Automating the Coordination of Interprocessor Communication

Jingke Li and Marina Chen

YALEU/DCS/TR-829
October, 1990

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Yale University
Department of Computer Science

Automating the Coordination of Interprocessor Communication

Jingke Li and Marina Chen

YALEU/DCS/TR-829
October, 1990

This work has been supported in part by the Office of Naval Research under Contract N00014-89-J-1906, N00014-90-J-1987.

Automating the Coordination of Interprocessor Communication

October 29, 1990

Jingke Li
Department of Computer Science
Portland State University
P.O.Box 751
Portland, OR 97207
Tel. (503) 725-4053
Email: li@cs.pdx.edu

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

To appear in "Programming Languages and Compilers for Parallel Computing," Research Monographs in Parallel and Distributed Computing series, MIT Press, Gelernter et al., eds, 1990.

Contents

Abstract

This paper presents methods for ensuring correct synchronization and scheduling of message-passing in the context of compiling shared-memory programs onto distributed-memory machines. We show that from a given source loop nest, there corresponds a *maximum granularity* where the computation can go on without the need for any communication, and a *communication window* within which a communication command must occur and can occur anywhere legally. Better overall efficiency can then be achieved by playing with the granularity parameter using more frequent communication than that for the maximum granularity case.

1 Introduction

In compiling a shared-memory program to a distributed-memory target machine, explicit communication commands to achieve interprocessor data transfer must be generated from the references in the source program. The generation process has three parts to it: first, selecting appropriate communication primitives (the *synthesis* part), placing the calls to these primitives in appropriate location in the target program text to ensure correct statement sequencing and scope (*scheduling*), and setting up correct conditions for invoking these primitives (*synchronization*). This paper addresses the latter two issues, referred to together as the issues of *coordinating* interprocessor communication. We present a solution which ensures that the data dependency of the original shared-memory program is

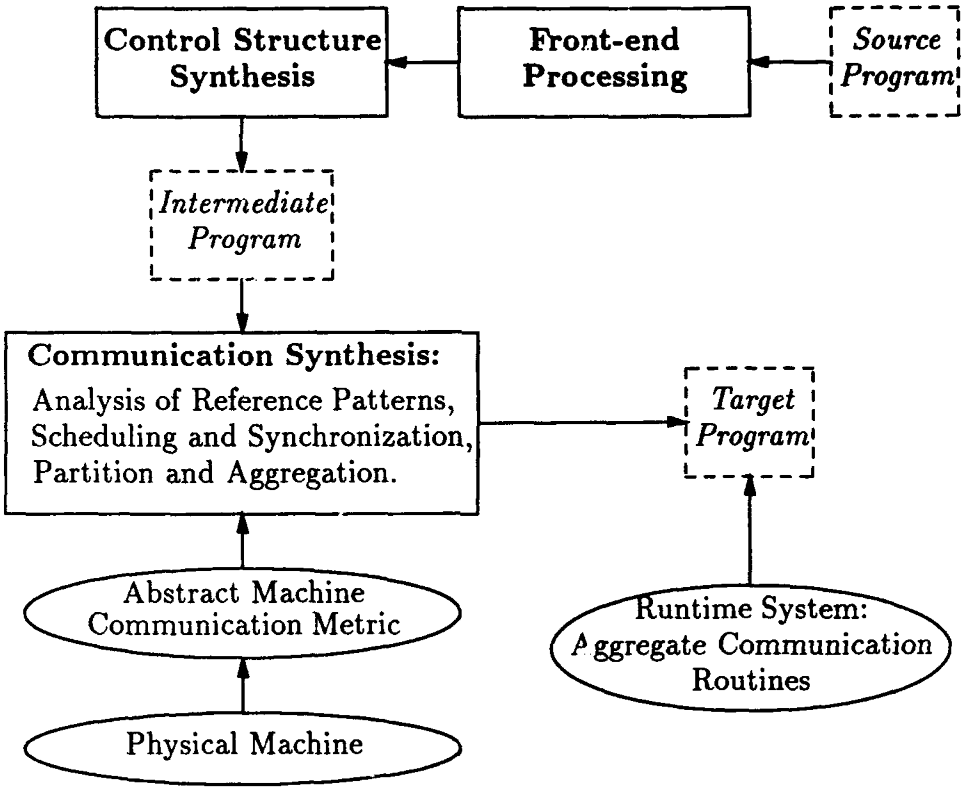

Control Structure Synthesis
Front-end Processing
Source Program
Intermediate Program
Communication Synthesis: Analysis of Reference Patterns, Scheduling and Synchronization, Partition and Aggregation.
Target Program
Abstract Machine Communication Metric
Runtime System: Aggregate Communication Routines
Physical Machine

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

preserved in the target message-passing program. Two important notions are developed for coordination: the *maximum granularity* of a loop nest where the computation can go on without the need for any communication, and a *communication window* within which a communication command must occur and can occur anywhere legally.

First we provide some background for our work on coordination of interprocessor communication.

Crystal Approach The Crystal approach to programming parallel computers 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. Our approach to compilation consists of the following components:

1. *Control structure synthesis:* deriving a parallel control structure from a functional specification or a sequential program. This component as shown in Figure 1 generates an intermediate program in which parallel schedule and flow of control are made explicit, but the references are still based on a global shared memory.

2. *Data distribution:* 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 interprocessor communication (we call this process *domain alignment*).

3. *Communication synthesis:* translating all references to data structures to either local memory accesses or interprocessor communication. The reference patterns of the intermediate program are matched with a library of aggregate communication routines and those which minimize network congestion and overhead are chosen. Once such calls to communication routines are determined, they need to appear in the program text under the correct conditions, statement sequence, and scope. This issue of coordinating interprocessor communication is the topic of this paper.

