Supporting Shared Data Structures on Distributed Memory Architectures

Chalres Koelbel

Piyush Mchrotra

SUPPORTING SHARED DATA STRUCTURES ON DISTRIBUTED MEMORY ARCHITECTURES

Charles Koelbel
Piyush Mehrotra

CSD-TR 915
October 1989

Supporting Shared Data Structures on Distributed Memory Architectures*

Charles Koelbel
chk@cs.purdue.edu

Piyush Mehrotra†
pm@icase.edu

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907.

Abstract

Programming nonshared memory systems is more difficult than programming shared memory systems, since 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 of the processors in the machine, and with all communication explicitly specified by low-level message-passing primitives. This paper presents a new programming environment for distributed memory architectures, providing a global name space and allowing direct access to remote parts of data values. In order to retain efficiency, we provide a system of annotations allowing the user to retain control over aspects of the program critical to performance, such as data distributions and load balancing. This paper describes the analysis and program transformations required to implement this environment, and shows the efficiency of the resulting code with an example program tested on an NCUBE hypercube.

1 Introduction

Distributed memory architectures promise to provide very high levels of performance for scientific applications at modest costs. However, they are extremely awkward to program. The programming languages currently available for such machines directly

*This research was supported by the Office of Naval Research under contract ONR N00014-88-M-0108, and by the National Aeronautics and Space Administration under NASA contract NAS1-18605 while the authors were in residence at ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665.

†On leave at ICASE, NASA Langley Research Center, Hampton, Va 23665.

reflect the underlying hardware in the same sense that assembly languages reflect the registers and instruction set of a microprocessor.

The basic issue is that programmers tend to think in terms of manipulating large data structures, such as grids, matrices, etc. With the current message-passing languages, each process can access only the local address space of the processor on which it is executing. Thus the programmer must decompose each data structure into a collection of pieces, each piece being "owned" by a single processor. All interactions between different parts of the data structure must then be explicitly specified using the low-level message-passing constructs supported by the language.

Decomposing all data structures in this way, and specifying communication explicitly can be extraordinarily complicated and error prone. However, there is also a more subtle problem here. Since the partitioning of the data structures across the processors must be done at the highest level of the program, and each operation on these distributed data structure turns into a sequence of "send" and "receive" operations intricately embedded in the code, 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.

In this paper we present a programming environment, called Kali[1], which is designed to simplify the problem of programming distributed memory architectures. Kali provides a software layer supporting a global name space on distributed memory architectures. The computation is specified via a set of parallel loops using this global name space exactly as one does on a shared memory architecture. The danger here is that since true shared memory does not exist, one might easily sacrifice performance. However, by requiring the user to explicitly control data distribution and load balancing, we force awareness of those issues critical to performance on nonshared memory architectures. In effect, we acquire the ease of programmibility of the shared memory model, while retaining the performance characteristics of nonshared memory architectures.

In Kali, one specifies parallel algorithms in a high-level, distribution independent manner. The compiler then analyzes this high-level specification and transforms it into a system of interacting tasks, which communicate via message-passing. 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. Preliminary results suggest that the performance of the resulting message-passing code is in many cases virtually identical to that which would be achieved had the user programmed directly in a message-passing language.

The remainder of this paper is organized as follows. Section 2 describes Kali, the language in which we have implemented our ideas. Section 3 presents the analysis needed to map a Kali program onto a nonshared memory architecture. If enough information is available, the compiler can perform this analysis at compile-time. Oth-

[1]Kali is the name of a Hindu goddess of creation and destruction who possesses multiple arms, embodying the concept of parallel work.

