Communication Optimization and Code Generation for Distributed Memory Machines

Saman P. Amarasinghe and Monica S. Lam
Computer Systems Laboratory
Stanford University, CA 94305

Abstract

This paper presents several algorithms to solve code generation and optimization problems specific to machines with distributed address spaces. Given a description of how the computation is to be partitioned across the processors in a machine, our algorithms produce an SPMD (single program multiple data) program to be run on each processor. Our compiler generates the necessary receive and send instructions, optimizes the communication by eliminating redundant communication and aggregating small messages into large messages, allocates space locally on each processor, and translates global data addresses to local addresses.

Our techniques are based on an exact data-flow analysis on individual array element accesses. Unlike data dependence analysis, this analysis determines if two dynamic instances refer to the same value, and not just to the same location. Using this information, our compiler can handle more flexible data decompositions and find more opportunities for communication optimization than systems based on data dependence analysis.

Our technique is based on a uniform framework, where data decompositions, computation decompositions and the data flow information are all represented as systems of linear inequalities. We show that the problems of communication code generation, local memory management, message aggregation and redundant data communication elimination can all be solved by projecting polyhedra represented by sets of inequalities onto lower dimensional spaces.

1 Introduction

Compiling to machines with a distributed address space can be separated into two major phases. The first phase determines how to decompose the computation and data across the processors. The goal is not just to parallelize the applications, but also to ensure that the programs have data locality so as to minimize the communication overhead. This step is common to all large-scale parallel machines with non uniform memory access times. If the machine has a distributed address space, then the compiler is faced with the additional problem of managing the memory and communication explicitly. The second phase of the compilation is to generate the code so that each processor will execute its allotted computation and communicate correctly and efficiently. These algorithms can also be useful to enhance the memory system performance of machines with a shared address space.

Many systems being developed rely on the user to supply the data decompositions. Languages such as High Performance FORTRAN[24], FORTRAN-D[16] and Vienna FORTRAN[8] allow the programmer to annotate the sequential program with data decompositions. The compiler is responsible for generating the computation decomposition and an SPMD (Single Program Multiple Data) program with explicit communication.

We are developing a compiler system that automatically parallelizes a sequential program for shared and distributed memory machines. The input language to our compiler is sequential FORTRAN-77, although many of the techniques developed are also applicable to optimizations within and especially across FORTRAN-90 statements[14].

Our data and computation decomposition phase tries to maximize parallelism and minimize communication[3]. If all the available parallelism cannot be exploited without communication, this phase first tries to trade off excess degrees of parallelism to eliminate communication. The algorithm understands “doacross” parallelism, where processors are organized as a pipeline and they synchronize and communicate during the course of the computation. The algorithm will choose to use the “doacross” form of parallelism if such a choice minimizes the overall communication cost. Finally, if necessary, the algorithm will reorganize the data dynamically; it tries to insert data reorganizations into the least frequently executed parts of the program.

The decomposition phase generates a full specification of the computation decompositions. It also specifies the data decompositions before and after major data reorganizations; it does not generate data decompositions for those sections of code using “doacross” form of parallelism, where data are often allocated to different processors at different times.

Data reorganizations such as matrix transposes are implemented using collective communication routines[18]. The algorithms presented in this paper focus on generating the code and communication between reorganizations. Within each region of code requiring no major data reorganization, there may still be fine-grained communication. Sections of the arrays may be replicated and allocated to different processors at different times. The given data decompositions are used as specifications for the initial and final layouts to each code region. The algorithm uses the computation decomposition to guide the local memory management process. Within each region, our algorithm may change the data layout according to the demands of the given computation decompositions.

The data and computation decomposition schemes that a user can specify in languages like High Performance FORTRAN on dense matrices is a subset of those generated by our compiler. Thus, the techniques described in this paper are also applicable to generating code from user-supplied data decompositions.

In the Proceedings of The ACM SIGPLAN ’93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, June 1993.
Given a computation decomposition, and an initial and a final data decomposition, the techniques described in this paper automatically produce an SPMD (single program multiple data) program to be run on each processor. Our compiler generates the necessary receive and send instructions, optimizes the communication by eliminating redundant communication and aggregating small messages into large messages, allocates space locally on each processor and translates global data addresses to local addresses.

This paper presents three main results. First, we propose a value-centric approach to deriving the necessary communication for machines with a distributed address space. Previous approaches are location-centric: communication is derived from data decompositions; optimizations are performed using data dependence analysis, an analysis that determines if accesses may refer to the same location. We derive our code generation from computation decompositions using a data-flow analysis technique that is based on values instead of locations. This approach allows for a more general set of data and computation decompositions and allows for more communication optimizations.

Second, we have developed a mathematical model for code generation which is applicable to both the location- and value-centric approaches. We represent data decompositions, computations decompositions, and communication as systems of linear inequalities. We show that the various code generation and communication optimization problems can be solved by projecting the polyhedra represented by systems of inequalities onto lower dimensional spaces.

Third, we have developed several communication optimizations within the value-oriented framework. These optimizations include eliminating redundant messages, aggregating messages, and hiding the communication latency by overlapping the communication with computation. These optimizations are essential to achieving an acceptable performance on distributed memory machines.

The organization of our paper is as follows. In Section 2, we describe the conventional way of generating communication code from user-specified data decompositions and some of the limitations of the approach. In Section 3, we describe our value-centric approach to communication generation. We formally describe the domain of our technique and present a mathematical representation of the problem in Section 4. We describe our code generation technique and our communication optimizations in Sections 5 and 6. We finally close with a detailed example to illustrate the entire code generation process.

2 A Location-Centric Approach

Many of the existing compilers developed for distributed memory machines have a similar basic approach in how they generate code from user-specified data decompositions. The FORTRAN-D compiler appears to be the most mature; it employs techniques such as data dependence analysis and regular array sections for communication optimizations, and substantial experimentation results have been reported. In the following, we first review the techniques used in the FORTRAN-D compiler. We then discuss some of the limitations of this approach to motivate the need for more advanced analysis and optimization.

2.1 The Basic Technique

For simplicity, in the following discussion we assume that there is only one loop nest which contains one read access and one write access to the same array. The argument obviously holds for the general case with multiple loops and arrays. There are three domains that are manipulated in the compilation process: the iteration space $\mathcal{I}$, the array elements space $\mathcal{A}$, and the processor space $\mathcal{P}$. Each array access function in the source program specifies the data used by each iteration in the loop. That is, each read or write access function, denoted by $f_r, f_w: \mathcal{I} \rightarrow \mathcal{A}$ respectively, maps an iteration to the array indices of the data read or written. The user-specified data decomposition $D: \mathcal{A} \rightarrow \mathcal{P}$ maps each array location to a processor. From the read and write access functions and the data decomposition, the compiler automatically derives the computation decomposition $C: \mathcal{I} \rightarrow \mathcal{P}$ which maps each iteration in the loop to a processor.

To derive the computation decomposition, the compiler applies the owner-computes rule: each assignment statement is performed by the processor that owns the data. Therefore, given a write access function $f_w$ and a data decomposition $D$, the computation decomposition is $C = D f_w$. Under the owner-computes rule, no communication is needed to implement the write accesses. Communication is needed for a read access in iteration $i$ if the data read is resident on a different processor, i.e., $C i \neq D f_w i$. The relationships between iteration space, array space, and the processor space are shown in Figure 1(a); processor $p_r$ receives data from processor $p_s$ if $p_r \neq p_s$.

