NASA Contractor Report 187453
ICASE Report No. 90–69

NASA-CR-187453
19910002904

ICASE

PROGRAMMING DISTRIBUTED MEMORY ARCHITECTURES USING KALI

Piyush Mehrotra
John Van Rosendale

Contract No. NAS1–18605
October 1990

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

Operated by the Universities Space Research Association

NASA
National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Programming Distributed Memory Architectures Using Kali*

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, in part because of the relatively low level of current programming environments for such machines. This paper presents a new programming environment, Kali, which provides a global name space and allows direct access to remote data values. In order to retain efficiency, Kali provides a system of annotations, allowing the user to control those aspects of the program critical to performance, such as data distribution and load balancing. This paper describes the primitives and constructs provided by our language, and also discusses some of the issues raised in translating a Kali program for execution on distributed memory systems.

*This research was supported 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

Current programming environments for distributed memory machines provide very little support for the distribution of data and code across the processors. This makes programming such machines very difficult since the user has to encode all the low-level details required to implement the algorithm. Thus, the resulting program is extraordinarily complex and also inflexible.

We have been working for several years on a programming environment, called Kali, designed to alleviate this problem in the context of scientific computation. The goal of our approach is to allow programmers 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.

This paper describes the Kali environment in relative detail, focusing on a Fortran-style syntax for our language primitives. This kind of language extension can be done with almost any sequential, procedural language supporting arrays. The version of these primitives here, known as KF1 (Kali Fortran 1), is natural in scientific programming, since most scientific programming is done in Fortran. Syntax is, however, not the issue; in earlier papers [9, 10] we have discussed this kind of language extension in the context of BLAZE [14], a Pascal-like data-flow language. We focus here on Fortran, primarily because of its greater acceptance in scientific computing.

The remainder of this paper is organized as follows. Section 2 describes the programming model employed by Kali. Section 3 describes the primitives provided by the language, while section 4 addresses the issues of portability and scalability. Section 5 presents an overview of the analysis and transformations needed to map a Kali program to a nonshared memory architecture. Finally, section 6 compares our work with other approaches, and section 7 gives a brief set of conclusions.

2 Kali Programming Model

The Kali programming environment is targeted to scientific applications. Such applications frequently consists of parallel operations on large "structures" such as grids, matrices, and so forth. These operations can generally be expressed as parallel loops manipulating arrays representing these structures.

The fundamental goal of Kali is to allow programmers to treat distributed data structures as single objects. Kali thus provides a software layer supporting a global name space on distributed memory architectures. The computation is then 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, Kali requires the user to explicitly control data distribution and load balancing, thus forcing awareness of those issues critical to performance on nonshared memory architectures. In effect, the user retains the ease of programmability of the shared memory model, while exploiting 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 translates it into a system of processes which communicate via messages. The generated code runs in what has been termed the SPMD mode (Single Program Multiple Data). That is, identical process code is down loaded onto each processors of the architecture. These processes then execute asynchronously, interacting via message-passing. These exchanges of data through messages are generally enough to maintain the semantics of the source code. However there are a few situations where barrier synchronizations are required in order to produce the correct results (see for example doall loops in section 3.3).

There are two major issues in restructuring Kali source code for parallel execution. First, the parallel loops must be partitioned across the processors. This is generally easy, since the compiler distributes the parallel loop iterations based on annotations provided by the user, as described in the next section. Second, all remote accesses have to be "compiled" into message passing communication. That is, the compiler must analyze all references to identify potentially nonlocal accesses. It then generates communication phases, causing the data to be moved to the processes requiring it.

There are two ways to translate sequential code for execution on a distributed memory architecture. One can either specify a "king processor," which executes all sequential code, communicating the results to all others, or one can replicate the sequential operations on all processors. We followed the latter approach, since it generally yields better performance.

The other issues to be addressed as part of our programming model are the issues of scaling and portability. These issues will be addressed in section 4, where we can discuss them in the context of specific Kali constructs.

3 Kali Language Primitives

In this section we describe the primitives provided by Kali. A Kali programmer must specify three things, in addition to the original (sequential) algorithm: a) the processor array on which the program is to be executed, b) the distribution of the data structures across these

processors, and c) the parallel loops and where they are to be executed. The following subsections describe each of these specifications in more detail.

