# Compile-Time Generation of Communications for Scientific Programs

*Charles Koelbel*

**CRPC-TR91089**
**January, 1991**

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

# Compile-Time Generation of Communications for Scientific Programs

Charles Koelbel*
Center for Research on Parallel Computation
Rice University
Houston, TX 77251-1892

January 18, 1991

## Abstract

Programming nonshared memory systems is more difficult than programming shared memory systems, largely because there is no support for shared data structures. Current programming languages for distributed memory architectures force the user to decompose all data structures into separate pieces, with each piece "owned" by one processor, and with all communication explicitly specified by low-level message-passing primitives. This paper presents a new programming language for these architectures which provides a global name space and allows direct access to remote parts of data values. We give a general framework for translating programs written in this environment into message-passing code suitable for direct execution on distributed memory machines. We then use this framework to derive an analysis suitable for use in the compiler. Performance results from such a compiler are shown for the iPSC/860 multiprocessor.
**Keywords:** Compiling, Distributed Memory Architectures, Shared Memory Model, Parallel Computation, Programming Languages, Compile-time Analysis.

## 1 Introduction

Distributed memory architectures promise high levels of performance for scientific applications at modest costs. Unfortunately, they are currently awkward to program. The available programming languages for such machines reflect the underlying hardware in the same way that assembly languages reflect the registers and instruction set of a microprocessor. In particular, each process can access only the local address space of the processor on which it is executing. In contrast, programmers tend to think of their programs in terms of manipulating large data structures, such as vectors, arrays, and so on. To implement such a view in a message-passing language, the programmer must decompose each data structure into a collection of pieces with each piece "owned" by a single process. All interactions between different parts of the data structure must then be explicitly specified using the low-level message-passing constructs supplied by the language.

Explicitly decomposing all data structures in this way can be extremely complicated and error prone. Moreover, program flexibility and portability are reduced by such low-level specifications. Because the partitioning of the data structures across the processors must be done at the highest level of the program, and because each operation on these distributed data structure turns into an

*This work was supported by NSF Cooperative Agreement No. CCR-8809615. Additional support was given under NASA Contract No. NAS1-18605 while the author was in residence at the Institute for Computer Applications in Science and Engineering, NASA Langley Research Center.

intricate sequence of communication statements, programs become highly inflexible. This makes the parallel program not only difficult to design and debug, but also "hard wires" all algorithm choices, inhibiting exploration of alternatives. Similar reasons inhibit porting message-passing programs to new architectures. Machine-dependent details may change the optimal data distribution, but implementing this change within the message-passing program will be difficult.

In this paper we present a different paradigm for programming distributed memory architectures. The goal is to provide a software layer supporting a global name space for the programmer. This allows specification of the computation via a set of parallel loops exactly as one does on a shared memory architecture. The compiler analyzes this high-level specification and transforms it into a system of tasks communicating by messages. We thus acquire the ease of programmability of the shared memory model and retain the high performance of nonshared memory architectures. This approach allows the programmer to focus on high-level algorithm design and performance issues while relegating the minor but complex details of interprocessor communication to the compiler and run-time environment.

A danger of this approach is that since true shared memory does not exist, one might easily sacrifice performance. We avoid this by requiring the user to explicitly control data distribution and load balancing, thus forcing awareness of those issues critical to performance on nonshared memory architectures. In the future we hope to extend the system so that even these issues can be handled by the compiler. Another danger is that the compiler may not be able to generate efficient message-passing code. In this paper we show that this is not the case for an important class of algorithms. Our compile-time analysis produces code that is virtually identical to that which would be achieved had the user programmed directly in a message-passing language. Other work [11, 12, 20] has shown techniques which apply to other important classes of algorithms.

The remainder of this paper is organized as follows. Section 2 presents a general framework for reasoning about distributed data arrays, and illustrates the model using the Kali[1] programming language. Although we use the Kali syntax here, our research is not specific to Kali; it can be applied to any imperative language extended with parallel loops and data distribution constructs. Section 3 uses this model to derive an analysis for use by the compiler. Each subsection of Section 3 derives the analysis needed for one particular pattern of subscripts and illustrates the use of this analysis on a well-known numerical algorithm. Section 4 shows performance data for those programs and compares them to hand-written implementations of the same algorithms. Finally, Section 5 compares our work with other groups, and Section 6 gives our conclusions.

## 2 A Model for Data Decomposition

This section develops a notation for describing the compilation of parallel loops accessing a shared name-space into code that can be directly executed on nonshared-memory machines. In Section 2.1, we give a general description of the source and target code and informally introduce the model. Sections 2.2 through 2.5 define the components of the model more formally. Finally, Section 2.6 briefly describes implications of this model for code generation.

### 2.1 The Structure of Generated Code

This section describes in general terms the structure of code generated for a single **forall** loop. A **forall** is essentially a **for** loop with no inter-iteration data dependences. This lack of dependences allows iterations of the loop to be executed in any order, including parallel execution.

For concreteness, we will base our discussion on the model program given in Figure 1. The program copies elements of array $A$ to $New\_A$; specifics of the syntax will be clarified in Sections 2.2 through 2.5. Figure 1 has some significant simplifying assumptions:

[1]The Kali language is named for a Hindu goddess possessing eight arms. The multiple arms symbolize the parallelism that we hope to exploit.

```
processors Procs : array[ 0..P−1 ] with P in 1..max_procs;

var A, New_A : array[ 0..N−1 ] of real dist by [ block ] on Procs;

forall i in low..high on New_A[i].loc do
   New_A[i] := A[ f(i) ];
end;
```

Figure 1: Example **forall** statement for definition of sets

1. $A[f(i)]$ is the only array reference in the loop that can induce communication. If there are several references, they can be analyzed separately and the results combined.
2. $A[f(i)]$ is an r-value rather than an l-value, that is, it is read rather than written. If assignments to nonlocal array elements are allowed, extra communication is needed at the end of the translated **forall** to send the nonlocal values to their home processors. The analysis to produce this communication is analogous to that shown here.
3. $A[f(i)]$ is always accessed in the loop, that is, there are no conditionals to alter control flow around the reference. Control flow can be handled by using conservative approximations, that is, by assuming that all references are always made.

In contrast, the specific distributions of data and computation are not simplifying assumptions. The expressions derived in Sections 2.2 through 2.5 will apply to any distributions.

Each processor must complete three major tasks in order to correctly implement the program of Figure 1:

- Generate information needed for passing messages and controlling iteration.
- Exchange data with other processors so that all processors have the data needed for their computations.
- Execute computations using local data and data from received messages.

A more detailed outline of how these tasks are ordered is given in Figure 2.

The first step of Figure 2 is to generate the information that will be used later. Each processor needs four pieces of information to complete the above tasks.

1. The set of array elements that it stores locally.
2. The set of **forall** iterations that it must execute.
3. The sets of array elements that must be sent and received in messages.
4. Two subsets of the set of iterations: those which access only local data and those which access some nonlocal data.

The usefulness of the first two sets is obvious. On processor $p$ they are called $local(p)$ and $exec(p)$. The next pair of sets is needed to control the communication with other processors. For every pair of processors $p$ and $q$ there will be sets $send\_set(p,q)$ and $recv\_set(p,q)$, which have the obvious meanings. We will refer to these sets as the communication sets. Finally, the iteration subsets are used to overlap computation and communication, as explained below. The iterations on processor $p$ that need no data from other processors are collected in $local\_iter(p)$. Iterations on $p$ that access any data from any other processor make up $nonlocal\_iter(p)$ We will refer to these sets as the iteration sets.

Code on processor $p$:

- Generate communication and iteration sets
  - $local(p)$ = Array elements stored on $p$.
  - $exec(p)$ = Iterations to be performed on $p$.
  - For all $q \neq p$, $send\_set(p,q)$ = Array elements sent from $p$ to $q$.
  - For all $q \neq p$, $recv\_set(p,q)$ = Array elements received by $p$ from $q$.
  - $local\_iter(p)$ = Iterations on $p$ that access only local data.
  - $nonlocal\_iter(p)$ = Iterations on $p$ that access some nonlocal data.
- For all $q$ with $send\_set(p,q) \neq \phi$, send message containing $send\_set(p,q)$ to $q$.
- Execute computations for iterations in $local\_iter(p)$, accessing only local arrays.
- For all $q$ with $recv\_set(p,q) \neq \phi$, receive message with $recv\_set(p,q)$ from $q$.
- Execute computations for iterations in $nonlocal\_iter(p)$, accessing local arrays and message buffers.

Figure 2: Implementing a **forall** on a nonshared memory machine

The next logical task in implementing a **forall** is to perform any necessary communication. This is split into two parts in Figure 2; sending the messages is done first, and receiving messages comes later. Since $A[f(i)]$ in Figure 1 is an r-value, the only messages needed in the implementation will be for reading nonlocal data. Since there are no inter-iteration dependences in a **forall**, the data for these messages will be available at the beginning of the loop and will not be overwritten within the loop. Thus, the data can be passed as messages at any time before it is needed. Our implementation sends the messages as soon as the data is available, that is, as soon as $send\_set(p,q)$ is known. This provides the maximum time for messages to reach their destinations before they are needed. Similarly, messages can be received at any time before their actual use. Our implementation performs all receives in a block immediately before the first nonlocal value is needed. This strategy is called prefetching and is quite effective, but it is not the only possible strategy. We will review other possibilities in our discussion of related work.

The final logical task in implementing a **forall** is the actual computation. This task is split into two parts and interwoven with the communication task. This organization is used to gain efficiency. If some iterations of the **forall** on processor $p$ use only data stored on $p$, then those iterations can be executed before any incoming messages have been received. This observation can be exploited to overlap computation and communication by grouping all iterations which use only local data together. The remaining iterations, which depend on data received in messages, must be executed after the messages have been received. Combining this overlap strategy with the prefetching strategy explained above results in the alternation of communication and computation shown in Figure 2.

Note that each processor only needs to generate its own sets. For example, processor 1 needs no information about $local\_iter(2)$ or $send\_set(3,6)$. This reduces the amount of information and analysis required on each processor. It should also be noted that the sets need not be explicitly generated in all cases; Section 2.6 explores this issue in more depth.

## 2.2 Data Distribution

The fundamental task of data distribution is to specify which processors in a nonshared memory machine will store each element of a shared data structure in their private memories. This is done by

```
processors    Procs : array[ 0..P−1 ] with P in 1 .. max_procs;

var    A : array[ 0..N−1 ] of real dist by [ block ] on Procs;
       B : array[ 0..N−1 ] of real dist by [ cyclic ] on Procs;
       C : array[ 0..N−1, 0..M−1 ] of real dist by [ block, * ] on Procs;
```

Figure 3: Array declarations in Kali

providing a mapping between the set of processors on a parallel machine and the set of data items to be stored. This mapping is not necessarily one-to-one. Going from data items to processors, there will usually be more data items than processors, so each processor must store more than one datum. In the other direction, it is sometimes advantageous to store several copies of the same data item. Two particular cases of this are of interest:

1. Scalars and small arrays are usually duplicated across all processors.
2. In practice, it is common to have a small area of "overlap" between the regions stored on neighboring processors to reduce communication.

A general model of data distribution must allow these types of copying.

We describe a data distribution by giving the set of array elements stored on each processor. Mathematically, this is a function from processors to sets of array elements which we call the *local* function.

**Definition 1** *Let Procs be the set of processors and Elem the set of elements of an array A. Then*

$$local : Procs \to 2^{Elem} : local(p) = \{a \in Elem \mid a \text{ is stored on } p\} \tag{1}$$

(Here, $2^S$ is the class of subsets of set $S$.) Note that using this scheme there is no problem with multiple copies of array elements; the only consequence is that the *local* sets of distinct processors are not disjoint. In this paper, however, we will assume that each array element is stored on exactly one processor. In the examples that follow, we will represent *Procs* and *Elem* by their index sets, which will be tuples of integers. Also, if there is an ambiguity as to the identity of the array, we will use the array or distribution name as a subscript.

