $\tilde{\alpha}$ in ALIGN-II

$$\tilde{\alpha}([a_{ij}])_{D_2\to D_3} \circ \tilde{\alpha}([b_{ij}])_{D_1\to D_2} = \tilde{\alpha}([a_{ij}] \times [b_{ij}])_{D_1\to D_3}$$

Rule 13 Left-Trans Reduction

For all $\tilde{\alpha}$ in ALIGN-II

$$\tilde{\alpha}([a_{ij}])_{D_2\to D_3} \circ \texttt{TRANS}([b_{ij}])_{D_1\to D_2} = \tilde{\alpha}([a_{ij}] \times [b_{ij}])_{D_1\to D_3}$$

Rule 14 Right-Trans Reduction

For all $\tilde{\alpha}$ in ALIGN-II

$$\texttt{TRANS}([a_{ij}])_{D_2\to D_3} \circ \tilde{\alpha}([b_{ij}])_{D_1\to D_2} = \tilde{\alpha}([a_{ij}] \times [b_{ij}])_{D_1\to D_3}$$

Composition of ALIGN-II Operators and Multi-D Operators

The exchange of **TRANS** and Multi-D operators states that applying a Multi-D operator and then a transposititon is equivalent to applying the transposition and then the permutation of the Multi-D operator on the permuted domain. Similarly, the linear property of ALIGN-I and ALIGN-II operators allows the exchange of **SKEW** and Multi-D **EOSHIFT** ($A(X + C) = AX + AC$, and $AX + C = A(X + A^{-1}C)$), the exchange of **SKEW** and Multi-D **REFLECT** ($A(N - X) = AN - AX$, and $N - AX = A(A^{-1}N - X)$), and the exchange of **CSKEW** and Multi-D **CSHIFT** ($A((X + C) \bmod N) \bmod N = ((AX \bmod N) + C) \bmod N$).

Rule 15 Exchange of ALIGN-II and Multi-D Operators

(1) Let $\kappa([d_i])_{D_1\times\cdots\times D_n\to E_1\times\cdots\times E_n} = (\kappa_{d_1} \times \cdots \times \kappa_{d_n})_{D_1\times\cdots\times D_n\to E_1\times\cdots\times E_n}$ where κ_i are 1-D operators

$$\texttt{TRANS}([a_{ij}])_{E_1\times\cdots\times E_n\to E_{d_1}\times\cdots\times E_{d_n}} \circ \kappa([d_i])_{D_1\times\cdots\times D_n\to E_1\times\cdots\times E_n}$$

$$= \kappa([a_{ij}] \times [d_i])_{D_{d_1}\times\cdots\times D_{d_n}\to E_{d_1}\times\cdots\times E_{d_n}} \circ \texttt{TRANS}([a_{ij}])_{D_1\times\cdots\times D_n\to D_{d_1}\times\cdots\times D_{d_n}}$$

(2) Let $\texttt{SH}([c_i])_{D_1\times\ldots\times D_n\to(D_1+c_1)\times\ldots\times(D_n+c_n)}$

$$= (\texttt{EOSHIFT}(c_1) \times \ldots \times \texttt{EOSHIFT}(c_n))_{D_1\times\ldots\times D_n\to(D_1+c_1)\times\ldots\times(D_n+c_n)}$$

$$\texttt{SKEW}([a_{ij}])_{(D_1+c_1)\times\ldots\times(D_n+c_n)\to E_1\times\ldots\times E_n} \circ \texttt{SH}([c_i])_{D_1\times\ldots\times D_n\to(D_1+c_1)\times\ldots\times(D_n+c_n)}$$

$$= \texttt{SH}([a_{ij}] \times [c_i])_{E'_1\times\ldots\times E'_n\to E_1\times\ldots\times E_n} \circ \texttt{SKEW}([a_{ij}])_{D_1\times\ldots\times D_n\to E'_1\times\ldots\times E'_n}$$