3.1 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. A processor array is declared in the main program in KF1 using a syntax similar to that of Fortran array declarations:

```
parameter(maxprocs = 128 )
parameter(p ~ maxprocs)

processors procs(p)
```

Here *procs* has been declared as a one-dimensional array of processors. The size of this processor array, *p*, is an integer constant whose value is between 1 and *maxprocs*, as set by the **parameter** statement. **Parameter** statements are standard Fortran, but the tilda notation here is new. It's meaning is that the value of *p* is dynamically chosen in this range by the run-time system, and will remain constant throughout the program's execution.

Allowing the size of the processor array to be dynamically chosen is important, since it allows programs to be parametrized by the number of processors. Our approach provides portability, scalability, and avoids dead-lock in case fewer processors are available than expected. There are, however, other approaches one could take to achieving this same end, an issue we will return to in the next section.

KF1 also allows multi-dimensional processor arrays to be declared:

```
parameter(px ~ 128)
parameter(py = 2*px)

processors procs(px, py)
```

Here *procs* is a two-dimensional array which has twice as many processors in the second dimension as in the first. Processors can be referenced in the code like any other array element, e.g., *procs(i, j)* is the *(i,j)*th processor.

3.2 Data Distribution Primitives

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,

though "lists" may be supported in future versions of the language. Scalar variables and arrays that are not distributed are simply replicated, with one copy assigned to each of the processors in the processor array.

A **dist** statement can be used to specify the distribution functions for arrays. Kali provides notations for the most common distribution patterns: **block**, **cyclic**, and **block_cyclic**:

```
processors procs(p)
real A(N), B(N), C(N)
dist A(block), B(cyclic), C(block_cyclic(b))
```

Here, arrays *A*, *B*, and *C* are distributed across the *p* processors of the one-dimensional processor array *procs*. Array *A* is distributed by blocks such that each processor receives a contiguous block of elements of the array. Conversely, array *B* has its rows cyclically distributed. 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. Array *C* is distributed in a block-cyclic fashion with size *b*. That is, the elements of *C* are first divided into blocks of size *b*, and then these blocks are cyclically distributed across the set of processors.

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 *D1, D2* and *E* shown here:

```
processors procs(p)
real D1(N, N), D2(N, N), E(N, M, L)
dist D1(block, *), D2(*, block), E(*, block, *)
```

Since the processor array *procs* is one-dimensional, only one dimension of the arrays *D1, D2* and *E* is distributed. In the case of array *D1*, the first dimension is distributed across the *p* processors, i.e., the rows are blocked with a block of rows assigned to each processor. Array *D2*, on the otehr hand, has its second dimension distributed. Similarly, for array *E*, the second dimension is distributed with each processor getting a slice of the array.

User defined distributions

In addition to system defined distributions, KF1 supports general user defined distributions. The **map** statement uses syntax similar to the Fortran statement function, and allows the user to provide a mapping from array indices to processor indices. For example, in the code fragment below, the distribution function *ublock* is equivalent to the system defined **block** distribution.

```
processors procs(p)

real F(N), G(N,M)

map ublock(i) = p*(i-1)/N + 1
dist F(ublock), G(ublock, *)
```

Here *ublock* takes one argument, an array index, and produces a corresponding processor index. The identifier *i* is a dummy argument, local to the **map** function. Such user defined distributions can then be used in the **dist** statement in place of the system defined distribution patterns.

The **map** statement is special in that it can return multiple values, as required for processor indices in the case of multi-dimensional processor arrays. This is illustrated in the following code fragment by the distribution function *cyc2d*, which takes as argument two integers and returns two integer values to be used as processor indices.

```
processors procs(p, p)

real H(N, N), P(N, M, L), Q(N, N, N)

map cyc1d(i) = mod(i-1, p)+1
map cyc2d(i,j) = mod(i-1, p)+1, mod(j-1, p) + 1
dist H(cyc2d), P(cyc2d, *), Q(block, cyc1d, *)
```

As shown here, array distributions can be specified using a combination of system defined distributions, user defined distributions, and asterisks which denote undistributed dimensions. This system is completely general, yet convenient for the commonly occurring cases.