![Figure 1: Different approaches to code generation for distributed memory machines.](image)

To minimize the communication cost, the compiler tries to maximize the intervals between communication. All the data needed within the interval are sent in one message. This optimization is based on data dependence analysis. Two accesses are data dependent if one of the accesses is a write operation, and if they may refer to the same location. The dependence is carried at level $k$ if the dynamic instances accessing the same location belong to the same iteration of the first $k − 1$ outermost loops but not the $k$th loop. The maximum level of a dependence between two references is simply the maximum loop nest level that carries a dependence between the references. If the maximum depth of all dependences involving a read access is $k$, the compiler needs to communicate only once in each iteration of the $k$th loop. Thus, the maximum depth information is useful for reducing the communication frequency. All the data accessed within the interval requiring communication are summarized by a regular section description. In this way, the same data used multiple times within the interval are only transferred once.

In summary, this approach deduces the computation decompositions from the user-specified data decompositions, using the owner-computes rule; it uses data dependence analysis to reduce the number of messages; finally, it uses the concept of regular sections to reduce redundant data transfers.

2.2 Limitations of Existing Approaches

While the approach described above can handle many regular dense matrix computations, there are still many aspects in which this approach can be improved.
2.2.1 Data Decompositions and the Owner-computes Rule

Implicit in the use of the owner-computes rule is the assumption that iterations writing to the same location must be executed by the same processor. This is unnecessarily restrictive. Consider the following example:

```plaintext
for i = 0 to n do
  for j = 1 to n do
    X[i, 0] = X[i, 0] + X[i, j]
```

Suppose the array X has been distributed as blocks of columns to facilitate the parallelization of other phases. The best way to parallelize this code is to pipeline the execution of the second loop. That is, the variable X[i,0] passes from processor to processor, and each processor contributes the sum of its section of the rth row to the value of X[i,0]. It is easy to express the computation decomposition to a unique address. Consider the following example:

```plaintext
for i = 0 to n do
  for j = 1 to n do
    X[i, 0] = 1/3 * (X[i] + X[i-1] + X[i+1])
```

A typical parallelization is to partition X into contiguous blocks, and assign to each processor the computation of one of these blocks. Since the range of data read extends beyond the block of data written, the borders of the blocks are often replicated on adjacent processors. Such data decompositions, however, would render the “owner-computes” rule inoperative, since the written data are replicated on multiple processors.

The use of the owner-computes rule plays a major role in simplifying the generation of communication code. The sender of any data is always the processor that owns the data. If we relax the owner-computes rule, deciding which processor has the desired value at any one time is much harder. If we were to replicate the written data, or move the written data dynamically, or choose not to compute at the owner’s site, we would not be able to just derive the sender of the data from the data decomposition.

2.2.2 Data Dependence Analysis

Data dependence analysis is an “alias” analysis. A read operation is dependent on a write operation as long as they may refer to the same location, even if none of the written values are used by any of the read instances[5]. The lack of value information may reduce the opportunities for parallelism and communication optimization. Consider the following example:

```plaintext
for i = 0 to 100 do
  for j = 0 to 100 do
    work[j] = ... (S1)
  for j = 0 to 100 do
    ... = work[j] (S2)
```

The data dependence analyzer does not know if a write operation covers, or kills, an earlier write operation. Data dependence analysis will yield a level 1 dependence between statements S1 and S2 because the locations written in each outer iteration of the loop (work[0:100]) overlap with the locations read (work[0:100]). This information would serialize the outermost loop. On analyzing the flow of the data, it is obvious that data written within each iteration in the outer loop affect only the computation within the same iteration. This outer loop can be parallelized by giving each processor a private copy of the work array. This optimization, known as array privatization, has been shown to be essential to parallelize many real programs successfully[6].

Data dependence analysis does not distinguish between the different instances of the accesses. The analyzer only knows that a dependence exists; it does not know which pairs of instances are dependent. When using data dependence information for communication optimizations, the maximum depth of the dependence determines the communication interval. All the non-local data accessed within the interval must be communicated, even though some of the data may not participate in the data dependence at the same maximum depth. Thus, the same value may be transferred multiple times unnecessarily. Consider the following example:

```plaintext
for i = 0 to n do
  X[i] = ... (S1)

for i = 0 to n do
  Y[i] = Y[i] + X[i - 1] (S2)
```

Suppose matrices X and Y are distributed across the processors by blocks. Using the owner-computes rule, statement S1 in iteration i of the outer loop is executed by the owner of X[i]; iteration j of the inner loop is executed by the owner of Y[j]. The data dependence between S1 and S2 indicates that communication is needed in each iteration of the outermost loop. Processors executing the second loop may need to make one non-local data reference. However, except for the processor executing the first iteration of the inner loop, none of the non-local data accessed has changed its value from the last time the loop was executed. Thus, at most one word needs to be transferred in each iteration of the outermost loop. Data dependence analysis cannot identify the instances that carry the dependence. Thus, if only data dependence information is available, we must conclude that communication is needed for every processor executing the inner loop.

Finally, compilers that use data dependence analysis to aggregate messages are limited by the resolution of the data dependence analysis. Since only the loop level information is available, the communication must necessarily take place at the iteration boundaries of the loop carrying the dependence. Potentially, if the compiler knows the precise iteration that produces the data to be communicated, we can send the data to the receiver as soon as all the data in a message have been generated. Such optimization is significant as it reduces the chances a processor is stalled waiting for data to arrive.

2.2.3 Regular sections

The contents of a message are defined by the regular section descriptor that encompasses all the non-local array elements touched between communication points. This technique may create additional traffic when the set of data accessed cannot be precisely described by a regular section. Consider the following simple example:

```plaintext
for i = 1 to 100 do
  for j = i to 100 do
    ... = A[1000 i + j]
```

Representing the data accessed as a regular section descriptor would increase the amount of communication by a factor of 20.

3 Our Approach

We have been exploring the use of a more sophisticated analysis technique that determines if two accesses refer to the same value, and not just to the same location. In the following, we describe the information obtained from the analysis and show how this can be used to eliminate the limitations of data dependence analysis discussed above.
3.1 Accurate Data-Flow Analysis

Instead of data dependence analysis, our optimizations are based on finding exact data-flow information on individual instances of array accesses. Let us use the simple example in Figure 2 to illustrate our technique.

for $t = 0$ to $T$ do
  for $i = 3$ to $N$ do
    $X[i] = X[i - 3]$

Figure 2: A simple 2-deep loop nest.

Data dependence analysis on this program will produce the dependence vectors $[[+, 3], [0, 3]]$, meaning that the read access in iteration $[t, i]$ may be data dependent on all iterations $[t_w, i_w]$ such that $i_w = i - 3$, and $t_w \leq t$. Perfect data flow analysis, however, is able to determine precisely that the first three iterations of the innermost loop read data defined outside the loop, and the rest of the iterations use the value defined three iterations earlier, i.e., $[t_w, i_w] = [t, i - 3]$.

The problem of finding precise array data-flow information was first formulated by Feautrier[10][11][12]. Feautrier proposed a parametric integer programming algorithm that can find such perfect information in the domain of loop nests where the loop bounds and array indices are affine functions of loop indices. We have developed an algorithm that can find the same precise information for many common cases more efficiently[19][20].

The perfect data-flow information is captured by a representation we called a Last Write Tree (LWT). The LWT for the above example is shown in Figure 3. The leaf M1 represents the first three iterations that read data $X[0:2]$, whose values are not generated by this program. The leaf M2 represents the rest of the iterations where the data read are written three iterations earlier:

$$
\begin{align*}
0 \leq t & \leq T \\
3 \leq i & \leq N \\
i & \geq 6 \\
\text{false} & \text{true} \\
& \text{Dependence Level} \\
M2 & \text{M1} \\
& \text{i} = \text{i}_r - 3 \\
& \text{t} = \text{t}_r
\end{align*}
$$

Figure 3: The Last Write Tree for the example in Figure 2.