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

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

forall i in 1..N−1 on A[i].loc do
   A[i] := A[i+1];
end;
```

Figure 1: Kali language primitives

erwise the compiler produces run-time code to generate the required information. We close this section with an example illustrating the latter situation. Section 4 shows the performance achieved by the sample program on an NCUBE/7. Finally, Section 5 compares our work with other groups, and Section 6 gives our conclusions.

2 Kali Language Primitives

The goal of our approach is to allow the programmers to treat distributed data structures as single objects. In our approach, the programmer must specify three things:

a) The processor topology on which the program is to be executed

b) The distribution of the data structures across these processors

c) The parallel loops and where they are to be executed

The following subsections describe each of these specifications. Figure 1 gives an example of these declarations in Kali, a language we created as a testbed for our ideas [5, 7]. We point out that these primitives can be added to any sequential language, such as FORTRAN [8].

2.1 Processor Arrays

The first thing that needs to be specified is a "processor array." This is an array of physical processors across which the data structures will be distributed, and on which the algorithm will execute. The **processors** line in Figure 1 declares this array. This particular declaration allocates a one-dimensional array *Procs* of P processors, where P is an integer constant between 1 and *max_procs* dynamically chosen by the run-time system. (Our current implementation chooses the largest feasible P; future implementations might use fewer processors to improve granularity or for other reasons.) Multi-dimensional processor arrays can be declared similarly.

This construct provides a "real estate agent," as suggested by C. Seitz. Allowing the size of the processor array to be dynamically chosen is important here, since it provides portability and avoids dead-lock in case fewer processors are available than expected. The basic assumption is that the underlying architecture can support multi-dimensional arrays of physical processors, an assumption natural for hypercubes and mesh connected architectures.

2.2 Defining a Distribution Pattern

Given a processor array, the programmer must specify the distribution of data structures across the array. Currently the only distributed data type supported is distributed arrays. Array distributions are specified by a *distribution clause* in their declaration. This clause specifies a sequence of distribution patterns, one for each dimension of the array. Scalar variables and arrays without a distribution clause are simply replicated, with one copy assigned to each of the processors in the processor array.

Mathematically, the distribution pattern of an array can be defined as a function from processors to sets of array elements. If *Proc* is the set of processors and *Arr* the set of array elements, then we define

$$local : Proc \rightarrow 2^{Arr}$$

as the function giving, for each processor p, the subset of *Arr* which p stores locally. In this paper we will assume that the sets of local elements are disjoint; that is, if $p \neq q$ then $local(p) \cap local(q) = \phi$. This reflects the practice of storing only one copy of each array element. We also make the convention that collections of processors and array elements are represented by their index sets, which we take to be vectors of integers.

Kali provides notations for the most common distribution patterns. Once the processor array *Procs* is declared, data arrays can be distributed across it using **dist** clauses in the array declarations, also shown in Figure 1. Array A is distributed by blocks, giving it a *local* function of