Issues in Coordinating Interprocessor Communication We use the following example to illustrate some issues arising in automating the coordination of interprocessor communication. The following loop nest contains a segment of a shared-memory program with explicit parallel control using forall loops:

```
for $(t : [0..n])$
  forall $((i,j) : [1..n] \times [1..n])$
    $a(i,j,t) =$ if $(t > 1)$ and $(i > j) \rightarrow a(i+1, j-2, t-1);$
                else $\rightarrow i + j;$
```

The program is written in a single-assignment, C-like notation which will be described in more detail in Section 2. The array a has been expanded with an extra dimension so as to allow the single assignment form of statement, however, its implementation will actually be a two-dimensional array using side-effecting assignment statements.

Suppose we distribute the forall loops over the two-dimensional domain to a two-dimensional network of processors. Let each processor be denoted by a pair (x, y), and let processor (x, y) be responsible for a range of iterations specified by the intervals $D_1 = [I_l(x,y)..I_u(x,y)]$ and $D_2 = [J_l(x,y)..J_u(x,y)]$. The following is the program for each processor (x, y) which is assigned a portion of array a:

```
Program for processor $(x,y)$ :
for $(t : [0..n])$
  forall $((i,j) : D_1 \times D_2)$
    if $(t > 1)$ && $(i > j)$
      $a[i][j][t] = a[i+1][j-2][t-1];$
    else $a[i][j][t] = i + j;$
```

For any given iteration $t : [0..n]$, the reference pattern of the program indicates that for every element (i, j) in the domain $[1..n] \times [1..n]$, the value of a needs to be shifted to another element which is offset by $(1, -2)$ steps from itself.

Recognizing such a pattern symbolically, we can match these references with the following **send** and **receive** commands, and the target message passing program will look like

```
Program for processor $(x, y)$ :
for $(t : [0..n])$ {
  forall $(\langle (i, j)$ on boundary of $D_1 \times D_2 \rangle)$ {
    if $(t > 1)$ && $(i - 1) > (j + 2)$
      send$((-1, 2), a)$;
    if $(t > 1)$ && $(i > j)$
      receive$((1, -2), a)$;
  }
  forall $((i, j) : D_1 \times D_2)$
    if $(t > 1)$ && $(i > j)$
      $a[i][j][t] = a[i + 1][j - 2][t - 1]$;
    else $a[i][j][t] = i + j$;
}
```

As we see above, the destination addresses of the **send** and **receive** commands are the inverses (to be defined later) of each other, and so are the predicates of the statements in which they appear. Since such information is not explicit in the shared-memory program, it needs to be derived symbolically. Also, there are many possible choices for when a communication can occur; for instance, it can occur immediately before the statement where the transferred data are to be used or it can occur much earlier. In addition, the actual target code generated will be more complex than what is shown above due to the need to aggregate many individual send commands into a single send command to avoid overhead involved in each invocation of a communication command. There are alternative ways to aggregate which can have significant performance impact.

Related Work The problem of automatically generating communication for distributed-memory machines from program references is addressed by several research projects [1, 8, 12, 13, 14, 15, 17] In terms of coordinating interprocessor communication, most of the proposed ideas use *run-time analysis* to solve the synchronization problem, which works as follows. A processor sends out a request whenever there is a need for a piece of data which is not available locally, and the request will interrupt the source processor. In this setting, scheduling is by default (communication occurs whenever there is a request). Rogers and Pingali [14] also describe a *compile-time resolution* in which processors are assigned to every computation node in the abstract-syntax tree of the source program at compile-time so that the source and destination of each message can be determined. The scheduling problem is not specifically discussed in that paper.

2 Context

To make this paper self-contained, we first describe the framework for generating communication commands from source program references. We introduce concepts and notations that are pertinent to the problems addressed in this paper. We first define the form of the input programs, called *shared-memory programs.* Then we introduce the notions and representations of *index domains* and *reference patterns*, which provide essential information regarding the latent parallelism and communication in a program. Next, we describe a set of run-time communication routines, which will be used by the compiler to implement interprocessor communication. We also describe *standard data partition strategies* since data layout affects the communication to be generated. Finally, we describe the method for selecting calls to run-time communication routines based on analyzing the reference patterns appearing in the shared-memory program.

2.1 Shared-Memory Programs

The input program to the communication synthesis module shown in Figure 1 is given in a C-like notation augmented with parallel control structures to be described below. Suitable pre-processing can be applied to programs written in exiting parallel shared-memory languages or sequential languages augmented with parallel control structures (see [16] for FORTRAN extensions) to obtain this form (with un-essential syntactic variations). An example shared-memory program is given in Figure 2.

We have the following assumptions on the form shared-memory programs:

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 subscripts are index variables: Array subscript expressions on the left-hand side of an assignment statement must be index variables. 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: Within each loop nest, arrays are aligned to have a common *index domain* (see [11] for a detailed discussion on domain alignment). The boundaries of each individual array are appropriately adjusted according to the alignment. In the example shown in Figure 2, the index domain of the loop nest is the Cartesian product of intervals $[1..n] \times [1..n] \times [0..n]$.