For simplicity, our discussion below assumes that there is only one loop nest and one write access in the program. The LWT algorithm can handle multiple write operations and multiple loop nests in general.

The LWT is a function that maps an instance of a read operation to the very write instance that produces the value read, provided such a write exists. We denote a read and a write instance by the values of their loop indices, $i_r$ and $i_w$, respectively. The domain of the LWT function is the set of read iterations $i_r$ that satisfy the constraints imposed by the loop bounds. Each internal node in the tree contains a further constraint on the value of the read instance $i_r$. It partitions the domain of read instances into those satisfying the constraint, represented by its right descendants, and those not satisfying the constraint, represented by its left descendants. We define the context of a node to be the set of read iterations that satisfy the constraints of a node’s ancestors. The tree is constructed such that either all the values read within the context of a leaf are written within the loop, or none of them are. In the former case, the leaf node defines a last-write relation that relates each read instance $i_r$ within the context to the write instance $i_w$ that produces the value read. All the read-write pairs share the same dependence levels. If the instances within a context do not read any value written within the loop, we denote the lack of a writer by $\perp$.

The LWT information differs from the data dependence analysis in two major ways. First, LWTs can distinguish between the different instances of the same array access. For example, the LWT analysis can determine that the first three iterations of the inner loop in Figure 2 have different dependence relationships than all the other iterations. Second, the LWT is a function that specifies precisely the last write instance that generates the value read by a particular read instance. Data dependence analysis, on the other hand, cannot discriminate between writes to the same location.

3.2 Using LWTs in a Distributed Memory Compiler

We have developed an array privatization algorithm based on the LWT analysis[19]. In this algorithm, parallelization is based on only the data-flow dependences generated by the LWT information. If the parallelized loop carries any anti-dependences or output dependences, then privatization is necessary. On machines with a shared address space, the compiler needs to create a private copy of the data for every processor. The private copies may need to be initialized and the final results may be written back to the original array. On machines with a distributed address space, the privatized array in a processor’s local memory is no different from any other data allocated in its local memory. By generating communication only for the data-flow dependences, writes irrelevant to the computation on other processors will not affect the other processors.

Besides enhancing parallelization by enabling optimizations such as array privatization, LWTs are especially useful for efficient code generation for distributed memory machines, as explained below.

3.2.1 Deriving Communication from LWTs

There are two kinds of leaves in an LWT: $\perp$ leaves that do not read any of the values written within the code being analyzed, and leaves that do. In the former case, the compiler can simply load all the non-local data onto a processor before executing any of the code. Given a computation decomposition produced by an earlier compiler phase, the technique to generate the necessary communication code is no different from that used in the location-centric approach.

Communication and computation are more tightly-coupled for the other kind of leaves. The LWT specifies all the pairs of iterations that share a producer and consumer relationship. By applying the computation decomposition function on the related iterations, we can derive the identity of the processors that write and read the same value. If the writer and reader are different processors, then communication is necessary. Our technique is depicted in Figure 1(b).

The data decompositions generated from the earlier compiler phase serve only as interfaces with other sections of the program. In general, we generate the necessary communication from LWTs and computation decompositions, and not data decompositions. Our algorithm will change the data layout when called for by the computation decompositions. Thus, our compiler can support a wider range of data decompositions. Locations written to can be replicated or mapped to different processors across time. Furthermore, our algorithm does not rely on the restriction of the owner-computes rule.
3.2.2 Communication Optimizations

Using the LWT information, we can easily eliminate redundant data transfers. While accessing the same location may require multiple data transfers since the value at the location may or may not have changed, each value needs to be transferred once and only once. Moreover, the perfect producer and consumer information enables the compiler to issue the send immediately after the data are produced and to issue the receive just before the data are used. This maximizes the chances that the communication is overlapped with computation.

Even when a source program appears to be very simple, handling the iterations at the boundary of the loops properly can be tricky. The LWT analysis automatically partitions the read instances into sets that share similar communication characteristics. This partitioning not only makes generating correct code routine, it also enhances optimizations. For example, all iterations of each context in an LWT have the same dependence level, and read instances requiring communication within a loop are separated from those that do not. The uniformity in each of the contexts allows us to develop simple and powerful algorithms to optimize the communication. Without perfect information, the compiler may need to treat all instances uniformly and conservatively, and observe the most stringent of the requirements for all the instances.

4 Problem Formulation

In this section, we formally define the scope of our technique. We show how we can represent all the information useful for communication and computation code generation as sets of linear inequalities. This model and our techniques discussed in Section 5 are useful for both value-centric and location-centric approaches.

4.1 The Problem Domain

The scope of our technique is limited to programs consisting of a set of loop nests, where the bounds of the loop nests are affine expressions of outer loop indices and symbolic constants. The array accesses are also affine functions of loop indices and symbolic constants. Our technique can also handle conditional statements that do not contain any loops. Each assignment within the conditional statement is treated as an unconditional assignment; depending on the outcome of the condition, it assigns to the variable either the newly computed value or the variable’s current value.

We can handle loops of the following form:

\[
\begin{align*}
\text{for } i_1 &= l_1(\vec{v}) \text{ to } h_1(\vec{v}) \text{ do} \\
\text{for } i_2 &= l_2(\vec{v}, i_1) \text{ to } h_2(\vec{v}, i_1) \text{ do} \\
&\quad \text{...} \\
\text{for } i_k &= l_k(\vec{v}, i_1, \ldots, i_{k-1}) \text{ to } h_k(\vec{v}, i_1, \ldots, i_{k-1}) \text{ do}
\end{align*}
\]

where \( \vec{v} \) is a symbolic constant vector (variables unchanged within the loop) and \( l_k, h_k \) are affine functions. The iteration set of \( \vec{v} \) is thus

\[
I = \{ i = (i_1, \ldots, i_n) \in A \mid \forall k = 1, \ldots, n, \quad i_k \geq l_k(\vec{v}, i_1, \ldots, i_{k-1}) \land i_k \leq h_k(\vec{v}, i_1, \ldots, i_{k-1}) \}.
\]

The index set of a data array with dimensions \( u_1, \ldots, u_m \) is

\[
A = \{ \vec{a} = (a_1, \ldots, a_m) \in A \mid \forall k = 1, \ldots, m, \quad 0 \leq a_k < u_k \}.
\]

The read and write access functions, \( \vec{f_r} = (f_{r_1}, \ldots, f_{r_n}) \) and \( \vec{f_w} = (f_{w_1}, \ldots, f_{w_n}) \) are affine functions of the form

\[
f(\vec{v}, i_1, \ldots, i_n) = (a_1, \ldots, a_m),
\]

where \( (i_1, \ldots, i_n) \in I, \ (a_1, \ldots, a_m) \in A \) and \( \vec{v} \) is a symbolic constant vector.

