Yale University
Department of Computer Science

Synthesis of Explicit Communication from Shared-Memory Program References

Jingke Li and Marina Chen

YALEU/DCS/TR-755
May 1990

This work has been supported in part by the Office of Naval Research under Contract N00014-86-K-0310, N00014-86-K-0564 and N00014-89-J-1906.

Synthesis of Explicit Communication from Shared-Memory Program References

Jingke Li Marina Chen

Department of Computer Science
Yale University
P.O. Box 2158, Yale Station
New Haven, CT 06520
Tel. (203) 432-4099
li-jingke@cs.yale.edu chen-marina@cs.yale.edu

May 1990

Abstract

This paper addresses the problem of data distribution and communication synthesis in generating parallel programs targeted for massively parallel, distributed-memory machines from sequential programs, functional programs, or parallel programs based on a shared-memory model. We present a novel compilation technique for synthesizing explicit communication commands. Our major contributions to the problem of generating communication are: (1) the idea of analyzing source program references and matching these syntactic patterns with aggregate communication routines which can be implemented efficiently on the target machine, (2) the notion of communication metric and the optimizations performed to reduce communication overhead, and (3) the development of a target program style for which the compiler-generated communication are provably deadlock free.

Key Words

Parallelizing compiler, inter-processor communication, parallel architecture, distributed memory, shared memory, reference pattern, data distribution, scheduling, synchronization.

Contents

1 Introduction

Distributed memory machines are playing an increasingly more important role in high performance computation as parallelism is exploited at an increasingly larger scale. By now it is widely recognized that directly programming this class of machines using explicit communication commands is not the way to go: it requires explicit control and microscopic management of parallel resources; it is tedious, error-prone, and often unwieldy for producing and maintaining efficient application code.

The Crystal approach to this problem is to begin with a machine-independent, high-level problem specification. A sequence of transformations, either suggested by the programmer or generated by the compiler, are then applied to this specification. These transformations are tuned for each particular machine architecture so that efficient target code with explicit communication can be generated.

To seek a balance between the automatable and the effort required of the programmer, Crystal provides the programmer with language constructs for specifying high-level algorithmic strategies and a suitably abstract model of the target architecture. We believe that such a balanced approach will allow the programmer to explore and devise suitable mapping and communication strategies that take advantage of the global characteristic of the underlying machine without dealing with the error-prone micro-management of resource allocation and synchronization.

Our approach to automation consists of the following components:

Control Structure Synthesis: One major task of a parallelizing compiler is to derive a parallel control structure from a functional specification or a sequential program. After the control structure is determined, the source program is transformed into an intermediate program in which parallel schedule and flow of control are made explicit. One can think of this intermediate program as a parallel program for multiple processors with a global shared memory. What remains to be done is to distribute data and generate appropriate communication.

Data Distribution: We distribute data over the fragmented memory in two stages, first mapping the program data structure to a virtual network and then embedding the virtual network into the physical network. Specifically, we consider virtual networks which are multi-dimensional grids and use the standard Gray code embedding of a grid into a hypercube.

The mapping from data structures to the virtual network consists of (1) *partitioning* the program data structures into appropriate grain sizes in such a way that communication overhead is reduced and workload is balanced, and (2) determining the relative locations of data structures so as to minimize inter-processor communication (we call this process *domain alignment*).

Communication Synthesis: Finally, all references to data structures must be translated to local memory accesses or inter-processor communication. The reference patterns of the intermediate program are then matched with a library of aggregate communication routines and those which minimize network congestion and overhead are chosen. The cost of inter-processor communication is modeled by a communication metric on which optimizations are based.

This paper addresses the issues involved in generating explicit communication commands. Although the techniques described here are developed within the context of Crystal (a functional language) [5], they can be applied to Fortran-based parallelizing systems [1, 2, 13, 16] as well.

Issues in Synthesizing Communication We now discuss the issues involved in establishing correct and efficient communication on a distributed-memory machine.

The primary issue is the intertwined relationship between data distribution and communication cost, since the locations of data determine the source, the destination and the pattern of a communication, or whether a communication is necessary at all.

At any given time during execution, the *global message pattern* affects the communication cost due to possible collisions of messages in the network. Note that given a fixed data distribution, there are still many possible choices of communication patterns which result in different communication costs.

Because of the fixed overhead incurred by sending a message, *message granularity* is an important factor in determining the cost of communication. Aggregating a collection of small messages into a single communication must be considered if the overhead per message is high.

On MIMD distributed-memory machines, *synchronization* between processors is achieved via matched "send" and "receive" pairs. The generation of communication commands must ensure that no deadlock is introduced.

Finally, the trade-off between efficient space consumption, data distribution, and message granularity must be carefully considered because they are often conflicting demands.

An Example We now illustrate the main ideas of our synthesis approach with a simple example. The following is a program segment (written in a pseudo, parallel C notation which will be described in more detail in Section 2):

$$
\begin{array}{l}
\textsf{for } (t : [0..n]) \\
\quad \textsf{forall } (i : [1..n], j : [1..n]) \\
\quad\quad a[i][j][t] = b[i][3][t] + a[i][j][t-1];
\end{array}
$$

The interpretation of the **for** loop is that the iterations are executed in sequential order, while that of a **forall** loop is that the iterations can be executed in parallel. We use the notation $[lb..ub]$ to denote the range of a loop index, with lb and ub being its lower and upper bounds, respectively.

Suppose that the iterations of the above nested **forall** loops are assigned to different processors. Denote each processor by a pair (x, y), and let processor (x, y) be responsible for a range of iterations specified by the intervals $[I_l(x, y)..I_u(x, y)]$ and $[J_l(x, y)..J_u(x, y)]$. The mappings from indices to processor ids are simple functions and can be computed on every processor. We use $\textsf{idx\_to\_pid}(i_1, \ldots, i_n)$ to denote the processor id which corresponds to index tuple $(i_1, \ldots, i_n)$.

A straightforward approach is to generate a communication for each instance of an array reference with an explicit communication statement:

$$
\begin{array}{l}
\text{Program for processor } (x, y) : \\
\textsf{for } (t = 0; t <= n; t++) \\
\quad \textsf{for } (i = I_l(x, y); i < I_u(x, y); i++) \\
\quad\quad \textsf{for } (j = J_l(x, y); j < J_u(x, y); j++) \ \{ \\
\quad\quad\quad \textsf{if } (\textsf{idx\_to\_pid}(i, 3, t) \neq (x, y)) \\
\quad\quad\quad\quad \langle \text{get } b[i][3][t] \text{ from processor } \textsf{idx\_to\_pid}(i, 3, t) \rangle; \\
\quad\quad\quad a[i][j][t] = b[i][3][t] + a[i][j][t-1]; \}
\end{array}
$$

The added statement checks, for each index pair (i, j), if the referenced data $b[i][3][t]$ happens to be stored in the local memory of the same processor where computation associated with the index pair (i, j) is assigned. If not, a two-way communication consisting of sending a request and receiving the corresponding answer takes place.

This straightforward approach would generate correct, but quite inefficient communication because the test is done for each reference, resulting in high runtime overhead. In addition, the messages generated are fine-grained and they incur high startup overhead.

The approach we are about to present is based on matching program references with communication primitives. In this approach, messages are aggregated and organized globally. For this example, a collection of *broadcast* communication will be issued for all instances of the reference as defined in the broadcast statement below:

```
Program for processor (x, y) :
for (t = 0; t <= n; t++) {
    Column_Broadcast(idx_to_pid(*, 3, t), b[*][3][t]);
    for (i = I_l(x, y); i < I_u(x, y); i++)
        for (j = J_l(x, y); j < J_u(x, y); j++)
            a[i][j][t] = b[i][3][t] + a[i][j][t - 1];
    }
```

The broadcast routine spreads the third column of array b, i.e. $b[i][3][t]$, $1 \leq i \leq n$ to all other columns. More specifically, broadcasting takes place concurrently and independently within all the rows of the processor network, and the processors on which segments of that column reside spread the segments to other processors in same row. Each processor will receive a segment of the column $b[i][3][t]$, $I_l(x, y) \leq i \leq I_u(x, y)$ in a single message.

Related Work The problem of automatically generating communication for distributed-memory machines from program references is addressed by several research projects. In Callahan and Kennedy's system [4], a target program with explicit communication is generated from a parallelized Fortran program along with user directives that specify how arrays are distributed over processors. The *DINO* project at Colorado University [15] requires the user to provide information regarding both data distribution and communication. The user directives are given at a level that is machine independent. Other systems for generating communication based on user directives or annotations include the Kali project at Purdue University [9, 10], the Superb project at Bonn University [17], and the work of Ramanujan and Sadayanppan [14].

