NASA Contractor Report 181981
ICASE Report No. 90–7

ICASE

SUPPORTING SHARED DATA STRUCTURES ON DISTRIBUTED MEMORY ARCHITECTURES

Charles Koelbel
Piyush Mehrotra
John Van Rosendale

Contract No. NAS1–18605
January 1990

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665–5225

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Supporting Shared Data Structures on Distributed Memory Architectures*

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

Piyush Mehrotra†
John Van Rosendale
ICASE, NASA Langley Research Center
Hampton, Va 23665.

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. We describe the analysis and program transformations required to implement this environment, and present the efficiency of the resulting code on the NCUBE/7 and IPSC/2 hypercubes.

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

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 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. In contrast, in 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 process. 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*, 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 programmability 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

*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

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. Otherwise 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 the NCUBE/7 and IPSC/2. 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 programmers to treat distributed data structures as single objects. We assume the user is designing data-parallel algorithms, which can be specified as parallel loops. Our system then translates this high level specification into an SPMD-style program which can execute efficiently on a distributed memory architecture. 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

By specifying these items, the user retains control over aspects of the program critical to performance, such as data distributions and load balancing.

The following subsections describe each of these specifications. Figure 1 gives an example of these declarations in Kali, a Pascal-like language we created as a testbed for these techniques [4, 6]. These primitives can as easily be added to FORTRAN, as described in [7], or any other sequential language.

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 processes.

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. Kali also supports block-cyclic distributions and provides a mechanism for user-defined distributions. 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 [5]. 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 2 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 [2]. 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

```
forall i ∈ Index_set on A[f(i)].loc do
    ... R_1 ...
    ... R_2 ...
        ⋮
    ... R_n ...
end;
```

Figure 2: Pseudocode loop for subscript analysis

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 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 1. 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

$$R_k \equiv A[g_k(i)]$$

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 1 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. Each 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 1, for example, processor p will execute all iterations in $Index\_set \cap f^{-1}(local(p))$. This intersection is often equal to $exec(p)$ except for boundary conditions; the name $exec(p)$ was chosen to reflect this close association. For simplicity, in this paper we will assume that p executes exactly the iterations in $exec(p)$, as is generally the case. 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: local accesses may be more amenable to optimization than general accesses, and we can overlap communication with computation in iterations that access only local array elements. 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. Note that iterations in $exec(p) \cap ref(p)$ are executed on processor p and access only p's local memory. Thus, 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. In other words, iterations in $exec(p) - ref(p)$ cause nonlocal accesses on p. The first stage of the 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 first uses the $ref(p)$ sets defined 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

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

The second, simpler way is to 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

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

We can now describe the organization of the message passing code derived from simple **forall** statements. Figure 2 shows this for the program fragment in Figure 1, assuming only one reference $R \equiv A[g(i)]$. 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. Compile-time analysis, however, 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 [3], 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. However, the impact of the overhead from this computation can be lessened by noting that the variables controlling the communications sets often do not change their values between repeated 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

Code executed on processor p:

```
-- Sets used in message passing code
exec(p) ≡ f^{-1}(local(p)) ∩ Index_set
ref(p) ≡ g^{-1}(local(p))
in(p,q) ≡ g(exec(p)) ∩ ref(q) for each q ∈ Proc − {p}
out(p,q) ≡ g(exec(q)) ∩ ref(p) for each q ∈ Proc − {p}

-- Send messages to other processors
for each q ∈ Proc do
    if out(p,q) ≠ φ then send( q, out(p,q) ); end;
end;

-- Do local iterations
for each i ∈ exec(p) ∩ ref(p) do
    ... A[g(i)] ...
end;

-- Receive messages from other processors
for each q ∈ Proc do
    if in(p,q) ≠ φ then
        tmp[ in(p,q) ] := recv( q );
    end;
end;

-- Do nonlocal iterations
for each i ∈ exec(p) − ref(p) do
    ... tmp[g(i)] ...
end;
```

Figure 3: Message passing pseudocode for Figure 1

```
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 arrays 'adj' and 'coef'

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

end;
```

Figure 4: Nearest-neighbor relaxation on an unstructured grid

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 2. 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 a variant of Fox's Crystal router [2] which handles such communications without creating 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 2 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 3. 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*

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

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

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 3 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 2, 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 methods shown in Section 3, we implemented the run-time analysis in the Kali compiler. The compiler produces C code for execution on the NCUBE/7 and iPSC/2 multiprocessors. We then compiled the program shown in Figure 2 using various constants for the sizes of the arrays and ran the resulting programs for several sizes of the hypercube, measuring the times for various sections of the codes.