Dynamic Distributions

Along with the static mechanisms for data distributions described above, KF1 allows the distribution of array elements to be changed dynamically. This can be done by using the **distribute** statement:

```
processors procs(p)

real U(N), V(M), W(N)

map static(i) = mod(i-1, p) + 1
map dyn(i) = p * (i-1) / k + 1
dist U(block), V(static), W(dyn)

...
k = ...
...
distribute U, V, W
...
```

The effect of the **distribute** statement depends on the distributions associated with the respective arrays. If the mapping function is based on constants, then the distribution statement is a null operation. For example, the array U is mapped using system defined distribution **block** while the array V is distributed using the map *static* which involves only constants. Hence the distribution of the arrays U and V will remain unchanged when the **distribute** statement is executed.

On the other hand, the mapping function *dyn* is dependent on the variable k. Thus when the **distribute** statement is executed, the system will redistribute W if and only if the value of the variable k has changed since the last redistribution. When the mapping function involves an array (possibly distributed) or a function call, the compiler may use the conservative approach and redistribute without checking. Note that arrays that have dynamic distribution functions, that is, the distribution function depends on variables rather than constants, have to be *distributed* before they can be accessed or modified. That is, such arrays must occur in a **distribute** statement before being referenced in the code.

In general, the redistribution of an array implies that the data elements comprising the array are moved to new processors. As a compiler optimization, in some cases it can be determined that the array elements are not being accessed before being redefined, so that the values themselves do not need to be transported Note that this is slightly different from live-variable analysis, since all the elements of the array have to be "dead" for this optimization to be applicable.

3.3 Doall Loops

Operations on distributed data structures are specified by **doall** loops:

```
      processors procs(p)

      real A(N)
      dist A(block)

      doall 10 i = 1, N on owner(A(i))
          A(i) = ...
          ...
10    continue
```

The iterations of a **doall** loop cannot have inter-iteration dependencies. That is, any memory location assigned to in one iteration cannot be accessed or modified in any other iteration. This allows the iterations to be logically executed in parallel. There is an implied synchronization at the beginning and the end of the parallel loop, i.e., all the parallel threads start concurrently and all threads have to finish execution of their iterations before any other statement is executed.

In addition to the range specification in the header of the **doall**, there is also an **on** clause. The expression associated with the 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. The system defined function **owner** returns the home processor of its argument. 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, as shown below, where the ith iteration is executed on processor $P(i)$.

```
      processors procs(p)

      real A(N)
      dist A(block)

      doall 10 i = 1, p on (procs(i))
          ...
10    continue
```

In KF1, the loop headers of perfectly nested **doall** loops can be combined into a single header as shown:

```
      processors procs(p,p)

      real A(N, M)
      dist A(block, block)

      doall 10 (i, j) = [1, N]*[1, M] on owner( A(i, j) )
          ...
          A(i, j) = ...
          ...
10    continue
```

Here, a product of ranges is used to specify that for each value of the outer loop index, i, in the range $[1, N]$, the inner loop index, j, assumes each of the values in the range $[1, M]$.

3.4 Parallel Subroutines

In addition to ordinary Fortran subroutines and functions, KF1 supports parallel subroutines, which manipulate distributed data structures in parallel. For example, the header of a parallel subroutine looks like:

```
      parsub jacobi(u, f, n; procs)

      processors procs(p, p)

      ...
```

The keyword, **parsub** declares *jacobi* to be a parallel subroutines. For a parallel subroutine, the processor array on which the subroutine will execute has to be passed in as a special parameter, in the case here, *procs*. In this case, *procs* is a two-dimensional processor array of size p by p. The size of the processor array argument is "open", and is determined by the actual size of the processor array passed at the point of call. It does not need to be explicitly passed into the subroutine. The identifier p reflects this value, and can be used as a constant in the body of the subroutine.