(3) Let $\texttt{REF}([n_i])_{D_1\times\ldots\times D_n\to D_1\times\ldots\times D_n}$

$$= (\texttt{REFLECT} \times \ldots \times \texttt{REFLECT})_{D_1\times\ldots\times D_n\to D_1\times\ldots\times D_n}$$

$$\texttt{SKEW}([a_{ij}])_{D_1\times\ldots\times D_n\to E_1\times\ldots\times E_n} \circ \texttt{REF}([n_i])_{D_1\times\ldots\times D_n\to D_1\times\ldots\times D_n}$$

$$= \texttt{REF}([a_{ij}] \times [n_i])_{E_1\times\ldots\times E_n\to E_1\times\ldots\times E_n} \circ \texttt{SKEW}([a_{ij}])_{D_1\times\ldots\times D_n\to E_1\times\ldots\times E_n}$$

(4) Let $\texttt{CSH}([c_i])_{D_1\times\ldots\times D_n\to D_1\times\ldots\times D_n}$

$$= (\texttt{CSHIFT}(c_1) \times \ldots \times \texttt{CSHIFT}(c_n))_{D_1\times\ldots\times D_n\to D_1\times\ldots\times D_n} \text{ where } |D_1| = |D_2| = \ldots = |D_n|$$

$$\texttt{CSKEW}([a_{ij}])_{D_1\times\ldots\times D_n\to D_1\times\ldots\times D_n} \circ \texttt{CSH}([c_i])_{D_1\times\ldots\times D_n\to D_1\times\ldots\times D_n}$$

$$= \texttt{CSH}([a_{ij}] \times [c_i])_{D_1\times\ldots\times D_n\to D_1\times\ldots\times D_n} \circ \texttt{CSKEW}([a_{ij}])_{D_1\times\ldots\times D_n\to D_1\times\ldots\times D_n}$$

Rule 16 Reduction of Non-Adjacent ALIGN-II Operators

(1) $\texttt{TRANS}([a_{ij}])_{E_{p_1}\times\cdots\times E_{p_n}\to E_{q_1}\times\cdots\times E_{q_n}} \circ \kappa([d_i])_{D_{p_1}\times\cdots\times D_{p_n}\to E_{p_1}\times\cdots\times E_{p_n}}$
$\circ\ \texttt{TRANS}([b_{ij}])_{D_1\times\cdots\times D_n\to D_{p_1}\times\cdots\times D_{p_n}}$
$= \kappa([a_{ij}]\times[d_i])_{D_{q_1}\times\cdots\times D_{q_n}\to E_{q_1}\times\cdots\times E_{q_n}} \circ \texttt{TRANS}([a_{ij}]\times[b_{ij}])_{D_1\times\cdots\times D_n\to D_{q_1}\times\cdots\times D_{q_n}}$

(2) $\texttt{SKEW}([a_{ij}])_{(E_1+c_1)\times\ldots\times(E_n+c_n)\to E'_1\times\ldots E'_n} \circ \texttt{SH}([c_i])_{E_1\times\cdots\times E_n\to(E_1+c_1)\times\ldots\times(E_n+c_n)}$
$\circ\ \texttt{SKEW}([b_{ij}])_{D_1\times\cdots\times D_n\to E_1\times\cdots\times E_n}$
$= \texttt{SH}([a_{ij}]\times[c_i])_{F_1\times\ldots\times F_n\to E'_1\times\ldots\times E'_n} \circ \texttt{SKEW}([a_{ij}]\times[b_{ij}])_{D_1\times\cdots\times D_n\to F_1\times\cdots\times F_n}$

(3) $\texttt{SKEW}([a_{ij}])_{E_1\times\ldots\times E_n\to E'_1\times\ldots E'_n} \circ \texttt{REF}([n_i])_{E_1\times\cdots\times E_n\to E_1\times\ldots\times E_n}$
$\circ\ \texttt{SKEW}([b_{ij}])_{D_1\times\cdots\times D_n\to E_1\times\cdots\times E_n}$
$= \texttt{REF}([a_{ij}]\times[n_i])_{E'_1\times\ldots\times E'_n\to E'_1\times\ldots\times E'_n} \circ \texttt{SKEW}([a_{ij}]\times[b_{ij}])_{D_1\times\cdots\times D_n\to E'_1\times\cdots\times E'_n}$