Our major contributions to the problem of generating communication are: (1) the idea of analyzing source program references and matching these syntactic patterns with aggregate communication routines which can be implemented efficiently on the target machine, (2) the notion of communication metric and the optimizations performed to reduce communication overhead, and (3) the development of a target program style for which the compiler-generated communication are provably deadlock free.

Organization of the Paper The remaining part of this paper is organized as follows. In Section 2, we give an overview of our approach to communication synthesis. We define precisely the form of input programs to the communication synthesis module. We then define the notion

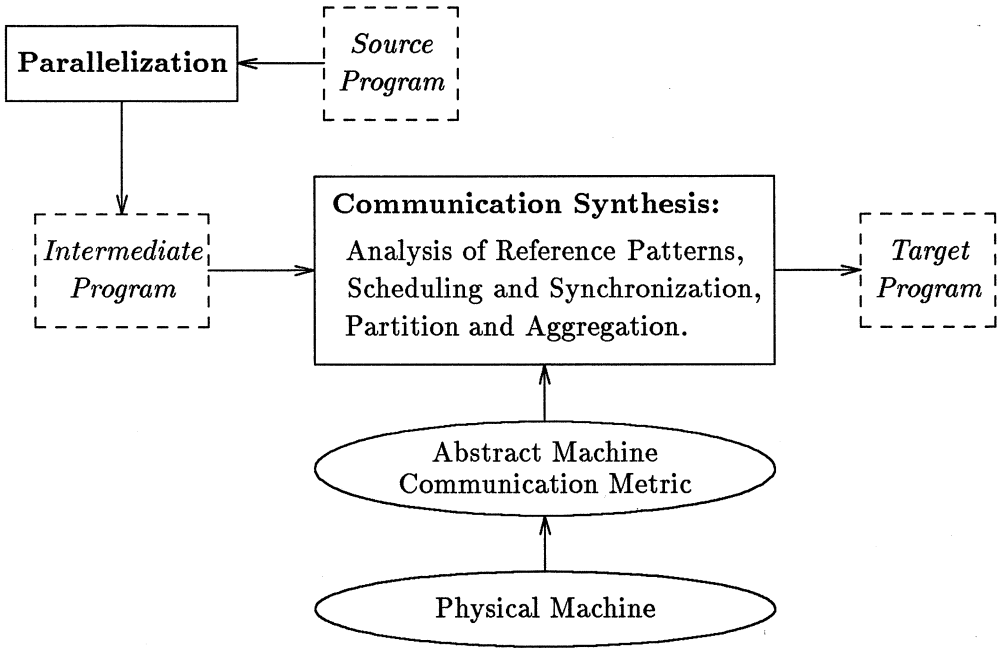


Figure 1: A High Level View of the Communication Synthesis Module

of communication metric and describe an abstract machine which models the class of physical machine under consideration. We then describe the *standard* partition strategies we use to distribute data over processors and their effects on the generation of communication. In Section 3, the algorithm for matching reference patterns with communication routines is given. Synchronization and correctness of the generated communication are discussed in Section 4. Finally, we summarize our approach in Section 5.

2 Our Approach to Communication Synthesis

The role of the communication synthesis module in a parallelizing compiler (e.g. the Crystal compiler [11]) is shown in Fig 1. The first part of the compiler consists of the *parallelization* module, which handles dependence analysis and the generation of parallel control structures. A source program is transformed into a *shared-memory program* containing explicit control structures, but no distributed data, nor explicit communication statements. The second part of the compiler consists of the *communication synthesis* module, which handles the generation of communication commands and data partition and distribution. It transforms a shared-memory program into a target program. An *abstract machine*, with a communication metric which reflects the architecture and communication cost of the target machine, is used to guide the optimizations.

We generate communication statements in three major steps. The first step is to analyze reference patterns of the input program and match them with efficient communication routines. The second step deals with scheduling and synchronization of **send** and **receive** pairs, as well as synchronization of global aggregate communication routines. The third step handles the problem of partitioning the index domain of the input program over the target processors. An approach for selecting a partition from a small set of standard strategies is described.

```
for (t : [0..n]) {
    forall (i : [1..n], j : [1..n])
        b(i,j,t) = if (j = t)  → \ + {a(i,x,t-1) | 1 ≤ x ≤ n};
                   else  → 1;
    forall (i : [1..n], j : [1..n])
        a(i,j,t) = if (t = 0)  → 0;
                   else if (i = b(0,t,t))  → a(t,j,t-1);
                   else  → b(i,t,t);}
forpipe (i : [1..n])
    forpipe (j : [1..n])
        c(i,j) = b(i-1,j,n) + b(i,j-1,n);
```

Figure 2: A Shared-Memory Program

2.1 Shared-Memory Programs

The input program to the communication synthesis module can be written in any parallel shared-memory language or sequential language augmented with parallel control structures. In this paper, we use a C-like notation augmented with parallel control structures (to be described below).

Parallel Control Structure Three types of loops are used: **for**, **forall**, and **forpipe**. A **for** loop is just a sequential loop in which iterations are executed in sequential order. In a **forall** loop, all iterations can be executed in parallel, i.e. there is no data dependence between the iterations. A **forpipe** loop is a special type of sequential loop, where data dependence is only between adjacent iterations. The implementation of a single **forpipe** loop is really no different from that of a **for** loop. But if there are two nested **forpipe** loops, the loop iterations can be executed in a pipelined fashion across multiple processors as described in the following example.

Figure 2 shows a shared-memory program with two separate loop structures. The first one consists of nested **for** and **forall** loops, and the second consists of nested **forpipe** loops. The execution order of the loop iterations of the nested **forpipe** loops can be described by a sequence of sets of loop index pairs: $\{(1,1)\}$, $\{(1,2),(2,1)\}$, $\{(1,3),(2,2),(3,1)\}$, ..., $\{(i,j) \mid i+j = n+1\}$, where computations associated with the index pairs belonging to the same set can be executed in parallel.

For and **forall** loops correspond to **do** and **doall** loops in parallel Fortran programs [16], while **forpipe** is somewhat similar to **doacross**.

Assumptions on the Form of Shared-Memory Programs For the sake of simplicity, we assume that a shared-memory program is preprocessed and transformed into the following restricted form:

Single assignment: Each array element can be assigned to only once. However, an array can appear on the left-hand side of many assignment statements (so long as different array elements are assigned to each time).

Left-hand side array indices must be formals: An array index expression on the left-hand side of an assignment statement must be a formal parameter. For instance, the following statement

$$a(i, j-1) = b(i+2, j)$$

should be written as

$$a(i, j) = b(i+2, j+1).$$

Arrays are aligned: Arrays appearing in the same loop structure (called a *π-block* in the parallelizing Fortran literature [3]) are aligned, i.e. the relative locations of these arrays are fixed. The alignment can either be specified by the user or be generated automatically by an alignment algorithm [12].

Index Domains For each program loop structure, all arrays are aligned and therefore all can be thought of as defined over the same range of indices, where the boundaries of arrays are appropriately adjusted according to the alignment. Such a range of indices is called an *index domain.* We restrict ourselves to index domains which are Cartesian products of the interval domains.

In the example shown in Figure 2, the index domain of the first loop structure is the Cartesian product of intervals $[1..n] \times [1..n] \times [0..n]$; and that of the second loop structure is $[1..n] \times [1..n]$.

Reference Patterns In the following, scalar expansion (c.f. [16]) is assumed to have been done. Those scalars which are not expanded to arrays will not be considered here, since they will be mapped to local variables on each processor and will not directly affect inter-processor communication. In other words, we will focus on references between arrays.

For each pair of array references appearing on the two sides of an assignment statement in a loop,

$$\begin{array}{l} \textsf{for } (i_1 : D_1, \ldots, i_n : D_n) \\ \quad a(i_1, \ldots, i_n) = \textsf{if } \gamma \ \rightarrow \cdots b(\tau_1, \ldots, \tau_n) \cdots; \\ \qquad\qquad\qquad\quad \textsf{else } \rightarrow \cdots; \end{array}$$

the symbolic form (as a quoted string of characters)

$$\ulcorner a(i_1, \ldots, i_n) \leftarrow b(\tau_1, \ldots, \tau_n) : \gamma \urcorner$$

is called a *reference pattern*, where the formals $(i_1, \ldots, i_n)$ are quantified over the index domain $D_1 \times \cdots \times D_n$, and γ is the guard of the conditional branch that $b(\tau_1, \ldots, \tau_n)$ is in.