The calling sequence for parallel subroutines is the same as that of ordinary subroutines, except for the special parameter representing the set of processors on which it will execute. For example, in the code fragment below, the subroutine *jacobi* is passed in the whole set of processors, *procs*.

```
      processors procs(p, p)
      ...
      call jacobi(u, f, n; procs)
```

Subsets of processors can also be passed:

```
      processors procs(p, p)

      real A(N, N)
      dist A(block, block)
      ...
      doall 10 i = 1, N on owner(A(i,*))
          call Q( A(i, *), N; owner(A(i, *)) )
10    continue
      ...
      stop
      end


      parsub Q(B, N; pr)

      processors pr(p)

      real B(N)
      dist B(block)
      ...
      return
      end
```

Here, subroutine Q accepts as argument a real vector B, distributed by blocks across a one-dimensional processor array *pr*, of size p. In the calling routine, we have a two-dimensional array A distributed by blocks on a two-dimensional processor array *procs*. The ith iteration of the **doall** loop conceptually executes on the "owner" of the ith row of the array A (represented by $A(i, *)$). That is a whole row of processors execute the i th iteration. In this example, the subroutine Q is called by each processor in the row in a distributed manner, each with its own piece of the i th row of the array A. The set of processors passed to subroutine Q is again supplied by the system function **owner**. In this example, the **doall** loop is supplying one level of parallelism, while the second level is provided by the parallel subroutine Q

Note that the set of processors passed in to a parallel subroutine must cover all the distributed data structures being passed as arguments. That is, the set of processors owning the individual pieces of distributed arguments need to be passed to the parallel subroutine

3.5 An Example: ADI Iteration

As an example of how these constructs fit together, we show here how to program a simple example, an ADI algorithm. More detailed versions of this example are given in [15]. This

example is appropriate here, since while straight forward in KF1, the analogous message-passing code is quite awkward. It is also typical of the kind of algorithms one must support in any scientific programming language.

ADI is a well known method for solving partial differential equations in multiple dimensions, in which one solves tridiagonal linear systems along the x- and y-lines of a grid at every step. The KF1 code for this algorithm is straight forward, as shown in Figure 1. In this version of the algorithm, we employ a sequential tridiagonal solver, *seqtri*, transposing the data arrays *vx* and *vy* so that either x-lines or y-lines do not cross processor boundaries. Then the sequential tridiagonal solver will run without interprocessor communication. The transpose here is implicit. Assigning array *vx* distributed **(block, *)** to *vy* distributed **(*, block)** induces the required interprocessor communication.

The notation **dynamic** here is new. Arrays declared dynamic are dynamically (stack) allocated. Sun Microsystems Fortran supports a similar construct, but standard Fortran does not. Thus one typically emulates dynamic allocation by hand using space in common blocks. KF1 does not currently support common blocks, and even when it does, this use of common arrays would be rather messy. This **dynamic** construct is natural here, and much simpler than dealing with distributed common blocks.

The version of ADI here is only one of a number of ways of distributing the computation; alternatives are given in [6, 12, 15]. The point is that all versions of this algorithm are equally easy to express in KF1. Moreover, changing distributions, or changing from calls to the sequential tridiagonal solver used here to calls to a parallel tridiagonal solver, is completely trivial. That is the power of this kind of language, in marked contrast to the situation with message-passing languages, where such minor changes typically induce weeks of programming.

4 Portability and Scalability

Portability and scalability are central concerns in parallel programming. The necessity of portability is clear; it is a real waste of human effort to repeatedly program the same algorithms for machines from different manufactures, or machines having slightly different topologies. Scalability is of equal importance; it is, after all, the raison d'êetre for distributed memory architectures. Thus a programming environment should allow programs to run essentially unchanged on architectures having varying numbers of processors, and should make as few assumptions as possible about the underlying architecture and its topology. At the same time, there needs to be enough specificity in the programming environment to adequately exploit machine performance, and to allow programmers to "tune" programs for

```
      parsub adi (u,f,nx,ny; procs)
c
c ..... procedure which performs one step of ADI iteration
c
      processors procs(nprocs)

      real        u(nx,ny),  f(nx,ny)
      dynamic real vx(nx,ny), vy(nx,ny)

      dist u(*, block), f(*, block), vx(*, block)
      dist vy(block, *)
c
c ..... compute residual
c
      call resid(vx,u,f,nx,ny; procs)
c
c ..... perform tridiagonal solves in x direction
c
      doall 100 j = 1,ny on owner(vx(*, j))
        call seqtri(vx(*, j), nx)
100   continue
c
c ..... perform transpose across processors
c
      vy = vx
c
c ..... perform tridiagonal solves in y direction
c
      doall 200 i = 1,nx on owner(vy(i, *))
        call seqtri(vy(i, *), ny)
200   continue
c
c ..... perform transpose
c
      u = vy

      return
      end
```