$$local_A(p) = \left\{ i \mid (p-1) \cdot \left\lceil \frac{N}{P} \right\rceil + 1 \leq i \leq p \cdot \left\lceil \frac{N}{P} \right\rceil \right\}$$

This assigns a contiguous block of array elements to each processor. Array B has its rows cyclically distributed; its *local* is

$$local_B(p) = \{(i,j) \mid i \equiv p \pmod{P}\}$$

Here, if P were 10 processor 1 would store elements in rows 1, 11, 21, and so on, while processor 10 would store rows which were multiples of 10. The number of dimensions of an array that are distributed must match the number of dimensions of the underlying processor array. Asterisks are used to indicate dimensions of data arrays which are not distributed as in the case of B as shown in Figure 1.

2.3 Forall Loops

Operations on distributed data structures are specified by **forall** loops. The **forall** loop here is similar to that in BLAZE [6]. The example in Figure 1 shows a loop which performs $N-1$ loop invocations, shifting the values in the array A one space to the left. The semantics here are "copy-in copy-out," in the sense that the values on the right hand side of the assignment are the old values in array A, before being modified by the loop. Thus the array A is effectively "copied into" each invocation of the **forall** loop, and then the changes are "copied out."

In addition to the range specification in the header of the **forall** there is an **on** clause. This clause specifies the processor on which each loop invocation is to be executed. In the above program fragment, the **on** clause causes the ith loop invocation to be executed on the processor owning the ith element of the array A. Although this is the most common use of the **on** clause, it is also possible to name the processor directly by indexing into the processor array.

2.4 Global Name Space

Given the **processors**, **dist**, and **forall** primitives, a programmer can specify a data parallel algorithm at a high level, while still retaining control over those details critical to performance. For example, the code fragment in Figure 4 in Section 3 shows a typical numerical computation. It is important to note that there are no message passing statements in either that program or Figure 1; instead, the programmer can view the program as operating within a global name space. The compiler analyses the program and produces the low level details of the message passing code required to support the sharing of data on the distributed memory machines.

The support of a shared memory model provides a distinct advantage over message passing languages; in those languages, communications statements often substantially increase the program size and complexity [3]. The global name space model used here allows the bodies of the **forall** loops to be independent of the distribution of the data and processor arrays used. If only local name spaces were supported, this would not be the case, since the communications necessary to implement two distribution patterns would be quite different. With our primitives a variety of distribution patterns can easily be tried by trivial modification of this program. Such a modification in a message passing language would involve extensive rewriting of the communications statements. Thus, Kali allows programming at a higher level of abstraction, since the programmer can focus on the general algorithm rather than the machine-dependent details of its implementation.

3 Analysis of the Program

Given a Kali program written using the distribution patterns and **forall** loops described above, the compiler must generate code that implements the message passing