We use the notation $(t : [0..n])$ to denote a for or forall loop indexed by t with lower bound 0 and upper bound n, and the notation of Cartesian product of intervals as in $((i, j) : [0..n] \times [0..n])$ for doubly nested loops. A for loop is just a conventional sequential

```
for ($t : [0..n]$) {
  forall ($(i,j) : [1..n] \times [1..n]$)
    $b(i,j,t) = $ if ($j = t$) $\rightarrow$
                  $\backslash + \{a(i,x,t-1) \mid 1 \leq x \leq n\}$;
                else $\rightarrow$ 1;
  forall ($(i,j) : [1..n] \times [1..n]$)
    $a(i,j,t) = $ if ($t = 0$) $\rightarrow$ 0;
                else if ($i = b(0,t,t)$) $\rightarrow a(i,j,t-1)$;
                else $\rightarrow b(i,t,t)$;
}
```

Figure 2: A Shared-Memory Program

loop. A forall is a parallel loop whose iterations can be executed in parallel with the assumption that proper synchronization is enforced. For example, given the following loop

```
forall ($i \in D_1$)
  for ($j \in D_2$)
    $a(i,j) = a(2,j-1)$,
```

the iterations of the forall loop can be executed in parallel, however, they have to be synchronized by the iterations of the for loop, i.e. no iteration of the forall loop should go on to the $j+1$th iteration of the for loop, unless all the iterations finish the jth iteration.

General data structures using pointers as in C are not allowed in the shared-memory program defined above. Notice also that arrays in a shared-memory program are expanded with extra dimensions so as to allow the single assignment form of statements. In the code generation stage, however, the extra dimensions of an array will be collapsed, and the array restored to its original shape.

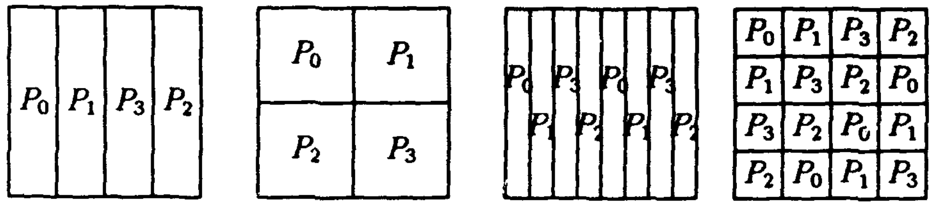
Reference Patterns For each pair of array references appearing on the two sides of an assignment statement in a loop,

```
for ($i_1 : D_1, \ldots, i_n : D_n$)
  $a(i_1, \ldots, i_n) = $ if $\gamma \rightarrow \cdots b(\tau_1, \ldots, \tau_n) \cdots$;
                    else $\rightarrow \cdots$;
```

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.



(a) Strip (b) Block (c) Interleaving

Figure 3: Standard Partition Strategies

The following reference patterns can be derived from the program in Figure 2,

$$\begin{aligned}
P_1 &: \ulcorner b(i,j,t) \leftarrow a(i,x,t-1) : j = t \text{ and } 1 \leq x \leq n\urcorner,\\
P_2 &: \ulcorner a(i,j,t) \leftarrow b(0,t,t) : t \neq 0\urcorner,\\
P_3 &: \ulcorner a(i,j,t) \leftarrow c(i,j,t-1) : t \neq 0 \text{ and } i = b(0,t,t)\urcorner,\\
P_4 &: \ulcorner a(i,j,t) \leftarrow b(i,t,t) : t \neq 0 \text{ and } i \neq b(0,t,t)\urcorner.
\end{aligned}$$

A reference pattern represents a collection of data dependencies. We emphasize this aggregate form rather than each instance of a reference because the key for generating efficient target message-passing programs is to extract correlated references and issue a single aggregate communication routine that is optimized both for the pattern and the target architecture.

2.2 Partition Strategies

Clearly, distribution of arrays affects what data need to be passed by messages between processors. Generally speaking, optimizing for data layout and minimizing communication overhead are two inter-dependent activities. We use a straightforward approach here by considering a few often-used, standard data layouts for programs with regular data structure, called *standard partition strategies*, which include ***block partition***, ***strip partition***, and *interleaving partition* (Figure 3). In each strategy, an index domain is partitioned into more or less equal-size sub-domains. Strategies differ in the resulting shape and size (the granularity) of the partitioned sub-domains. In the following, we call a dimension of an index domain *spatial* if the subdomain along that dimension is mapped to different processors; otherwise we call it *temporal*. In Figure 3, the horizontal dimension of the index domain is partitioned in all of the four cases, while the vertical dimension is partitioned only in the second and fourth cases.

Partitioned Shared-Memory Program Applying a partition strategy to a shared-memory program results in a *partitioned* shared-memory program. For example, suppose that the forall loops in the program in Figure 2 are partitioned over a two-dimensional network of processors such that processor (x,y) is responsible for a sub-domain $[I_l(x,y)..I_u(x,y)] \times [J_l(x,y)..J_u(x,y)]$ of (i,j). Figure 4 shows the partitioned version of the program.

```
for ($t$ : [0..$n$]) {
    forall (($i,j$) : [$I_l(x,y)..I_u(x,y)$] × [$J_l(x,y)..J_u(x,y)$])
        $b(i,j,t)$ = if ($j = t$) →
                      $\backslash + \{a(i,x,t-1) \mid 1 \le x \le n\}$;
                  else → 1;
    forall (($i,j$) : [$I_l(x,y)..I_u(x,y)$] × [$J_l(x,y)..J_u(x,y)$])
        $a(i,j,t)$ = if ($t = 0$) → 0;
                  else if ($i = b(0,t,t)$) → $a(i,j,t-1)$;
                  else → $b(i,t,t)$;
}
```

Figure 4: A Partitioned Shared-Memory Program

Although the partitioned program is to be executed sequentially by each individual processor, we still use **forall** to represent sections of parallel loops derived from a **forall** loop. The reason here is that knowing the absence of data dependence of a **forall** loop helps in scheduling communication in such a way that better performance can be obtained.