Note that a reference pattern represents a collection of data dependencies. We emphasize this aggregate form rather than each instance of a reference because data dependencies between elements of index domains are sources of communication and they need to be aggregated for performance reasons.

Example From the program in Figure 2, we can derive the following reference patterns:

$$\ulcorner b(i, j, t) \leftarrow a(i, x, t-1) : j = t \textsf{ and } 1 \leq x \leq n \urcorner,$$
$$\ulcorner a(i, j, t) \leftarrow b(0, t, t) : t \neq 0 \urcorner,$$

$$\ulcorner a(i,j,t) \leftarrow a(t,j,t-1) : t \neq 0 \text{ and } i = b(0,t,t) \urcorner,$$
$$\ulcorner a(i,j,t) \leftarrow b(i,t,t) : t \neq 0 \text{ and } i \neq b(0,t,t) \urcorner,$$
$$\ulcorner c(i,j) \leftarrow b(i-1,j,n) \urcorner,$$
$$\ulcorner c(i,j) \leftarrow b(i,j-1,n) \urcorner.$$

Spatial Reference Patterns When a loop with nested levels is mapped to the target machine, some levels of the loop will be mapped over different processors while others will be mapped to a sequential loop to be executed by each individual processor. We call the indices corresponding to the former *spatial* indices, and the those corresponding to the latter *temporal* indices. The spatial part of a reference pattern, corresponding to the spatial indices, leads to potential inter-processor communication. For the purpose of determining communication patterns, it is sufficient to consider only the spatial part. We hence introduce the notion of *spatial reference pattern.* In terms of notation, the temporal part in a reference pattern is dropped and the arrow is reversed (for reasons that will be clear later). For convenience and in situations where there is no confusion, we will simply call the spatial reference pattern a reference pattern for the rest of this paper.

Suppose that indices i, j are spatial for the first reference pattern in the above example; then its spatial part is

$$\ulcorner a@(i,x) \Rightarrow (i,j) : j = t \text{ and } 1 \leq x \leq n \urcorner.$$

Other Related Concepts A canonical form of an expression is a syntactic form in which variables appear in a predefined order and constants are partially evaluated. For example, $\ulcorner 2-i+j \urcorner$ and $\ulcorner j-i+3-1 \urcorner$ would have the same canonical form $\ulcorner -i+j+2 \urcorner$. The process of deriving a canonical form is called *normalization*, and involves symbolic transformations and partial evaluations.

A Boolean predicate P over an index domain is said to be *space-invariant* if P always evaluates to the same value with respect to different values of the spatial indices. A non-space-invariant predicate may contain space-invariant sub-predicates. For example, suppose that indices i, j are spatial; then the space-invariant component in the predicate $\ulcorner t > 1 \text{ and } i \neq j \urcorner$ is $\ulcorner t > 1 \urcorner$.

In the rest of this paper, we use Greek letters α, β, γ etc to denote arbitrary expressions in a shared-memory program.

2.2 Abstract Machine Model

The class of physical target machines under our consideration is large-scale, distributed-memory machines, including various hypercubes such as iPSC/2 and NCUBE, transputer arrays, WARP and iWARP systolic arrays, and the Connection Machine.

We define an abstract machine for this class of machines. The abstract machine is configured as an n-dimensional grid of size $N_1 \times \cdots \times N_n$ and modeled as an index domain D which is a Cartesian product of interval domains $D = [1..N_1] \times \cdots \times [1..N_n]$.

2.2.1 Communication Patterns

Definition Given an n-dimensional index domain D, let $(i_1, \ldots, i_n)$ range over D. Let σ_p, δ_p where $1 \leq p \leq n$ be expressions of indices $i_1, \ldots, i_n$, and let γ be a boolean predicate over variables $i_1, \ldots, i_n$.

The following form

$$\ulcorner a@(\sigma_1, \ldots, \sigma_n) \Rightarrow (\delta_1, \ldots, \delta_n) : \gamma \urcorner$$

is called a *communication pattern*, which represents the collection of data movements that bring data pointed to by a from $(\sigma_1, \ldots, \sigma_n)$ to $(\delta_1, \ldots, \delta_n)$ for all the elements in D where γ is true.

Tuple $(\sigma_1, \ldots, \sigma_n)$ is called the *source* expression, and $(\delta_1, \ldots, \delta_n)$ the *destination* expression. There are two special forms of communication patterns. In the *sender's form*, the source expression consists of the formals $(i_1, \ldots, i_n)$ ranging over domain D. In the *receiver's form*, the destination expression consists of the formals $(i_1, \ldots, i_n)$:

$$\begin{aligned} \text{Sender's form:} \quad & \ulcorner a@(i_1, \ldots, i_n) \Rightarrow (\delta'_1, \ldots, \delta'_n) \urcorner, \\ \text{Receiver's form:} \quad & \ulcorner a@(\sigma'_1, \ldots, \sigma'_n) \Rightarrow (i_1, \ldots, i_n) \urcorner. \end{aligned}$$

Tuples $(\sigma'_1, \ldots, \sigma'_n)$ and $(\delta'_1, \ldots, \delta'_n)$ are related in the following way. Suppose we can write the source and destination expressions as

$$\begin{aligned} (\delta'_1, \ldots, \delta'_n) &= T_1(i_1, \ldots, i_n) \\ (\sigma'_1, \ldots, \sigma'_n) &= T_2(i_1, \ldots, i_n) \end{aligned}$$

where T_1 and T_2 are well-defined functions. Then T_1 and T_2 must be inverses of each other.

Example Communication pattern $\ulcorner a@(i, j+1) \Rightarrow (i-1, j) \urcorner$ can be transformed into

$$\begin{aligned} \text{Sender's form:} \quad & \ulcorner a@(i, j) \Rightarrow (i-1, j-1) \urcorner, \\ \text{Receiver's form:} \quad & \ulcorner a@(i+1, j+1) \Rightarrow (i, j) \urcorner. \end{aligned}$$

When $(\sigma'_1, \ldots, \sigma'_n)$ and $(\delta'_1, \ldots, \delta'_n)$ are linear expressions of the indices, is is possible to symbolically determine the sender's and receiver's forms. But in general a compiler would not be able to do so. Our restriction on the array index expressions on the left-hand side of an assignment statement (the 2nd assumption on the shared-memory program) is to assure that at least the receiver's form is readily available to the compiler. In case $(\delta'_1, \ldots, \delta'_n)$ is not computable by the compiler, we allow the user to specify it via the *communication form* construct in which the functions T_1 and T_2 are specified and used in references wherever needed.

2.2.2 Communication Primitives

The original ideas of communication primitives are from Fox *et al.* [6], and Johnsson and Ho [7, 8]. They have developed a collection of efficient communication routines for hypercube machines, and have shown that using such synchronous communication is more efficient than using asynchronous message passing (i.e. individual **send** and **receive** pairs) in most scientific and engineering applications.

The goal is to devise an algorithm which can analyze source reference patterns and generate automatically such synchronous communication. We select a small set of communication routines as primitives, as shown in Tables 1 and 2. Primitives in Table 1 are called *general primitives.* Those in Table 2 are called *simple primitives.* Each simple primitive takes a dimension index p as input, and confines data movement to the pth dimension of domain D. In the table entries,

Primitive	*Pattern*	*Cost*	*Type*
One-All-Broadcast$(D, \mathbf{s}, a)$	$\ulcorner a@\ \mathbf{s} \Rightarrow \mathbf{i} \urcorner$	$\mathcal{O}(B \log \lvert N \rvert)$	B
All-One-Reduction$(D, \mathbf{d}, a, \oplus)$	$\ulcorner a@\ \mathbf{i} \Rightarrow \mathbf{d} \urcorner$	$\mathcal{O}(B \log \lvert N \rvert)$	R
All-All-Broadcast(D, a)	$\ulcorner a@\ \mathbf{i} \Rightarrow \mathbf{j} \urcorner$	$\mathcal{O}(B \lvert N \rvert)$	B
Single-Send-Receive$(D, \mathbf{s}, \mathbf{d}, a)$	$\ulcorner a@\ \mathbf{s} \Rightarrow \mathbf{d} \urcorner$	$\mathcal{O}(B)$	P
Uniform-Shift$(D, \mathbf{c}, a)$	$\ulcorner a@\ \mathbf{i} \Rightarrow \mathbf{i} + \mathbf{c} \urcorner$	$\mathcal{O}(B \log \lvert N \rvert)$	P
Affine-Form$(D, M, \mathbf{c}, a)$	$\ulcorner a@\ \mathbf{i} \Rightarrow M\mathbf{i} + \mathbf{c} \urcorner$	$\mathcal{O}(B \log \lvert N \rvert)$	P