```
forall $i \in Index\_set$ on $A[f(i)]$.loc do
    $\ldots R_1 \ldots$
    $\ldots R_2 \ldots$
        $\vdots$
    $\ldots R_n \ldots$
end;
```

Figure 2: Pseudocode loop for subscript analysis

necessary to run the program on a nonshared memory machine. This entails an analysis of the subscripts of array references to determine which ones may cause access to nonlocal elements. We will describe such an analysis in this section and then discuss how it can be efficiently accomplished.

3.1 General Outline of the Analysis

The type of loop we are considering has the form shown in Figure 2. Iteration i of the loop is executed on the processor storing $A[f(i)]$. In many cases, f will be the identity function, but we allow other functions for generality. Each R_k represents an array reference of the form

$$R_k = A[g_k(i)]$$

For simplicity, we will assume here that only one array A is referenced. The general case of multiple arrays does not alter the goals of the analysis, although it may complicate the analysis itself if the arrays have different distribution patterns. The g_k functions may depend on other program variables, so long as those variables are invariant during the execution of the **forall** loop.

The set of iterations executed on processor p, denoted by $exec(p)$ is determined by the **on** clause associated with the **forall** loop. For example, in Figure 2 because of the **on** clause, "$A[f(i)]$.**loc**," this set is a subset of iterations i such that $A[f(i)]$ is be local to processor p. We define this set mathematically as

$$exec(p) = f^{-1}(local(p))$$

where *local* is the distribution function associated with array A. In general, processor p will execute every iteration in $exec(p)$ which is in the **forall**'s range, that is, the intersection of the range with $exec(p)$. In the loop of Figure 2, for example, processor p will execute all iterations in $Index\_set \cap f^{-1}(local(p))$. This intersection is often equal to $exec(p)$; the name $exec(p)$ was chosen to reflect this close association. In the following discussion, we will assume that p executes exactly the iterations in $exec(p)$, as is generally the case except for boundary conditions. In cases where this is not true it is generally only necessary to intersect $Index\_set$ with $exec(p)$ in the following equations.

We first identify the **forall** iterations that can cause nonlocal array references. There are two reasons for doing this: we can overlap communication with computation in iterations that access only local array elements, and local accesses may be more amenable to optimization than general accesses. For each processor p and reference $R = A[g(i)]$ we define the set

$$ref(p) = g^{-1}(local(p))$$

This is the subset of the (unbounded) iteration space where R is always a local reference. If $exec(p) \subseteq ref(p)$ then the reference R can always be satisfied locally on processor p. Otherwise, any element a such that $a \in exec(p)$ but $a \notin ref(p)$ represents an iteration on p that may reference an array element not on p; this element must be communicated to p via messages. Thus, iterations in $exec(p) \cap ref(p)$ are executed on processor p and access only p's local memory. Iterations in $exec(p) - ref(p)$ cause nonlocal accesses on p. The first stage of analysis therefore finds $ref(p)$ for each reference R and processor p and determines how they intersect with the loop range sets $exec(p)$.

If $exec(p) \not\subseteq ref(p)$ for some p, then more analysis must be done to generate the messages received and sent by each processor. For each pair of processors p and q we must compute the sets $in(p,q)$, the set of elements received by p from q, and $out(p,q)$, the set of elements sent from p to q. This can be done in two ways. The simpler way is to simply note that processor p can only access elements in $g(exec(p))$. Since every element has a "home" processor, we can identify the sources of these elements using the *local* functions. Every nonempty set $g(exec(p)) \cap local(q)$ where $q \neq p$ represents a set of elements which processor p must receive as messages from processor q. Conversely, every nonempty set $g(exec(q)) \cap local(p)$ represents a set of elements that p must send to q. Thus, we can define

$$\begin{aligned} in(p,q) &= g(exec(p)) \cap local(q) \\ out(p,q) &= g(exec(q)) \cap local(p) \end{aligned}$$

The same sets can be constructed using the $ref(p)$ sets used above. Here, we note that those sets cover the iteration space. Thus, $exec(p)$ can be divided into parts by its intersections $exec(p) \cap ref(q)$. Any of these sets which is nonempty represents a region of iteration space executed on processor p and accessing array elements on processor q. The sets of elements to be received by p are $g(exec(p) \cap ref(q))$ for all q; similarly, the sets of elements that p must send are $g(exec(q) \cap ref(p))$. The communications sets can therefore also be defined as

$$\begin{aligned} in(p,q) &= g(exec(p) \cap ref(q)) \\ out(p,q) &= g(exec(q) \cap ref(p)) \end{aligned}$$

We can now describe the organization of the message passing code derived from simple **forall** statements. Figure 3 shows this for the program fragment in Figure 2,