Canonical Form of Loop Nests For reasons which will become clear later, we want to perform loop interchange (see [16] for a survey on this topic) on a partitioned shared-memory program to obtain a *canonical form* where all **for** loops appear outside of **forall** loops. Note that there is a top-level **forall** loop that ranges over the processors and is invisible from each processor's standpoint. The **forall** loops in a canonical form reveal the maximum granularity between synchronization points between iterations.

The validity of the interchange of two adjacent **for** and **forall** loops can be easily shown for the partitioned shared-memory program as defined (due to the single assignment form of the array assignment statements). Suppose that we are given a loop nest and its interchanged version, as shown below:

```
for ($i : D_1$)                 forall ($j : D_2$)
  forall ($j : D_2$)              for ($i : D_1$)
    $S(i,j)$;                       $S(i,j)$;
```

Consider two arbitrary instances of the loop body, $S(i_1, j_1)$ and $S(i_2, j_2)$, where $i_1 < i_2$. In the first nest, $S(i_1, j_1)$ will be executed first, since i is the outer loop. In the second nest, the same is also true, since the **for** loop appears in every iteration of the **forall** loop, and it forces synchronization between the iterations. For two instances with the same i value, $S(i, j_1)$ and $S(i, j_2)$, the execution order does not matter, since there is no data dependence between them.

Spatial Reference Patterns As described above, a given partition of the index domain of a loop nest also separates the domain coordinates into two kinds: spatial and temporal. Clearly, for the purpose of determining the form of interprocessor communication, it is sufficient to consider only the spatial part of a reference pattern (called a *spatial reference pattern*). Using the program and partition strategy in Figure 4 as an example, we can derive the following spatial reference patterns:

$$\begin{aligned}
&\ulcorner a@(i,x) \Rightarrow (i,j) : j = t \text{ and } 1 \leq x \leq n\urcorner,\\
&\ulcorner b@(0,t) \Rightarrow (i,j) : t \neq 0\urcorner,\\
&\ulcorner a@(i,j) \Rightarrow (i,j) : t \neq 0 \text{ and } i = b(0,t,t)\urcorner,\\
&\ulcorner b@(i,t) \Rightarrow (i,j) : t \neq 0 \text{ and } i \neq b(0,t,t)\urcorner.
\end{aligned}$$

Compared with the reference patterns above, we see that different notations are used: in a reference pattern, we use $b(x) \leftarrow a(y)$ to denote that $a(y)$ is needed to compute $b(x)$. By contrast, in a spatial reference pattern, we use $a@y' \Rightarrow x'$, where x' and y' contain only the spatial coordinates to denote that the element of array a in spatial location y' needs to be sent to spatial location x'.

For convenience and in situations where there is no confusion, we will simply call the spatial reference pattern a reference pattern in the rest of this paper.

2.3 Communication Primitives

We define an abstract distributed-memory machine to which shared-memory programs will be compiled. The abstract machine will then be embedded into the target machine. 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 $D = [1..N_1] \times \cdots \times [1..N_n]$.

We select a set of aggregate communication routines[1] defined over D as primitives, as shown in Tables 1 and 2. Let B denote 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 **i**, **s**, **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.

Primitives in Table 1 are called *general primitives*. Those in Table 2 are called *simple primitives*. Each simple primitive describes collective communication which is confined in a single dimension (denoted by index p) of the multi-dimensional grid of the abstract machine. Each simple primitive has a corresponding general primitive, but its data movement is constrained. Tables 1 and 2 can be extended to include more primitives, such as gather, scatter, and shuffle-exchange.

[1]called *collective communication routines* by Fox *et al.*[5], and Johnsson and Ho[6, 7]. They have developed a collection of efficient collective communication routines for hypercube machines, and have shown that programs using these routines are more efficient than those using asynchronous message passing (i.e. individual send and receive pairs) in many scientific and engineering applications.

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

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

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

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

These communication primitives can be implemented as part of a run-time system on a specific target machine. Each primitive uses a routing algorithm that takes advantage of its particular pattern of communication, and is carefully tuned for performance for each specific target machine. From a compiler's point of view, each communication primitive has a unique pattern characteristic that the compiler can identify symbolically, and issue a call to this primitive when a match with a spatial reference pattern is found. Note that a composition of these primitives can generate many more complex communication patterns. Thus a complex spatial reference pattern may be decomposed to match a composition of communication primitives.

Communication Patterns The data movement of each communication primitive is described by a *communication pattern*

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

It is interpreted as follows: Let $(i_1, \ldots, i_n)$ range over index domain D. Variables σ_p and δ_p where $1 \leq p \leq n$ are expressions of indices $i_1, \ldots, i_n$, and γ is a boolean predicate defined over domain D.

The communication pattern 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 a communication pattern. 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 $(i_1, \ldots, i_n)$:

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

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

$$(\delta'_1, \ldots, \delta'_n) = T_1(i_1, \ldots, i_n)$$
$$(\sigma'_1, \ldots, \sigma'_n) = T_2(i_1, \ldots, i_n)$$

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

Both of the special forms are needed in synchronizing some of our communication primitives (this point will be elaborated later). When $(\sigma'_1, \ldots, \sigma'_n)$ and $(\delta'_1, \ldots, \delta'_n)$ are linear expressions of the indices, is is possible to determine symbolically the sender's and receiver's forms. But in general a compiler would not be able to do so. 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 whenever needed.