Table 1: General Communication Primitives over Domain D and Their Costs

Primitive	*Pattern*	*Cost*	*Type*
Spread(D, p, s, a)	$\ulcorner a@(l_1, s, l_2) \Rightarrow (l_1, i, l_2) \urcorner$	$\mathcal{O}(B \log \lvert N_p \rvert)$	B
Reduction$(D, p, d, a, \oplus)$	$\ulcorner a@(l_1, i, l_2) \Rightarrow (l_1, d, l_2) \urcorner$	$\mathcal{O}(B \log \lvert N_p \rvert)$	R
Multi-Spread(D, p, a)	$\ulcorner a@(l_1, i, l_2) \Rightarrow (l_1, j, l_2) \urcorner$	$\mathcal{O}(B \lvert N_p \rvert)$	B
Copy(D, p, s, d, a)	$\ulcorner a@(l_1, s, l_2) \Rightarrow (l_1, d, l_2) \urcorner$	$\mathcal{O}(B)$	P
Shift(D, p, c, a)	$\ulcorner a@(l_1, i, l_2) \Rightarrow (l_1, i + c, l_2) \urcorner$	$\mathcal{O}(B \log \lvert N_p \rvert)$	P

Table 2: Simple Communication Primitives over Domain D and Their Costs

B denotes the message size, N the number of virtual processors modeled by the index domain D, and N_p the number of processors along the pth dimension of the domain. We also use bold face letters $\mathbf{i}$, $\mathbf{s}$, $\mathbf{d}$ as shorthand for index tuples $(i_1, i_2, \ldots, i_n)$, $(s_1, s_2, \ldots, s_n)$, $(d_1, d_2, \ldots, d_n)$. In Table 2, l_1 and l_2 denote lists of indices $(i_1, \ldots, i_{p-1})$ and $(i_{p+1}, \ldots, i_n)$, respectively.

All of these communication primitives can be implemented efficiently on a target hypercube machine. In addition, each communication primitive has a unique pattern characteristic that the compiler can identify symbolically. We now give a brief description of each of the general primitives.

One-All-Broadcast$(D, \mathbf{s}, a)$: The data pointed to by buffer pointer a in virtual processor $\mathbf{s}$ in domain D is sent to all the other virtual processors in D. This pattern can be identified by the presence of a constant tuple in the source expression and formals in the destination expression.

All-One-Reduction$(D, \mathbf{d}, a, \oplus)$: A primitive based on the **reduction** operator in APL. Data pointed to by a in every virtual processor in D are combined using the binary associative operator $\oplus$ and sent to virtual processor $\mathbf{d}$. This pattern is identified by the presence of a constant tuple in the destination expression and formals in the source expression.

All-All-Broadcast(D, a): Data pointed to by a in every virtual processor in D are duplicated over every other processor. This pattern is identified by formals appearing in both the source and the destination expressions.

Single-Send-Receive$(D, \mathbf{s}, \mathbf{d}, a)$: Data pointed to by a in virtual processor $\mathbf{s}$ is sent to virtual processor $\mathbf{d}$. This pattern is identified by constant tuples appearing in both the source and destination expressions.

Uniform-Shift($D, \mathbf{c}, a$): Data pointed to by a is sent from every virtual processor $\mathbf{i}$ in D to virtual processor $\mathbf{i}+\mathbf{c}$. This pattern is identified by the presence of the same constant offset between the source and destination expressions over all virtual processors in D.

Affine-Form($D, M, \mathbf{c}, a$): Input M is a constant $n \times n$ matrix, where n is the dimensionality of D. Data pointed to by a in every virtual processor $\mathbf{i}$ in D is sent to virtual processor $M \cdot \mathbf{i} + \mathbf{c}$. This pattern is identified by deriving both the sender's and the receiver's forms of the pattern, and verifying the relationship between the two forms. Note that Transpose is a special case of this primitive.

The simple primitives in Table 2 are used for describing collective communication within a single dimension of the multi-dimensional grid of the abstract machine. Each simple primitive has a corresponding general primitive, but its data movement is constrained. The correspondence is as follows:

$$\begin{aligned} \text{Spread} &\iff \text{One-All-Broadcast}, \\ \text{Reduction} &\iff \text{All-One-Reduction}, \\ \text{Multi-Spread} &\iff \text{All-All-Broadcast}, \\ \text{Copy} &\iff \text{Single-Send-Receive}, \\ \text{Shift} &\iff \text{Uniform-Shift}. \end{aligned}$$

For example, Spread(D, p, s, a) means spreading data in the pth dimension from virtual processors which have address s in the pth dimension. In the two dimensional case, Spread($D, 1, 2, a$) means spreading data from the virtual processors in the second row to all the other rows, where row i refers to the elements $(i, *)$.

The communication primitives we are considering are classified into three types according to whether they reduce, preserve, or broadcast messages.

Type R (message-Reducing): Reduction, All-One-Reduction.

Type P (message-Preserving): Copy, Shift, Uniform-Shift, Single-Send-Receive, Affine-Form.

Type B (message-Broadcasting): Spread, Multi-Spread, One-All-Broadcast, All-All-Broadcast.

The classification is used in optimizing the compositions of communication primitives.

Note that these primitives span the complete spectrum of source and destination patterns — one-to-many (one source, many destinations), many-to-one, many-to-many, one-to-one, and multiple one-to-one. They can be composed to form more complex communication patterns. It may be worthwhile to expend this list. For instance, we plan to implement other important primitives such as Shuffle-Exchange.

2.2.3 Communication Metric

A Metric Based on Pattern Uniformity We define the notion of communication metric for defining communication costs on the target machine. With this metric, the cost of any communication pattern of the abstract machine can be calculated. We first present a simple metric which is based on the "uniformity" of communication patterns.

Definition Given a communication pattern (written in sender's form),

$$\ulcorner a@(i_1, \ldots, i_n) \Rightarrow (\delta_1, \ldots, \delta_n) \urcorner$$

we say it is *uniform* in the pth dimension if $\delta_p \cong \ulcorner i_p + c \urcorner$, where c is a constant independent of the size of the index domain, and $\cong$ denotes that the two expressions have the same canonical form.

With this concept, we can classify communication patterns with respect to their *uniformity*, namely patterns that are uniform in every dimension; those that are non-uniform in one, two, three, etc dimensions; and those that are non-uniform in every dimension. We illustrate this classification for the three dimensional case in the following table.

Pattern Example	*Uniformity*
$\ulcorner a@(i,j,k) \Rightarrow (i,j,k) \urcorner$	Memory access
$\ulcorner a@(i,j,k) \Rightarrow (i+c_1, j+c_2, k+c_3) \urcorner$	Uniform in 3 dimensions
$\ulcorner a@(i,j,k) \Rightarrow (\delta_1, j+c_2, k+c_3) \urcorner$	Uniform in 2 dimensions
$\ulcorner a@(i,j,k) \Rightarrow (\delta_1, \delta_2, k+c_3) \urcorner$	Uniform in 1 dimensions
$\ulcorner a@(i,j,k) \Rightarrow (\delta_1, \delta_2, \delta_3) \urcorner$	Non-uniform

A memory access within a processor is often far faster than inter-processor communication. The difference in cost can be as big as 2 or 3 orders of magnitude. To communicate with nearby processors within a constant distance, a message needs only to be routed through a small constant number of processors. Message collisions can be avoided, so neighborhood communication is the most efficient inter-processor communication. Non-uniformity implies non-local communication, which is likely to involve message collisions.

This metric is used in two ways: in the index domain alignment module, it guides the optimization of aligning multiple arrays, and, in the communication synthesis module, it forms the basis for the idea of simple and general primitives and of matching a general reference pattern with a composition of simple primitives.

A Metric Based on Communication Primitives To guide the optimization of selecting partition parameters and determining communication patterns, a more sophisticated communication metric is needed. We define a new communication metric based on the communication primitives. We assign a cost to each communication primitive and derive costs for other patterns from them.

When a specific target architecture is under consideration, the cost of the primitives can be estimated. For instance, suppose the target architecture is a hypercube and the index domain over which the primitives are defined is a multi-dimensional grid embedded in the hypercube using some Gray coding. Then the complexity formulas (using the big $\mathcal{O}$ notation) can be given for each primitive as shown in the third columns of Tables 1 and 2.

When a specific target machine is under consideration, a even more accurate cost function can be obtained. By experimenting with communication primitive routines on an actual machine, we can determine for each primitive the constant factors in the complexity formula.