```
Code executed on processor $p$

-- *Sets used in message pasing*
$exec(p) \equiv f^{-1}(local(p)) \cap Index\_set$
$ref(p) \equiv g^{-1}(local(p))$
$in(p,q) \equiv g(exec(p)) \cap ref(q)$ for each $q \in Proc - \{p\}$
$out(p,q) \equiv g(exec(q)) \cap ref(p)$ for each $q \in Proc - \{p\}$

-- *Send messages to other processors*
for each $q \in Proc$ do
    if $out(p,q) \neq \phi$ then send( $q$, $out(p,q)$ ); end;
end;

-- *Do local computations*
for each $i \in exec(p) \cap ref(p)$ do
    $\ldots A[g(i)] \ldots$
end;

-- *Receive messages from other processors*
for each $q \in Proc$ do
    if $in(p,q) \neq \phi$ then tmp[ $in(p,q)$ ] := recv( $q$ ); end;
end;

-- *Do nonlocal computations*
for each $i \in exec(p) - ref(p)$ do
    $\ldots tmp[g(i)] \ldots$
end;
```

Figure 3: Pseudocode message passing version of Figure 2

assuming only one reference $R_k = A[g(i)]$. Only high-level pseudocode for the computation on processor p is shown. Using the *in* and *out* sets, the processor sends all its messages, performs the iterations which do not require nonlocal data, receives all its messages, and finally performs the iterations requiring nonlocal data. These sets can be computed at either compile-time or run-time. In the next subsection, we characterize these two situations and then provide a detailed example requiring run-time analysis.

3.2 Run-time Versus Compile-time Analysis

The major issue in applying the above model is the analysis required to compute $exec(p)$, $ref(p)$, and their derived 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 cases we can analyze the program at compile-time and precompute the sets symbolically. Such an analysis requires the subscripts and data distribution patterns to be of a form such that closed form expressions can be obtained for the communications sets. If such an analysis is possible, no set computations need be done at run-time. Instead, the expressions for the sets can be used directly. The price paid for this improvement is lack of generality. Compile-time analysis is only possible when the compiler has enough information about the distribution function, *local*, and the subscripting functions f and g_k to produce simple formulas for the sets. In this paper we will not pursue this optimization; interested readers are referred to [4], which gives some flavor of the analysis.

In many programs the $exec(p)$ and $ref(p)$ 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. The cost of this computation can be lessened, however, by noting that the variables controlling the communications sets often do not change their values between executions of the **forall** loop. Our run-time analysis takes advantage of this by computing the $exec(p)$ and $ref(p)$ sets only the first time they are needed and saving them for later loop executions. This amortizes the cost of the run-time analysis over many repetitions of the **forall**, lowering the overall cost of the computation. This method is generally applicable and, if the **forall** is executed frequently, acceptably efficient. The next section shows how this method can be applied in a simple example.

3.3 Run-time Analysis

In this section we apply our analysis to the program in Figure 4. This models a simple partial differential equation solver on a user-defined mesh. Arrays a and *old_a* store values at nodes in the mesh, while array *adj* holds the adjacency list for the mesh and $coef$ stores algorithm-specific coefficients. This arrangement allows the solution of PDEs on irregular meshes, and is quite common in practice. We will only consider the computational core of the program, the second **forall** statement.

The reference to *old_a*[*adj*[i, j]] in this program creates a communications pattern dependent on data (*adj*[i, j]) which cannot be fully analyzed by the compiler. Thus, the $ref(p)$ sets and the communications sets derived from them must be computed at run-time. We do this by running a modified version of the **forall** called the *inspector* before running the actual **forall**. The inspector only checks whether references to distributed arrays are local. If a reference is local, nothing more is done. If the reference is not local, a record of it and its "home" processor is added to a list of elements to be received. This approach generates the $in(p,q)$ sets and, as a side effect, constructs the sets of local iterations ($exec(p) \cap ref(p)$) and nonlocal iterations ($exec(p) - ref(p)$). To construct the $out(p,q)$ sets, we note that $out(p,q) = in(q,p)$. Thus, we need only route the sets to the correct processors. To avoid excessive communications overhead we use Fox's Crystal router [3] which handles such communications without creating

```
processors Procs : array[ 1..P ] with P in 1..n;
var a, old_a : array[ 1..n ] of real dist by [ block ] on Procs;
   count : array[ 1..n ] of integer dist by [ block ] on Procs;
   adj : array[ 1..n, 1..4 ] of integer dist by [ block, * ] on Procs;
   coef : array[ 1..n, 1..4 ] of real dist by [ block, * ] on Procs;

-- code to set up adjacency and coefficient arrays 'adj' and 'coef' goes here

while ( not converged ) do

   -- copy mesh values
   forall i in 1..n on old_a[i].loc do
         old_a[i] := a[i];
   end;

   -- perform relaxation (computational core)
   forall i in 1..n on a[i].loc do
         var x : real;
         x := 0.0;
         for j in 1..count[i] do
            x := x + coef[i,j] * old_a[ adj[i,j] ];
         end;
         if (count[i] > 0) then a[i] := x; end;
   end;