2.4 Selecting Communication Routines

We have developed an algorithm for matching reference patterns with communication primitives[10]. The algorithm applies to the reference patterns of a shared-memory program together with a partition strategy selected *a priori*, and generates communication primitives that implement the data movement of the reference patterns. The algorithm works as follows. It first identifies the symbolic characteristic of the reference pattern, e.g. decides whether the data is to be moved from a single point in the domain or from multiple points. It then searches through the list of communication primitives for a matching one. The search is conducted in such a way that if there are multiple matching primitives, the most economical one (based on a communication metric) will be encountered first and will hence be selected. In the case where no matching primitive can be found, the algorithm will break the reference pattern into simpler sub-patterns, and work on each of them recursively. There are many interesting issues in the matching process, such as how to optimize the breakdown of a complex pattern, and how to define a communication metric. Interested readers are referred to [10].

Forced Communication One important issue, however, needs to be pointed out here. The communication routines discussed in Section 2.3 are all defined over regular index domains. By regular, we mean that a domain is either an interval or a Cartesian product of intervals. But reference patterns of a partitioned shared-memory program may have *associated* predicates, which means the required data movement may only occur in a selected part of a regular domain. For example, consider the following reference pattern

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

defined over a two-dimensional spatial domain D. The required data movement is confined in a triangular sub-domain of D, specified by the predicate $i > j$. As far as matching communication routines are concerned, such predicates are ignored, i.e., those processors which do not need data are forced to participate in the aggregate communication. Thus the above reference pattern will be matched with the communication primitive

$$\textsf{One-All-Broadcast}(D, (2,3), a).$$

As a result, some processors will be getting data they do not need. In the implementation, these extraneous data are discarded as soon as they arrive at the processor in order to free up the buffer space of the processor.

The matching algorithm generates only calls to communication primitives. The issue of where and under what conditions these primitives should be invoked in the target program is not addressed. The remainder of this paper is devoted to these issues.

2.5 Message Aggregation

In most cases, a sub-domain of elements will be mapped to each processor as the result of partitioning an index domain. Due to such partitioning, the actual code for communication would be more complicated than what is shown above. For instance, suppose in the above example the index domain D is partitioned over a two-dimensional network of processors, such that a processor (x,y) is responsible for a sub-domain $E = [I_l(x,y)..I_u(x,y)] \times [J_l(x,y)..J_u(x,y)]$. The parameters to the communication routine must contain this domain information, in addition to the information related to data allocation within a processor and how the abstract machine is embedded in the physical network. For instance, the indices $(2,3)$ must be translated to the node address of the processor network. The pointer to the data, in some cases, is to a local buffer instead of the array itself. (In general, there are also other housekeeping chores such as loading and unloading buffers, discarding unwanted data, etc.) Thus the actual code for the above example would be of the form

$$\begin{array}{l} \langle \text{pre-comm statements} \rangle; \\ \textsf{One-All-Broadcast}(E, \textsf{idx\_to\_pid}(2,3), BUF\_a); \\ \langle \text{post-comm statements} \rangle; \end{array}$$

In the rest of this paper, to keep the presentation of the key ideas clear, aggregations are not shown explicitly in the target program with communication calls.

3 Synchronizing Communication Primitives

For the purpose of discussing message synchronization, communication primitives can be classified into two groups: Group A consists of primitives that are implemented by pairs of send and receive commands, such as Copy, Shift, and Single-Send-Receive. To implement a primitive in this group both the sender's and the receiver's forms of a reference pattern are needed. The synchronization issue for such a primitive is to make sure that the parameters to the send and receive commands are correctly set up so that they are matched one-to-one when the primitive is invoked. Group B consists of primitives that are implemented by building message combining trees among a pre-defined set of processors (such as a row or a column), for example Spread, Reduce, One-All-Broadcast and All-All-Broadcast. The synchronization issue for a Group B primitive is to make sure that the primitive is invoked under the same condition on all the participating processors, so that they will all reach whatever communication commands that implement the primitive. The actual time a processor reaches the communication commands however, may differ from processor to processor on an asynchronous multiprocessor like the iPSC/2, since there is no global clock. We illustrate each case with examples.

3.1 Synchronizing Group A Primitives

For a Group A primitive, the critical issue is to derive both the sender's form and the receiver's form of the communication pattern corresponding to the communication primitive. Due to our selection, every primitive in Group A corresponds to a communication pattern that can be symbolically transformed into both sender's and receiver's forms by a compiler. We show the synchronization process through an example.

Example Given the following shared-memory program,

```
for ($t : [0..n]$)
  forall ($(i,j) : [1..n] \times [1..n]$)
    $a(i,j,t) =$ if ($t > 1$) $\rightarrow b(i+1, j-2, t)$;
                 else $\rightarrow a(i,j,t-1)$;
```

One spatial reference pattern derived is

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

and is matched with a Uniform-Shift$(E, (-1,2), b)$, where $E = [1..n] \times [1..n]$. The target program would look like

```
Program for processor $(x,y)$ :
for ($t : [0..n]$) {
  if ($t > 1$)
    Uniform-Shift($E, (-1,2), b$);
forall $((i,j) : [I_l(x,y)..I_u(x,y)] \times [J_l(x,y)..J_u(x,y)])$
   if $(t > 1)$
      $a[i][j] = b[i+1][j-2]$;
}
```

The aggregate communication routine Uniform-Shift$(E, (-1,2), b)$ is actually implemented by the following pair of **send** and **receive** commands[2] in every processor participating in the aggregate communication.

$$\text{send}((-1,2), b);$$
$$\text{receive}((1,-2), b).$$

Notice 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. Since the source program contains the receiver's form already, the compiler only needs to derive the sender's form

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

The derivation involves a matrix inversion. Under the condition that the matrix is full rank (which is met by all Uniform-Shift cases), the inversion can be performed symbolically, and can be derived automatically.