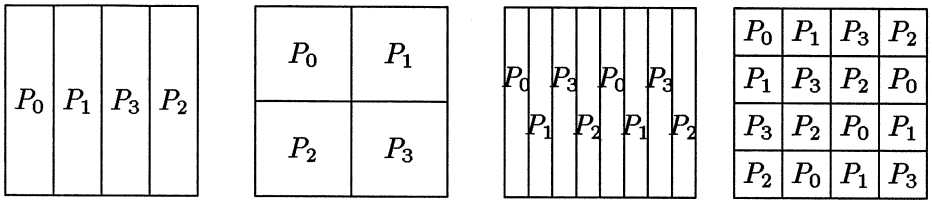


Figure 3: Standard Partition Strategies

2.3 Mapping an Abstract Machine to a Target Machine

2.3.1 Standard Partition Strategies

The partition strategies under consideration here are the standard ones: *block partition*, *strip partition*, and *interleaving partition* (Figure 3). These strategies partition an index domain into equal-size sub-domains. The differences between them are in the shape and the granularity of the sub-domains. In the following, we call a dimension of an index domain *partitioned* if with respect to that dimension the domain is mapped to different processors, otherwise we call the dimension *sequentialized.* In Figure 3, the horizontal dimension of the index domain is partitioned in all the four cases, while the vertical dimension is partitioned only in the second and forth cases.

Given an index domain with fixed sizes, all possible different partitions using the above strategies can be enumerated. If the number of processors of the target machine is also given, the possibilities can be further reduced. For example, suppose an index domain $D = [0..63] \times [0..15]$ is given, and the target machine consists of 32 processors. Then the possible partitions are the following ones: 32×1, 16×2, 8×4, 4×8, 2×16, and 1×32.

The compiler tries each of the candidate partitions, estimates the corresponding communication cost, and then selects the one which has the minimum cost. The process is shown in Fig 4. For large programs consisting of many sub-programs or multiple index domains, global optimization is needed to determine the partition. This is beyond the scope of this paper and will be addressed in a separate paper. The compiler can also rely on the user to provide partitioning parameters. In this mode, our compiler generates parameterized target programs, which at runtime take in parameters provided by the user. Partial evaluation of the target program can further optimize the generated code.

2.3.2 Aggregating Communication

Once a partition strategy is chosen, the communication patterns of the shared-memory program and their matching primitives need to be adjusted, because many communication become unnecessary.

Internalization For those dimensions of an index domain that are *sequentialized*, communication between elements along these dimensions would become local memory accesses. For each partitioned dimension, a range of indices is mapped to a processor. No communication is needed between the elements of the same range. We call this *internalization.*

For example, given a reference pattern

$$\ulcorner a@(3, j+1) \Rightarrow (i,j) : j > 0 \urcorner,$$

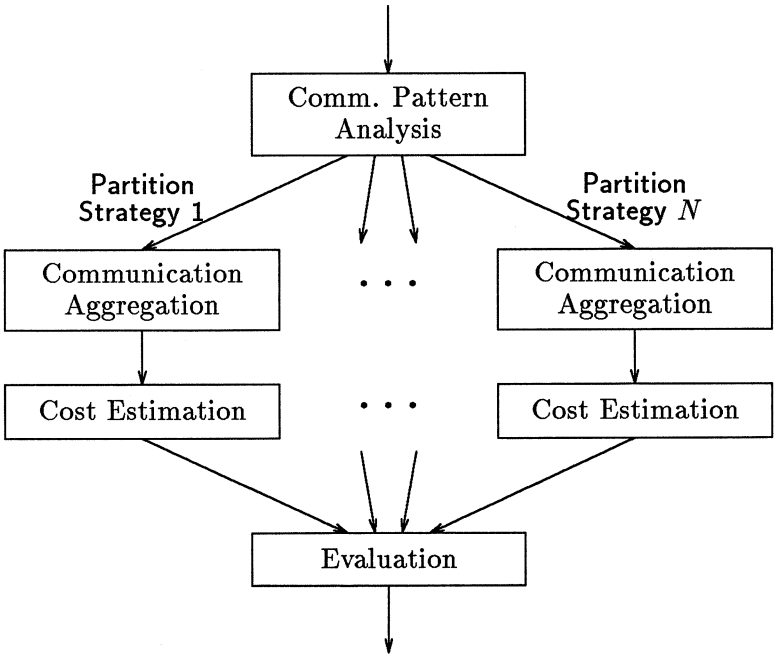


Figure 4: Relation between Partition and Communication

suppose that the first dimension is partitioned while the second is sequentialized. The communication corresponding to the second dimension of the domain is completely internalized while that corresponding to the first dimension is partially internalized. As far as inter-processor communication is concerned, the reference pattern can be transformed to

$$\ulcorner a@(3) \Rightarrow (i) : j > 0 \urcorner.$$

Elimination Look at the following slightly modified reference pattern,

$$\ulcorner a@(3, j) \Rightarrow (i, j) : j > 0 \urcorner.$$

Suppose that the second dimension is partitioned. The reference pattern can then be transformed to

$$\ulcorner a@(j) \Rightarrow (j) : j > 0 \urcorner,$$

resulting in a pattern for which no communication need take place. Thus this reference pattern can be eliminated.

2.3.3 Estimating Communication Costs

In general, performance estimation is a non-trivial problem, and deserves a treatment of its own. However, for applications that are regular and static in nature [6], simple approaches can work quite well. In our case, when the partition strategy for an index domain is known, the size of the sub-domain on each processor can be derived, and then, individual message size can be calculated. The communication cost of a program hence can be estimated by adding up the costs of communication primitives used in the program. If the partition strategy is given as a set of

parameters whose values are unknown at compile time, the difficulty is that the sub-domain sizes can no longer be determined. However, a symbolic expression of the size can still be obtained at compile time and the communication cost can be given in an unevaluated symbolic form. For simple cases, it is possible to compare expressions symbolically and determine the relationship between their values. In general, due to conditionals and unknowns, techniques such as profiling would be needed for estimations.

3 Algorithm for Matching Reference with Communication

In this section, we describe an algorithm for matching reference patterns of a shared-memory program with communication primitives of an abstract machine. For a given program, we define the abstract machine to be of the same shape as the spatial part of the index domain of the program. The matching algorithm is applied to one pattern at a time. Each reference pattern is matched with either a single primitive or a composition of primitives.

3.1 Definition of Pattern Matching

Definition Given an index domain D and a communication primitive defined over D, the *set of instantiated source-destination pairs* of the *primitive* is defined as the set of pairs of the elements of domain D which hold the source-destination relationship with respect to the primitive.

Example Suppose that $D = [1..N_1] \times [1..N_2]$. Recall that $\mathbf{i}$ is a bound variable over D while $\mathbf{s}$, $\mathbf{d}$ are free variables. Primitive One-All-Broadcast$(D, \mathbf{s}, a)$ has the following set of instantiated source-destination pairs:

$$\{(\mathbf{s},(1,1)),(\mathbf{s},(1,2)),\ldots,(\mathbf{s},(1,N_2)),(\mathbf{s},(2,1)),\ldots,(\mathbf{s},(N_1,N_2))\}.$$

For primitive All-All-Broadcast(D, a), the set contains all the possible pairs of elements of D:

$$\{((1,1),(1,1)),((1,1),(1,2)),\ldots,((N_1,N_2),(N_1,N_2))\};$$

and for primitive Single-Send-Receive$(D, \mathbf{s}, \mathbf{d}, a)$, the set has only one element $\{(\mathbf{s}, \mathbf{d})\}$.

Definition Given an index domain D and a reference pattern defined over it, the *set of instantiated source-destination pairs* of the *pattern* is defined as the set of pairs of elements of domain D obtained by replacing the formals in the pattern by all possible values, disregarding the predicates.

Example Reference pattern

$$\ulcorner a@(2,3) \Rightarrow (i,j) : i > j \urcorner$$

over domain $[1..N_1] \times [1..N_2]$ has the following set of instantiated source-destination pairs,

$$\{((2,3),(1,1)),((2,3),(1,2)),\ldots,((2,3),(N_1,N_2))\}.$$

Definition A primitive is said to *match* a reference pattern if the set of instantiated source-destination pairs of the primitive for some choices of the free variables over D is a superset of that of the reference pattern. A primitive is said to *perfectly match* a reference pattern if the two sets of pairs are exactly the same.

Example Reference pattern

$$\ulcorner a@(2,3) \Rightarrow (i,j) : i > j \urcorner$$

over a two-dimensional spatial domain D is matched with both of the communication primitives, One-All-Broadcast$(D,(2,3),a)$ and All-All-Broadcast(D,a), but is perfectly matched with only the former.