Since our primary interest is unstructured grids, our program allows general *adj* and *coef* arrays. However, in the tests here the grids used were simple rectangular

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 ∈ local_a(p) do
          flag := true;
          for each j ∈ {1,2,...,count[i]} do
            if ( adj[i,j] ∉ 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;
     Form send_list using recv_lists from all processors
       (requires global communication)
end;
for each msg ∈ send_list do      -- Send messages to other processors
     send( msg );
end;
for each i ∈ local_list do                      -- Do local iterations
     Original loop body
end;
for each msg ∈ recv_list do              -- Receive messages from other
processors
     recv( msg ) and add contents to msg_list
end;
for each i ∈ nonlocal_list do                   -- Do nonlocal iterations
     x := 0.0;
     for each j ∈ {1,2,...,count[i]} do
          if ( adj[i,j] ∈ 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: Message passing pseudocode for Figure 4

Time (in seconds) for 100 sweeps over 128 × 128 mesh				
processors	total time	executor time	inspector time	inspector overhead
2	246.07	244.04	2.03	0.8%
4	127.46	126.12	1.34	1.1%
8	68.38	67.28	1.10	1.6%
16	38.95	37.88	1.07	2.7%
32	24.36	23.21	1.15	4.7%
64	17.71	16.42	1.29	7.3%
128	12.64	11.19	1.45	11.5%

Figure 7: Performance of run-time analysis for varying number of processors on an NCUBE/7

Time (in seconds) for 100 sweeps over 128 × 128 mesh				
processors	total time	executor time	inspector time	inspector overhead
2	60.69	60.34	0.34	0.56%
4	31.20	31.02	0.18	0.57%
8	16.23	16.13	0.10	0.60%
16	8.88	8.82	0.06	0.64%
32	5.27	5.23	0.04	0.70%

Figure 8: Performance of run-time analysis for varying number of processors on an iPSC/2

Time (in seconds) for 100 sweeps on 128 processors					
mesh size	total time	executor time	inspector time	inspector overhead	speedup
64 × 64	4.97	3.56	1.38	27.8%	23.9
128 × 128	12.64	11.19	1.45	11.5%	37.3
256 × 256	34.13	32.52	1.61	4.7%	55.2
512 × 512	93.78	91.68	2.10	2.2%	80.4
1024 × 1024	305.03	301.31	3.72	1.2%	98.9

Figure 9: Performance of run-time analysis for varying problem size on an NCUBE/7

Time (in seconds) for 100 sweeps on 32 processors					
mesh size	total time	executor time	inspector time	inspector overhead	speedup
64 × 64	1.88	1.86	0.02	0.85%	15.7
128 × 128	5.27	5.23	0.04	0.70%	22.5
256 × 256	17.65	17.54	0.11	0.62%	26.8
512 × 512	65.17	64.79	0.38	0.58%	29.1
1024 × 1024	249.75	248.34	1.41	0.56%	30.3

Figure 10: Performance of run-time analysis for varying problem size on an iPSC/2

grids, on which we performed 100 Jacobi iterations with the standard five point Laplacian. For this test problem, the optimal static domain decomposition is obvious, so we did not have to cope with the added complication of load balancing strategies. Except for issues of load balancing and domain decomposition, we ran the program exactly as we would for an unstructured grid. The only significant difference is that the node connectivity is higher for unstructured grids; nodes in a two dimensional unstructured grid have six neighbors, on average, rather than the four assumed here. Thus all costs, execution, inspection, and communication, would be somewhat higher for an unstructured grid.

Figures 4 and 5 show how the execution time varies when the problem size remains constant and the number of processors is increased. The inspector overhead is defined here as the proportion of time spent in the inspector; that is, the overhead is the inspector time divided by the total time. The tables show that the overhead from the inspector is never very high; for the NCUBE it varies from less than 1% to about 12% of the total computation time, while on the iPSC it is always less than 1% of the total. These numbers obviously depend on the number of iterations performed, since the inspector is always executed only once. We assumed 100 iterations, since this is typical of many numerical algorithms.

For some problems, there are numerical algorithms requiring fewer relaxation iterations. Such algorithms tend to be much more complex, requiring incomplete LU factorizations or multigrid techniques, and we suspect our approach would be less useful in such cases. In the worst case, where one performs only one sweep, the inspector overhead on the NCUBE would range from 45% on 2 processors to 93% on 128 processors, while on the iPSC it ranges from 35% to 41%. These numbers illustrate the importance of saving inspector information to avoid recomputation. They also suggest that with this kind of hardware/software environment algorithm choices might shift in favor of simpler algorithm with more repetitive inner loops.