(4) $\texttt{CSKEW}([a_{ij}])_{D_1\times\ldots\times D_n\to D_1\times\ldots D_n} \circ \texttt{CSH}([c_i])_{D_1\times\cdots\times D_n\to D_1\times\ldots\times D_n}$
$\circ\ \texttt{CSKEW}([b_{ij}])_{D_1\times\cdots\times D_n\to D_1\times\cdots\times D_n}$
$= \texttt{CSH}([a_{ij}]\times[c_i])_{D_1\times\ldots\times D_n\to D_1\times\ldots\times D_n} \circ \texttt{CSKEW}([a_{ij}]\times[b_{ij}])_{D_1\times\cdots\times D_n\to D_1\times\cdots\times D_n}$

Composition of ALIGN-II Operators and ALIGN-III Operators

Rules 17 reduces the composition of ALIGN-II and ALIGN-III operators to an ALIGN-III operator, and Rule 18 implies the reduction of nonadjacent ALIGN-II operators.

Rule 17 Reduction of ALIGN-II and ALIGN-III

(1) $\texttt{TRANS}([a_{ij}])_{E_1\times\ldots\times E_k\to E_{d_1}\times\ldots\times E_{d_k}} \circ \texttt{EMBED}([b_{ij}], E_1\times\ldots\times E_k)_{D_1\times\ldots\times D_n\to E_1\times\ldots\times E_k}$
$= \texttt{EMBED}([a_{ij}]\times[b_{ij}], E_{d_1}\times\ldots\times E_{d_k})_{D_1\times\ldots\times D_n\to E_{d_1}\times\ldots\times E_{d_k}}$

(2) $\texttt{SKEW}([a_{ij}])_{E_1\times\ldots\times E_k\to E'_1\times\ldots\times E'_k} \circ \texttt{EMBED}([b_{ij}], E_1\times\ldots\times E_k)_{D_1\times\ldots\times D_n\to E_1\times\ldots\times E_k}$
$= \texttt{EMBED}([a_{ij}]\times[b_{ij}], E'_1\times\ldots\times E'_k)_{D_1\times\ldots\times D_n\to E'_1\times\ldots\times E'_k}$

(3) $\texttt{TRANS}([a_{ij}])_{E_1\times\ldots\times E_k\to E_{d_1}\times\ldots\times E_{d_k}} \circ \texttt{RESHAPE}([m_i],[b_{ij}],[l_i],[c_{ij}])_{D_1\times\ldots\times D_n\to E_1\times\ldots\times E_k}$
$= \texttt{RESHAPE}([m_i],[b_{ij}],[a_{ij}]\times[l_i],[a_{ij}]\times[c_{ij}])_{D_1\times\ldots\times D_n\to D_{d_1}\times\ldots\times D_{d_k}}$

(4) $\texttt{RESHAPE}([m_i],[b_{ij}],[l_i],[c_{ij}])_{D_{d_1}\times\ldots\times D_{d_n}\to E_1\times\ldots\times E_k} \circ \texttt{TRANS}([a_{ij}])_{D_1\times\ldots\times D_n\to D_{d_1}\times\ldots\times D_{d_n}}$
$= \texttt{RESHAPE}([a_{ij}]\times[m_i],[a_{ij}]\times[b_{ij}],[l_i],[c_{ij}])_{D_1\times\ldots\times D_n\to E_1\times\ldots\times E_k}$

Rule 18 Exchange of ALIGN-II and ALIGN-III

Let $\texttt{SEC}([l_i],[u_i],[s_i]) = \texttt{SECTION}(l_1,u_1,s_1)\times\ldots\times\texttt{SECTION}(l_n,u_n,s_n)$

$\texttt{TRANS}([a_{ij}])_{E_1\times\ldots\times E_n\to E_{d_1}\times\ldots\times E_{d_n}} \circ \texttt{SEC}([l_i],[u_i],[s_i])_{D_1\times\ldots\times D_n\to E_1\times\ldots\times E_n}$
$= \texttt{SEC}([a_{ij}]\times[l_i],[a_{ij}]\times[u_i],[a_{ij}]\times[s_{ij}])_{D_{d_1}\times\ldots\times D_{d_n}\to E_{d_1}\times\ldots\times E_{d_n}} \circ \texttt{TRANS}([a_{ij}])_{D_1\times\ldots\times D_n\to D_{d_1}\times\ldots\times D_{d_n}}$

Expansion Rules for Exposing Communication Idioms

The expansion rules are used to expose communication idioms.