To support cyclic decompositions where data or computation are distributed to processors in a round-robin manner, we introduce the notion of a virtual processor array. The computation and data decompositions map the computation and data to the virtual processor space. Let \( u_1, \ldots, u_k \) be the dimensions of the virtual processor space, the index set of this virtual processor array is thus

\[
P = (\vec{p} = (p_1, \ldots, p_d) \in P \mid \forall k = 1, \ldots, q, \ 0 \leq p_k < u_k).
\]

Our physical processor array has the same number of dimensions as the virtual processor array. Let \( u'_1, \ldots, u'_k \) be the physical processor array dimensions, \( u'_k \leq u_k, \forall k = 1, \ldots, q \). The physical processor index set is

\[
P' = (\vec{p}' = (p'_1, \ldots, p'_d) \in P \mid \forall k = 1, \ldots, q, \ 0 \leq p'_k < u'_k).
\]

The 0th dimension of data elements or loop iterations are distributed across the physical processors in a cyclic manner whenever \( u'_k < u_k \). The mapping from the virtual to the physical processor space, \( \pi: P \rightarrow P' \) is defined as

\[
\pi(\vec{p}) = \vec{p}' \text{ where } \forall k = 1, \ldots, q, \ p'_k = p_k \mod u'_k.
\]

Since it is only in the latter stages of the optimizations will the compiler be operating in the physical processor space, we will simply refer to virtual processors as the processors.

4.2 Data Decompositions

**Definition 1**: The data decomposition relation \( D \) is a set of array element and processor pairs \( (\vec{a}, \vec{p}) \), such that \( (\vec{a}, \vec{p}) \in D \) iff the processor \( \vec{p} \) has a copy of the array element \( \vec{a} \). Data decompositions can be written as

\[
D = \{ (\vec{a}, \vec{p}) \in A \times P \mid U(\vec{a} - \vec{b}) \geq (\vec{b} + B\vec{a}) \vec{p} - \vec{d}_1 \}
\]

where \( U \) is an extended unimodular matrix, \( \vec{b}, \vec{d}_1, \vec{d}_2, \vec{d}_3 \) are integer vectors, \( B \) is an integer matrix and \( \vec{a} \) is a vector of symbolic constants such that \( \vec{b} + B\vec{a} \geq 0 \) and \( \vec{d}_1, \vec{d}_2, \vec{d}_3 \geq 0 \).

Our scope of data decompositions is larger than those typically used in existing distributed memory machine compilers. The matrix \( U \) determines if the array is reversed or skewed. When the array have more dimensions than that of the processor space, the 0 columns of the extended unimodular matrix \( U \) choose the dimensions to be mapped on the same processor. The entire array can be shifted with respect to the processor array using the integer vector \( \vec{b} \). Since the data block size is often a function of the number of processors engaged in the computation, it is useful not to determine the block size at compile time. We can handle some symbolic block sizes of the form \( \vec{b} + B\vec{a} \); the scope of our technique is discussed in Section 5.1. The overlap of array elements between processors is determined by vectors \( \vec{d}_1, \vec{d}_2, \vec{d}_3 \geq 0 \). Figure 4 illustrates how we can use this scheme to describe several common data decompositions. The 2 x 2 grid in each example represents the first 2 x 2 processors in the system; each panel is a picture of the entire data array, and the shaded portion represents the data allocated locally to that specific processor.

4.3 Computation Decompositions

Computation decompositions have a similar scope as data decompositions, except that an iteration can only be mapped onto one processor.

**Definition 2**: The computation decomposition relation \( C \) is a set of iteration and processor pairs \( (i, \vec{p}) \) such that processor \( \vec{p} \)
executes iteration \( i \) iff \((i, \vec{p}) \in C\). Computation decompositions can be written as
\[
C = \{(i, \vec{p}) \in I \times P \mid \langle \vec{b} + \vec{B}\vec{u} \rangle \vec{p} \leq U(i - i) < \langle \vec{b} + \vec{B}\vec{u} \rangle \langle \vec{p} + \vec{1} \rangle \}
\]
where \( U \) is an extended unimodular matrix, \( i, \vec{b} \) are integer vectors, \( B \) is an integer matrix and \( \vec{u} \) is a vector of symbolic constants such that \( \vec{b} + \vec{B}\vec{u} > 0 \).

**Theorem 1:** Assuming that written data are not replicated, the computation decomposition as derived from data decomposition is a set of elements \((i_r, \vec{p}_r, i_s, \vec{p}_s, a) \in I \times P \times I \times P \times A\)
where \((i_r, \vec{p}_r, i_s, \vec{p}_s, a) \in M\)
iff processor \( \vec{p}_r \) needs to send the value in location \( a \) in iteration \( i_s \)
to processor \( \vec{p}_s \) for use in iteration \( i_r \).

### 4.4 Communication

We define the communication between processors formally as a communication set:

**Definition 3:** A communication set \( M \) is a set of elements \((i_r, \vec{p}_r, i_s, \vec{p}_s, a) \in I \times P \times I \times P \times A\),
where \((i_r, \vec{p}_r, i_s, \vec{p}_s, a) \in M\)
iff processor \( \vec{p}_r \) needs to send the value in location \( a \) in iteration \( i_s \)
to processor \( \vec{p}_s \) for use in iteration \( i_r \).

#### 4.4.1 Data Decompositions and the Owner-computes Rule

If we use the owner-computes rule, no communication is necessary for write operations. We use Theorem 2 to find all the necessary communication for each read access within the loop nest. We use the user-specified data decomposition to find the owner of the data read; we use the computation decomposition derived from Theorem 1 to find the processor reading the data; if these two processors are not the same, communication is needed.

**Theorem 2:** The communication set required by the access function \( f \) for a set of iterations \( \tau \) under computation decomposition \( C \) and data decomposition \( D \) is the set of elements \((i_r, \vec{p}_r, i_s, \vec{p}_s, a) \in I \times P \times I \times P \times A\),
where \((i_r, \vec{p}_r) \in C, (\vec{a}, \vec{p}_s) \in D, i_s \in \tau, \vec{a} = f_s(\vec{a}, i_s), i_r = i_r, \vec{p}_r \neq \vec{p}_s\).

#### 4.4.2 Computation Decompositions and Last Write Trees

**Definition 4:** An LWT partitions the iteration set of a loop nest into the contexts of its leaves. If the values read by the iterations in a context \( \tau \subseteq \{1\} \) are written within the loop, then the context has a last-write relation \( \mu \). The last-write relation \( \mu \) of the context \( \tau \) is a set of iteration pairs \((i_r, i_s)\) such that \((i_r, i_s) \in \mu\) iff \( i_r \in \tau \) and \( i_s \in \tau \) is the iteration that generates the value read in iteration \( i_r \).

A context \( \tau \) can be written as \((i \in \tau \mid \vec{a}(\vec{a}, i) \geq 0)\) and a read-write relation \( \mu \) can be written as \((\vec{a}, i_s) \in I \times \vec{a} \geq 0)\) where \( \vec{a} \) and \( \vec{a}' \) are vectors of affine expressions.

It is sometimes necessary to introduce auxiliary variables so that the last-write relations can be represented as linear inequalities. Some of the read-write relations produced by the LWT analysis algorithm are expressed as \( i = \vec{b} \pmod{\alpha} \) or \( i \neq \vec{b} \pmod{\alpha} \), where \( \alpha, \vec{b} \) are integers. We can introduce an auxiliary variable \( u \) and rewrite these relations as \( i = \vec{b} + au \) and \( \alpha u < i - \vec{b} < \alpha u + \alpha \); respectively.

If a context of an LWT reads data written within the loop, then communication is needed only if the iterations sharing the read-write relation are executed by different processors (Theorem 3). If the context of an LWT uses data written outside the loop, then we use the initial data decomposition to determine the owner of the data (Theorem 4). Theorem 4 is similar to Theorem 2, except that the sends can precede the computation of the loop since the values needed are not generated within the loop.

**Theorem 3:** The communication set that satisfies the last-write relation \( \mu \) under computation decomposition \( C \) is the set of elements \((i_r, \vec{p}_r, i_s, \vec{p}_s, a) \in I \times P \times I \times P \times A\)
where \((i_r, i_s) \in \mu, \vec{a} = f_s(\vec{a}, i_s), \vec{p}_s \neq \vec{p}_r\).