3.2 Synchronizing Group B Primitives

The implementation of a Group B primitive relies on building dynamic broadcasting (or reduction) trees among a pre-defined set of processors. Take one-all-broadcast as an example. In the first step, the processor which holds the source data sends the data to one of its neighbors; in the second step, the two processors send the data to two new neighboring processors; in the third step, four processors send to another four processors; and so on. If there are n processors, the broadcasting can be done in $\lceil \log n \rceil$ steps. Efficient algorithms for synchronizing and coordinating messages to implement a broadcasting or a reduction on hypercube machines have been developed (e.g. [6, 7]). The implementation of a Group B routine assumes a set of participating processors specified by domain parameter D and uses a pre-determined structure on these processors to accomplish the communication.

As we mentioned in Section 2.4, certain predicates in a reference pattern are ignored in the process of matching communication routines. The major issue in synchronizing group B communication routines is to determine exactly the type of predicates that should be ignored, i.e. the domain parameter D contains all the participating processors. For each and every one of such participating processors, whether forced or not, a call to the communication routine must be issued.

A Boolean predicate P of a reference pattern is said to be *space-invariant* with respect to a domain partition strategy if the value of P is invariant with respect to the values of

[2]ignoring the issues of the address translation and message aggregation for the moment.

the spatial indices, otherwise, it is said to be ***space-variant.*** For example, suppose indices (i, j, t) are defined over domain $D_1 \times D_2 \times D_3$ where D_1 and D_2 are partitioned. Then predicate $i > j$ is space-variant since for different values of i and j, $i > j$ can have different values. On the other hand, predicate $t > 1$ is a space-invariant predicate.

When a space-invariant predicate is in conjunction with a space-variant predicate, as in $(t > 1)$ and $(i > j)$, it is lifted outside of the call to the communication routine while the space-variant predicate is ignored. Since the space-invariant predicate will evaluate to the same value for all participating processors, all, or none, of the processors will participate in the communication, as shown in the following example:

Example From the shared-memory program

```
for $(t : [0..n])$
    forall $((i, j) : [1..n] \times [1..n])$
        $a(i, j, t) =$ if $(t > 1)$ and $(i > j) \rightarrow b(3, j, t)$;
                      else $\rightarrow a(i, j, t - 1)$;
```

the compiler derives a spatial reference pattern

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

which is then matched with a Spread$(E, 1, 3, b)$, where $E = [1..n] \times [1..n]$, by ignoring the predicate $i > j$. The corresponding target code looks like

```
Program for processor $(x, y)$ :
for $(t : [0..n])$ {
    if $(t > 1)$ {
        Spread$(E, 1, 3, b)$;
        (discard data if forced);
        }
    forall $((i, j) : [I_l(x, y)..I_u(x, y)] \times [J_l(x, y)..J_u(x, y)])$
        if $((t > 1)$ && $(i > j))$
            $a[i][j] = b[3][j]$;
    }
```

It is worth noting that the data sent to any processor that is forced to participate are discarded as soon as they arrive to free up the buffer space at the processor.

3.3 Computing the Inverse of a Reference Pattern

Synchronizing Group A primitives depends on computing the inverse of a reference pattern. In case the inverse is not computable at compile-time, our current solution is to use a Group

B primitive instead. Such a primitive requires every member in a well-defined subset of the network of processors (such as a column) to participate, including those who do not really need the data. Since every processor is able to identify its position in the network, synchronization can be easily achieved in this case.

However, this simple solution may incur high performance cost in some cases. For example, suppose the reference pattern (over domain $D = [1..n] \times [1..n]$)

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

contains an indirect reference $c(i,j)$ whose value cannot be determined at compile-time. Our pattern matching algorithm would match it with a Spread, but a Copy would suffice if $c(i,j)$ was known to be constant at compile-time.