Rule 19 Composition-Product Exchange

For all $\bar{\alpha}_1,\bar{\alpha}_2,\bar{\alpha}_3,\bar{\alpha}_4$ in ALIGN-I
where $\bar{\alpha}_1$ and $\bar{\alpha}_3$ are of the same kind or $\bar{\alpha}_2$ and $\bar{\alpha}_4$are of the same kind

$(\bar{\alpha}_1\circ\bar{\alpha}_2)\times(\bar{\alpha}_3\circ\bar{\alpha}_4)$
$= (\bar{\alpha}_1\times\bar{\alpha}_3)\circ(\bar{\alpha}_2\times\bar{\alpha}_4)$

Rule 20 Partition Expansion for Multi-D ALIGN-I

For all $\bar{\alpha}_1,\ldots,\bar{\alpha}_d$ in ALIGN-I and of the same kind

$\Pi^d(\gamma\circ\beta)\circ((\ldots\times\ldots)\circ)^*\ (\bar{\alpha}_1\times\ldots\times\bar{\alpha}_d)\circ\Pi^d(\beta^{-1}\circ\gamma^{-1})$
$= (\Pi^d(\gamma\circ\beta)\circ((\ldots\times\ldots)\circ)^*\circ\Pi^d(\beta^{-1}\circ\gamma^{-1}))\circ(\Pi^d(\gamma\circ\beta)\circ(\bar{\alpha}_1\times\ldots\times\bar{\alpha}_d)\circ\Pi^d(\beta^{-1}\circ\gamma^{-1}))$

Rule 21 Partition Expansion for 1-D ALIGN-I

Let $\bar{\alpha}_1$ and $\bar{\alpha}_2$ be in ALIGN-I

$\gamma \circ \beta \circ (\bar{\alpha}_1 \circ \bar{\alpha}_d) \circ \beta^{-1} \circ \gamma^{-1}$

$= (\gamma \circ \beta \circ \bar{\alpha}_1 \circ \beta^{-1} \circ \gamma^{-1}) \circ (\gamma \circ \beta \circ \bar{\alpha}_2 \circ \beta^{-1} \circ \gamma^{-1})$

Appendix C: Sample Derivation of Layout Conversion and Data Motion for the Least-Square Approximation

Derivation 1 Boundary Extraction for Fine-Grain Array A_TEMP

ω_a is the boundary expression for the fine-grain shifting, and $M_1 \times M_2$ the local index domain

$\omega_a = \mathtt{bd}(id \times (\mathtt{BLOCK}(b_2) \circ \mathtt{CSHIFT}(1) \circ \mathtt{BLOCK}(b_2)^{-1}))$

(by Prop 8 Distribute bd over Product)

$= M_1 \times \mathtt{bd}(\mathtt{BLOCK}(b_2) \circ \mathtt{CSHIFT}(1) \circ \mathtt{BLOCK}(b_2)^{-1})$

(by BD 1 Boundary for ALIGN-I)

$= M_1 \times \mathtt{interval}(\mathtt{ub}(M_2) - 1, \mathtt{ub}(M_2))$

Derivation 2 Intra-Block Data Motion for Coarse-Grain Array A_TEMP

θ_a is the communication expression extracted from line 14 of subroutine mm

$\theta_a = (\mathtt{BLOCK}(b_1) \times \mathtt{BLOCK}(b_2)) \circ ((id \times \mathtt{CSHIFT}(1))^{b_2})^{p_2} \circ (\mathtt{BLOCK}(b_1) \times \mathtt{BLOCK}(b_2))^{-1}$

(by Rule ?? Partition Expansion)

$= (\mathtt{BLOCK}(b_1) \times \mathtt{BLOCK}(b_2)) \circ (id \times \mathtt{CSHIFT}(1))^{b_2} \circ ((\mathtt{BLOCK}(b_1) \times \mathtt{BLOCK}(b_2))^{-1})^{p_2}$