Figure 1: ADI Algorithm in KF1

efficient execution on specific machines Balancing these conflicting requirements is quite difficult.

On the issue of portability, Kali assumes a distributed memory architecture on which one can allocate processor arrays of at least two dimensions. With this assumption, both hypercube and mesh connected architectures suffice. Programs that use only one or two dimensional arrays will run well on either architecture. Programs using processor arrays of higher dimensions may suffer performance degradation on mesh architectures, depending on compiler implementation details, and the way in which the processor array is used in the program. This level of performance penalty is unavoidable since portability entails some compromise in performance. Given the great importance of portability, this level of compromise seems acceptable.

On the issue of scalability, the simplest approach is to provide arrays of virtual processors, which can be of any size desired. This is the approach taken, for example, in the Connection Machine language, C*. This approach is natural, and is easy to use, but one soon runs into subtle semantics and performance issues when one has multiple data arrays of various sizes and shapes [8]. Either the compiler must guess at the appropriate size for the virtual processor array, at the mapping from virtual processors to physical processors, and at the distributions of the data arrays, or one must provide annotations allowing the user to specify things more precisely. That is, in order to achieve performance, one must break the abstraction of virtual processors, and show the user at least part of the mechanism by which they are implemented.

The alternative followed here is to provide arrays of physical processors rather than virtual processors at the language level. This approach is simpler, and allows full control of the mapping. On the other hand it is occasionally awkward, since programs must be written to accommodate the varying sizes of processor arrays selected by the runtime system. With virtual processors, the programmer is sheltered from this issue, at the cost of serious potential performance penalties. In effect, we are avoiding potentially serious performance penalties, through the sacrifice of a small amount of language elegance.

To see why we have followed this path, consider the code fragment for a tree summation as shown in Figure 2. Here the first N/p steps of the summation are done sequentially on each processor, while the final $log_2(p)$ steps require interprocessor communication. The point is that one needs to know p in order to produce this code Without knowing p, it is impossible to know how far the sequential operations on each processor should be carried before one resorts to interprocessor operations. Thus one could only program this as though p and N were equal, incurring a substantial overhead.

This particular example is "made up" in the sense that there is a built in primitive **sum=**

```
      processors procs(p)

      integer U(N), Rep(p)
      dist U(block), Rep(block)

c
c ..... perform summation locally on each processor
c

      doall 10  i = 1, p on procs(i)
         Rep(i) = 0
         do 100  j = N*i-1+1, N*i
            Rep(i) = Rep(i) + U(j)
100       continue

10    continue

c
c ..... perform logarithmic tree summation across processors
c

      do 20  k = 1, log(p)
         istep = 2 ** k
         doall 200  i = 1, p on procs(i)
            if (mod(i, istep) .eq. 0) then
               Rep(i) = Rep(i) + Rep(i-istep)
            endif
200       continue
20    continue
```

Figure 2: Code for Tree Summation

which accomplishes this tree summation without all of this code. However the same issue arises in a variety of contexts, including fast tridiagonal solvers, Fast Fourier transforms, and adaptive quadrature algorithms [15]. Given only virtual processor arrays, one cannot write programs which are nearly as efficient, in most of these cases.

5 Program Transformation

In this section we describe some of the transformations performed by the compiler to restructure high-level Kali code for execution on distributed memory machines. As indicated before, the compiler produces SPMD-style code. The distributed data structures are partitioned such that each process gets the appropriate portion of the data structure. The scalar and non-distributed variables are replicated, and a current copy is maintained by each process. This is done by replicating the sequential parts of the source code in each process and inserting send-receive pairs whenever distributed variables are accessed.