Asynchronous Communication One 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 ($i = 2$) {
    ⟨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:
$T(i,j) = (c(i,j),j) = \{(i \text{ div } j, j)\}$
$T\_inv(i,j) =$ 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 ($i = 2$)
   ⟨Send msg to processor idx_to_pid($T(i, j)$)⟩;
if (p ∈ {idx_to_pid($T\_inv(i, j)$)})
   ⟨Receive msg from processor idx_to_pid($2, j$)⟩;
```

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

4 Scheduling Communication Primitives

In this section, we consider the problem of the location where a communication primitive should appear in the target program. We first introduce the notions of *maximum granularity*, *computation segment* and *communication segment*. We then define *communication window*, specifying the range in which a communication primitive can legally appear. We then discuss the various trade-offs in choosing the best placement for a call to the communication routine.

4.1 Computation Segments

Assume that all the loop nests in a partitioned shared-memory program are in the canonical form as defined in Section 2. Given an n-level loop nest defined over domain $D = D_1 \times D_2 \cdots \times D_n$, then the first k levels are for loops while the inner loops from level $k+1$ to n are forall loops. For the purpose of our discussion, we assume, in addition, that the inner forall loops are distributed over the array assignment statements in the body of the loop nest. Such loop distribution will not be done actually to generate the target code, in other words, we perform loop distribution over the array assignment statements now and then perform the inverse operation after we have determined the placement of the communication calls.

We use the following notation

$$\mathsf{S}(a, E, i_1, \ldots, i_k)$$

to denote the inner loop nest from level $k+1$ to n over an assignment statement of array a and call it a *computation segment* for a, where $E = D_{k+1} \times \cdots \times D_n$, and $(i_1, \ldots, i_k)$ are the index tuple of the first k for loops. An instance of $(i_1, \ldots, i_k)$ is called a *time-stamp* of a computation segment. An example is shown in Table 3.

Since the iterations of forall in a computation segment are fully independent, no communication has to occur inside each computation segment if the data it needs are fetched before it begins execution. However, between two different computation segments, there might be data dependences, hence communication might be needed. Thus a computation segment represents maximal granularity of the loop nest under consideration.

Comp. Segment	Original Code
$\mathsf{S}(b,E,t)$	forall $((i,j) : D_2 \times D_3)$ $b(i,j,t) =$ if $(j = t) \rightarrow$ $\backslash + \{a(i,x,t-1) \mid 1 \le x \le n\}$; else $\rightarrow$ 1;

$\mathsf{S}(a,E,t)$	forall $((i,j) : D_2 \times D_3)$
	$a(i,j,t) =$ if $(t = 0) \rightarrow 0$;
	else if $(i = b(0,t,t)) \rightarrow a(t,j,t-1)$;
	else $\rightarrow b(i,t,t)$;

Table 3: Computation segments for the example program

Comm. Segment	Corresponding Code
$\mathsf{C}(a,P_1,E,t-1)$	if $(t > 0)$ Reduce$(E,2,t,a,+)$;
$\mathsf{C}(b,P_2,E,t)$	if $(t > 0)$ One-All-Broadcast$(E,(0,t),a)$;
$\mathsf{C}(b,P_4,E,t)$	if $(t > 0)$ Spread$(E,2,t,b)$;

Table 4: Communication segments for the example program

4.2 Communication Segment

Communication in a partitioned program can be represented in a similar way. Recall that in the canonical form of a partitioned shared memory program, the first k domain coordinates are temporal. Given a reference pattern

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

where $E = D_{k+1} \times \cdots \times D_n$, and $(\delta_1,\ldots,\delta_k)$ is the time-stamp of the segment. We use the following notation

$$\mathsf{C}(b,P,E,\delta_1,\ldots,\delta_k)$$

to denote the *communication segment* generated for P, including the calls to primitive routines, statements for loading message buffers, etc.. An example of communication segments is given in Table 4. For simplicity, we will leave out the code for the usual housekeeping activities before and after the call to the communication routines.

Granularity of Segments Consider the case where a processor computes some values and then sends them to other processors. The processor can either compute and send one value at a time or it can compute many values first and then send. The difference is the memory usage (data need to be stored if they don't get sent) and the communication overhead (more frequent, small messages incur more fixed cost such as message startup time and time for the calls to the operating system kernel. With respect to the above example, the inner loop nests can all be broken down to smaller segments. For instance, the following three loop nests are possible decompositions of computation segment $\mathsf{S}(b, E, t)$:

```
forall ($i \in D_2$)          forall ($j \in D_3$)          forall ($(i,j) \in D_2 \times D_3$)
    $\mathsf{S}(b, D_3, t, i)$;        $\mathsf{S}(b, D_2, t, i)$;        $\mathsf{S}(b, \mathsf{nil}, t, i, j)$;
```

Segments $\mathsf{S}(b, D_3, t, i)$, $\mathsf{S}(b, D_2, t, i)$, and $\mathsf{S}(b, \mathsf{nil}, t, i, j)$ are all of smaller granularity than $\mathsf{S}(b, E, t)$.

Selecting appropriate granularity of computation segments and consequently the size and frequency of message-passing requires cost-driven optimization based on both the target machine parameters and cost estimation of the program. The formulation here provides the framework for doing so. In what follows, we present our method using computation segments and communication segments both at their maximum granularity. The optimization issue can be addressed separately and the following method works for any given granularity.

4.3 Scheduling Communication Segments

The main issue of scheduling is to place a communication segment in the appropriate location in a sequence of computation segments. The potential locations for communication segments are points between the computation segments. However, not every such point is legal. A communication should happen no earlier than the time when the transmitted data is ready and no later than the time when the transmitted data is used. We define the notion of a *communication window* specifying the range in which a communication segment must be placed and can be placed anywhere legally.

Communication Window Given a communication segment, $\mathsf{C}(b, P, E, \delta_1, \ldots, \delta_k)$, the *top* of the window is the point immediately after the last of the set of computation segments including $\mathsf{S}(b, E, \delta_1, \ldots, \delta_k)$ and those which compute the indirect array references occurring in $\ulcorner\delta_1, \ldots, \delta_n\urcorner$. The *bottom* of the window is the point immediately before the earliest of the set of computation segments in which $a(i_1, \ldots, i_n)$ is used.

For instance, given reference pattern

$$P : \ulcorner a(i,j,t) \leftarrow b(i, c(i,j,t), t-1)\urcorner$$

the top of the communication window for communication segment $\mathsf{C}(b, P, E, t-1)$ is the point immediately after computation segments $\mathsf{S}(b, E, t-1)$ and $\mathsf{S}(c, E, t)$. Since the time-stamp of $\mathsf{S}(c, E, t)$ is newer, the top is the point right after it.

For the communication segments in Table 4, the compiler would derive the following communication windows:

```
for ($t : D_1$) {
    S($b, E, t$);
    C($b, P_2, E, t$);
    C($b, P_4, E, t$);
    S($a, E, t$);
    C($a, P_1, E, t$);
}
```

(a) Simple strategy 1