The Kali syntax for data distribution is shown in Figure 3. The **processors** declaration declares and names the set of processors which will execute the program. The declaration shown declares *Procs* to be a one-dimensional array of $P$ processors, where $P$ is an integer constant between 1 and *max_procs* dynamically chosen by the run-time system. (The current Kali implementation chooses the largest feasible $P$.) This is equivalent to defining the *Procs* set in the definition of *local* to be

$$Procs = \{0, 1, 2, \ldots, P-1\}$$

The definitions of data distributions below will use $P$ as the number of processors and will assume 0-based indexing. It is also possible to declare a processor array with two or more dimensions.

The most common distributions patterns for one-dimensional arrays are **block** and **cyclic**. Array $A$ in Figure 3 is distributed by **block**, which assigns a contiguous block of array elements to each processor. For example, if $N = 64$ and $P = 4$, then *Procs*[0] stores elements 0 through 15, *Procs*[1] would store elements 16 to 31, and so on. The general form for *local* functions for block-distributed arrays is given below.

**Definition 2** *Let an array have N elements indexed starting with 0, and the set of processors have P processors also with 0-based indexing. Then* **block** *distribution of the array implies the following local function.*

$$local_{\mathbf{block}}(p) = \left\{ i \;\middle|\; (p-1) \cdot \left\lceil \frac{N}{P} \right\rceil + 1 \leq i \leq p \cdot \left\lceil \frac{N}{P} \right\rceil \right\} \tag{2}$$

```
processors      Procs : array[ 0..P−1 ] with P in 1..max_procs;

var     A, B : array[ 0..N ] of real dist by [ block ] on Procs;

forall i in low..high on A[i].loc do
   A[i] := B[f(i)];
end;
```

Figure 4: **Forall** loop in Kali

Array $B$ in the figure is distributed by **cyclic**, which assigns every $P$th element to the same processor. For example, if $P$ were 4 then $Procs[0]$ would store elements 0, 4, 8, and so on, while $Procs[1]$ would store elements 1, 5, 9, etc. The counterpart to Definition 2 for **cyclic** distributions is

**Definition 3** *Let an array have N elements indexed starting with 0, and the set of processors have P processors also with 0-based indexing. Then* **cyclic** *distribution of the array implies the following local function.*

$$local_{\textbf{cyclic}}(p) = \{i \mid i \equiv p \pmod P\} \tag{3}$$

Other static distribution patterns are available in Kali, and we are working on extensions to dynamic distribution patterns. In this paper we will only consider block and cyclic distributions, however.

Distributions of multi-dimensional arrays in Kali is achieved by applying **block** and **cyclic** distributions to each dimension of the array independently. The number of dimensions so distributed cannot exceed the dimensionality of the processor array. Other dimensions are grouped together on the same processor and denoted by an asterisk in the **dist** clause. The declaration of array $C$ in Figure 3 gives an example of this. It is distributed by blocks of rows; its *local* function is

$$local_C(p) = \left\{(i,j) \;\middle|\; (p-1)\cdot\left\lceil\frac{N}{P}\right\rceil + 1 \le i \le p\cdot\left\lceil\frac{N}{P}\right\rceil\right\}$$

If the processor array *Procs* has two or more dimensions, several dimensions of the data array can be distributed, leading to patterns such as the familiar two-dimensional blocked distribution.

## 2.3 Iteration Distribution

The next task in implementing shared name-spaces on non-shared memory machines is to divide the iterations of a loop among the processors. This can be modeled mathematically by a function similar to the *local* function.

**Definition 4** *Let Procs be the set of processors and Iter the set of iterations of a* **forall** *loop. Then*

$$exec : Procs \rightarrow 2^{Iter} : exec(p) = \{i \in Iter \mid i \text{ is executed on } p\} \tag{4}$$

Again, we assume that iteration sets are disjoint and that iterations are represented by the value of the loop index.

A feature of the Kali **forall** not found in other languages is the explicit specification of the location of the computation. The **on** clause of a Kali **forall** specifies the processor to execute each iteration of the loop. In effect, this determines the *exec* function for that **forall**. In Figure 4, the loc expression specifies that iteration $i$ of the **forall** will be executed on the processor storing element $i$ of the $A$ array. Thus, in this example,

$$exec(p) = local_A(p) \cap \{low, low+1, \ldots, high\}$$

More complex **on** clauses are also possible; in these cases, the *exec* function is essentially the inverse of the expression in the **on** clause [10].

## 2.4 Communication Sets

We now turn our attention to the communication sets. The purpose of this section is to define and derive expressions for *send_set*$(p,q)$ and *recv_set*$(p,q)$ in terms of the *local* and *exec* functions. This differs from the last two sections, in which we only defined the functions. This is because the *local* and *exec* functions were defined by the Kali program text; the communication sets are not so readily available We will not discuss the practicalities of computing the sets here. Implementation methods will be discussed in general in Section 2.6 and in detail in Section 3.

The information needed to generate messages in the implementation of a Kali **forall** can be encapsulated in the two set-valued functions given in Definition 5.

**Definition 5** *Let Procs be the set of processors, and Elem the set of elements of array A. Then*

$$send\_set : Procs \times Procs \rightarrow 2^{Elem} : send\_set(p,q) = \{a \in Elem \mid p \text{ must send } a \text{ to } q\} \quad (5)$$
$$recv\_set : Procs \times Procs \rightarrow 2^{Elem} : recv\_set(p,q) = \{a \in Elem \mid p \text{ must receive } a \text{ from } q\} \quad (6)$$

Note that $send\_set(p,q) = recv\_set(q,p)$ for all $p$ and $q$, reflecting the fact that every message has a sender and a receiver.

To derive useful formulas for the communication sets, we first make one auxilary definition.

**Definition 6** *Let Proc and Elem be as they were in Definition 5. Then*

$$ref : Procs \rightarrow 2^{Elem} : ref(p) = \{e \in Elem \mid p \text{ accesses } e\} \quad (7)$$

We observe that, under the assumptions of Figure 1, the only way for processor $p$ to access array element $e$ is for $e = A[f(i)]$ for some $i \in exec(p)$. Thus, we can give a simple formula for $ref(p)$:

$$\begin{aligned} ref(p) &= \{f(i) \in Elem \mid i \in exec(p)\} \\ &= f(exec(p)) \end{aligned}$$

We first derive an expression for $recv\_set(p,q)$. An array element $e$ must be in $recv\_set(p,q)$ if two conditions are met: processor $q$ must store $e$, and processor $p$ must access $e$. The first condition is satisfied by $e \in local(q)$. The second condition is equivalent to $e \in ref(p)$ Combining the two conditions stated above, we have that $e \in recv\_set(p,q)$ if $e \in local(q)$ and $e \in ref(p)$ or, equivalently,

$$recv\_set(p,q) = local(q) \cap ref(p)$$

We will take this as the general formula for $recv\_set(p,q)$. The symmetry of the definitions of *send_set* and *recv_set* then produces the corresponding formula for *send_set*.

$$send\_set(p,q) = local(p) \cap ref(q)$$

The expressions for *ref* and the communication sets are collected in the following theorem for reference.

**Theorem 1** *Let* $A[f(i)]$ *be an array reference in a* **forall** *statement. Then*

$$\begin{aligned} ref(p) &= f(exec(p)) & (8) \\ recv\_set(p,q) &= local_A(q) \cap ref(p) & (9) \\ send\_set(p,q) &= local_A(p) \cap ref(q) & (10) \end{aligned}$$

**Proof.** See above discussion. □

Figure 5 shows a visualization of the communication set analysis for **block** distributions in two dimensions. The diagram represents a portion of the array space used in the Kali program above it, where each circle represents one element of array A. The *local*$(p)$ sets are squares in the data space. Because of the form of the **on** clause, the *exec*$(p)$ sets are the same as the *local*$(p)$ sets. These sets

are represented by the large dashed rectangles in the figure. Only four of these sets are shown; the other data elements are contained in the *local*($p$) sets for other processors. Subscripting functions shift and deform these rectangles to produce the *ref*($p$) sets. Here the subscripting function is $f(i,j) = \langle i-1, j-1 \rangle$, which shifts the rectangles up and left without deformation. One such set is shown as the dotted square in the figure. Intersections between the sets are shown as solid rectangles. In this case there are three nonempty *recv_set*($p,q$) sets for each processor $p$. Symmetric arguments allow one to visualize the *send_set*($q,p$) sets.

## 2.5 Iteration Sets

The purpose of this section is to derive expressions for the iteration sets in the same way that the last section derived expressions for the communication sets. Again, we will leave the practicalities of computing the sets for later sections.

Formally, the iteration sets are defined as two set-valued functions.

**Definition 7** *Let Procs be the set of processors, and Iter the set of iterations of a* **forall**. *Then*

$$local\_iter(p) : \ Procs \rightarrow 2^{Iter} : \ local\_iter(p) = \{i \in exec(p) \mid i \text{ uses only data on } p\} \quad (11)$$
$$nonlocal\_iter(p) : \ Procs \rightarrow 2^{Iter} : \ nonlocal\_iter(p) = \{i \in exec(p) \mid i \text{ uses data not on } p\} \quad (12)$$

Note that both iteration sets are subsets of *exec*($p$).

As before, we need an auxillary definition to perform our analysis.

**Definition 8** *Let Procs and Iter be as they were in Definition 7. Then*

$$deref(p) : Procs \rightarrow 2^{Iter} : deref(p) = \{i \in Iter \mid i \text{ accesses only data on } p\} \quad (13)$$

Although their definitions are very similar, it is not the case that *deref*($p$) is the same as *local_iter*($p$). The difference is that *deref*($p$) may include iterations not executed on processor $p$. For example, if all iterations of the **forall** accessed the same array element $e$ (and no other elements), then *deref*($p$) would be all of *Iter* for the processor storing $e$. For the program of Figure 1, there is only one way that any array element $e$ can be accessed: if $e = A[f(i)]$ for some iteration $i$. Thus, we have

$$\begin{aligned} deref(p) &= \{i \in Iter \mid f(i) \in local(p)\} \\ &= f^{-1}(local(p)) \end{aligned}$$

As was the case for *recv_set*($p,q$), two conditions must be satisfied in order for an iteration $i$ to be in *local_iter*($p$): processor $p$ must execute $i$, and iteration $i$ must access only data stored on $p$. The first condition is satisfied when $i \in exec(p)$ and the second when $i \in deref(p)$. Combining the above two conditions, we have

$$local\_iter(p) = exec(p) \cap deref(p)$$

This is the general formula for *local_iter*($p$) that we sought. Since iterations on processor $p$ which do not fall in *local_iter*($p$) must fall into *nonlocal_iter*($p$), we can define *nonlocal_iter*($p$) by set complement:

$$nonlocal\_iter(p) = exec(p) - local\_iter(p)$$

Rewriting and simplifying, we find

$$\begin{aligned} nonlocal\_iter(p) &= exec(p) - local\_iter(p) \\ &= exec(p) - (exec(p) \cap deref(p)) \\ &= (exec(p) - exec(p)) \cup (exec(p) - deref(p)) \\ &= \phi \cup (exec(p) - deref(p)) \\ &= exec(p) - deref(p) \end{aligned}$$

We take the last form as our definition of *nonlocal_iter*.