**Theorem 4:** The communication set required by an access function \( f \) within a \( \perp \)-context \( \tau \) of an LWT, under computation decomposition \( C \) and an initial data decomposition \( D \), is the set of elements \((i_r, \vec{p}_r, i_s, \vec{p}_s, a) \in I \times P \times I \times P \times A\),
where \((i_r, i_s) \in \{C, (\vec{a}, \vec{p}_s) \in D, i_s \in \tau, \vec{a} = f_s(\vec{a}, i_s), i_s = 0, \vec{p}_s \neq \vec{p}_r\).
Communication decompositions, data decompositions, iteration contexts, access functions and last-write relations can all be expressed as systems of linear inequalities. The \( p_s \neq p_t \) constraint, however, cannot be expressed as a conjunction of inequalities. We break down the inequality into a set of disjunctive conditions. For example, for a one-dimensional processor array, the constraint \( p_s \neq p_t \) is represented by \( p_s > p_t \lor p_s < p_t \). We represent the necessary communication as a set of communication sets, with each one satisfying all the other inequalities and one of the disjunctive conditions.

Suppose the second loop in our program in Figure 2 is distributed as blocks of 32 iterations across a linear array of processors. That is, processor \( p \) executes iteration \( [t, i] \) if

\[
32p \leq \left[ \frac{0}{1} \right] \left[ \frac{t}{i} \right] < 32(p+1)
\]

(We will use this decomposition throughout the rest of the paper.) Figure 5 shows the communication sets for context M2 from Figure 3.

<table>
<thead>
<tr>
<th>Context</th>
<th>( t_i \geq 0 )</th>
<th>( T - t_i \geq 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( i_r - 3 \geq 0 )</td>
<td>( N - i_r \geq 0 )</td>
</tr>
<tr>
<td></td>
<td>( i_r - 6 \geq 0 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( t_i - t_i \geq 0 )</td>
<td>( t_i - t_i \geq 0 )</td>
</tr>
<tr>
<td></td>
<td>( i_s - i_s + 3 \geq 0 )</td>
<td>( i_r - i_s - 3 \geq 0 )</td>
</tr>
</tbody>
</table>

| Access function | \( i_r - 3 - a \geq 0 \) | \( a - i_r + 3 \geq 0 \) |

| Computation decomposition for read iterations | \( i_r - 32p_i \geq 0 \) | \( 32p_r + 31 - i_r \geq 0 \) |

| Computation decomposition for write iterations | \( i_s - 32p_s \geq 0 \) | \( 32p_s + 31 - i_s \geq 0 \) |

| Constraint \( p_s \neq p_r \) | \( p_s > p_r \) | \( p_s < p_r \) |

Figure 5: Inequalities defining the communication sets for context M2 in Figure 3.

4.4.3 Finalization

Data produced within the loop nest may need to be written back to their home locations in the “final” data layout. The problem of identifying which written values are live at exit is a sub-problem in calculating last write trees[19]. The set of inequalities generated by this sub-problem, in conjunction with the final data distribution, defines the communication set for finalization.

5 Code Generation

Before we describe how to generate the computation and communication code, we first review the techniques of projection and scanning a polyhedron in Section 5.1. We then show how to generate a working program. We first generate the code for each communication set individually, merge these code fragments into an SPMD program, and manage the local memory on each processor. Optimizations are discussed in the next section.

5.1 Projection

We can represent all possible values of a set of variables \( (v_1, v_2, ..., v_n) \in Z^n \) as an \( n \)-dimensional discrete cartesian space, where the \( k \)-th dimension corresponds to variable \( v_k \). Coordinate \( (x_1, ..., x_n) \in Z^n \) corresponds to the value \( v_1 = x_1, ..., v_n = x_n \).

A set of linear inequalities in \( (v_1, v_2, ..., v_n) \) defines a polyhedron \( S^n \) in this \( n \)-dimensional space. All the solutions satisfying the inequalities correspond to the integer points within the polyhedron. Suppose we project the polyhedron onto the \( n \)-1 dimensional subspace orthogonal to the axis representing variable \( v_n \). The resulting polyhedron in the \( (n-1) \)-dimensional subspace, \( S^{n-1} \), can be represented by a set of linear inequalities involving \( (v_1, v_2, ..., v_{n-1}) \).

If \( [x_1, ..., x_n] \in S^n \) then \( [x_1, ..., x_{n-1}] \in S^{n-1} \). However, given \( [x_1, ..., x_{n-1}] \in S^{n-1} \), there may or may not exist an \( x_n \) such that \( [x_1, ..., x_n] \in S^n \). Consider the example where \( S^n \) has a single constraint involving \( v_n \): \( v_1 = 2v_n \). We know that \( v_1 \) must be even. However, this constraint is not captured in the projected polyhedron \( S^{n-1} \) and \( v_1 \) can be an odd number in \( S^{n-1} \).

Projection of an \( n \)-dimensional polyhedron onto an \( (n-1) \)-dimensional space can be achieved using a single step of Fourier-Motzkin elimination[23]. Fourier-Motzkin elimination can produce a large number of superfluous constraints. We can determine if a constraint is superfluous as follows. We replace the constraint in question with its negation, and if the new system does not have an integer solution then the constraint is superfluous. To check if a system has an integer solution we again use Fourier-Motzkin elimination. Since the Fourier-Motzkin elimination algorithm checks if a real solution exists for a system, a branch-and-bound technique is needed to check for the existence of integer solutions[23].

We have extended the technique of projection to handle some simple non-linear inequalities so that we can handle symbolic block sizes. We allow the coefficients in the linear inequalities to be of arbitrary size. We allow the coefficients in the linear inequalities to be of arbitrary size. We allow the coefficients in the linear inequalities to be of arbitrary size. We allow the coefficients in the linear inequalities to be of arbitrary size.

5.2 Scanning a Polyhedron

**Definition 5:** Given \( i, j \in Z^n \), \( i \) is lexicographically less than \( j \) iff there exists \( k \leq n \) such that \( \forall 0 \leq l < k \; i_l = j_l \) and \( i_k < j_k \).

Given a system of linear inequalities \( S \) with unknowns \( (v_1, v_2, ..., v_n) \), we wish to create a loop nest that will enumerate all the solutions to \( S \) in lexicographic order. The desired code is in the form of an \( n \)-deep loop nest. The index of the \( k \)-th outermost loop is \( v_k \); it is incremented by one every iteration, and has a finite lower and upper bound. The loop bounds are expressions of symbolic constants in the code and outer loop indices. This loop structure guarantees that the indices of the iterations executed are in lexicographic order. The problem that remains is: what should the bounds of the loops be such that the loop nest contains all the solutions of \( S \)?

Ancourt and Irigoin showed that the problem can be done by a series of projections[1][2]. In the following, we briefly describe
their algorithm, then discuss a minor extension that is useful in our domain.

We find the bounds of the loop nest in reverse order. To find the bounds for \(v_n\), we rewrite the constraints in the form of \(c_i v_n \geq k (v_1, ..., v_{n-1})\) and \(c_i v_n \leq h (v_1, ..., v_{n-1})\). Any inequalities not involving \(v_n\) need not be considered here. The integer lower and upper bounds for \(v_n\) are given simply by:

\[
\text{MAX} \left[ \frac{k (\ldots)}{c_i} \right] \leq v_n \leq \text{MIN} \left[ \frac{h (\ldots)}{c_i} \right]
\]

We next project the original polyhedron onto the \((v_1, v_2, ..., v_{n-1})\) space, we obtain an \((n - 1)\)-dimensional polyhedron represented by a set of constraints involving only variables \((v_1, v_2, ..., v_{n-1})\). We can then repeat the process for variables \(v_{n-1}, ..., v_1\).