   -- code to check convergence goes here

end;
```

Figure 4: Nearest-neighbor relaxation

```
record
   from_proc: integer;          -- processor sending the message
   to_proc: integer;            -- processor receiving the message
   low: integer;                -- lower bound of array range
   high: integer;               -- upper bound of array range
   buffer: ^real;               -- pointer to memory containing message
end;
```

Figure 5: Representation of *in* and *out* sets

bottlenecks. Once this is accomplished, we have all the sets needed to execute the communications and computation of the original **forall**, which are performed by the part of the program which we call the *executor*. The executor consists of the two **for** loops shown in Figure 3 which perform the local and nonlocal computations.

The representation of the $in(p, q)$ and $out(p, q)$ sets deserves mention, since this representation has a large effect on the efficiency of the overall program. We represent these sets as dynamically-allocated arrays of the record shown in Figure 5. Each record contains the information needed to access one contiguous block of an array stored on one processor. The first two fields identify the sending and receiving processors. On processor p, the field $from\_proc$ will always be p in the *out* set and the field to_proc will be p in the *in* set. The *low* and *high* fields give the lower and upper bounds of the block of the array to be communicated. In the case of multi-dimensional arrays, these fields are actually the offsets from the base of the array on the home processor. To fill these fields, we assume that the home processors and element offsets can be calculated by any processor; this assumption is justified for static distributions such as we use. The final $buffer$ field is a pointer to the communications buffer where the range will be stored. This field is only used for the *in* set when a communicated element is accessed. When the *in* set is constructed, it is sorted on the $from\_proc$ field, with the *low* field serving as a secondary key. Adjacent ranges are combined where possible to minimize the number of records needed. The global concatenation process which creates the *out* sets sorts them on the to_proc field, again using *low* as the secondary key. If there are several arrays to be communicated, we can add a *symbol* field identifying the array; this field then becomes the secondary sorting key, and *low* becomes the tertiary key.

Our use of dynamically-allocated arrays was motivated by the desire to keep the implementation simple while providing quick access to communicated array elements. An individual element can be accessed by binary search in $O(\log r)$ time (where r is the number of ranges), which is optimal in the general case here. Sorting by processor *id* also allowed us to combine messages between the same two processors, thus saving on the number of messages. Finally, the arrays allowed a simple implementation of the concatenation process. The disadvantage of sorted arrays is the insertion time of $O(r)$ when the sets are built. In future implementations, we may replace the arrays by binary trees or other data structure allowing faster insertion while keeping the same access time.

The above approach is clearly a brute-force solution to the problem, and it is not clear that the overhead of this computation will be low enough to justify its use. As explained above, we can alleviate some of this overhead by observing that the communications patterns in this **forall** will be executed repeatedly. The *adj* array is not changed in the **while** loop, and thus the communications dependent on that array do not change. This implies that we can save the $in(p,q)$ and $out(p,q)$ sets between executions of the **forall** to reduce the run-time overhead.

Figure 6 shows a high-level description of the code generated by this run-time analysis for the relaxation **forall**. Again, the figure gives pseudocode for processor p only. In this case the communications sets must be calculated (once) at run-time. The sets are stored as lists, implemented as explained above. Here, *local_list* stores $exec(p) \cap ref(p)$; *nonlocal_list* stores $exec(p) - ref(p)$; and *recv_list* and *send_list* store the $in(p,q)$ and $out(p,q)$ sets, respectively. The statements in the first **if** statement compute these sets by examining every reference made by the **forall** on processor p. As discussed above, this conditional is only executed once and the results saved for future executions of the **forall**. The other statements are direct implementations of the code in Figure 3, specialized to this example. The locality test in the nonlocal computations loop is necessary because even within the same iteration of the **forall**, the reference $old\_a[adj[i,j]]$ may be sometimes local and sometimes nonlocal. We discuss the performance of this program in the next section.

4 Performance

To test the efficacy of the analysis in Section 3, we hand-translated the relaxation program shown in Figure 6 into C for execution on the NCUBE/7 in our department. We ran the resulting programs for several sizes of the hypercube, measuring the times for various sections of the codes. The results of those tests are given in the subsections below. We are currently debugging a compiler which can perform these transformations automatically, and will present the performance data from programs compiled with this system in future reports. However, since the code generated by the compiler is virtually identical to that produced by hand, we expect no surprises.

Figure 7 tabulates the execution times for the program of Figure 6; Figure 8 graphs this data and data from a variation of the program. The *adj* and *coef* matrices used describe the ordinary five-point Jacobi iterations on a regular 64×64 grid repeated 100 times. Nearly linear speedups can be seen in both the table and the graph, despite the overhead of run-time analysis. The graph plots the total execution times from Figure 7 as a dotted line. The times for computation only and message passing only (which does not include the time for the inspector loop) are plotted as dashed lines; the total time for the program closely follows the computation line. To show the advantage of not recomputing the communications sets, we modified the program to execute the inspector loop on every **forall** iteration. These times are shown as the solid line. Note the separation of this line from the "Saved Analysis" and "Computation Only"