Boolean Predicates Presently, predicates in reference patterns are not used in the matching process. This is because the primitives described in this paper are not general enough to take parameters to select processors. We plan to implement communication primitives over a set of selected processors and incorporate predicates as one of the parameters of the primitives.

3.2 The Matching Algorithm

We first describe the major procedures in the matching algorithm.

Identifying a Perfect Matching Identifying a perfect matching is by normalization and symbolic comparison. Suppose we are given a general pattern over index domain D

$$\ulcorner a@\sigma \Rightarrow \delta : \gamma \urcorner$$

where σ and δ are the source and destination expressions and γ is the predicate. The following matching steps will be applied:

Pattern Characteristics	*Matching Primitives*
$\sigma \cong \delta$	(Local Memory Access)
const$(\sigma) \wedge$ const(δ)	Single-Send-Receive(D,σ,δ,a)
const$(\delta - \sigma)$	Uniform-Shift$(D,\delta-\sigma,a)$
const$(\sigma) \wedge$ formal(δ)	One-All-Broadcast(D,σ,a)
formal$(\sigma) \wedge$ const(δ)	All-One-Reduction$(D,\delta,a,\oplus)$
formal$(\sigma) \wedge$ formal(δ)	All-All-Broadcast(D,a)
affine(σ,δ)	Affine-Form$(D,M,\mathbf{c},a)$

Recall that $\cong$ denotes two expressions having the same canonical form, and that symbolic simplifications and partial evaluations are needed to obtain canonical forms. The two predicates, const(σ) and formal(σ), are for testing whether an expression σ contains constants only or formals only, respectively. Note that an expression containing temporal indices is considered a constant expression in the matching. The predicate affine(σ,δ) is for testing if the two vector expressions σ and δ have an affine relationship, i.e. if there exits a constant matrix M and a constant tuple $\mathbf{c}$, such that $\sigma = M\delta + \mathbf{c}$. If a pattern fails to satisfy any of the predicates in the above table, it is not perfectly matched with a general primitive and the next matching step will be taken.

Matching a Simple Pattern with the Lowest-Cost Primitive A reference pattern whose source and destination tuples are exactly the same except for one pair of corresponding index expressions is called a *simple reference pattern.* A simple pattern is always matched with a single primitive. Suppose the following simple pattern over domain D is given:

$$\ulcorner a@(\sigma_1,\ldots,\sigma_{p-1},\sigma_p,\sigma_{p+1},\ldots,\sigma_n) \Rightarrow (\sigma_1,\ldots,\sigma_{p-1},\delta_p,\sigma_{p+1},\ldots,\sigma_n) : \gamma \urcorner$$

It will be tested for the following pattern characteristics:

Pattern Characteristics	*Matching Primitives*
$\sigma_p \cong \delta_p$	(Local Memory Access)
$\text{const}(\sigma_p) \wedge \text{const}(\delta_p)$	$\textsf{Copy}(D, p, \sigma_p, \delta_p, a)$
$\text{const}(\delta_p - \sigma_p)$	$\textsf{Shift}(D, p, \delta_p - \sigma_p, a)$
$\text{const}(\sigma_p)$	$\textsf{Spread}(D, p, \sigma_p, a)$
$\text{const}(\delta_p)$	$\textsf{Reduction}(D, p, \delta_p, a, \oplus)$
otherwise	$\textsf{Multi-Spread}(D, p, a)$

The predicates in the left column are not mutually exclusive, so the order in which they are tested is important. We order them so that the lowest cost primitive will be matched first.

Decomposing a Reference Pattern A general reference pattern in an n-dimensional index domain can be thought of as a composition of n simple patterns, each describing data movement along one dimension. We call a composition of a subset of these simple patterns a *sub-pattern* of the general pattern. When a general reference pattern cannot be perfectly matched with a single primitive, it will be decomposed into sub-patterns and the then matching algorithm will be applied to these sub-patterns recursively.

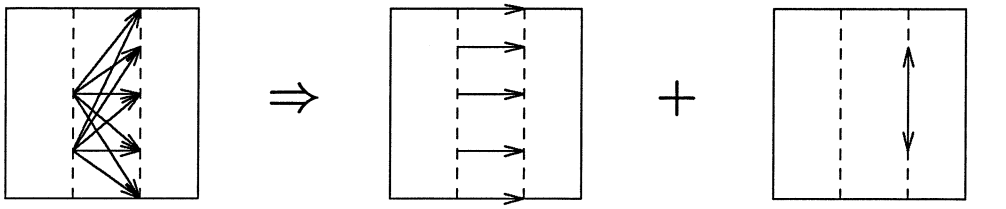
Example Reference pattern

$$\ulcorner a@(c(i,j), j-3) \Rightarrow (i,j) \urcorner$$

over domain D cannot be perfectly matched with a single primitive. It is therefore decomposed into two sub-patterns, each of which is matched with a single primitive:

$$\begin{array}{ll} \ulcorner a@(i, j-3) \Rightarrow (i,j) \urcorner & \textsf{Shift}(D, 2, 3, a) \\ \ulcorner a@(c(i,j), j) \Rightarrow (i,j) \urcorner & \textsf{Multi-Spread}(D, 1, a) \end{array}$$

The composition of these two primitives, $\textsf{Shift}(D, 2, 3, a)$ and $\textsf{Multi-Spread}(D, 1, a)$, is the result of the match (Fig 5).



$\ulcorner a@(c(i,j), j-3) \Rightarrow (i,j) \urcorner$ $\quad$ $\textsf{Shift}(D, 2, 3, a)$ $\quad$ $\textsf{Multi-Spread}(D, 1, a)$

Figure 5: Decomposing a Reference Pattern

In general, when the index domain is of high dimensionality, there will be many ways to decompose a reference pattern. To find the optimal composition of primitives, dynamic programming techniques are used.

Optimizing the Composition of Primitives Notice that the ordering of the primitives in the composition does not affect the correctness of the target program. However, it does affect the cost of communication. For the above example, we can have two orderings:

Case 1: $\mathsf{Shift}(D,2,3,a) \circ \mathsf{Multi\text{-}Spread}(D,1,a)$ (applying multi-spread in dimension one first; shift in dimension two second) — the message size for multi-spread is the original message size B, and the cost of it is $\mathcal{O}(B|D_1|)$. However, the result of multi-spread is that every processor gets a whole column of data, hence the data size for shift becomes $B|D_1|$. The corresponding cost is $\mathcal{O}(B|D_1|)$.

Case 2: $\mathsf{Multi\text{-}Spread}(D,1,a) \circ \mathsf{Shift}(D,2,3,a)$ (applying shift in dimension two first; multi-spread in dimension one second) — The message size for shift is B, so the cost is $\mathcal{O}(B)$. The message size for multi-spread is the same, and the cost is $\mathcal{O}(B|D_1|)$. The total cost is less than that of the first case.

The principle for ordering primitives in a composition is to have them appearing in the following order: type R followed by type P primitives, and finally type B primitives.

We now summarize the above with the reference pattern matching algorithm below:

Algorithm (Matching Reference Patterns).

Step 1. *For a given pattern, search through the list of primitives and try to find a perfect match;*

Step 2. *For a simple pattern which fails Step 1, find the lowest cost matching primitive;*

Step 3. *For a general pattern which fails Step 1,*

- **3.1** *decompose it into sub-patterns;*
- **3.2** *recursively apply the algorithm on the sub-patterns;*
- **3.3** *optimize the composition of the resulting primitives.*

3.3 Optimizing Reference Patterns

Reference patterns derived directly from the input program are usually not in the most efficient form. In this following, we introduce several optimizations which transform the original set of reference patterns into ones that are better suited for implementation.

Narrowing Scopes By constant propagation, the communication primitive that perfectly matches a given reference pattern may change to another with lower cost. For example, a perfect match of the reference pattern

$$\ulcorner a@(3,j) \Rightarrow (i,j) : i = 4 \urcorner$$

would result in a Spread. However, the information that i is a constant or it is space-invariant results in the reference pattern

$$\ulcorner a@(3,j) \Rightarrow (4,j) \urcorner,$$

which matches perfectly with a Copy, which is less costly than a Spread.

Combining Identical Patterns Two reference patterns can be combined if they are equivalent, disregarding the guards. Clearly, the advantage of combining is to reduce the number of

communication statements and thus the number of messages. For example, the following two reference patterns

$$\ulcorner a@(i+3,j) \Rightarrow (i,j) : i > 53 \urcorner,$$
$$\ulcorner a@(i+3,j) \Rightarrow (i,j) : i < 7 \urcorner$$

can be combined as

$$\ulcorner a@(i,j) \Rightarrow (i+3,j) : i > 53 \text{ or } i < 7 \urcorner,$$

thus eliminating a **Shift**.