The expressions for *deref* and the iteration sets are collected below for reference.

```
var A : array[ 1..N, 1..N ] of real dist by [ block, block ] on Procs;

forall i in 2..N, j in 2..N on A[i,j].loc do
        ⋮
   ... A[i−1,j−1] ...
        ⋮
end;
```

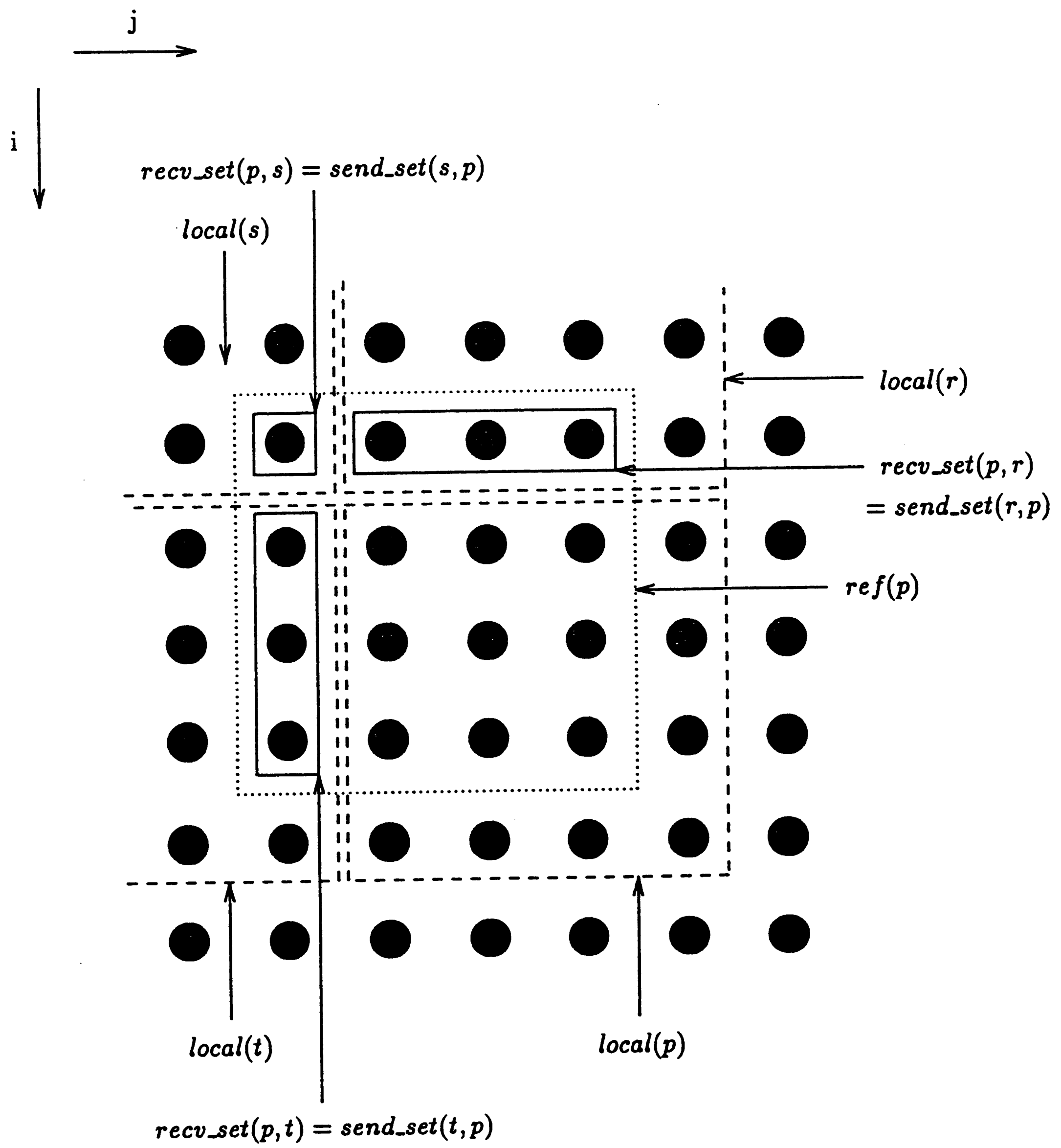

Figure 5: Visualizing communication sets

**Theorem 2** *Let $A[f(i)]$ be an array reference in a* **forall** *statement. Then*

$$deref(p) = f^{-1}(local(p)) \tag{14}$$
$$local\_iter(p) = exec(p) \cap deref(p) \tag{15}$$
$$nonlocal\_iter(p) = exec(p) - deref(p) \tag{16}$$

**Proof.** See above discussion. □

The iteration sets can be visualized for two-dimensional **block** distributions in much the same way as the communication sets, as shown in Figure 6. Here, the small squares represent **forall** iterations. One $exec(p)$ set is shown as a dashed rectangle; because of the **on** clause in the **forall**, this is the same as the $local(p)$ set. The inverses of subscript functions deform this set in much the same way that subscript functions did in the last section. In this case, $f^{-1}(i,j) = \langle i+1, j+1 \rangle$ and the effect of $f^{-1}$ is to shift the $local(p)$ set down and right. This produces $deref(p)$, shown as a dotted square. The $local\_iter(p)$ and $nonlocal\_iter(p)$ sets are shown as the solid square and solid L-shaped region, respectively.

## 2.6 Run-time Versus Compile-time Analysis

The major issue in applying the above model is the analysis required to compute the communication and iteration sets. It is clear that a naive approach to computing these sets at run-time will lead to unacceptable performance, in terms of both speed and memory usage. This overhead can be reduced by either doing the analysis at compile-time or by careful optimization of the run-time code.

In some programs the $ref(p)$ and $deref(p)$ sets of a **forall** loop depend on the run-time values of the variables involved. In such cases, the sets must be computed at run-time at obvious expense. We refer to the process of generating code for this task as *run-time analysis*. A key feature of run-time analysis is reducing the overhead of building the sets. This can be accomplished by careful data structure design and reusing the results of the analysis when possible. The reuse amortizes the cost of the run-time analysis over many repetitions of the **forall**, lowering the overall cost of the computation. This strategy is investigated in depth in [12, 10, 15, 20].

In many other programs, the distributions and subscript functions used in a loop are simple enough that analytic expressions for the communication and iteration sets are available. If such expressions can be found, the compiler can either compute the sets itself or emit short sequences of integer instructions to compute them at run time. We refer to such analysis as *compile-time analysis*. In the next section we give a form of this analysis applicable to a large number of scientific computations. As the examples in that section show, programs amenable to this analysis can be compiled into quite efficient programs.

# 3 Compile-time Analysis

Many scientific applications have very regular array access patterns. These access patterns may arise from either the underlying physical domain being studied or the algorithm being used. Examples of such applications include

1. Dense matrix factorizations, such as Gaussian elimination [9].
2. Relaxation algorithms for PDEs on regular meshes [6].
3. Alternating Direction Implicit (ADI) methods for solving PDE [23].

The distributions and subscripts used in such applications tend to be simple: **block** or **cyclic** distribution, and linear subscript functions. With such functions, the communication and iteration sets can be described by a few scalar parameters (such as low and high bounds on a range). Compile-time analysis exploits this simple structure to quickly compute those parameters, making set generation quite efficient.

The general methodology of compile-time analysis is

**var** A : **array**[ 1..N, 1..N ] **of real dist by** [ **block**, **block** ] **on** Procs;

**forall** i **in** 1..N, j **in** 1..N **on** A[i,j].loc **do**

```
          ⋮
   ... A[i−1,j−1] ...
          ⋮
```

**end**;

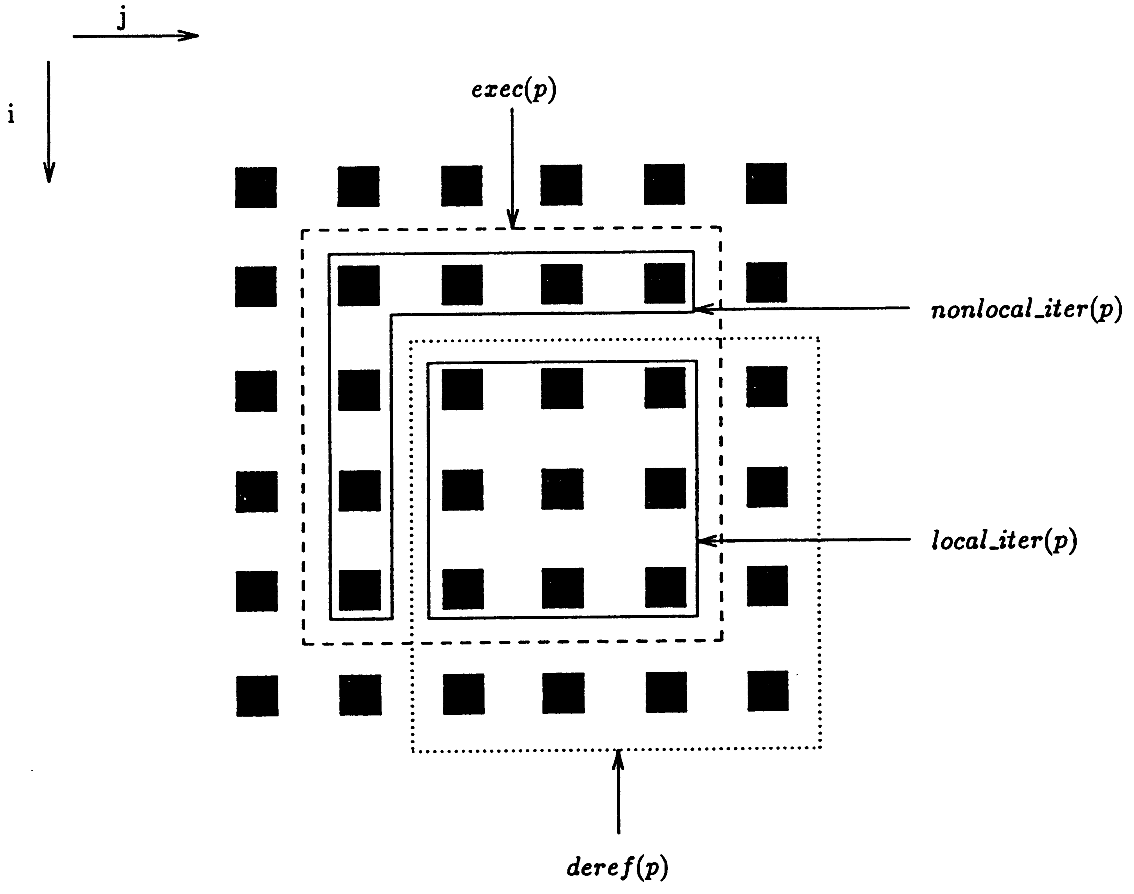

Figure 6: Visualizing iteration sets

1. Restrict attention to specific forms of subscript (e.g, linear functions of **forall** indices) and distributions (e.g, **block** distribution).
2. Derive theorems giving closed-form expressions for the communication and iteration sets based on the subscript and distribution forms chosen.
3. Generate code to evaluate the closed-form expressions given by the theorems.
4. Control communication and iteration in the compiled code by using the results of the expressions above.

Steps 1 and 2 are done when the compiler is designed, and steps 3 and 4 are part of the code-generation strategy. In this paper we examine three common cases to which compile-time analysis can be applied. Section 3.1 first defines the notation we will use. Section 3.2 then gives the theorems for compiling constant subscripts with any array distributions, while Sections 3.3 and 3.4 give the results for linear subscript functions with **block** or **cyclic** distributions. The proofs will only be sketched here; the full proofs can be found in [10]. Each section also shows how its theorems can be applied to a typical numerical algorithm, and presents performance results for that program.

## 3.1 Notation

Ranges of integers will appear frequently in Sections 3.3 and 3.4, so we define a notation for them.

**Definition 9** *A contiguous range of integers is denoted by*

$$[A:B] = \{i \mid A \le i \le B\} \tag{17}$$

*Non-contiguous ranges with a constant (integer) step size are denoted by*

$$[A:B:C] = \{i \mid A \le i \le B \wedge i \equiv A \pmod{C}\} \tag{18}$$

*with $C$ restricted to be positive.*

In order to describe certain properties of ranges, it is convenient to define the following:

**Definition 10** *For integer $a$ and $b$ and positive integer $c$, let $nxt(a,b,c)$ be the smallest integer such that $nxt(a,b,c) \ge a$ and $nxt(a,b,c) \equiv b \pmod{c}$.*

Defining the modulo operator % to always return the positive remainder of its arguments, we can compute $nxt(a,b,c)$ by

$$nxt(a,b,c) = a + (b-a)\,\%\,c$$

The proof of this property is elementary.