An example illustrating the algorithm is shown in Figure 6. In the example, we have a polyhedron defined by inequalities involving variables \(i\) and \(j\). Steps 1 and 2 produce the bounds for the loop nest that scans the solution lexicographically in \((i, j)\) order; whereas steps 3 and 4 produce the bounds for the loop nest that scans the solution lexicographically in \((j, i)\) order.

As discussed in Section 5.1, there may exist an \([x_1, ..., x_{n-1}]\) in the projected polyhedron in \((v_1, v_2, ..., v_{n-1})\) space that does not correspond to any solution in the original system. Since all constraints involving \(v_n\) are used in the derivation of its lower and upper bounds, the lower bound of \(v_n\) will be larger than the upper bound when \(v_1 = x_1, ..., v_{n-1} = x_{n-1}\). Thus, there is a one-to-one correspondence between the iterations executed and the solutions in the system.

While the algorithm described above is correct, the generated code can be inefficient. An outer loop may contain iterations that do not have any useful computation; they simply compute the bounds of their inner loop just to find that the inner loop has no iterations. The auxiliary variables introduced by the LWT functions can often cause such inefficiency. We can usually eliminate this form of inefficiency as follows. We need not create a loop nest for \(v_n\), if the bounds on \(v_n\) can be expressed as \(\alpha v_n - \beta \leq v_n \leq \alphatilde v_n - \beta\), where \(\alpha\) and \(\beta\) are integers. We can simply replace all references to \(v_n\) by \(\alpha v_n - \beta\). If the bounds on \(v_n\) are of the form \(v_n - \beta \leq \alpha v_n \leq v_n - \beta\) where \(|\alpha| > 1\), we can still eliminate the loop nest for \(v_n\) by updating the bounds of \(v_n\) to:

\[
\text{for } v_k = \alpha \left\lfloor \frac{l - \beta}{\alpha} \right\rfloor + \beta \text{ to } h \text{ step } \alpha \text{ do}
\]

where \(l\) and \(h\) are the original lower and upper bounds of \(v_k\).

5.3 Generating Computation and Communication Code

To find the computation allotted to each processor, we scan the elements in a computation decomposition relation \(C\) lexicographically in \((p_1, ..., p_r, i_1, ..., i_p)\), or simply \((\vec{p}, \vec{i})\), order. The \(\vec{p}\) loops enumerate the processors. The inner \(i\) loops enumerate the iterations to be executed for each value of \(\vec{i}\). The SPMD code to be executed by each processor is as follows. Each processor checks if its processor number is within the bounds of the \(\vec{p}\) loops. If so, the code it executes is simply the \(l\) loops parameterized by its processor number. In the case where the computation decomposition is cyclic, each processor must iterate through the virtual processors it represents. Figures 7(a) and (b) show the computation code for our example in Figure 2. The rest of the figure shows the communica-

5.4 Merging Loop Nests

To generate the receive and send code for a communication set \(M\), we scan \(M\) lexicographically in \((\vec{p}_r, i_r, \vec{p}_l, i_l, \vec{a})\) and \((\vec{p}_r, i_r, \vec{p}_l, \vec{a})\) order, respectively. In the receive loop nest, the \(\vec{p}_r\) loops enumerate the processors involved in receiving data. The \(i_r\) loops specify the iterations when processor \(\vec{p}_r\) needs to receive data. By definition, the \(\vec{p}_r\), \(i_r\) and \(\vec{a}\) loops are degenerate loops containing only one iteration. The data to be received is the value in location \(\vec{a}\) on processor \(\vec{p}_r\) in iteration \(i_r\). Conversely, the \(\vec{p}_l\) loops in the send loop nest enumerate all the senders. The \(i_l\) loops specify the iterations when processor \(\vec{p}_l\) needs to send messages. The \(\vec{p}_l\) loops identify the receivers of each message. The \(i_l\) loops specify the iterations when processor \(\vec{p}_l\) needs the data. The \(\vec{a}\) loop is a degenerate loop containing the address of the data to be sent. If auxiliary variables have been introduced to handle modulo constraints in the LWTs, the auxiliary variables are placed last in the lexicographic order for both loops.

![Figure 6: Example of two projection sequences on a 2-dimensional polyhedron. The table contains the bounds on the variables eliminated by each projection.](image)
We have developed an algorithm that merges multiple nested loops together using the technique of loop splitting. Details of the algorithm are beyond the scope of this paper. If the relative magnitude between the bounds of the individual loops is not known at compile time, loop splitting can expand the program size by a significant amount. Our compiler only uses loop splitting on inner loops, and also when the relative magnitudes between the loop bounds are known. We have also developed a dynamic splitting scheme that we use on the outer loops. The compiler does not generate all the possible combinations statically. Instead, each processor determines its bounds for all the iteration sets, sorts the bounds, and interprets the sorted list to determine the loops it has to execute. Finally, for iteration sets that are a function of only outermost loop variables, we insert dynamic checks into the bodies of the outer loops.

5.5 Local Address Space

Typically, a processor on a parallel machine touches only a part of an array. Since data sets processed by these programs are often very large, it is essential that the compiler only allocates, on each processor, storage for the data used by the processor.

The following is a simple approach to the memory allocation problem. We allocate on each processor the smallest rectangular region that covers all the data read or written by the processor, and we copy all the received data from the communication buffer to their home locations in the array before they are accessed. Given a computation decomposition and an access function, the set of locations touched by processor is

\[
\{ \alpha \in A \mid \exists (i, p) \in C \land a = f(\bar{v}, i) \}
\]

By scanning the inequalities lexicographically in order, the bounds we obtain on \(a\) are the bounds for the kth dimension of the bounding box covering access \(f\). If there are multiple accesses to the same array, we simply find the bounding box of the rectangular boxes for all the accesses to the same array. Note that this formulation allows local data spaces on different processors to overlap.

The above algorithm is inadequate if the rectangular bounding box of the data accessed is larger than the available local memory on the processor, while the data actually used fit in the local memory. Also, a processor’s local memory may not be large enough to fit all the data that a processor will eventually use in a computation. In that case, we need to manage the memory dynamically.

The LWT information provides a more efficient way to manage the data that have been received from other processors. The compiler knows precisely which values are read by every instance of the read access. Instead of first copying all the received data to their home locations, a processor can simply read the values directly from the communication buffers. The compiler also has the information on when the buffer is no longer needed, and can manage the buffer space effectively.

6 Optimizations

Since the above algorithms generate a receive and a send message for every read access to remote data, the code is correct but inefficient. It is essential that we eliminate the redundant messages and amortize the message sending overhead by batching the communication.

6.1 Eliminating Redundant Communication

Ancourt has also studied the problem of eliminating redundant communication[2]. Given a set of iterations and accesses, Ancourt’s algorithm can construct a set of loop nests that fetches all the data touched without any duplication. This algorithm is adequate for removing redundant traffic if no communication is required within the loop nest. In general, transfers of data with the same address are redundant only if the values transferred are identical.
We separate redundancy into two categories. We say that there is self reuse when multiple instances of a single read access use the same data and group reuse when instances of different read accesses use the same data. We discuss each of these in turn below.

6.1.1 Redundant Communication Due to Self Reuse

Read instances that have different data-flow relationships often are amenable to different communication optimizations. By partitioning the read instances into different contexts according to their data-flow patterns, the LWT makes it easier to detect and eliminate redundancy. Our algorithm applies Theorem 5 to the communication set of each context to detect redundancy due to self reuse.