The major focus of the KF1 compiler is to distribute the parallel loops using message-passing to transfer data between processes executing the different iterations of the loop A parallel loop is "strip mined" across the processors based on the associated **on** clause, that is, each processor gets a set of iterations to execute sequentially. Based on the references made to distributed data structures made in this set of iterations, we can define $in(p, q)$ as the set of data elements that are referenced by processor p but stored in processor q. This set can be inverted into the set $out(p, q)$ which is the set of elements owned by processor p and required by processor q. Given the latter set, each processor can then send the appropriate data to the appropriate processors before the computation is performed.

In some cases we can analyze the program at compile-time and precompute the sets symbolically. Such an analysis requires the distribution of the referenced data structures, the **on** clause of the associated loops, and the subscripts in the array references to be of a form such that closed form expressions can be obtained for the communications sets If such an analysis is possible, the compiler generates the message-passing statements necessary to communicate the data between the processes. In this paper we will not pursue this optimization; interested readers are referred to [9], which gives some flavor of the analysis.

However, such compile time analysis is not possible for programs in which the array references in a **doall** loop depend on the run-time values of the variables involved. This situation arises, for example, in PDE solvers using an irregular grid. The grid is generally represented using adjacency lists which denotes the neighbors of a particular node of the grid. Figure 3 presents a KF1 subroutine to perform a "relaxation" operation on such an irregular grid.

```
      parsub relax(u, nbrs, coef, n; procs)

      processors procs(p)

      parameter(niters  = 100)
      parameter(maxnbrs = 12)

      real u(n)
      dynamic real utmp(n)

      real coef(n, maxnbr)
      integer nbrs(n, maxnbr)

      dist u(block), utmp(block), coef(block, *), nbrs(block, *)

      do 400 iter = 1, niters
c
c.....copy u into utmp
c
         utmp = u

c
c.....update u
c
         doall 300 i = 1,n on owner(u(i))

            u(i) = 0.0
            do 200 j = 1, maxnbrs
               if (nbrs(i,j) .eq. 0)  goto 300
               u(i) = u(i) + coef(i,j) * utmp( nbrs(i,j) )
200         continue

300      continue

400   continue

      return
      end
```

Figure 3: Sweep over an unstructured mesh

Here the n node grid is represented by the vector u. Each node has a maximum of *maxnbrs* neighbors with indices of these neighbors being stored in the two-dimensional array *nbrs*. We assume that the grid is generated on the fly by some algorithm and hence the values in the array *nbrs* are set at run-time before the routine *relax* is called. In the code, the arrays are shown distributed by block; proper distribution of the arrays in this case raises load balancing issues outside the scope of this paper. The **doall** loop ranges over the grid points updating the value with a weighted sum of the values at the grid point's immediate neighbors.

The important point here is that to access the values at neighboring nodes, the elements of the vector *utmp* are indexed by the array *nbrs*. Thus the compiler cannot determine at compile-time which elements will be accessed. In such cases, the communication sets must be computed at run-time. We do this by running a modified version of the **doall** called the *inspector* before running the actual **doall**. 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. 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 using a global communication phase. To avoid excessive communications overhead we use Fox's Crystal router [4] 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 **doall**, which are performed by the part of the program which we call the *executor*.

Such run-time analysis obviously adds overhead. It is not clear a priori whether this overhead will be low enough to justify the use of run-time analysis. However, the variables controlling the communications sets often do not change for many executions of the **doall** loop. We take advantage of this by computing the *in* and *out* 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 **doall**, lowering the overall cost of the computation. This method is generally applicable and relatively efficient. A detailed description of this approach and its performance is given in [10].

6 Related Work

Since the issue of effectively programming distributed memory architectures is so pressing, a number of other projects have sprung up in an attempt to provide cleaner environments for these machines. These projects fall into two major groups: those which attempt automatic

distribution of data values and code and those which require the user to explicitly specify data distribution. Our approach is, in a sense, intermediate: we currently require the user to explicitly specify distributions, but are studying ways to automate array distribution. To show how our work relates to that of other researchers in this field, we briefly describe the principal alternative approaches being explored.