(by Prop 5 Product-Composition Exchange)

$= ((\mathtt{BLOCK}(b_1) \circ id \circ \mathtt{BLOCK}(b_1)^{-1}) \times (\mathtt{BLOCK}(b_2) \circ (\mathtt{CSHIFT}(1))^{b_2} \circ \mathtt{BLOCK}(b_2)^{-1}))^{p_2}$

(by Prop 1 Inverse)

$= (id \times (\mathtt{BLOCK}(b_2) \circ (\mathtt{CSHIFT}(1))^{b_2} \circ \mathtt{BLOCK}(b_2)^{-1})^{p_2})$

(by Rule 5 Aggregate Communication)

$= (id \times (\mathtt{BLOCK}(b_2) \circ (\mathtt{CSHIFT}(b_2)) \circ \mathtt{BLOCK}(b_2)^{-1})^{p_2})$

(by Rule 6 Coarse-Grain Communication)

$= (id \times (\mathtt{CSHIFT}(1) \times id)^{p_2})$

Derivation 3 Boundary Extraction for Data Motion of Coarse-Grain Array A_TEMP

ω_a is the boundary expression for the coarse-grain shifting, and $M_1 \times M_2$ the local index domain

$\omega_a = \mathtt{bd}(id \times (\mathtt{BLOCK}(b_2) \circ \mathtt{CSHIFT}(b_2) \circ \mathtt{BLOCK}(b_2)^{-1}))$

$= \{\} + (M_1 \times M_2)$ (by Prop 8 Distribute bd over Product)

$= M_1 \times M_2$ (by definition of +)

Derivation 4 Boundary Extraction for Computation of Coarse-Grain array A_TEMP

ρ_a is the boundary expression for computation

$\rho_a = \mathtt{bd}(id \times (\mathtt{BLOCK}(b_2) \circ (\mathtt{CSHIFT}(1))^{b_2} \circ \mathtt{BLOCK}(b_2)^{-1}))$

(by Prop 8 Distribute bd over Product)

$= M_1 \times \mathtt{bd}(\mathtt{BLOCK}(b_2) \circ (\mathtt{CSHIFT}(1))^{b_2} \circ \mathtt{BLOCK}(b_2)^{-1})$

(by BD 4 Boundary for Composition)

$= M_1 \times \{\mathtt{bd}(\mathtt{BLOCK}(b_2) \circ \mathtt{CSHIFT}(b_2) \circ \mathtt{BLOCK}(b_2)^{-1}), \ldots, \mathtt{bd}(\mathtt{BLOCK}(b_2) \circ \mathtt{CSHIFT}(1) \circ \mathtt{BLOCK}(b_2)^{-1})\}$

(by definition of cross product in mathematics)

$= \{M_1 \times \mathtt{bd}(\mathtt{BLOCK}(b_2) \circ \mathtt{CSHIFT}(b_2) \circ \mathtt{BLOCK}(b_2)^{-1}), \ldots, M_1 \times \mathtt{bd}(\circ\mathtt{BLOCK}(b_2) \circ \mathtt{CSHIFT}(1) \circ \mathtt{BLOCK}(b_2)^{-1})\}$

Figure 9: Derivations for Matrix Multiplication with Different Schemes of Shifting

The first set of derivation shown in Figure 9 is for optimizing the data motion for array A_TEMP inside subroutine mm. Derivation 1 extracts the boundary (the rightmost column) for fine-grained cyclic shift,

without aggregating communications. Derivation 2 aggregates the fine-grained shifts into coarse-grained one. The expression (CSHIFT(1) $\times$ id) in θ_a matches with the idiom for coarse-grained cyclic shift, and thus a call to the communication routine **coarse-align** will be generated and inserted outside the inner loop and the message received will be stored in a two-dimensional temporary array. Derivation 3 decides the amount of communications, and Derivation 4 extracts the boundaries for the computation in each iteration of the inner loop (ρ_1 for the first iteration, ..., ρ_{b_2} for the last iteration).

The communication derivation and boundary derivation for array **B_TEMP**, in line 11 of subroutine **mm**, is similar.