Theorem 5: Given a communication set $M$, communications $(i_r, p_r, i_p, p_s, \bar{a}), (r, p_r, i_r, p_s, \bar{a}) \in M$ are redundant due to self reuse if $p_r = p'_r, p_s = p'_s, i_s = i'_s$ and $\bar{a} = \bar{a}'$.

All elements in a communication set with identical $i_r, p_r, i_p, p_s$ and $\bar{a}$ refer to the same values; all elements with identical $i_r, p_r, i_p, p_s$ and $\bar{a}$ are redundant messages. Thus, we wish to replace the set of redundant messages with $(\min(i_r), p_r, i_p, p_s, \bar{a})$. This can be achieved by projecting the set onto the $(p_r, i_r, p_s, \bar{a})$ space, and constraining the upper bound of $i_r$ to be identical to its lower bound. There are two complications. First, if the lower bound of $i_r$ is expressed as a conjunction of multiple inequalities involving outer loop indices, then the communication set containing the minimum $i_r$’s is no longer convex. The algorithm needs to divide the communication set into multiple convex sets. The second complication arises from the fact that a projected image may contain points that do not correspond to a solution to the original system. In many cases, a simple test can determine that no such degeneracies are present[19].

6.1.2 Redundant Communication Due to Group Reuse

Detection of reuse between arbitrary accesses to the same matrix is expensive. However, there is a prevalent form of reuse that can be incorporated and exploited easily within our model and that is the set of uniformly generated references[13]. Array index functions of uniformly generated references are affine functions of loop indices and symbolic constants, and they differ only in the constant terms. For example, $X[i]$ and $X[i+3]$ are uniformly generated references; so are $B[2i+3]+1, 3i+n+3$ and $B[2i+3]+10, 3i+n+2$, but not $C[i]$ and $C[j]$. Reuse between uniformly generated references has been exploited successfully in improving cache locality[27][28]. Uniformly generated references are quite common in real programs, so much so that specialized languages and compilers have been built to translate them to efficient code[7][17].

We can represent a set of uniformly generated references by their convex hull, and describe the data flow information for all the references by a single LWT. For example, suppose the example in Figure 2 has three more read accesses:

\[
\begin{align*}
\text{for } t = 0 \text{ to } T \\
\text{for } i = 3 \text{ to } N \\
X[i] &= f(X[i], X[i - 1], X[i - 2], X[i - 3])
\end{align*}
\]

Figure 8: The example in Figure 2 with multiple read accesses.

The set of accesses $X[i], X[i + 1], X[i - 2], X[i - 3]$ can be represented by the access function $f(i)$, where $f(i) = i - u$ and $0 \leq u \leq 3$. The LWT for all the accesses is given in Figure 9.

Note that the convex hull may contain more data than those accessed within an iteration; however, since a processor is typically responsible for a contiguous block of iterations, this method is unlikely to cause any significant unnecessary traffic.

6.1.3 Other forms of redundancies

Redundancy may also arise from cyclic decompositions, where a physical processor emulates multiple virtual processors. Given a virtual to physical processor mapping $\pi: p \rightarrow p'$, communication $(i_r, p_r, i_p, p_s, \bar{a}) \in M$ can be eliminated if $\pi(p_r') = \pi(p_s')$. Also, communications $(i_r, p_r, i_p, p_s, \bar{a})$, $(i_r', p_r', i_p', p_s', \bar{a}') \in M$ are redundant if $\pi(p_r') = \pi(p_s')$, $p_r = p_r'$, $i_s = i'_s$ and $\bar{a} = \bar{a}'$.

Communication sets derived from data decompositions that replicate data may also contain redundancy. In our definition of communication sets, we consider communication to be necessary as long as there is a processor that owns a copy of the data needed by another processor. That means communication is generated even if the processor already owns a copy of the data. To eliminate this redundancy, we eliminate all the communication elements $(i_r, p_r, i_p, p_s, \bar{a}) \in M$ such that $(\bar{a}, p_s) \in D$. Furthermore, two communication elements $(i_r, p_r, i_p, p_s, \bar{a}), (i_r', p_r', i_p', p_s', \bar{a}') \in M$ are redundant due to replicated data if $p_r = p_r'$, $p_s \neq p_s'$, $i_s = i'_s$ and $\bar{a} = \bar{a}'$. The technique to eliminate this redundancy is similar to that of removing redundant communication due to self-reuse.

6.2 Communication Aggregation

Whether aggregation of small messages into large messages is necessary depends on the machine architecture. For example, machines such as the iWarp and CM-5 support fine-grain communication, while machines such as the Intel iPSC have significant overhead in processing every message. Again, we classify message aggregation into two kinds: self aggregation, where messages generated by different instances of the same access are aggregated, and group aggregation, where messages generated by different accesses are aggregated. For group aggregation, we simply aggregate all messages that have the same sender, receiver and dependence level into one message.

While group aggregation reduces the number of messages by a small constant, self aggregation can potentially eliminate more messages. Our self aggregation algorithm also takes advantage of the partitions created by the LWT analysis. All instances
within the same communication set have the same dependence level. If the dependence level of a communication set is \( k \), it is obviously legal to batch all the messages within an iteration of loop \( k \) and send the data at the end of the iteration. This can result in significant overhead reduction if loop \( k \) is not the innermost loop.

The algorithm to aggregate the communication of a communication set at level \( k \) is as follows. To generate the send code, we scan the communication set lexicographically in 
\[(\tilde{p}_w, i_{w1}, \ldots, i_{wK}, \tilde{p}_r, i_{r1}, \ldots, i_{rK}, t, \delta)\]
order. Each instance of the loops \( \tilde{p}_w, i_{w1}, \ldots, i_{wK}, \tilde{p}_r, i_{r1}, \ldots, i_{rK}, t, \delta \) produces one message, and each instance of the loops \( \tilde{p}_w, i_{w1}, \ldots, i_{wK} \) contributes an item to the message. Redundancy elimination would have caused \( i_{w1} \) to take on only the value of the earliest iteration on the receiver side using the value. Similarly, we create the receive loop nest by scanning the polyhedron in 
\[(\tilde{p}_r, i_{r1}, \ldots, i_{rK}, \tilde{p}_w, i_{w1}, i_{w2}, \ldots, i_{wK}, t, \delta)\]
order. Iterations in loops \( i_{w1}, i_{w2}, \ldots, i_{wK} \) use the same data from the same message. Note that for each message, the order in which the sender packs the data is the same as the unpacking order. Figure 10 shows the receive and send code for the M2 context in Figure 3 after aggregation.

if \( p_w > 1 \) and \( p_r < N / 32 \) then
for \( t_w = 0 \) to \( T \) do
  \( p_w = p_w - 1 \)
  receive data into buffer from processor \( p_w \)
  index = 0
for \( i_{w1} = 32 \; p_w - 3 \) to \( \text{MIN}(32 \; p_w - 1, N - 3) \) do
  \( t_w = t_w \)
  \( i_{w1} = i_{w1} + 3 \)
  \( a = i_{w1} - 3 \)
  \( X[a] = \text{buffer}[\text{index}] \)
  index = index + 1
if \( p_r > 0 \) and \( p_r < (N - 32) / 32 \) then
for \( t_r = 0 \) to \( T \) do
  \( p_r = p_r + 1 \)
  index = 0
for \( i_{r1} = 32 \; p_r + 29 \) to \( \text{MIN}(32 \; p_r + 31, N - 3) \) do
  \( t_r = t_r \)
  \( i_{r1} = i_{r1} + 3 \)
  \( a = i_{r1} \)
  \( \text{buffer}[\text{index}] = X[a] \)
  index = index + 1
send the data in buffer to processor \( p_r \)

**Figure 10:** Aggregated communication for context M2 in Figure 3.

### 6.2.1 Multi-casting

Many systems provide optimized routines for multi-casting. To take advantage of these routines, we need to determine if the same message is sent to multiple processors. We scan a communication set to be aggregated at level \( k \) lexicographically in 
\[(\tilde{p}_w, i_{w1}, \ldots, i_{wK}, \tilde{p}_r, i_{r1}, \ldots, i_{rK}, t, \delta)\]
order. If the bounds of \( \tilde{p}_w \) are independent of \( \tilde{p}_r \), the data sent to each processor are identical.