Among those following the first approach of automatic distribution of data and code, Quinn and Hatcher [18], and Reeves et al. [3, 19] compile languages based on SIMD semantics. These groups perform automatic distribution, while attempting to minimize the interprocessor synchronizations inherent in SIMD execution. The compiler for SPOT [23], an SIMD language designed specifically for point-iterative methods, performs compile-time analysis similar to ours to aggregate mesh point operations into larger processes. The AL compiler [24], targeted to one-dimensional systolic arrays, distributes only one dimension of the arrays. Based on the one dimensional distribution, this compiler allocates the iterations to the cells of the systolic array in a way that minimizes inter-cell communications.

Another approach is that of Linda [1], which is an explicit tasking language requiring the programmer to write code for each process. However, it does provide a shared name space called a *tuple space*, which can be implemented on distributed memory architectures. The language provides special primitives to access and modify this tuple space.

There are also a variety of projects following the second approach of leaving the distribution of data to the user. Pingali and Rogers [20] extend the dataflow language Id Nouveau with array distributions. Their compiler analysis appears similar to ours, and they also suggest run-time resolution of communications. However, they have not attempted to save run-time information over repeated execution of the parallel loops, and hence conclude that run-time resolution is "fairly inefficient". The Onyx environment [13], on the other hand, uses run-time analysis similar to that described here to handle communication patterns based on run-time data.

Griswold et al. [5] introduce the concept of *ensembles* to partition data, code and communication ports. The data is partitioned into sections, each section being mapped to a processor. This allows the data to scale with the number of processors as is the case with our system. The communication graph and the actual movement of data has to be explicitly specified by the programmer along with code for each processor. This allows them not only to distinguish between global and local computation but also to support the pure MIMD semantics.

Kennedy and coworkers [2, 7] and the SUPERB project [25] extend Fortran with array distributions. The approach there is to decompose the array into partitions and then specify a one-to-one mapping of partitions to processors. They utilize optimizing compiler trans-

formations similar to our compile-time analysis to generate the processes and the necessary communications. The programmer can specify blocks of local data which have images on other processors. These blocks can then be explicitly "moved" by the programmer allowing the efficient copying of data across processors.

The language DINO [21] proposed by Rosing and Schnabel takes a similar approach by extending a C-like language with data distribution primitives. The user can then invoke parallel functions on these distributed data structure, however, the user needs to annotate off-processor data accesses. The language allows users to specify portions of data which are mapped to several processors. The copies are kept consistent automatically by the system thus allowing overlapping distributions to be programmed fairly easily. The compile-time analysis required for their approach is again similar to ours, however, as in most other approaches they do not provide any support for run-time analysis of communication patterns.

Run-time analysis of communications patterns has also been proposed in [16, 22], while [11, 17] study the efficiency of data structures required to support such run-time extraction of communication patterns.

7 Conclusions

Current programming environments for distributed memory architectures provide inadequate support for mapping applications to the machine. In particular, the lack of a global name space forces algorithms to be specified at a relatively low level This greatly increases the complexity of programs, and also locks in algorithm design choices, inhibiting experimentation with alternate algorithm choices or problem decompositions. Thus, the relatively low-level of the programming environment leads, perhaps paradoxically, to poor performance, as well as burdening the programmer unnecessarily

In this paper, we have described an environment which allows the user to specify data parallel algorithms at a high level, while still retaining control over those details critical to performance. It is important to note that there are no message-passing statements in KF1. Instead, the programmer views 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 [4]. The global name space model used here allows the bodies of the **doall** 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. In a message-passing language, changing distribution patterns would involve extensive rewriting of the communications statements, and perhaps of the whole program. With our primitives, a variety of distribution patterns can easily be tried by trivial modification of this program. In this sense, Kali allows programming at a higher level of abstraction. Kali allows one to focus on the global algorithm, and worry less about machine-dependent details of the implementation.

The paper also outlined the transformations performed by the Kali compiler in order to efficiently map the KF1 programs to distributed memory machines. Our system relies on both compile-time and run-time analysis of the program. Compile-time analysis results in faster programs, but is only applicable when the compiler has adequate information. Run-time analysis is much more general, but also entails performance penalties. However, in many cases, the costs of run-time analysis can be effectively masked by the number of loop iterations performed, resulting in negligible overhead.