Using $nxt(a,b,c)$ we can derive several properties of ranges

**Lemma 1** *For any $c_1 > 0$, $c_2 > 0$, let $n_1$, $n_2$ be integers such that $c_1n_1 + c_2n_2 = \gcd(c_1,c_2)$ and let*

$$m = nxt\left(\max(a_1,a_2),\ \frac{c_1n_1(a_2-a_1)}{\gcd(c_1,c_2)} + a_1,\ lcm(c_1,c_2)\right)$$

*Then*

$$[a_1:b_1:c_1] \cap [a_2:b_2:c_2] = \left\{ \begin{array}{ll} [m:\min(b_1,b_2):lcm(c_1,c_2)] & \textit{if } a_2 \equiv a_1 \pmod{\gcd(c_1,c_2)} \\ \phi & \textit{otherwise} \end{array} \right. \tag{19}$$

$$\begin{aligned} [a_1:b_1] \cap [a_2:b_2] &= [\max(a_1,a_2):\min(b_1,b_2)] & (20)\\ [a_1:b_1] - [a_2:b_2] &= [a_1:\min(b_1,a_2-1)] \cup [\max(a_1,b_2+1):b_2] & (21)\\ [a_1:b_1] \cap [a_2:b_2:c] &= [\max(nxt(a_1,a_2,c),a_2):\min(b_1,b_2):c] & (22) \end{aligned}$$

```
processors procs : array[ 1..NP ] with NP in 1..max_procs;

const    c = 10;

var      A, B : array[ 1..N ] of real dist by [ block ] on procs;

for j in 1..10
   var k : integer;
do
   k := (j*j + j) / 2;
   forall i in low..high on A[i].loc
      var bad : integer;
   do
      bad := round( sqrt( i ) );
      A[i] := B[ c ];          -- OK; compile-time constant
      A[i] := B[ j ];          -- OK; forall loop invariant treated as constant
      A[i] := B[ k ];          -- OK; forall loop invariant treated as constant
      A[i] := B[ j+k ];        -- OK; value of j+k is loop invariant
      A[i] := B[ bad ];        -- Trouble; not forall loop invariant
   end;
end;
```

Figure 7: Forall-invariant subscripts

The proof relies only on elementary number theory. We omit it, as it is tangential to the main thrust of this paper.

The precise conditions that we impose on subscripts are worth noting. In the following, some values used in subscript expressions will be treated as constants. We assume these constants are integers. This is natural, since the subscript itself must be an integer. It is not necessary that these values be computable at compile-time, however. Instead, it suffices for these values to be any expression which is invariant during the **forall** statement. Such invariant expressions can be detected by standard compiler techniques. Figure 7 shows several examples of these. The references $B[c]$, $B[j]$, $B[k]$ and $B[j+k]$ are forall-invariant and can all be handled by the analysis in Section 3.2. Reference $B[bad]$, however, is outside the scope of our analysis because the value of *bad* changes within the same execution of the forall.

## 3.2 Constant Subscripts

The first case that we attack is constant subscripts. Referring back to Figure 1, this is the case in which $f(i) = c$, where $c$ is a constant (or a **forall** invariant, as explained above). No restrictions are necessary on the distributions of arrays $A$ and $New\_A$ or on the form of the **on** clause. The basic results for this class of subscripts are given in Theorem 3.

**Theorem 3** *If the subscripting function in a* **forall** *is* $f(i) = c$ *for some constant* $c$*, then*

$$recv\_set(p,q) = \begin{cases} \{c\} & \text{if } c \in local(q) \text{ and } exec(p) \neq \phi \\ \phi & \text{otherwise} \end{cases} \tag{23}$$

$$send\_set(p,q) = \begin{cases} \{c\} & \text{if } c \in local(p) \text{ and } exec(q) \neq \phi \\ \phi & \text{otherwise} \end{cases} \tag{24}$$

$$local\_iter(p) = \begin{cases} exec(p) & \text{if } c \in local(p) \\ \phi & \text{otherwise} \end{cases} \tag{25}$$

$$nonlocal\_iter(p) = \begin{cases} \phi & \text{if } c \in local(p) \\ exec(p) & \text{otherwise} \end{cases} \tag{26}$$

**Proof.**

We first derive expressions for the *ref* and *deref* functions.

$$\begin{aligned}
ref(p) &= f(exec(p)) \\
&= \{f(i) \mid i \in exec(p)\} \\
&= \{c \mid i \in exec(p)\} \\
&= \begin{cases} \{c\} & \text{if } exec(p) \neq \phi \\ \phi & \text{if } exec(p) = \phi \end{cases}
\end{aligned}$$

$$\begin{aligned}
deref(p) &= f^{-1}(local(p)) \\
&= \begin{cases} Iter & \text{if } c \in local(p) \\ \phi & \text{if } c \notin local(p) \end{cases}
\end{aligned}$$

Direct applications of these expressions to to Equations 9 through 16 produce the desired results.

$$\begin{aligned}
recv\_set(p,q) &= local(q) \cap ref(p) \\
&= \begin{cases} \{c\} & \text{if } exec(p) \neq \phi \text{ and } c \in local(q) \\ \phi & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
send\_set(p,q) &= local(p) \cap ref(q) \\
&= \begin{cases} \{c\} & \text{if } exec(q) \neq \phi \text{ and } c \in local(p) \\ \phi & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
local\_iter(p) &= exec(p) \cap deref(p) \\
&= \begin{cases} exec(p) & \text{if } c \in local(p) \\ \phi & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
nonlocal\_iter(p) &= exec(p) - deref(p) \\
&= \begin{cases} \phi & \text{if } c \in local(p) \\ exec(p) & \text{otherwise} \end{cases}
\end{aligned}$$

□

The expressions for *recv_set*$(p,q)$ and *send_set*$(p,q)$ indicate that one processor is sending a value to all other processors. This type of broadcast is precisely the behavior that we expect from a program repeatedly accessing a fixed array element. It is also possible to exploit this observation by using an efficient broadcast mechanism (such as hardware broadcast or a fan-out tree) instead of sending individual messages to all processors.

A few other implementation issues are easily resolved. The amount of memory needed to store the received values can be found directly from the maximum size of *recv_set*$(p,q)$; a scalar variable is sufficient for the loop of Figure 1. Separate loops for local and nonlocal iterations can be avoided by copying $B[c]$ to the temporary location on the sending processor and considering all iterations on every processor to be nonlocal.

As a realistic example of a program amenable to compile-time analysis, we chose Gaussian elimination without pivoting. As Figure 8 shows, both forward and back substitution were included in the program. The Kali compiler produces a C program with explicit message passing primitives suitable for execution on the iPSC/860. Figure 9 shows a Kali translation of the generated code for the **for** $jj$ loop, which is much more readable than the actual C code. The most important features of that figure are the communication statements and the range of the **for** $i$ loop. Both are found directly from Theorem 3. The translation of the rest of Figure 8 is similar, but produces broadcasts for both the pivot row of $a$ and the pivot element of $x$. A mature compiler would combine these two broadcasts; we are currently pursuing such optimizations. We will discuss the performance of this program in Section 4.

```
processors
   Procs : array[ 1..P ] with P in 1..32;

const
   N : integer = iargv(1);          -- size of matrix (from command line)

var
   a : array[ 1..N, 1..N ] of double dist by [ cyclic, * ] on Procs;
   x : array[ 1..N ] of double dist by [ cyclic ] on Procs;

-- gaussian elimination without pivoting
for k in 1..N-1 do
   forall i in k+1 .. N on a[i,1].loc do
         for j in k+1 .. N do
            a[i,j] := a[i,j] - a[k,j] * a[i,k] / a[k,k];
         end;
         x[i] := x[i] - x[k] * a[i,k] / a[k,k];
   end;
end;
-- back substitution
for jj in 0..N-1
   var j : integer;
do
   -- reverse sense of loop, since Kali doesn't have negative steps
   j := N - jj;
   x[j] := x[j] / a[j,j];
   forall i in 1..j-1 on x[i].loc do
         x[i] := x[i] - x[j] * a[i,j];
   end;
end;
```

Figure 8: Kali program for Gaussian elimination with forward and back substitution

```
-- Code on processor p

for jj in 0..N-1
   var j : integer;
        temp_x : double;                          -- compiler temporary
do
   j := N - jj;

   -- only perform assignment on processor storing x[j]
   if ( j%P = p ) then
        x[j] := x[j] / a[j,j];
   end;

   -- communications statements: broadcasting
   if ( j%P = p ) then
        temp_x := x[j];
        send( temp_x, procs[*] );
   else
        temp_x := recv( procs[*] );
   end;
   -- computation statements (all iterations nonlocal)
   for i in p..j-1 by P do
        x[i] := x[i] - temp_x * a[i,j];
   end;
end;
```

Figure 9: Compiled form of back substitution in Figure 8

## 3.3 Linear Subscripts with Block Distributions

We next consider programs which subscript block-distributed arrays using linear functions of the forall index. The particular restrictions that we place on Figure 1 in this case are

1. All arrays in the program have a block distribution and the same size.
2. The computation is performed on the processor storing element $i$ of one of the arrays.
3. The subscripting function is $f(i) = c_0 i + c_1$, that is, a linear function of the forall index. The discussion on page 13 described our assumptions about $c_0$ and $c_1$.

Programs with these features include relaxation and ADI algorithms for solving partial differential equations on regular grids. In order to generate useful expressions for the communication and iteration sets, it is necessary to consider two general cases for the value of $c_0$: $c_0 > 0$ and $c_0 < 0$. In this paper we will only present the results for $c_0 > 0$, which is by far the more common case. Theorem 4 gives the formulas for this case. The analysis for $c_0 < 0$ is similar in outline; the reader is referred to [10] for a full account of this case. Many programs have the additional property that $|c_0| = 1$; therefore, we will derive special forms of all our equations when $f(i) = i + c$ in Theorem 5

The analysis requires expressions for *local*($p$) and *exec*($p$). The *local* function for block distributions is given by Equation 2. Because of the simple form of the on clause (*New_A*[$i$].loc), the set *exec*($p$) is a restriction of *local*($p$). If we let $M = \lceil \frac{N}{P} \rceil$ be the number of elements on each processor, then we have

$$\begin{aligned} local(p) &= \{i \mid Mp \le i \le Mp + M - 1\} \\ &= [Mp : Mp + M - 1] \qquad (27) \\ exec(p) &= [low : high] \cap local(p) \\ &= [\max(low, Mp) : \min(high, Mp + M - 1)] \qquad (28) \end{aligned}$$

Because the bounds of *exec(p)* will be used so frequently, we designate names for them.

**Definition 11** *Define bot(p) and top(p) as*

$$\begin{aligned} bot(p) &= \max(low, Mp) & (29)\\ top(p) &= \min(high, Mp + M - 1) & (30)\end{aligned}$$

We will also find the following lemma useful

**Lemma 2** *If all arrays in a forall are distributed by block, the subscripting function is $f(i) = c_0 i + c_1$ and $c_0 > 0$, then*

$$\begin{aligned} ref(p) &= [c_0 bot(p) + c_1 : c_0 top(p) + c_1 : c_0] & (31)\\ deref(p) &= \left[\left\lceil \frac{Mp - c_1}{c_0} \right\rceil : \left\lfloor \frac{Mp + M - 1 - c_1}{c_0} \right\rfloor\right] & (32)\end{aligned}$$

**Proof.** Substitute $f(i) = c_0 i + c_1$ and Equations 27 and 28 into Equations 8 and 14 by and simplifying. The floor and ceiling functions come from restricting the expressions to be integers. □

Given this, we can derive the major result of this section.