```
Code executed on processor $p$

if ( first_time ) then                                   -- Compute sets for later use
     local_list := nonlocal_list := send_list := recv_list := NIL;
     for each $i \in local_a(p)$ do
          flag := true;
          for each $j \in \{1, 2, \ldots, count[i]\}$ do
             if ( $adj[i,j] \notin local_{old\_a}(p)$ ) then
                Add old_a[ adj[i,j] ] to recv_list
                flag := false;
             end;
          end;
          if ( flag ) then Add i to local_list
                    else Add i to nonlocal_list
                    end;
     end;
     Globally combine recv_list from all processors into send_list
end;
for each $msg \in send\_list$ do                 -- Send messages to other processors
     send( msg );
end;
for each $i \in local\_list$ do                               -- Do local computations
     Original loop body
end;
for each $msg \in recv\_list$ do           -- Receive messages from other processors
     recv( msg ) and add contents to msg_list
end;
for each $i \in nonlocal\_list$ do                         -- Do nonlocal computations
     x := 0.0;
     for each $j \in \{1, 2, \ldots, count[i]\}$ do
          if ( $adj[i,j] \in local_{old\_a}(p)$ ) then
             tmp := old_a[ adj[i,j] ];
          else
             tmp := Search msg_list for old_a[ adj[i,j] ]
          end;
          x := x + coef[i,j] * tmp;
     end;
     if (count[i] > 0) then a[i] := x; end;
end;
```

Figure 6: Pseudocode message passing version of Figure 4

lines; this indicates the large overhead of the repeated analysis.

The overhead due to the run-time analysis depends strongly on the number of processors and whether its results can be reused. If the run-time analysis is performed only once, its overhead varies from 10% for one processor down to 2% for 128 processors. This is equivalent to 80% of one extra iteration in the single-processor case and 10 extra iterations in the 128-processor case; we consider this to be an acceptable overhead. The repeated analysis situation, however, has an 80% overhead for one processor and a 1000% overhead for 128 processors. Such "efficiency" would certainly be unacceptable.

5 Related Work

There are many other projects concerned with compiling programs for nonshared memory parallel machines. Three in particular break away from the message passing paradigm and are thus closely related to our work.

Kennedy and his coworkers [1] compile programs for distributed memory by first creating a version which computes its communications at run-time. They then use standard compiler transformations such as constant propagation and loop distribution to optimize this version into a form much like ours. Their optimizations appear to fail in our run-time analysis cases. If significant compile-time optimizations are possible, their results appear to be similar to our compile-time analysis in [4]. We extend their work in our run-time analysis by saving information on repeated communications patterns. It is not obvious how such information saving could be incorporated into their method without devising new compiler transformations. We also provide a more top-down approach to analyzing the communications, while their optimizations can be characterized as bottom-up.

Rogers and Pingali [9] suggest run-time resolution of communications for the functional language Id Nouveau. They do not attempt to save information between executions of their parallel constructs, however. Because the information is not saved, they label run-time resolution as "fairly inefficient" and concentrate on optimizing special cases. These cases appear to correspond roughly to our compile-time analysis. We extend their work by saving the communications information between **forall** executions and by providing a common framework for run-time and compile-time resolution.