References

[1] S. Ahuja, N. Carriero, and D Gelernter. Linda and friends. *IEEE Computer*, 19.26–34, August 1986.

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

[3] A. L. Cheung and A. P. Reeves. The paragon multicomputer environement: A first implementation. Technical Report EE-CEG-89-9, Cornell University, July 1989.

[4] 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.

[5] W. Griswold, G. Harrison, D. Notkin, and L. Snyder. Scalable abstractions for parallel programming In *Proceedings of the Fifth Distributed Memory Computing Conference*, April 1990.

[6] S. L. Johnsson, Y. Saad, and M. H. Schultz. Alternating direction methods on architectures. Technical Report YALEU/DCS/RR-382, Yale Research Report, October 1985.

[7] K. Kennedy and H. Zima Virtual shared memory for distributed-memory machines. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, March 1989.

[8] K. Knobe, J. D. Lukas, and G. L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, 1990.

[9] 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.

[10] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures. In *2nd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, pages 177–186, March 1990.

[11] C. Koelbel, P. Mehrotra, J. Saltz, and H. Berryman. Parallel loops on distributed machines. In *Proceedings of the Fifth Distributed Memory Computing Conference*, April 1990.

[12] D. S. Lim and R. V. Thanakij. A survey of ADI implementations on hypercubes In *Proceedings of the Second Conference on Hypercube Multiprocessors*, 1987.

[13] R. Littlefield. Efficient iteration in data-parallel programs with irregular and dynamically distributed data structures. Technical Report 90-02-06, University of Washington, Seattle, February 1990.

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

[15] 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.

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

[17] S Mirchandaney, J. Saltz, H. Berryman, and P. Mehrotra. Parallel loops on distributed machines. In *Proceedings of the Fifth Distributed Memory Computing Conference*, April 1990.

[18] M. Quinn and P. Hatcher. Implementing a data parallel language on a tightly coupled multiprocessor. Technical Report 89-60-20, Oregon State University, Corvallis, OR, 1989.

[19] A. P. Reeves. Paragon: a programming paradigm for multicomputer systems. Technical Report EE-CEG-89-3, Cornell University, January 1989.

[20] 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.

[21] M. Rosing and R. Schnabel. An overview of DINO - a new language for numerical computation on distributed memory multiprocessors Technical Report CU-CS-385-88, University of Colorado, Boulder, 1988.

[22] 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(2):303–312, 1990.

[23] D. Socha. An approach to compiling single-point iterative programs for distributed memory computers. In *Proceedings of the Fifth Distributed Memory Computing Conference*, April 1990.

[24] P.-S. Tseng. An AL compiler for the Warp sytolic computer. In *Proceedings of the Fifth Distributed Memory Computing Conference*, April 1990.

[25] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1 Report No	2 Government Accession No	3 Recipient's Catalog No
NASA CR-187453 ICASE Report No. 90-69		

4 Title and Subtitle

PROGRAMMING DISTRIBUTED MEMORY ARCHITECTURES USING KALI

5 Report Date

October 1990

6 Performing Organization Code

7 Author(s)

Piyush Mehrotra
John Van Rosendale

8 Performing Organization Report No

90-69

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

Proceedings of the Workshop on Languages and Compilers for Parallel Computers, MIT/Pitman Press, Fall 1990

Final Report

16 Abstract

Programming nonshared memory systems is more difficult than programming shared memory systems, in part because of the relatively low level of current programming environments for such machines. This paper presents a new programming environment, Kali, which provides a global name space and allows direct access to remote data values. In order to retain efficiency, Kali provides a system of annotations, allowing the user to control those aspects of the program critical to performance, such as data distribution and load balancing. This paper describes the primitives and constructs provided by our language, and also discusses some of the issues raised in translating a Kali program for execution on distributed memory systems.

17 Key Words (Suggested by Author(s))	18 Distribution Statement
parallel language constructs, distributed memory architectures	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	23	A03

NASA FORM 1626 OCT 86

NASA-Langley, 1990

End of Document