### 7 A Detailed Example

We have implemented the communication optimization and code generation algorithms described in this paper in the Stanford SUIF compiler system[25]. However, these algorithms have not been fully integrated with all the other phases in the compiler. We have been able to experiment with our algorithms on the Intel iPSC/860 machine by doing the following. We enter the decompositions manually. From the decompositions, our compiler pass generates all the necessary communication and computation loop nest structures and their bounds in C. We then insert in the original loop body and compile the program using the vendor’s C compiler. In this section, we show how our algorithm compiles and optimizes the LU decomposition kernel in Figure 11.

```plaintext
for i1 = 0 to N do
  for i2 = i1 + 1 to N do
    X[i1][i2] = X[i1][i2] / X[i1][i1]
  for i3 = i1 + 1 to N do
    X[i3][i2] = X[i3][i2] - X[i3][i1] * X[i1][i3]
```

**Figure 11:** Sequential version of LU decomposition.

To improve load balance, we use a cyclic computation and data decomposition in this program. The computation decomposition \( C \) and initial data decomposition \( D \) first map the computation and data, respectively, to a virtual processor space, which is later mapped to the physical processor space. Let \( i = (i_1, i_2, i_3) \), the decompositions used are
\[ C = \{(i, p) \in X \times p | p \leq \left[ \begin{array}{c} 0 \ 1 \ 0 \ \end{array} \right]^{T} < p + 1 \} \]
and
\[ D = \{(d, p) \in A \times p | p \leq \left[ \begin{array}{c} 1 \ \delta \ \end{array} \right]^{T} < p + 1 \} \]
That is, the \( k \)th virtual processor executes the \( k \)th iteration of the second loop and owns the \( k \)th row of array \( X \).

First, our algorithm generates the LWTs for all the five read accesses in the program. Every tree has two leaves, one \( \perp \) leaf and one leaf reading from either the \( X[i_1][i_3] \) or \( X[i_2][i_3] \) write operation. The LWT for the read access \( X[i_1][i_3] \) is shown in Figure 12.

![LWT Diagram](image)

**Figure 12:** The LWT for the read access \( X[i_1][i_3] \).

Second, our compiler calculates the communication set for every leaf of each tree. It determines that both leaves on the LWTs for \( X[i_1][i_3] \) and \( X[i_2][i_3] \) have non-empty communication sets. All other leaves need no communication, either because the values read are produced by the same processor or they are already in the local memory due to the initial data decomposition.

The communication sets specify the related read and write dynamic instances precisely. We know precisely that it is the first iteration of the second loop that produces all the data used by \( X[i_1][i_3] \) and \( X[i_1][i_1] \) in the next iteration of the outermost loop. Knowing this information enables us to issue the send immediately after the first iteration of the second loop. If we had used only data dependence analysis, we would have to delay the send until all the
iterations of the second loop have been executed. Sending the data early maximizes the potential for overlap between communication and computation.

Next, the compiler performs several communication optimizations. It first aggregates all the messages within each of the two communication sets of \( X[i_1][i_2] \). Then, the communication sets for the \( \bot \) leaves of both \( X[i_1][i_2] \) and \( X[1][1] \) are aggregated. The same is done for the other two leaves. At this point, each virtual processor receives only a single message in each iteration of the outermost loop. The compiler then discovers that the message contents are independent of the identity of the receiving processor, thus the data can be multi-cast. Since the computation decomposition is cyclic, the compiler optimizes the communication further by sending the data only to one virtual processor in each physical processor.

Lastly, the compiler decides on how to manage the local memory. While each virtual processor always writes to the same row in the matrix, every processor would have read the entire matrix by the end of the computation. The compiler creates a local array for the write accesses, and uses a buffer to hold the data a processor receives. The program uses the data in the buffer directly. The dimensions of the local array on each physical processor are \( (N+P)/P \times (N+1) \) where \( P \) is the number of physical processors in the system. Note that the middle dimension of this local array can be trivially eliminated. The communication buffer is of size \( N+1 \), the size of the largest aggregated message.

Figure 13 is a pretty-printed version of the information generated by the compiler. In this code, the variable “\text{myp}” identifies the physical processor executing the code. Most of the loop overhead present in this code can be eliminated using classical optimizations.

```c
float X[(N+P)/P][N+1], B[N+1]
{ First processor sends initial data }
if N >= 1 and myp = 0 then
  for t = 0 to N do
    B[t] = X[0][t]
  for p1 = 1 to MIN(P - 1, N) do
    send B of size N + 1 to (p1 mod P)
for i1 = 0 to N - 1 do
  { iterate over the virtual processor set }
  for p = P*(i1 + 1 - myp + P - 1)/P + myp to N step P do
    i2 = p
    c = i2/P
    { local address }
    { receive data, once for each physical processor }
    if i1 >= i2 - P + 1 then
      receive to B of size N - i1 + 1
      X[c][i1] = X[c][i2]/B[0]
    for i2 = 1 + i1 to N do
      X[c][i2] = X[c][i1] - X[c][i2]*B[i2 - i1]
    { send data }
    if i2 <= N - 2 and i2 = i1 + 1 then
      for t = i1 to N do
        B[t - i2] = X[c][t]
      for p2 = i2 + 1 to MIN(i2 + P - 1, N) do
        send B of size N - i2 + 1 to (p2 mod P)
Figure 13: Compiler generated SPMD code for LU decomposition.
```

Our compiler pass took 2.9 seconds to generate the computation and communication code. Our 2048 \times 2048 \text{LU}-decomposition code runs at 250 MFLOPS (single-precision) on a 32-processor Intel iPSC/860 machine. Figure 14 shows the performance of our code for two matrix sizes on different numbers of processors.

8 Summary and Conclusion

This paper introduces a value-centric approach to communication generation for machines with a distributed address space. We use a precise data-flow analysis that determines the exact value read by every instance of a read access; we derive communications from computation decompositions. This approach has several advantages over the more conventional approach of deriving communications from data decompositions, and optimizing communications using data dependence analysis. Our approach supports more flexible data decomposition schemes: data may not have a static home, data may be replicated, and the owner-computes rule can be relaxed. Our approach can also support more communication optimizations. The LWT analysis automatically partitions the read instances into sets that share similar communication characteristics. This simplifies the handling of boundary conditions and also exposes more opportunities for parallelism.

We have developed a uniform framework to solve the problems of communication code generation, local memory management, message aggregation and redundant data communication elimination. We represent data decompositions, computation decompositions and the data flow information all as systems of linear inequalities. The various code generation and communication optimization problems can be solved by projecting polyhedra represented by sets of inequalities onto lower dimensional spaces. Our techniques are applicable to both the value-centric approach presented in this paper as well as the conventional location-centric approach used in other systems.

Acknowledgments

We would like to thank Jennifer Anderson, Amy Lim, Dror Maydan and Daniel Scales for their contributions to this work. We acknowledge the support of the SUIF compiler group, and Robert Dutton and Dan Yergeau for providing and supporting the iPSC/
References