Figure 4 also shows how the time taken by the inspector varies. As can be seen, the time for the inspector starts high, decreases to a minimum at 16 processors, and then increases slowly. This behavior can be explained by the structure of the inspector itself. It consists of two phases: the loop identifying nonlocal array references, and the global communication phase to build the receive lists. The time to execute the loop is proportional to the number of array references performed, and thus in this case inversely proportional to the number of processors. The global communications phase, on the other hand, requires time proportional to the dimension of the hypercube, and thus is logarithmic in the number of processors. When there are few processors, the inspector time is dominated by the array reference loop, and is thus inversely proportional to the number of processors. However, as more processors are added, the increasing time for the communications phase eventually overtakes the decreasing loop time and the total time begins to rise.

This behavior is not seen in Figure 5 because the locality-checking loop always dominates the computation on the iPSC. For sufficiently many processors the communication phase would also become significant there. In general, the iPSC inspector

overheads are much less than of the NCUBE. This appears to be primarily due to the relatively lower cost of communications for small messages on the iPSC. This increases the cost of the global combining in the inspector on the NCUBE, thus increasing its overhead in relation to the iPSC.

Figures 6 and 7 keep the number of processors constant and vary the problem size. Inspector overhead is defined as above, while speedup is given relative to the executor time on one processor. This represents the closest measurement we have to an optimal sequential program, since it does not include any overhead for either the inspector or for communication. As can be seen, the inspector overhead decreases and the speedup increases when the number of processors is increased. The decrease in inspector overhead can again be explained by the structure of the inspector. As the problem size increases, the number of iterations of the locality-checking loop also increases, making that phase of the inspector more dominant in the total inspector time. Thus, our inspector-executor code organization can be expected to scale well as problem size increases. The increases in speedup reflect decreasing overheads in the executor loop. Our parallel programs have two overheads associated with nonlocal references: the cost of sending and receiving data in messages, and data structure overhead from the searches for nonlocal array elements. Any program written for a distributed memory machine will have the communications overhead, however, the search overhead is unique to our system. This search overhead is primarily responsible for suboptimal speedups. Also, this overhead is much less for the iPSC than for the NCUBE, probably because of the faster procedure calls on the iPSC. We are working to both analyze these results and to improve the data structure performance on both machines.

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 [3]. 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 [8] suggest run-time resolution of communications for the functional language Id Nouveau. They do not attempt to save information between execu-

tions 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.

Saltz et al [9] 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 (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 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 more general (but less efficient) case of run-time analysis. Our run-time analysis generates messages by performing an *inspector* loop before the main computation, which records any nonlocal array references. The *executor* loop subsequently uses this information to transmit information efficiently while performing the actual computation.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181981 ICASE Report No. 90-7		

4. Title and Subtitle

SUPPORTING SHARED DATA STRUCTURES ON DISTRIBUTED MEMORY ARCHITECTURES

5. Report Date

January 1990

6. Performing Organization Code

7. Author(s)

Charles Koelbel
Piyush Mehrotra
John Van Rosendale

8. Performing Organization Report No.

90-7

10. Work Unit No.

505-90-21-01

9. Performing Organization Name and Address

Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665-5225

11. Contract or Grant No.

NAS1-18605

13. Type of Report and Period Covered

Contractor Report

12. Sponsoring Agency Name and Address

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Final Report

To appear in the Proceedings of the 2nd SIGPLAN Symposium on Principles and Practice of Parallel Programming, March 1990.

16. 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. We describe the analysis and program transformations required to implement this environment, and present the efficiency of the resulting code on the NCUBE/7 and IPSC/2 hypercubes.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
distributed memory architectures; language constructs; runtime optimizations	59 - Mathematical and Computer Sciences (General) 61 - Computer Programming and Software Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	19	A03

NASA FORM 1626 OCT 86

NASA-Langley, 1990

The inspector is clearly an expensive operation. However, if one amortizes the cost of the inspector over the entire computation, it turns out to be relatively inexpensive in many cases. This is especially true in cases where the computation is an iterative loop executed a large number of times.

The other issue effecting the overhead of our system is the extra cost incurred throughout the computation by the new data structures used. This is a serious issue, but one on which we have only preliminary results. In future work, we plan to give increased attention to these overhead issues, refining both our run-time environment and language constructs. We also plan to look at more complex example programs, including those requiring dynamic load balancing, to better understand the relative usability, generality and efficacy of this approach.

References

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

[2] 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.

[3] 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.

[4] 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 1987.

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

[6] 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.

[7] P. Mehrotra and J. Van Rosendale. Parallel language constructs for tensor product computations on loosely coupled architectures. In *Proceedings Supercomputing '89*, pages 616–626, November 1989.

[8] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Conference on Programming Language Design and Implementation*, pages 69–80. ACM SIGPLAN, June 1989.

[9] 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*, April 1990. (To appear).