Crowley et al [2] compute data-dependent communications patterns in a preprocessor, producing schedules for each processor to execute later. This preprocessing is done off-line, although they are currently integrating this with the actual computation as is done with our system. Their execution schedules also take into account inter-iteration dependencies, something not necessary in our system since we currently start with completely parallel loops. They do not give any performance figures for their preprocessor, although they do note that given its "relatively high" complexity, parallelization will be required in any practical system. Saving the information about **forall** communications between executions is very similar between our two works. A major difference from our work is that they explicitly enumerate all array references (

Nearest-neighbor relaxation, 64×64 mesh, 100 iterations				
processors	total time	setup time	communications time	computation time
1	107614	844	3	106600
2	54819	625	169	53951
4	29281	423	291	28531
8	15629	311	392	14902
16	8853	258	467	8087
32	5519	237	527	4681
64	3257	230	565	2333

Figure 7: Performance of run-time analysis

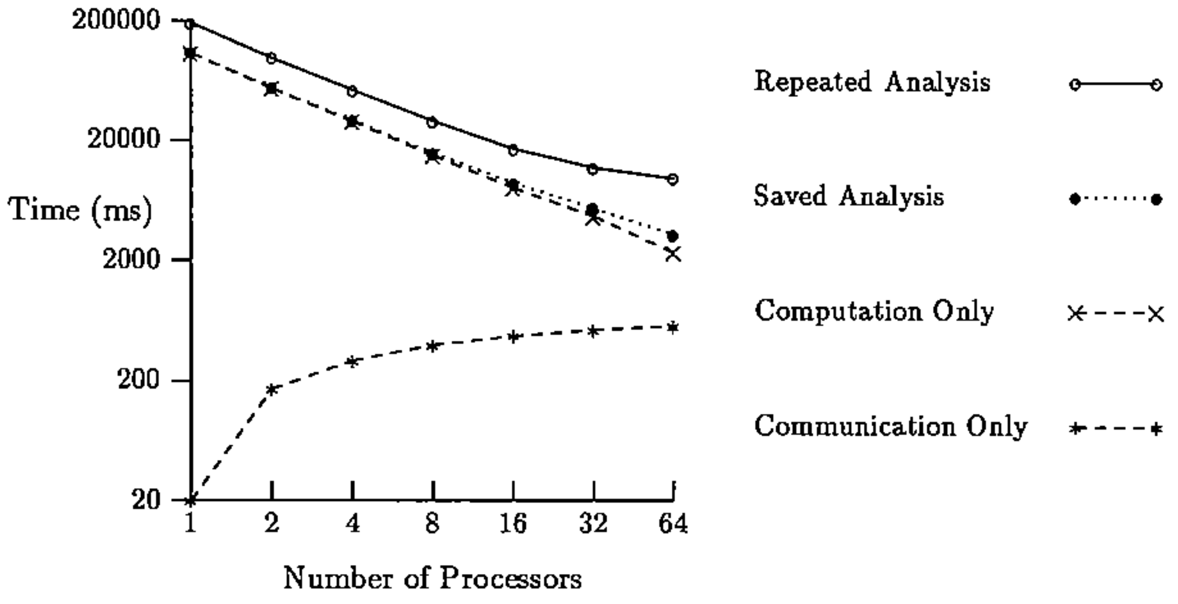

Figure 8: Performance graph of run-time analysis

local and nonlocal) in a "list". This eliminates the overhead of checking and searching for nonlocal references during the loop execution but requires more storage than our implementation. We also differ in that we consider compile-time optimizations, which they do not attempt.

6 Conclusions

Current programming environments for distributed memory architectures do not provide any support for mapping an application onto 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 not only increases the complexity of the program but also hard wires the choices inhibiting experimentation.

In this paper, we have described an environment which allows the user to specify algorithms at a fairly high-level using a global name space. The user has to make minimal additions to a sequential version of the algorithm; the low level details of local array indexing , message passing, etc. are left to the compiler.

We describe a system by which this transformation can be done automatically, and show that it can be implemented with acceptable efficiency. Our system allows the messages to be generated either at compile-time or at run-time. The compile-time analysis results in faster programs, but is only possible if the compiler has sufficient information about the program. We expect the performance of codes produced by compile-time analysis to be similar to that of hand-coded versions of the algorithm. Run-time analysis, while slower, is more general. Under certain circumstances even the run-time analysis can be executed efficiently by saving information. Our experience suggest that this is a viable approach for many classes of parallel programs.

The environment and language constructs described here are the first steps towards easing the task of programming nonshared memory machines. Further experience is needed with more complex and real applications to determine the usability, generality and efficacy of such an approach.

References

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

[2] K. Crowley, J. Saltz, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. ICASE Report 89-7, Institute for Computer Applications in Science and Engineering, Hampton, VA, January 1989.

[3] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors, Volume 1.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[4] C. Koelbel and P. Mehrotra. Compiler transformations for non-shared memory machines. In *Proceedings of the 4th International Conference on Supercomputing*, volume 1, pages 390–397, May 1989.

[5] P. Mehrotra. Programming parallel architectures: The BLAZE family of languages. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 289–299, December 1988.

[6] P. Mehrotra and J. V. Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, 1987.

[7] P. Mehrotra and J. Van Rosendale. Compiling high level constructs to distributed memory architectures. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, March 1989.

[8] P. Mehrotra and J. Van Rosendale. Parallel language constructs for tensor product computations on loosely coupled architectures. Technical Report 89-41, ICASE, September 1989. (To appear in Supercomputing '89).

[9] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Conference on Programming Language Design and Implementation*, pages 1–999. ACM SIGPLAN, June 1989.