**Theorem 4** *Let all arrays in a* forall *be distributed by* block, *and let $M = \lceil \frac{N}{P} \rceil$ be the size of the block on each processor. Let the subscripting function used in the* forall *be $f(i) = c_0 i + c_1$ where $c_0$ and $c_1$ are integer constants, $c_0 > 0$, and*

$$\begin{aligned} lb(p,q) &= \max(c_0 bot(p) + c_1, nxt(Mq, c_1, c_0))\\ ub(p,q) &= \min(c_0 top(p) + c_1, Mq + M - 1)\end{aligned}$$

*Then:*
*If $\lceil \frac{c_1+1}{M} \rceil + c_0 p - 1 \le q \le \lfloor \frac{c_1 - c_0}{M} \rfloor + c_0(p+1)$ then*

$$recv\_set(p,q) = [lb(p,q) : ub(p,q) : c_0] \tag{33a}$$

*Otherwise,*

$$recv\_set(p,q) = \phi \tag{33b}$$

*If $\left\lceil \frac{Mp + c_0 - c_1}{M c_0} \right\rceil - 1 \le q \le \left\lfloor \frac{Mp + M - c_1 - 1}{M c_0} \right\rfloor$ then*

$$send\_set(p,q) = [lb(q,p) : ub(q,p) : c_0] \tag{34a}$$

*Otherwise,*

$$send\_set(p,q) = \phi \tag{34b}$$

$$local\_iter(p) = \left[\max\left(bot(p), \left\lceil \frac{Mp - c_1}{c_0} \right\rceil\right) : \min\left(top(p), \left\lfloor \frac{Mp + M - 1 - c_1}{c_0} \right\rfloor\right)\right] \tag{35}$$

*If $c_1 \ge 0$ then*

$$nonlocal\_iter(p) = \left[\max\left(bot(p), \left\lfloor \frac{Mp + M - 1 - c_1}{c_0} \right\rfloor + 1\right) : top(p)\right] \tag{36a}$$

*Otherwise, if $c_1 \le (MP - 1)(1 - c_0)$ then*

$$nonlocal\_iter(p) = \left[bot(p) : \min\left(top(p), \left\lceil \frac{Mp - c_1}{c_0} \right\rceil - 1\right)\right] \tag{36b}$$

*Otherwise,*

$$nonlocal\_iter(p) = \begin{array}{l}\left[\max\left(bot(p), \left\lfloor \frac{Mp+M-1-c_1}{c_0}\right\rfloor + 1\right) : top(p)\right] \cup \\ \left[bot(p) : \min\left(top(p), \left\lceil \frac{Mp-c_1}{c_0}\right\rceil - 1\right)\right]\end{array} \tag{36c}$$

*Note that the conditions on Equations 36a and 36b are both true when* $c_0 = 1$ *and* $c_1 = 0$. *In this case,*

$$nonlocal\_iter(p) = \phi \tag{36d}$$

**Proof.**

Applying Lemma 2 to Equations 9 and 10 produces

$$\begin{aligned}
recv\_set(p,q) &= local(q) \cap ref(p) \\
&= [Mq : Mq + M - 1] \cap [c_0 bot(p) + c_1 : c_0 top(p) + c_1 : c_0] \\
&= [\max(c_0 bot(p) + c_1, nxt(Mq, c_1, c_0)) : \min(c_0 top(p) + c_1, Mq + M - 1) : c_0] \\
send\_set(p,q) &= local(p) \cap ref(q) \\
&= [Mp : Mp + M - 1] \cap [c_0 bot(q) + c_1 : c_0 top(q) + c_1 : c_0] \\
&= [\max(c_0 bot(q) + c_1, nxt(Mp, c_1, c_0)) : \min(c_0 top(q) + c_1, Mp + M - 1) : c_0]
\end{aligned}$$

These expressions apply to all values of $p$ and $q$, but are not in the most efficient form for computation. Many processor pairs will not need to communicate during the computation, and therefore many of the sets will be empty. We can make computation more efficient by finding conditions for $recv\_set(p,q) \neq \phi$ and $send\_set(p,q) \neq \phi$ and not constructing the sets for other values of $p$ and $q$. In outline, this involves fixing $p$ in the above expressions and finding bounds on $q$ based on comparisons of the upper and lower range bounds. We omit the details of this analysis here; the conditions in Equations 33 and 34 incorporate the results.

We now turn our attention to the iteration sets. We obtain expressions for these by applying Lemma 2 to Equations 15 and 16.

$$\begin{aligned}
local\_iter(p) &= exec(p) \cap deref(p) \\
&= [bot(p) : top(p)] \cap \left[\left\lceil \frac{Mp - c_1}{c_0}\right\rceil : \left\lfloor \frac{Mp + M - 1 - c_1}{c_0}\right\rfloor\right] \\
&= \left[\max\left(bot(p), \left\lceil \frac{Mp - c_1}{c_0}\right\rceil\right) : \min\left(top(p), \left\lfloor \frac{Mp + M - 1 - c_1}{c_0}\right\rfloor\right)\right] \\
nonlocal\_iter(p) &= exec(p) - deref(p) \\
&= [bot(p) : top(p)] - \left[\left\lceil \frac{Mp - c_1}{c_0}\right\rceil : \left\lfloor \frac{Mp + M - 1 - c_1}{c_0}\right\rfloor\right] \\
&= \left[bot(p) : \min(top(p), \left\lceil \frac{Mp - c_1}{c_0}\right\rceil - 1)\right] \cup \\
&\qquad \left[\max\left(bot(p), \left\lfloor \frac{Mp + M - 1 - c_1}{c_0}\right\rfloor + 1\right) : top(p)\right]
\end{aligned}$$

The equation for *local_iter(p)* is now in its final form, suitable for use in an implementation. The formula for *nonlocal_iter(p)*, however, is the union of two disjoint ranges, which is more difficult to use. This situation is analogous to the situation for the communication sets, in which we have a correct but computationally expensive formula. As we did then, we now identify simplifying cases; in particular, we find cases in which one of the unioned ranges is empty. We omit the full analysis and present the results in Equation 36.

□

We now specialize Theorem 4 to the important case of $c_0 = 1$.

**Theorem 5** *If all arrays in a* **forall** *are distributed by* **block** *with $M = \lceil \frac{N}{P} \rceil$ elements per processor and the subscripting function is $f(i) = i + c$, then let*

$$\begin{aligned} lb(p,q) &= \max(bot(p) + c, Mq) \\ ub(p,q) &= \min(top(p) + c, Mq + M - 1) \end{aligned}$$

*Then:*
*If* $\lceil \frac{c+1}{M} \rceil + p - 1 \leq q \leq \lfloor \frac{c-1}{M} \rfloor + p + 1$ *then*

$$recv\_set(p,q) = [lb(p,q) : ub(p,q)] \tag{37a}$$

*Otherwise,*

$$recv\_set(p,q) = \phi \tag{37b}$$

*If* $\lceil \frac{1-c}{M} \rceil + p - 1 \leq q \leq \lfloor \frac{-1-c}{M} \rfloor + p + 1$ *then*

$$send\_set(p,q) = [lb(q,p) : ub(q,p)] \tag{38a}$$

*Otherwise,*

$$send\_set(p,q) = \phi \tag{38b}$$

$$local\_iter(p) = [\max(bot(p), Mp - c) : \min(top(p), Mp + M - 1 - c)] \tag{39}$$

*If* $c > 0$ *then*

$$nonlocal\_iter(p) = [\max(bot(p), Mp + M - c) : top(p)] \tag{40a}$$

*If* $c < 0$ *then*

$$nonlocal\_iter(p) = [bot(p) : \min(top(p), Mp - c + 1)] \tag{40b}$$

*Otherwise,*

$$nonlocal\_iter(p) = \phi \tag{40c}$$

**Proof.** Substitute $c_0 = 1$ and $c_1 = c$ in Theorem 4. The bounds on *nonlocal_iter(p)* can be improved by using intermediate results from the full derivation rather than bounding the expressions in Equation 36. □

The formulas of Theorems 4 and 5 can be used directly in an implementation. Ranges of iterations correspond directly to for loop bounds and steps, while ranges of array subscripts describe sections of arrays. One issue of some subtlety is allocation of memory to hold off-processor data. The size of the necessary buffers must be calculated from the cardinality of the sets; given this information, temporary variables for the buffers can be statically declared or dynamically allocated. (The current Kali implementation uses dynamic allocation.) To avoid sparse use of storage, addressing into these buffers should ensure that the range is stored contiguously. This forces the range step size to be accounted for in the addressing formulas by using division in the address calculation. Some additional optimizations are possible if $|c_0| = 1$. Buffer addressing need not be complex, since the ranges for the communication sets have unit stride. Also, under the assumptions we have made in this section, the conditions on $q$ in the communication sets allow at most two sets to be nonempty. These values of $q$ can be kept explicitly for quick reference.

As a realistic example of a program amenable to compile-time analysis, we chose the cyclic reduction algorithm for solving a tridiagonal linear system. Figure 10 shows the program used.

```
processors
   Procs : array[ 0..P-1 ] with P in 1..32;

const
   N : integer = iargv(1);             -- size of matrix (from command line)

var
   l, d, u, x, y, l_tmp, u_tmp, y_tmp : array [ 0..N-1 ] of real
         dist by [block] on Procs;
   k : integer;

k := 1;
while (k < N) do
   -- normalize main diagonal to 1's
   forall i in 0..N-1 on d[i].loc
         var m : real;
   do
         m := 1.0 / d[i];
         d[i] := 1.0;
         y[i] := m * y[i];
         l_tmp[i] := m * l[i];
         u_tmp[i] := m * u[i];
         y_tmp[i] := y[i];
   end;
   -- eliminate non-main diagonal entries
   forall i in k..N-1 on d[i].loc do
         d[i] := d[i] - l_tmp[i]*u_tmp[i-k];
         y[i] := y[i] - l_tmp[i]*y_tmp[i-k];
         l[i] := - l_tmp[i] * l_tmp[i-k];
   end;
   forall i in 1..N-k-1 on d[i].loc do
         d[i] := d[i] - u_tmp[i]*l_tmp[i+k];
         y[i] := y[i] - u_tmp[i]*y_tmp[i+k];
         u[i] := - u_tmp[i] * u_tmp[i+k];
   end;
   -- update k and go to next iteration
   k mult= 2;
end;
-- final solution
forall i in 1..n on x[i].loc do
   x[i] := y[i] / d[i];
end;
```

Figure 10: Kali program for cyclic reduction with block distributions

Despite the name, cyclic reduction can be applied to arrays with any distribution pattern. As with the Gaussian elimination example, the Kali compiler used the expressions derived in Theorem 5 to produce a C program with explicit message passing. We show the translation of the second **forall** statement as a Kali program in Figure 11. In this case, $f(i) = i - k$, so $c = -k$ in Theorem 5. Declarations of variables and arrays are not shown. The translation of the third **forall** statement is similar; the first **forall** does not require communication. Since the subscripts in the expressions $u\_tmp[i-k]$, $y\_tmp[i-k]$, and $l\_tmp[i-k]$ are the same, the compiler can (and does) bundle three messages into one to reduce communication costs. The complexity of the communications setup is due to the fact that $k$, $N$, and $P$ are not known at compile-time, forcing the use of the general form of Theorem 5. If those values were true constants, the compiler could do the calculations and substitute the appropriate values where needed. The generated program also demonstrates the advantage of compiler generation of communications statements; if only a message-passing environment were available, then this mass of code would have to be produced by the programmer. It should be obvious that such code is difficult to write, understand, and debug. Section 4 discusses the performance of this program.

## 3.4 Linear Subscripts with Cyclic Distributions

The final case which we analyze is similar to the last section. We again restrict subscripts to linear functions of the **forall** index, that is, $f(i) = c_0 i + c_1$ in Figure 1. We also retain the restriction on form of the **on** clause. We now require arrays to be distributed by **cyclic** rather than **block**, however. As in Section 3.3, we derive closed-form expressions for the communication and iteration sets induced by these conditions. These results appear in Theorem 6. We also derive a simplified form of the expressions for the case of $c_0 = 1$ in Theorem 7.

Once again we will need expressions for $local(p)$ and $exec(p)$. The $local$ function for cyclic distributions is given by Equation 3, and $exec(p)$ is again a restriction of $local(p)$. We give those equations here for convenience.

$$\begin{aligned} local(p) &= \{i \mid i \equiv p \pmod{P}\} && (41) \\ exec(p) &= [low : high] \cap local(p) \\ &= [nxt(low, p, P) : high : P] && (42) \end{aligned}$$

As in the last section, the bounds on $exec(p)$ will be useful. Thus, we make the following definition.

**Definition 12** *Define $bot(p)$ and $top(p)$ as*

$$\begin{aligned} bot(p) &= nxt(low, p, P) && (43) \\ top(p) &= nxt(high - P + 1, p, P) && (44) \end{aligned}$$

The definition makes $top(p)$ the largest integer less than $high$ which is equivalent to $p$ modulo $P$. This is the exact upper bound on the range, rather than an upper bound which is not reached. We also require the following counterpart to Lemma 2.