Combining Subset Patterns If the set of instantiated source-destination pairs of one reference pattern is a subset of another, then it can be eliminated. For example, the following two reference patterns

$$\ulcorner a@(2,3) \Rightarrow (2,j) : j > 1 \urcorner,$$
$$\ulcorner a@(2,3) \Rightarrow (i,j) : j > 1 \urcorner$$

can be combined as

$$\ulcorner a@(2,3) \Rightarrow (i,j) : j > 1 \urcorner.$$

Aggregating Patterns In case there are many individual reference patterns sending messages from the same source to different destinations, it is often better to use **Spread** instead of many **Copy**s. For example, for the following reference patterns

$$\ulcorner a@(2,j) \Rightarrow (c_1,j) : j > 1 \urcorner,$$
$$\ulcorner a@(2,j) \Rightarrow (c_2,j) : j > 1 \urcorner,$$
$$\ldots$$
$$\ulcorner a@(2,j) \Rightarrow (c_k,j) : j > 1 \urcorner$$

where $c_1, c_2, \ldots, c_k$ are constants. When k is large, it is better to combine these reference patterns as

$$\ulcorner a@(2,j) \Rightarrow (i,j) : j > 1 \urcorner.$$

When to do this optimization depends on the relative costs of **Copy** and **Spread** and must be determined experimentally for each target machine.

3.4 Trade-off in Matching Patterns

Our approach may generate communication which are more costly than necessary due to indirect references. For example, suppose the reference pattern (over domain $D = [1..n] \times [1..n]$)

$$\ulcorner a@(2,j) \Rightarrow (c(i,j),j) : j > 1 \urcorner$$

contains an indirect reference $c(i,j)$ whose value can not be determined at compile-time. Our pattern matching algorithm would match it with **Spread**, but **Copy** would suffice if $c(i,j)$ was known to be constant at compile time. Let's examine some alternatives to our matching algorithm.

Asynchronous Communication An alternative approach is to generate a Request-Receive pair which interrupts the processor holding the requested value. The target program looks like

Program for processor p :
if $(j > 1)$ and $(i = 2)$ {
  ⟨Send a request to processor idx_to_pid$(c(i,j),j)$⟩;
  ⟨Wait for an answer from processor idx_to_pid$(c(i,j),j)$⟩;}

The Request-Receive pair works as follows: Whenever there is a request coming to a processor, an interrupt handler will send out the requested data if it is ready, otherwise it will queue the request and send out the value when it becomes available. The overhead of interrupt handling and queue management may be reduced if a separate communication co-processor is available in the hardware. In practice, message granularity in this approach is fine enough so that it incurs unacceptably high overhead on machines like the iPSC/2. In addition, asynchronous communication makes this approach far more error-prone. A working mechanism for asynchronous communication on this class of machines may incur additional system overhead.

User Directives Another alternative is to allow the user to provide enough information to generate efficient communication. It turns out that all that is needed is a pair of functions which are inverses of each other for specifying the sender's form and the receiver's form of a given reference pattern. Using the same example shown above, the user can say

Communication Forms:
$T(i,j) = (c(i,j),j) = \{(i \text{ div } j, j)\}$
$T\_inv(i,j) = \text{if } (i <= (n \text{ div } j)) \rightarrow \{(k,j) \mid i*j <= k < \min(n+1,(i+1)*j)\};$

The inverse T_inv can then be used to generate a send-receive pair for efficient communication. The corresponding target code will look like

Program for processor p :
if $(j > 1)$ and $(i = 2)$
  ⟨Send msg to processor idx_to_pid$(T(i,j))$⟩;
if $(j > 1)$ and (p $\in$ {idx_to_pid$(T_inv(i,j)$}$)$
  ⟨Receive msg from processor idx_to_pid$(2,j)$⟩;

We think this approach is the best and will support this in the future.

4 Synchronizing and Scheduling Communications

Once a communication primitive is chosen, we still need to schedule it and synchronize those processors that participate in the communication.

From the point of view of synchronization, communication primitives can be classified into two groups. *Group A* consists of primitives that require synchronization among more than two processors (i.e. using combining trees), e.g. Spread, Reduction, One-All-Broadcast and All-All-Broadcast. *Group B* consists of primitives that require synchronization only between pairs of processors (i.e.

using just pairs of **send** and **receive**), e.g. **Copy**, **Shift**, and **Single-Send-Receive**. This classification relates to the classification based on message type (Section 2) in the following way: the set of group A primitives is equivalent to the union of the sets of primitives of type R and B, and the set of group B primitives is equivalent to the set of primitives of type P.

4.1 Synchronizing Group A Communication Primitives

Take the following source program segment as an example,

$$
\begin{array}{l}
\textsf{for } (t : [0..n]) \\
\quad \textsf{forall } (i : [1..n], j : [1..n]) \\
\quad\quad a(i,j,t) = \textsf{if } (t > 1) \textsf{ and } (i > j) \rightarrow b(i+1, j-2, t); \\
\quad\quad\quad\quad\quad\quad\quad\quad \textsf{else} \rightarrow a(i,j,t-1);
\end{array}
$$

One spatial reference pattern derived is

$$\ulcorner b@(i+1, j-2) \Rightarrow (i,j) : t > 1 \textsf{ and } i > j \urcorner,$$

and is matched with a **Uniform-Shift**.

A **Uniform-Shift** consists of a set of **send** and **receive** pairs. To construct correct message passing on a distributed-memory machine, each **send** must be matched with a **receive**. Both the sender's and the receiver's forms of a reference pattern are needed. From the above reference pattern, we need to derive the sender's form

$$\ulcorner b@(i,j) \Rightarrow (i-1, j+2) : t > 1 \textsf{ and } (i-1) > (j+2) \urcorner.$$

The resulting target code (omitting the address translation and aggregation for simplicity) looks as follows:

$$
\begin{array}{l}
\textsf{if } (t > 1) \;\&\&\; (i-1) > (j+2) \\
\quad \textsf{send}(D, (-1,2), b); \\
\textsf{if } (t > 1) \;\&\&\; (i > j) \\
\quad \textsf{receive}(D, (1,-2), b); \\
\textsf{if } (t > 1) \;\&\&\; (i > j) \\
\quad a[i][j] = b[i+1][j-2];
\end{array}
$$

Note that the send statement is derived from the sender's form of the reference pattern while the receive statement is derived from the receiver's form.

The compiler generates a non-blocking **send** and a blocking **receive**, and places the **receive** statement right after its corresponding **send** statement. A blocking **receive** does not return until the message it is expecting arrives.

4.2 Synchronizing Group B Communication Primitives

When many processors participate in executing a communication primitive, the important thing is to make certain that all relevant processors execute the same predicate. For example, from the

source program segment

$$
\begin{array}{l}
\textsf{for } (t : [0..n]) \\
\quad \textsf{forall } (i : [1..n], j : [1..n]) \\
\quad\quad a(i,j,t) = \textsf{if } (t > 1) \textsf{ and } (i > j) \ \rightarrow \ b(3,j,t); \\
\quad\quad\quad\quad\quad\quad\quad\quad \textsf{else } \rightarrow \ a(i,j,t-1);
\end{array}
$$

we derive the reference pattern

$$\ulcorner b@(3,j) \Rightarrow (i,j) : t > 1 \textsf{ and } i > j \urcorner,$$

which is then matched with a $\textsf{Spread}(D,1,3,b)$, where $D = [1..n] \times [1..n]$, by disregarding the guard $i > j$. The corresponding target code looks like

Program for processor (x,y) :
if $(t > 1)$ {
  $\textsf{Spread}(D,1,3,b)$;
  if not $\langle (i,j) \in [I_l(x,y)..I_u(x,y)] \times [J_l(x,y)..J_u(x,y)]$ such that $i > j \rangle$
    $\langle$discard received data$\rangle$;
  if $(i > j)$
    $a[i][j] = b[3][j]$;
  }

In general, we extract *space-invariant* components out from the guard and disregard the rest in communication. Note that unnecessary data are discarded as soon as possible to free up message buffer space and to avoid using local memory in the processor.

4.3 Scheduling Communication

Recall that an intermediate, shared-memory program consists of multi-level loops. A target program generated by the compiler for processors of the target machine has similar structure. Disregarding the details of computation, a loop body can be thought of consisting of a sequence of segments, containing computation or communication.