Derivation 5 Boundary Extraction for Arrays b and C_TEMP in Subroutine mvm

The boundary expressions are extracted from line 9 and 10 of subroutine mvm

LetM_2 be the local index domain of array b, $M_1 \times M_2$be the local index domain of array C_TEMP

Array b : (by BD 2: Boundary for SPREAD)

$$\mathrm{bd}((\mathrm{BLOCK}(b_1) \times \mathrm{BLOCK}(b_2)) \circ \mathrm{SPREAD}([1,0],[1..m]) \circ (\mathrm{BLOCK}(b_1) \times \mathrm{BLOCK}(b_2))^{-1}) = M_2$$

Array C_TEMP : (by BD 2: Boundary for SUM)

$$\mathrm{bd}((\mathrm{BLOCK}(b_1) \times \mathrm{BLOCK}(b_2)) \circ \mathrm{SUM}([0,1]) \circ (\mathrm{BLOCK}(b_1) \times \mathrm{BLOCK}(b_2))^{-1}) = M_1$$

Figure 10: Derivations for Matrix-Vector Multiplication

Figure 10 shows the boundary extraction for array b and array C_TEMP inside subroutine **mvm**. The boundary derivation for the coarse-grained spread enables each processor to store only a one-dimensional temporary array instead of a two-dimensional temporary array. The boundary derivation for the reduction enables each processor to perform sequential reduction locally and then parallel reduction among processors, with each processor communicating by a vector of partial results. Boundary derivation for array C in subroutine **inverse** is similar.

The last set of derivations shown in Figure 11 is for layout conversion of arrays B, C and D. The normal form of θ_B (Derivation 6) matches with the idiom for cyclic skewing. θ_{C_1} matches with the idiom for change of partition. The normal form of θ_{C_2} (Derivation 7) matches with the idiom for change of partition and the idiom for cyclic skewing. θ_D is reduced to an identity function (Derivation 8), which implies no data motion is required.

Derivation 6 Conversion Derivation for Array B

$$\theta_B = (\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2)) \circ \texttt{CSKEW}\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \circ \texttt{CSKEW}\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}^{-1} \circ (\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2))^{-1}$$

(by Rule 11 Inverse of ALIGN-II)

$$= (\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2)) \circ \texttt{CSKEW}\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \circ \texttt{CSKEW}\begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \circ (\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2))^{-1}$$

(by Rule 12 Reduction of ALIGN-II)

$$= (\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2)) \circ \texttt{CSKEW}\begin{pmatrix} 0 & 1 \\ -1 & 1 \end{pmatrix} \circ (\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2))^{-1}$$

Derivation 7 Conversion Derivation for Array C

θ_{C_1} is the communication expression for layout conversion of C between first `mm` and `inverse`

θ_{C_2} is the communication expression for layout conversion of C between `inverse` and second `mm`

$$\theta_{C_1} = (\texttt{SEQ} \times \texttt{CYCLIC}(p)) \circ (\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2))^{-1}$$

$$\theta_{C_2} = (\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2)) \circ \circ\texttt{CSKEW}\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \circ (\texttt{SEQ} \times \texttt{CYCLIC}(p))^{-1}$$

(by Rule ?? Partition Expansion)

$$= ((\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2)) \circ \texttt{CSKEW}\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \circ (\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2))^{-1})$$

$$\circ ((\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2)) \circ (\texttt{SEQ} \times \texttt{CYCLIC}(p_2))^{-1})$$

Derivation 8 Conversion Derivation for Array D

$$\theta_D = (\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2)) \circ (\texttt{BLOCK}(b_1) \times \texttt{BLOCK}(b_2))^{-1}$$

$$= (\texttt{BLOCK}(b_1) \circ \texttt{BLOCK}(b_1)^{-1}) \times (\texttt{BLOCK}(b_2) \circ \texttt{BLOCK}(b_2)^{-1}) \quad \text{(by Prop 5 Product-Composition Exchange)}$$

$$= id_{(H(n_1) \times H(n_2)) \times (L_1 \times L_2)} \quad \text{(by Prop 1 Inverse)}$$

Figure 11: Derivations for Layout Conversion