```
for ($t : D_1$) {
    C($a, P_1, E, t-1$);
    S($b, E, t$);
    C($b, P_2, E, t$);
    C($b, P_4, E, t$);
    S($a, E, t$);
}
```

(b) Simple strategy 2

Figure 5: A partitioned program with explicit communication

Comm. Segment	*Communication Window*	
C$(a, P_1, E, t-1)$	S$(a, E, t-1)$	S(b, E, t)
C(b, P_2, E, t)	S(b, E, t)	S(a, E, t)
C(b, P_4, E, t)	S(b, E, t)	S(a, E, t)

A communication window specifies the range within which a communication segment can be inserted between any two computation segments. Notice that a communication window may cross loop iterations, which is the consequence of cross-iteration dependencies.

Issues of Scheduling Given a communication window, what is the best placement of a communication segment? Again, this problem involves trade-offs in communication cost, storage use, balanced network flow, etc., and needs to be answered by cost-driven optimizations based on a model of target machine characteristics. We describe a scenario here to illustrate the trade-off between processor idling time versus network message traffic.

A processor waiting for a message cannot progress with its own program unless the message is received. So the earlier the message is sent out by the processor which produces the required data, the better. On the other hand, if the production and consumption of the messages are too much off-balance, messages may start to saturate the network. Profiling and estimating computation time, message size, etc., should provide clues for where the communication segment should be placed to maintain a smooth flow of message traffic.

Consequently, we want to put communication segments in places where message aggregation can be performed. However, larger messages also mean larger buffers. For large applications, this could cause a shortage of memory. Again, to balance the issue, cost estimation and profiling are needed.

Simple Strategies Here we propose two very simple default strategies which do not take cost into consideration:

1. Place a communication segment at the top of its communication window.
2. Place a communication segment at the top of its communication window if the top is a computation segment with the same time-stamp; otherwise place it immediately before the first computation segment in the window that has the same time-stamp

Using the first strategy on the program shown in Figure 4, we obtain a schedule shown in Figure 5(a); using the second strategy, the result is shown in Figure 5(b).

The first strategy has an advantage in controlling granularity, since a *strip-mining* can be applied to a computation segment and the adjacent communication segments, while the second strategy is slightly easier to generate code for, since there are no cross-iteration dependencies between computation segments and communication segments.

5 Correctness of Communication Synthesis

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

The target code generated by the compiler consists of a host program and a node program. The single node program is in the so-called SPMD (single program multiple data) style. As we discussed in the previous section, the node program consists of a sequence of perfectly nested loop nests, each with a sequence of computation and communication segments as its loop body.

Assume that each computation segment is a single-entry single-exit segment (i.e. there are no **goto** or **break** statements), and is generated by the compiler based on semantics-preserving transformations which do not introduce deadlock. Provided that the source program is correct and all the data a computation segment requires are available, then its execution always terminates. Therefore, we only need to check the behavior of communication segments which ensure the availability of the data required by the computation segments.

We prove that for each loop nest, the communication segments so generated do not introduce deadlocks by induction on the sequence of communication segments.

The induction hypothesis is that up to the $N-1'th$ communication segment, the program is deadlock-free, i.e., all processors have reached the beginning of the Nth communication segment since any computation segments in between them terminates eventually.

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 primitive. 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 [3]. (1) Since the send and receive pair of the primitive is arranged as a non-blocking send followed by a blocking receive (Section 3.1), 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 3.2), every receive statement will terminate with received data. Therefore, the Nth communication segment eventually terminates, and so the program also terminates.

6 Concluding Remark

In this paper we considered generating a program with explicit communication commands from a program for shared-memory multiprocessors based on a set of standard data partition strategies. In particular, we discussed the problem of coordinating interprocessor communication. The communication synthesis approach has been implemented in the experimental Crystal compiler for the iPSC/2 and the NCUBE [9]. The compiler takes a Crystal program as input and generates C code with calls to communication primitives as output. All the communication primitives discussed in this paper, except for Affine-Transform, have been implemented on the iPSC/2. The preliminary results we have obtained on a few benchmark programs, including matrix multiplication [3], Gaussian elimination with partial pivoting [4], and a financial application [2], have shown that the performance of the compiler-generated code is within a factor of 1.7 - 2.8 of that of the their corresponding hand-crafted code.

When measured independently, an individual aggregate communication routine is far more efficient than corresponding send and receive pairs with the current generation of hypercube multiprocessors. As the design of communication network advances, however, the cost function may change over time.

As shown by Chuck Seitz, the communication system of the Mosaic system has extremely low latency for send and receive commands. In addition, the so called worm-hole style routing makes the communication done directly by the router much more competitive than user-programmed multi-phase routines due to their need to access the processor memory. One interesting architectural design question is whether the functionality of the routing system should be broadened to do special aggregate communication, taking advantage of the highly correlated communication patterns and the smart algorithms to do them.

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

References

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

[2] Marina Chen, Young-il Choo, Erik DeBenedictis, Jingke Li, and Janet Wu. Speedup of a financial application using the crystal compiler for hypercubes. Technical Report YALEU/DCS/TR-673, Dept. of Computer Science, Yale University, January 1989.

[3] Marina Chen, Young-il Choo, and Jingke Li. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing*, 1(2):171–207, July 1988.

[4] Marina Chen, Young-il Choo, and Jingke Li. Theory and pragmatics of compiling efficient parallel code. Technical Report YALEU/DCS/TR-760, Dept. of Computer Science, Yale University, December 1989.

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

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

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

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

[9] Jingke Li. *Compiling Crystal for Hypercube Machines.* PhD thesis, Yale University, (Expected Dec. 1990).

[10] Jingke Li and Marina Chen. Generating explicit communication from shared-memory program references. In *Supercomputing 90*, New York, NY, Nov. 1990.

[11] Jingke Li and Marina Chen. Index domain alignment: Minimizing cost of cross-reference between distributed arrays. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Computation*, College Park, Maryland, Oct. 1990.

[12] Michael J. Quinn, Philip J. Hatcher, and J.V. Rosendale. Compiling C* programs for a hypercube multicomputer. In *ACM/SIGPLAN PPEALS 1988*, New Haven, Connecticut, July 1988.

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

[14] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *SIGPLAN'89 Conference on Programming Language Design and Implementation*, June 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, Cambridge, MA, 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.