**Lemma 3** *Let the subscripting function be $f(i) = c_0 i + c_1$, and $c_0 > 0$, let $G = \gcd(P, c_0)$ and let $n$ and $m$ be such that $c_0 n + Pm = \gcd(P, c_0)$. If all arrays in a* **forall** *are distributed cyclically, then*

$$ref(p) = [c_0 bot(p) + c_1 : c_0 top(p) + c_1 : c_0 P] \qquad (45)$$

$$deref(p) = \begin{cases} \phi & \text{if } p \not\equiv c_1 \pmod{G} \\ \{i \mid i \equiv \frac{n(p-c_1)}{G} \pmod{P/G}\} & \text{if } p \equiv c_1 \pmod{G} \end{cases} \qquad (46)$$

*Note that top and bot in this lemma are those given by Definition 12, not 11*

**Proof.** The formulas are derived from Equations 8 and 14 by substituting $f(i) = c_0 i + c_1$, Equations 41, and 42 and simplifying. The conditions on $deref(p)$ come from elementary number theory.

```
-- Code on processor p
M = ceil( N / P );
my_bot := max( k, M*p );
my_top := min( N-1, M*p+M-1 );
recv_q1 := p + ceil( (1-k) / M ) - 1;          -- first proc to receive from
recv_low1 := max( my_bot-k, M*recv_q1 );
recv_hi1 := min( my_top-k, M*recv_q1+M-1 );
recv_q2 := p + floor( (-1-k) / M ) + 1;        -- other proc to recv from
recv_low2 := max( my_bot-k, M*recv_q2 );
recv_hi2 := min( my_top-k, M*recv_q2+M-1 );
send_q1 := p + ceil( (k+1) / M ) - 1;          -- first proc to send to
send_low1 := max( k-k, M*send_q1-k, M*p );
send_hi1 := min( N-1-k, M*send_q1+M-1-k, M*p+M-1 );
send_q2 := p + floor( (k-1) / M ) + 1;         -- other proc to send to
send_low2 := max( k-k, M*send_q2-k, M*p );
send_hi2 := min( N-1-k, M*send_q2+M-1-k, M*p+M-1 );
local_low := max( my_bot, M*p+k );             -- local iterations
local_hi := min( my_top, M*p+M-1+k );
nonlocal_low := my_bot;                        -- nonlocal iterations
nonlocal_hi := min( my_top, M*p+k-1 );
-- send to other processors
if ( legal_proc(send_q1) and send_low1<=send_high1 ) then
   send( Procs[send_q1], u_tmp[send_low1..send_hi1],
         y_tmp[send_low1..send_hi1], l_tmp[send_low1..send_hi1] );
end;
if ( legal_proc(send_q2) and send_low2<=send_high2 and send_q2<>send_q1 ) then
   send( Procs[send_q2], u_tmp[send_low2..send_hi2],
         y_tmp[send_low2..send_hi2], l_tmp[send_low2..send_hi2] );
end;
-- local computations
for i in local_low..local_high do
   d[i] := d[i] - l_tmp[i]*u_tmp[i-k];
   y[i] := y[i] - l_tmp[i]*y_tmp[i-k];
   l[i] := - l_tmp[i] * l_tmp[i-k];
end;
-- receive messages
if ( legal_proc(recv_q1) and recv_low1<=recv_hi1 ) then
   tmp1[recv_low1..recv_hi1], tmp2[recv_low1..recv_hi1],
         tmp3[recv_low1..recv_hi1] := recv( Procs[recv_q1] );
end;
if ( legal_proc(recv_q2) and recv_low2<=recv_hi2 and recv_q2<>recv_q1) then
   tmp2[recv_low2..recv_hi2], tmp2[recv_low2..recv_hi2],
         tmp3[recv_low2..recv_hi2] := recv( Procs[recv_q2] );
end;
-- nonlocal computations
forall i in k..N-1 on d[i].loc do
   d[i] := d[i] - l_tmp[i]*tmp1[i-k];
   y[i] := y[i] - l_tmp[i]*tmp2[i-k];
   l[i] := - l_tmp[i] * tmp3[i-k];
end;
```

Figure 11: Compiled form of part of Figure 10

Values of $n$ and $m$ such that $c_0 n + Pm = \gcd(P, c_0)$ can be found by the Extended Euclid's algorithm [1]. $\Box$

We can now prove Theorem 6.

**Theorem 6** *If all arrays in a forall are distributed cyclically, the subscripting function is $f(i) = c_0 i + c_1$ and $c_0 > 0$, then let $G = \gcd(P, c_0)$ and let $n$ and $m$ be such that $c_0 n + Pm = \gcd(P, c_0)$. Then:*
*If $q \equiv c_0 p + c_1 \pmod P$ then*

$$recv\_set(p,q) = [c_0 bot(p) + c_1 : c_0 top(p) + c_1 : c_0 P] \tag{47a}$$

*Otherwise,*

$$recv\_set(p,q) = \phi \tag{47b}$$

*If $p \equiv c_1 \pmod G$ and $q \in \left[\left(\frac{n(p-c_1)}{G}\right) \% \left(\frac{P}{G}\right) : P-1 : \frac{P}{G}\right]$ then*

$$send\_set(p,q) = [c_0 bot(q) + c_1 : c_0 top(q) + c_1 : c_0 P] \tag{48a}$$

*Otherwise,*

$$send\_set(p,q) = \phi \tag{48b}$$

*If $p \equiv c_1 \pmod G$ and $p \equiv \frac{n(p-c_1)}{G} \pmod{P/G}$ then*

$$local\_iter(p) = exec(p) \tag{49a}$$

*Otherwise,*

$$local\_iter(p) = \phi \tag{49b}$$

*If $p \not\equiv c_1 \pmod G$ or $p \not\equiv \frac{n(p-c_1)}{G} \pmod{P/G}$ then*

$$nonlocal\_iter(p) = exec(p) \tag{50a}$$

*Otherwise,*

$$nonlocal\_iter(p) = \phi \tag{50b}$$

**Proof.**
We obtain the communication set by applying Lemma 3 to Equation 9.

$$\begin{aligned} recv\_set(p,q) &= local(q) \cap ref(p) \\ &= \{i \mid i \equiv q \pmod P\} \cap [c_0 bot(p) + c_1 : c_0 top(p) + c_1 : c_0 P] \\ &= \begin{cases} [c_0 bot(p) + c_1 : c_0 top(p) + c_1 : c_0 P] & \text{if } c_0 bot(p) + c_1 \equiv q \pmod P \\ \phi & \text{otherwise} \end{cases} \\ &= \begin{cases} [c_0 bot(p) + c_1 : c_0 top(p) + c_1 : c_0 P] & \text{if } c_0 p + c_1 \equiv q \pmod P \\ \phi & \text{otherwise} \end{cases} \end{aligned}$$

(The last step follows because $bot(p) \equiv p \pmod P$.) This form is efficiently computable, since for each $p$ there will be exactly one $q$ with a nonempty $recv\_set(p,q)$, easily found by taking the remainder of $c_0 p + c_1$. The corresponding expression for $send\_set(p,q)$,

$$send\_set(p,q) = \begin{cases} [c_0 bot(q) + c_1 : c_0 top(q) + c_1 : c_0 P] & \text{if } c_0 q + c_1 \equiv p \pmod P \\ \phi & \text{otherwise} \end{cases}$$

is not acceptable, because there may be a number of nonempty sets which cannot be quickly computed from this form. We therefore characterize the solutions to $c_0 q + c_1 \equiv p \pmod{P}$. This can be done by number theory and is similar to the derivation of Equation 46; we omit the details of the analysis.

To derive the iteration sets, we apply Equation 46 to Equations 15 and 16.

$$\begin{aligned} local\_iter(p) &= exec(p) \cap deref(p) \\ &= [bot(p) : top(p) : P] \cap deref(p) \end{aligned}$$

Again, conditions allowing the intersection to be nonempty can be derived from number theory. Once this is done, the expression for *nonlocal_iter(p)* can be derived immediately using the fact that the iteration sets are complements. □

We now specialize Theorem 6 to the case of $c_0 = 1$.

**Theorem 7** *If all arrays in a forall are distributed cyclically and the subscripting function is $f(i) = i + c$, then:*

*If $q = (p + c) \% P$ then*

$$recv\_set(p, q) = [bot(p) + c : top(p) + c : P] \tag{51a}$$

*Otherwise,*

$$recv\_set(p, q) = \phi \tag{51b}$$

*If $q = (p - c) \% P$ then*

$$send\_set(p, q) = [bot(q) + c : top(q) + c : c_0 P] \tag{52a}$$

*Otherwise,*

$$send\_set(p, q) = \phi \tag{52b}$$

*If $c \% P = 0$ then*

$$local\_iter(p) = exec(p) \tag{53a}$$

*Otherwise,*

$$local\_iter(p) = \phi \tag{53b}$$

*If $c \% P \neq 0$ then*

$$nonlocal\_iter(p) = exec(p) \tag{54a}$$

*Otherwise,*

$$nonlocal\_iter(p) = \phi \tag{54b}$$

**Proof.** By simple substitution. □

The remarks at the end of Section 3.3 regarding the use of theorems in implementation apply here as well. Allocation and indexing of temporary arrays must take into account the fact that only every $P$th element will be accessed. This is handled by the mechanisms for allocating and addressing the local sections of cyclic-distributed arrays, namely using a division in the addressing formulas. (When the number of processors is a power of two, as it always is on hypercube machines, the division can be replaced with a shift.) In addition, we can use the technique of assuming all iterations are nonlocal to reduce the number of generated loops, as was done for the constant-subscript case.

As a realistic example of a program amenable to this analysis, we chose the cyclic reduction algorithm applied to arrays distributed by cyclic. The input program is identical to Figure 10

```
-- Code on processor p

my_bot := nxt( k, p, P );
my_top := nxt( N-P, p, P );

recv_q := (p-k) % P;        -- processor to receive from
recv_low := my_bot-k;
recv_hi := my_top-k;

send_q := (p+k) % P;        -- processor to send to
send_low := nxt( k, send_q, P ) - k;
send_hi := nxt( N-P, send_q, P ) - k;

-- send messages
if ( send_q <> p ) then
   send( Procs[send_q], u_tmp[send_low..send_hi],
         y_tmp[send_low..send_hi], l_tmp[send_low..send_hi] );
end;

-- no local computations

-- receive messages
if ( recv_q <> p ) then
   tmp1[recv_low..recv_hi], tmp2[recv_low..recv_hi],
         tmp3[recv_low..recv_hi] := recv( Procs[recv_q] );
else
   tmp1[recv_low..recv_hi by P] := u_tmp[recv_low..recv_hi by P];
   tmp2[recv_low..recv_hi by P] := y_tmp[recv_low..recv_hi by P];
   tmp3[recv_low..recv_hi by P] := l_tmp[recv_low..recv_hi by P];
end;

-- nonlocal computations
for i in my_bot..my_top by P do
   d[i] := d[i] - l_tmp[i]*tmp1[i-k];
   y[i] := y[i] - l_tmp[i]*tmp2[i-k];
   l[i] := - l_tmp[i] * tmp3[i-k];
end;
```

Figure 12: Compiled form of Figure 10 with cyclic distribution

except for the distributions of the arrays. The compiler applied Theorem 7 to compile this program. The code generated for the second **for** $i$ loop is shown in Figure 12. The process of compilation is similar to that for Figure 11. Because the underlying formulas differ, however, Figure 12 is very different from Figure 11. These changes illustrate another advantage of the Kali model of compilation: flexibility. In Kali, experimenting with different data distributions is simply a matter of changing a single declaration, while in a conventional message-passing system the entire structure of the program must be changed. When combined with systems for predicting performance such as [3], this offers a valuable tool for creating efficient nonshared-memory programs.

## 4 Performance

To evaluate the effectiveness of the Kali compiler, we compiled and ran the programs described in Sections 3.2, 3.3, and 3.4. All programs were run on an iPSC/860 with 8 Mbyte of memory per node. No other programs were executed for the duration of our tests. To ensure timing accuracy, an outer loop was added so that execution times were much larger than the granularity of the system clock. To ensure correctness, the linear systems to be solved were constructed with a known result vector, and the computed solution was checked against these values. No significant variation in timings or numerical errors occurred. The C programs output by the Kali compiler were compiled with the GreenHills C compiler with optimization turned on.

As a basis for comparison, we also implemented the same algorithms in C. The C versions were optimized as much as possible, subject to the constraint that they be written in a "similar" style to the Kali compiler output. Optimizations that were performed included expression simplification and combining messages to the same processor. We did not apply low-level transformations such as induction variable elimination; both the Kali and C versions would be amenable to these optimizations. Similarly, we did not change the algorithm or data distributions for the hand-written versions, although this might have improved performance. The hand-written C programs were compiled with the same options as the Kali compiler output.