Definition For an array assignment statement appearing in a multi-level loop of a shared memory program (without lost of generality, assume the outermost k loops are for loops),

$$
\begin{array}{l}
\textsf{for } (i_1 : D_1, \ldots, i_k : D_k) \\
\quad \textsf{forall } (i_{k+1} : D_{k+1}, \ldots, i_n : D_n) \\
\quad\quad a(i_1, \ldots, i_n) = \cdots;
\end{array}
$$

we call the set of statements in the target node program for implementing the assignment statement a *computation segment for array* a, and denote it by $\textsf{computation}(a, i_1, \ldots, i_k)$, with the indices of the outermost k loops specified as parameters.

A computation segment may contain conditionals, loops (e.g. inner-level **forall** loops), and local environments (e.g. compound statements in C). The parameters $i_1, \ldots, i_k$ are space-invariant and their significance will be discussed later.

Definition Given a reference pattern P as in the above definition,

$$P : \ulcorner a(i_1, \ldots, i_n) \leftarrow b(\delta_1, \ldots, \delta_n) : \tau \urcorner,$$

we call the set of synthesized communication statements (including the calls to primitive routines, statements for loading message buffers, etc.) for implementing it the *communication segment for pattern* P, and denote it by **communication**(P).

A sequence of computation segments within a multi-level loop consistent with the data dependence between the arrays is first obtained in the control structure synthesis module of the compiler. The task of scheduling communication is to determine the appropriate location for each communication segment within the sequence of computation segments. For this purpose, we define for each communication segment two sets of associated computation segments.

Given a reference pattern P as in the above definition, the set of *pre-segments* for **communication**(P) consists of (1) the computation segment in which $b(\delta_1, \ldots, \delta_n)$ is computed; and (2) the computation segments in which the indirect array references occurring in $\delta_1, \ldots, \delta_n$ are computed. The *post-segments* consists of the computation segments in which $a(i_1, \ldots, i_n)$ is computed.

A communication segment is scheduled before all of its post-segments and after all of its pre-segments.

Example Given the following program,

```
for (t : [0..n]) {
  forall (i : [1..n], j : [1..n])
    b(i,j,t) = if (t = 0) → 0;
               else → b(i,j,t-1);
  forall (i : [1..n], j : [1..n])
    a(i,j,t) = if (t > 0) and (i > j) → b(c(i,j,t-1),j,t);
               else → τ;
  forall (i : [1..n], j : [1..n])
    c(i,j,t) = a(i,j,t) + 3;
}
```

the sequence of computation segments of the target program can be derived as shown in Figure 6(a). The reference patterns of the program

$$P_1 : \ulcorner a(i,j,t) \leftarrow b(c(i,j,t-1),j,t) : t > 0 \text{ and } i > j \urcorner,$$
$$P_2 : \ulcorner a(i,j,t) \leftarrow c(i,j,t-1) : t > 0 \text{ and } i > j \urcorner,$$
$$P_3 : \ulcorner b(i,j,t) \leftarrow b(i,j,t-1) : t > 0 \urcorner,$$
$$P_4 : \ulcorner c(i,j,t) \leftarrow a(i,j,t) \urcorner.$$

determines the corresponding pre-segments and post-segments:

```
for (0 <= t <= n) {
    computation(b, t);
    computation(a, t);
    computation(c, t);
    }
```

(a) shared-memory program

```
for (0 <= t <= n) {
    communication(P3);
    computation(b, t);
    communication(P1);
    communication(P2);
    computation(a, t);
    communication(P4);
    computation(c, t);
    }
```

(b) with explicit communication

Figure 6: Outlines of a Target Program

	Pre-Segments	*Post-Segments*
communication(P_1)	computation(b, t) computation($c, t-1$)	computation(a, t)
communication(P_2)	computation($c, t-1$)	computation(a, t)
communication(P_3)	computation($b, t-1$)	computation(b, t)
communication(P_4)	computation(a, t)	computation(c, t)

The target program after the insertion of the communication segments is shown in Figure 6(b).

Cross-Iteration Dependence As we see in the above example, the pre- or post-segments of a communication segment may contain segments which are not computed in the current loop iteration. Thus a communication segment may be placed at the beginning or at the end of a loop body to satisfy the cross-iteration dependence. The space-invariant indices attached to each computation segment provides the information for doing so.

4.4 Correctness of Communication Synthesis

One important issue in synthesizing communication is to guarantee that no deadlock is introduced by the compiler. We prove this property of our synthesis procedure as follows:

The target code generated by the compiler consists of a host program (which we will not discuss), and a node program for all the processors in the so-called SPMD style. The node program consists of sequential loops; the body of each loop is a sequence of computation segments and communication segments.

The computation segments of a target program are single-entry single-exit segments (i.e. there are no **goto** or **break** statements). The compiler generates these computation segments based on semantics-preserving transformations which do not introduce deadlock, and the execution of such a segment eventually terminates. Therefore, we can focus on each individual communication segment, which consists of either a group A or a group B communication primitive. We prove

by induction on the sequence of communication segments. We assume that all processors have reached at the Nth communication segment.

Case 1: *The communication segment consists of a group A communication primitive.* Since such a communication primitive is guarded only by space-invariant predicates, all processors will execute the communication segment. Since the communication primitive is assumed to terminate, the entire segment terminates.

Case 2: *The communication segment consists of a group B communication primitive.* We assume that the message buffer is large enough to hold the entire data transmitted in a message. [1] (1) Since the **send** and **receive** pair of the primitive is arranged as a non-blocking **send** followed by a blocking **receive** (Section 4.1), every processor will execute a **send** statement first. (2) Due to the assumption on the buffer size, no deadlock due to buffer overflow will occur; therefore every processor entering the communication segment will eventually finish executing the **send** statement, and move on to the **receive** statement. (3) Since the predicates for the **send** and **receive** statements are arranged in such a way that for every message sent out to the network, there is a receiving statement matching it (Section 4.2), every **receive** statement will terminate with received data. Therefore, the the Nth communication segment eventually terminates, and so the program also terminates.

5 Summary

In this paper we consider generating a program with explicit communication commands from a program for shared-memory multiprocessors (called an intermediate program) based on a set of standard data partition strategies. The shared-memory references of the intermediate program are translated to either local memory accesses or inter-processor communication. These references, depending on their syntactic patterns, are matched with a library of aggregate communication primitives. Based on a communication metric that captures the cost of inter-processor communication, communication primitives with lowest cost will be chosen. We use a stylized SPMD target program form for incorporating communication commands together with the computation part of the program. For a communication between two processors, matched **send** and **receive** pairs must be correctly synchronized and sequenced in such a way that no deadlock is generated.

Acknowledgment

We thank Young-il Choo for suggesting the notation for reference patterns and many helpful discussions, and thank Joe Rodrigue for careful reading and commenting on an earlier version of the paper. Our thanks also go to Lennart Johnsson, Ching-Tien Ho, and Bill Gropp for helpful discussions on hypercube communication.

[1] This assumption can be relaxed if we take buffer size into consideration when generating communication.

References

[1] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. In *Proceedings of the First International Conference on Supercomputing*, 1987.

[2] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 1987.

[3] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[4] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2(2):151–170, 1988.

[5] M. C. Chen. A parallel language and its compilation to multiprocessor machines. In *The Proceedings of the 13th Annual Symposium on POPL*, January 1986.

[6] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors.* Prentice Hall, 1988.

[7] Ching-Tien Ho. *Optimal Communication Primitives and Graph Embeddings on Hypercubes.* PhD thesis, Yale University, 1990.

[8] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *J. Parallel and Distributed Computation*, 4(2), April 1987.

[9] C. Koelbel and P. Mehrotra. Compiler transformations for non-shared memory machines. In *4th International Conference on Supercomputing*, May 1989.

[10] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Semi-automatic process partioning for parallel computation. *International Journal of Parallel Computing*, 16, 1987.

[11] Jingke Li. *Compiling Crystal for Hypercube Machines.* PhD thesis, Yale University, (in preparation).

[12] Jingke Li and Marina Chen. Index domain alignment: Minimizing cost of cross-reference between distributed arrays. Technical report, Yale University, 1989.

[13] David A. Padua, David J. Kuck, and Duncan H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, C-29(9):763–776, September 1980.

[14] A. Ramanujan and P. Sadayappan. A methodology for parallelizing programs for complex memory multiprocessors. In *Supercomputing 89*, Reno, Nevada, Nov. 1989.

[15] Matthew Rosing and Robert B. Schnabel. An overview of Dino – a new language for numerical computation on distributed memory multiprocessors. Technical Report CU-CS-385-88, University of Colorado, March 1988.

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

[17] Hans P. Zima, Heinz J. Bast, and Michael Gerndt. Superb: A tool for semi-automatic SIMD/MIMD parallelization. *Parallel Computing*, 6:1–18, 1988.