Figures 13 and 14 show performance results for the Gaussian elimination programs. Figure 13 shows the execution times of both the Kali and C programs for various matrix and machine sizes. The solid lines with symbols are the times for compiled Kali code; the dotted lines immediately below them are the times for hand-written C code. The top performance curve is cut off because memory constraints prevented running the $1024 \times 1024$ problem on one or two processors. The Kali program is from 11% to 58% slower than the C version. The larger discrepancies occur when the amount of data per node is small. In these cases, communication dominates the total time. This gives a substantial advantage to the hand-written version, which combines the broadcasts of the pivot row and element of the solution vector. Figure 14 shows the parallel speedups for the Kali program for all matrix sizes that fit on a single node. Communications overhead and load balancing prevent good performance for small matrices spread over many processors, but performance for larger problems is acceptable. These performance limitations are inherent in the Gaussian elimination algorithm; they also appear in the C language version of the program.

Figures 15 and 16 show performance results for the program of Figure 10. In this case, we did not hand-write a C version because the performance of the algorithm with cyclic distribution is better. Analysis of the generated code, however, suggests that the hand-optimized version of the block-distributed algorithm would show less improvement than the cyclic version. This statement is based on the observation that there is less to improve in the block version. For example, the cyclic version used a more complex subscripting formula which the hand-written version eliminated, but the block subscripting formula has no exploitable complexity. The times shown here are unimpressive because the communication overhead is very high in comparison to the computation, as shown for $N = 500$ in Figure 16. Although the computation time does decrease approximately linearly, the communication time does not, preventing perfect speedup. Again, these effects were due to the nature of the algorithm, not to the compilation strategy. These effects would be less apparent for larger problems, but we were unable to test those cases due to an apparent bug in the C compiler.

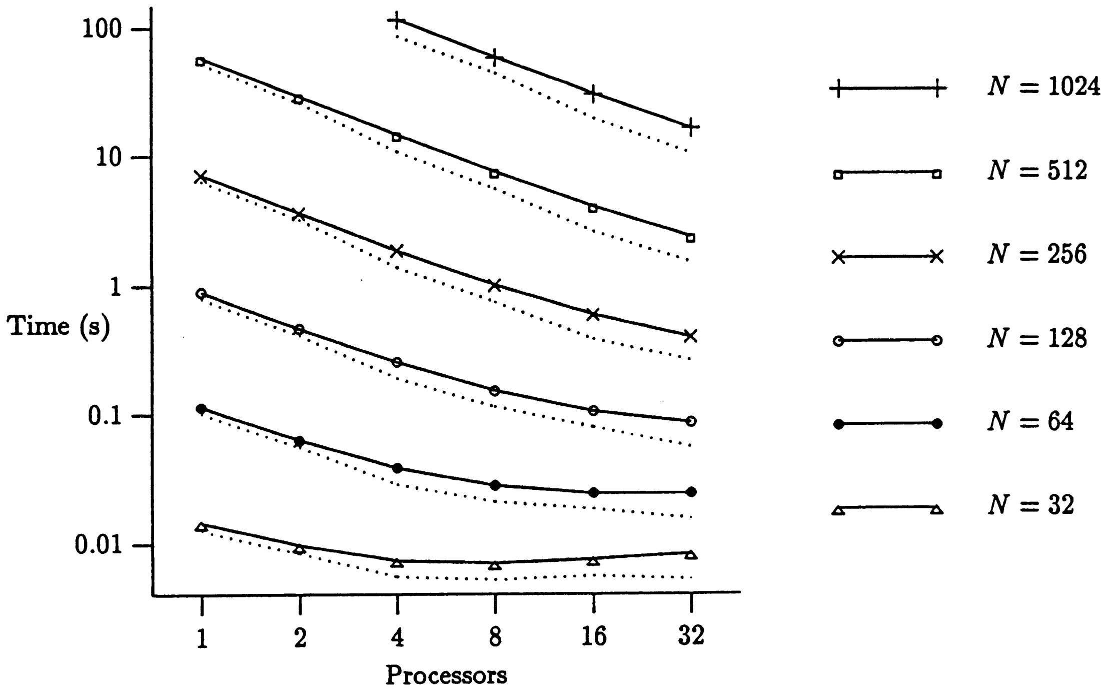

Figure 13: Execution times for Gaussian elimination

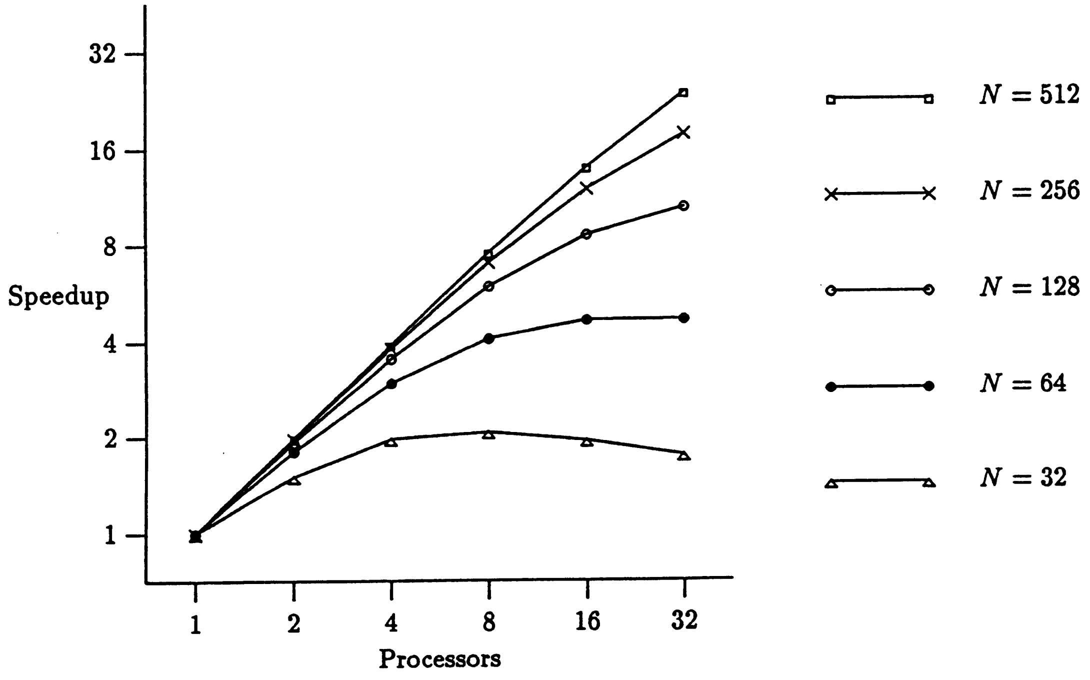

Figure 14: Parallel speedups for Gaussian elimination

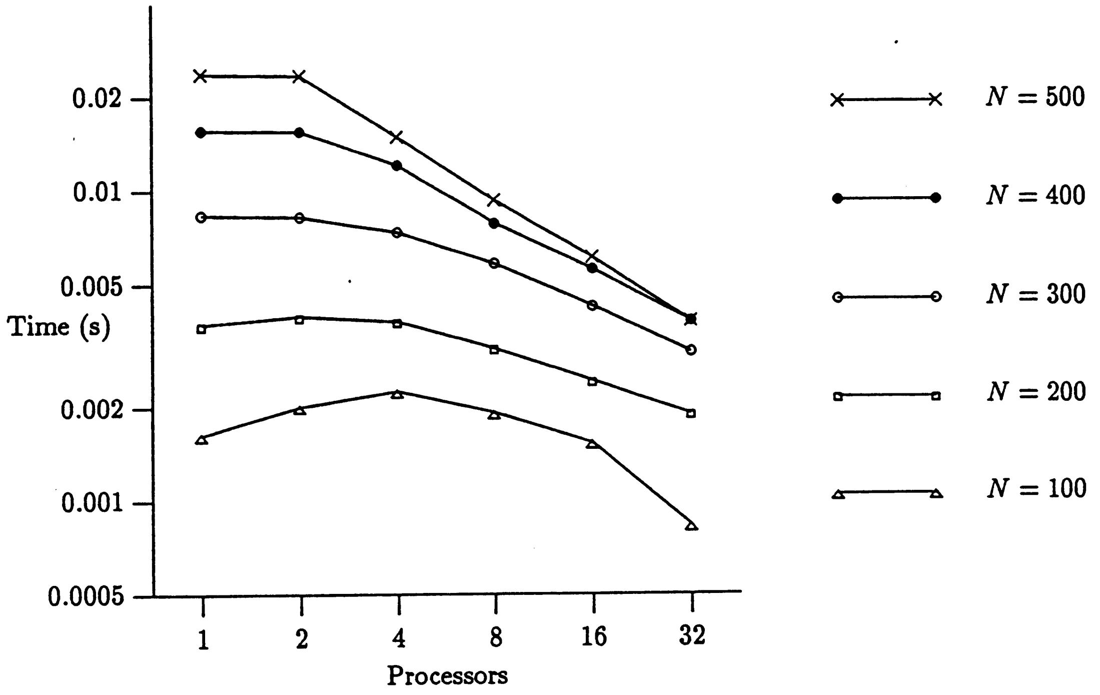

Figure 15: Execution times for cyclic reduction with **block** distribution

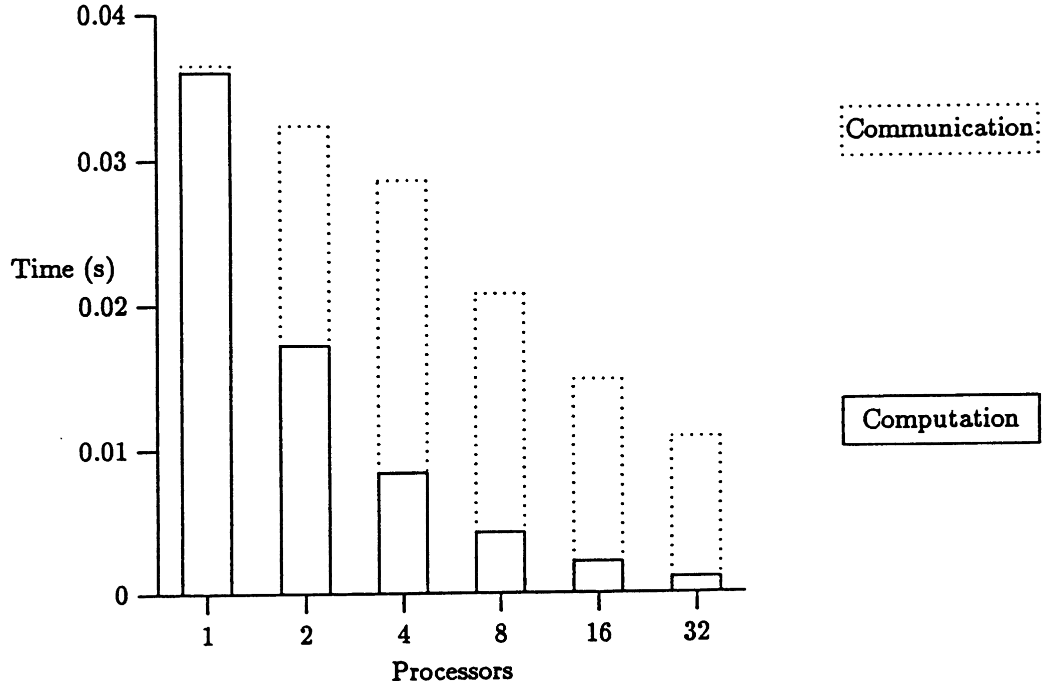

Figure 16: Computation and communication times for cyclic reduction with **block** distribution ($N = 500$)

Figures 17 and 18 show performance results for the **cyclic**-distributed version of cyclic reduction. As in Figure 13, the solid lines represent times for the Kali program and the dotted lines represent the C program. The Kali program ranged from 28% slower to 22% faster than the hand-coded program. We cannot explain all the cases in which the hand-coded program was slower, but the differences are repeatable and at least 10 times larger than the variance of the data. The fact that many of them occur for small per-node array sizes offers a clue. The Kali compiler implemented the array copies of Figure 12 in the natural way, by copying all elements. The hand-crafted program, on the other hand, merely set a pointer to the start of the local array section. For small arrays, this is a very minor savings, and could conceivably create second-order effects that mask it entirely. Parallel speedups are better than for the **block**-distributed program, primarily because the later stages of the reduction do not require communication, while all stages of the **block** version do. Figure 18 is the counterpart of Figure 16 for the **cyclic** variant. The communication cost shown for one processor corresponds to the cost of the array copies, since these would not be present in an optimized sequential implementation. Again, we see that computation scales nearly linearly, while the communication does not.

## 5 Related Work

There is a great deal of current research aimed at providing high-level languages for nonshared memory machines. In this section we will mention only the most closely related work.

Relatively few other groups have considered explicit models of data distribution for the purposes of compilation. Those who have [4, 7, 14, 18] have taken a different path toward formalizing the distribution. Generally, these approaches define a function

$$proc : Elem \rightarrow Procs : proc(a) = p, \qquad \text{where } p \text{ is the processor storing } a$$

If every element is stored on exactly one processor, then the two approaches are equivalent. (In this case, *local* is simply $proc^{-1}$.) If an element can be stored on more than one processor, however, the two methods are not equivalent. It is not obvious how such a distribution scheme could be modeled using a single-valued *proc* function. Our model therefore appears more general than others.

There are many other groups that translate shared-memory code with partitioning annotations into message-passing programs. Recent research from these groups includes [2, 4, 5, 8, 14, 16, 17, 19, 22, 24]. All of these groups produce highly efficient code for the problems to which our compile-time analysis applies. Some of their methods, particularly those in [2, 4, 14, 17, 22, 24], appear to be more generally applicable than ours. Their means of code generation differ significantly from ours and from each other. In general, others' methods work bottom-up, that is, from the innermost levels of loop nests outwards. This is particularly true for [2, 4, 8, 24], which insert communication primitives in the innermost loops and use aggressive program transformations to move those statements to outer loop levels. The bottom-up approach has the advantage that it can be applied to any loops, while our approach is specific to bf forall statements. For **forall** loops, however, our methods appear to require less compiler overhead. A hybrid approach of transforming general loops into **forall** loops and then applying the techniques of this paper could combine these advantages. We are researching this possibility.

Three of the above groups are particularly advanced. Gerndt [8] introduces the concept of "overlap" between the sections of arrays stored on different processors and shows how it can be automatically computed. This has advantages for accessing nonlocal array elements once they have been received. Such overlap areas seem to fit nicely into the formalism of Section 2, and we are investigating incorporating them into our research. Tseng [22] has developed the AL language and its compiler targeted to the WARP systolic array. An important advance over our work is that AL automatically generates the distribution for an array, including overlap information, given only the programmer's specification of which dimension is to be distributed. This is a significant improvement over the **block** and **cyclic** declarations used in Kali, as determining a good distribution for an array is a significant intellectual challenge. A large portion of Chen and Li's work [13, 14] is also concerned

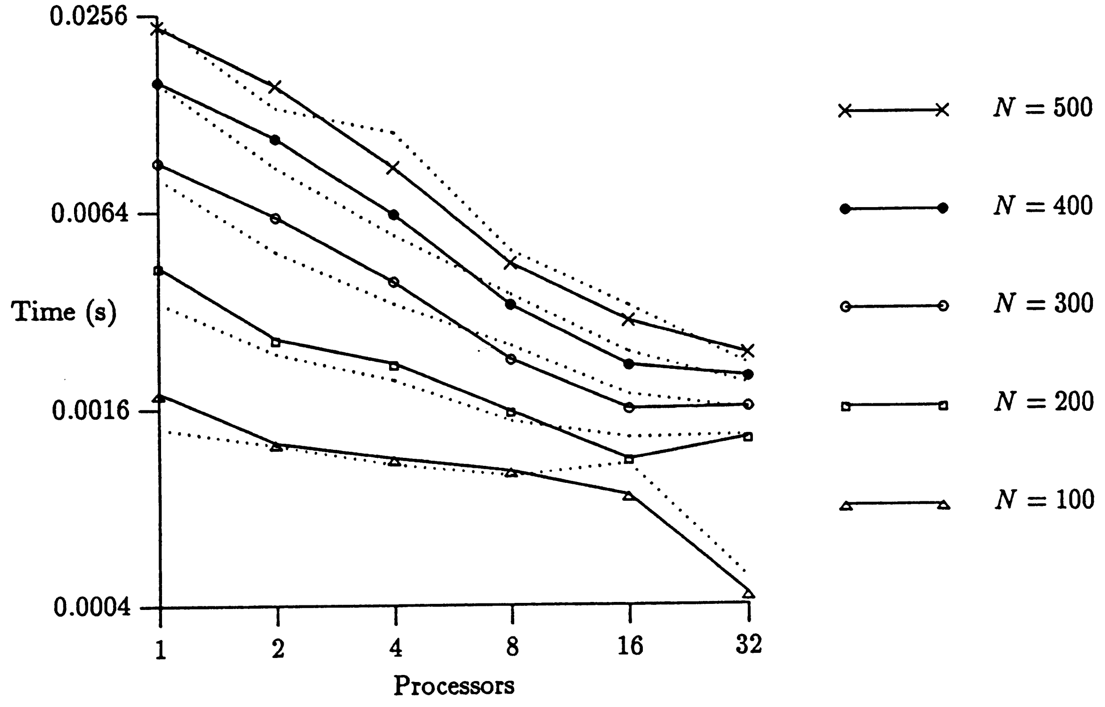

Figure 17: Execution times of cyclic reduction with cyclic distribution

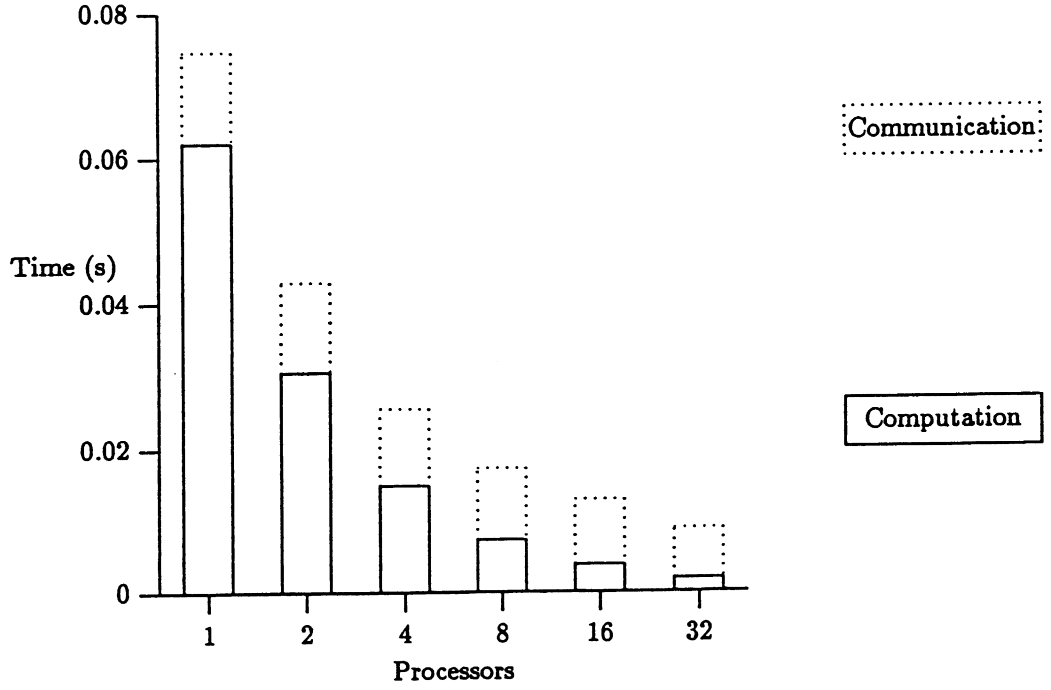

Figure 18: Computation and communication times for cyclic reduction with cyclic distribution ($N = 500$)

with automatically distributing data among processors. Their method, however, requires classifying the dimensions of an array as "temporal" and "spatial." These distinctions are clear in the single-assignment language Crystal used by Chen and Li, but it is less obvious how they can be applied to imperative languages.

Of the works mentioned above, only [5, 14, 17] explicitly consider run-time message generation like that mentioned in Section 2.6. Another group who has been very active in implementing this class of program is Saltz and his coworkers [15, 20, 21]. While such methods are beyond the scope of this paper, they are needed for generality in compilers for languages like Kali.

## 6 Conclusions

Current programming environments for distributed memory architectures provide little support for mapping applications to the machine. In particular, the lack of a global name space implies that the algorithms have to be specified at a relatively low level. This greatly increases the complexity of programs, and also hard wires the algorithm choices, inhibiting experimentation with alternative approaches.

In this paper, we described an environment which allows the user to specify algorithms at a higher level. By providing a global name space, our system allows the user to specify data parallel algorithms in a more natural manner. The user needs to make only minimal additions to a high level "shared memory" style specification of the algorithm for execution in our system; the low level details of message-passing, local array indexing, and so forth are left to the compiler. Our system performs these transformations automatically, producing relatively efficient executable programs.

The fundamental problem in mapping a global name space onto a distributed memory machine is generation of the messages necessary for communication of nonlocal values. In this paper, we presented a framework which can systematically and automatically generate these messages, using either compile time or run time analysis of communication patterns. In this paper we concentrated on the case of compile-time analysis. This method produces very efficient code in the cases where it can be applied. We have demonstrated that the compiler can produce code similar to that which a human programmer might produce, and validated its performance against a simple performance model.

## References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] F. André, J.-L. Pazat, and H. Thomas. PANDORE: A system to manage data distribution. In *International Conference on Supercomputing*, pages 380–388, June 1990.

[3] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. An interactive environment for partitioning and distribution. In *5th Distributed Memory Computing Conference*, pages 1160–1170, Charleston, SC, April 8-12 1990.

[4] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, 1988.

[5] A. Chueng and A. P. Reeves. The Paragon multicomputer environment: A first implementation. Technical Report EE-CEG-89-9, Cornell University Computer Engineering Group, Ithaca, NY, July 1989.

[6] G. Dahlquist and Å. Björck. *Numerical Methods.* Prentice-Hall, Eaglewood Cliffs, NJ, 1974.

[7] K. Gallivan, W. Jalby, and D. Gannon. On the problem of optimizing data transfers for complex memory systems. In *Conference Proceedings of the International Conference on Supercomputing*, pages 238–253, St. Malo, France, July 1988. ACM Press.

[8] H. M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, University of Bonn, December 1989.

[9] G. H. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MD, second edition, 1989.

[10] C. Koelbel. *Compiling Programs for Nonshared Memory Machines.* PhD thesis, Purdue University, West Lafayette, IN, August 1990.

[11] C. Koelbel, P. Mehrotra, J. Saltz, and S. Berryman. Parallel loops on distributed machines. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 9-12 1990.

[12] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 177–186, Seattle, WA, March 14-16 1990.

[13] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. Technical Report YALEU/DCS/TR-725, Yale University, New Haven, CT, November 1989.

[14] J. Li and M. Chen. Synthesis of explicit communication from shared-memory program references. Technical Report YALEU/DCS/TR-755, Yale University, New Haven, CT, May 1990.

[15] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 140–152, St. Malo, France, 1988.

[16] M. J. Quinn and P. J. Hatcher. Compiling SIMD programs for MIMD architectures. In *Proceedings of the 1990 IEEE International Conference on Computer Language*, pages 291–296, March 1990.

[17] A. Rogers. *Compiling for Locality of Reference.* PhD thesis, Cornell University, Ithaca, NY, August 1990.

[18] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 69–80, June 21-23 1989.

[19] M. Rosing, R. W. Schnabel, and R. P. Weaver. Expressing complex parallel algorithms in DINO. In *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers, and Applications*, pages 553–560, 1989.

[20] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[21] Joel Saltz, Harry Berryman, and Janet Wu. Multiprocessors and runtime compilation. ICASE report 90-59, Institute for Computer Applications in Science and Engineering, Hampton, VA, 1990.

[22] P. S. Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1989.

[23] D. M. Young. *Iterative Solution of Large Linear Systems.* Academic Press, New York, NY, 1971.

[24] H. Zima, H. Bast, and M. Gerndt. *Parallel Computing*, volume 6, chapter Superb: A Tool for Semi-Automatic MIMD/SIMD Parallelization, pages 1–18. North-Holland, Amsterdam, 1988.