NASA Contractor Report 181648

ICASE REPORT NO. 88-21

ICASE

PROGRAMMING PARALLEL ARCHITECTURES: THE BLAZE FAMILY OF LANGUAGES

Piyush Mehrotra

Contract No. NAS1-18107
March 1988

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

PROGRAMMING PARALLEL ARCHITECTURES: THE BLAZE FAMILY OF LANGUAGES

Piyush Mehrotra [†]
Dept. of Computer Science,
Purdue University,
West Lafayette, In. 47907.
pm@cs.purdue.edu.

Abstract

Programming multiprocessor architectures is a critical research issue. This paper gives an overview of the various approaches to programming these architectures that are currently being explored. We argue that two of these approaches, interactive programming environments and functional parallel languages, are particularly attractive, since they remove much of the burden of exploiting parallel architectures from the user.

This paper also describes recent work by the author in the design of parallel languages. Research on languages for both shared and nonshared memory multiprocessors is described, as well as the relation of this work to other current language research projects.

[†]Research supported by NASA Contract No. NAS1-18107 while the author was in residence at ICASE, NASA Langley Research Center.

1 INTRODUCTION

One of the insights that has emerged in parallel computing research is that the design of portable and efficient software for parallel systems is a critical issue. Our success in inventing new high performance parallel architectures has not been matched by equal success in learning how to program them. We need mechanisms (languages, compilers, libraries, and tools) that allow parallel algorithms to be mapped to high performance computers, and that fully exploit the variety of kinds of parallelism supported by current architectures.

Today, each multiprocessor system typically comes equipped with its own dialect of C or FORTRAN, containing a variety of language extensions for exploiting parallelism. These extensions are generally specific to a given architecture and may require programming techniques unique to that architecture. Thus, programmers wishing to use such a machine not only need to know the intricacies of their application, but must also become knowledgeable about the architecture and its programming environment.

While it is clearly useful to have parallel languages available for each new architecture, in the end we must get away from the idiosyncratic languages provided by manufacturers. For one thing, manufacturers face a host of hardware and software problems and rarely have the resources or inclination to create carefully designed and well thought through programming environments. Of equal importance, parallel architectures continue to proliferate, and today's high-end machine will soon be replaced by new generations of machines, having new and different architectures. There is a clear need for programming environments which are portable across machines of different makes and across the generations of architectures from each manufacturer.

1.1 Parallel Programming Environments

We are still some distance away from having portable and user-friendly parallel programming environments, but a great deal of progress has been made towards achieving them. Four basic approaches to the construction of such programming environments have emerged.

1. Explicit-tasking languages.
2. Direct compilation of existing sequential languages for multiprocessor execution.
3. Interactive program restructuring systems for existing sequential languages.
4. New high level parallel languages.

In practice, the distinctions between these four approaches are often quite blurred. For example, language constructs intended for high level parallel languages can easily migrate into interactive program restructuring systems. How-

ever, to keep the discussion here simple, we will treat these four approaches as distinct.

In the following paragraphs, we give a quick overview of current research in each of these areas. After that, we will focus more deeply on our own efforts in parallel language design.

Explicit-Tasking Languages

This approach is based on the view that it is the programmer's responsibility to control and manage the resources of the underlying parallel architecture. Explicit-tasking languages generally follow the concept of "communicating sequential processes," advocated by C.A.R. Hoare and embodied in the theoretical programming language, CSP [9]. In such explicit-tasking languages, the programmer defines and controls a system of interacting "tasks" or "processes." Depending on the underlying architecture, the interaction between tasks is either via synchronized sharing of data structures or via messages. In either case, the language provides mechanisms for the programmer to explicitly manage this interaction.

There is a clear advantage to this approach, since it allows complete control of the machine resources, and allows the programmer to fully exploit the target architecture. Efforts in this domain include the set of message-passing and synchronization primitives developed at Argonne National Laboratory [6] and a variety of new languages such as Occam [19], PISCES [20], FORCE [11], and LINDA [1].

The fact that explicit-tasking languages allow programs to exactly match the target architecture carries with it an attendant disadvantage: loss of portability. Portability is lost even with "portable languages" like PISCES and FORCE; though such languages can run on a variety of machines, programs generally need to be structured differently for different architectures. Moreover, the visibility of the underlying architecture makes such programs quite difficult to design and debug. The programmer is faced with a variety of load-balancing, resource allocation, and communication and synchronization issues which are not present on sequential machines.

Direct Compilation of Conventional Languages

The second approach to programming multiprocessors, direct compilation of conventional languages for parallel execution, provides a number of important advantages. First, it allows programmers to continue using familiar languages as they move to newer and more complex machines. Second, there is a large body of existing programs which can be transported to parallel architectures without change. Third, the details of the target architecture are invisible to the programmer, so the complex load-balancing and program design issues, which must be faced with the explicit-tasking languages, are not present.

This approach is, in a real sense, a direct outgrowth of successful research in construction of vectorizing compilers. It is being actively explored by major groups at at Illinois [18], Rice [4], and IBM [2], and by smaller groups elsewhere. Since the millions of lines of existing sequential programs cannot be easily replaced, nor are they readily modifiable, there is clear importance to this approach, and it will surely continue.

There are, however, a number of difficulties with this approach. The major one is that the semantics of conventional languages strongly reflects the sequential von Neumann architecture, making the task of automatic restructuring very difficult. Aliasing effects in virtually all current languages obscure data dependencies and severely limit the compiler's ability to extract parallelism. Moreover, existing languages, especially FORTRAN, encourage programming styles which make it extremely difficult for compilers to extract much parallelism. When arrays are freely "equivalenced," and passing of "pointers" is used to simulate dynamic allocation, the potential for parallel execution is quickly lost. The end result seems to be that direct compilation of sequential languages can extract only modest amounts of loop-level parallelism.

Interactive Restructuring Systems

The difficulties in direct compilation of sequential programs for multiprocessor execution has led to the exploration of a third approach to parallel programming, interactive restructuring systems. The problem with conventional sequential languages is that they do not provide the compiler with the "right" information for mapping programs to multiprocessors, and the information which they do provide is thoroughly hidden.

One way to alleviate this problem is to design a compiler which asks for "help" during the program transformation process. Programmers know far more about their programs than is directly visible in a program. For example, the typical number of invocations of a loop, the frequency with which a procedure is called, whether a procedures has "side effects," and so on, are all pieces of information a programmer might have. Interactive restructuring systems are systems which allow programmers to express this "deeper" knowledge to the compiler and thereby guide the compilation process.

There are several mechanisms through which the programmer can provide this information to the compiler. First, the programmer can modify the source code, inserting pragmas and assertions, to help the compiler extract parallelism [8]. Second, the programmer can communicate interactively with the compiler, through a window-based graphics system, which allows the programmer to view and modify the program at various stages of the transformation process [3] [18]. Third, a variety of interactive performance analysis and debugging tools are possible, which can provide rapid feedback to the programmer.

There are two principal disadvantages to interactive program restructuring systems. First, the user of such a system has to be quite knowledgeable about

the target architecture to be able to provide appropriate guidance. A naive user would not understand the nuances of the architecture and the program transformation process well enough to be of help. Second, the concept of a "program" as a file containing an algorithm expressed in a high level language is lost. Instead, the "program" is now the original source code, plus the sequence of hints and mouse clicks provided by the user during the interactive transformation process.

High Level Languages

The fourth approach to programming multiprocessors is to construct new high-level languages designed expressly for compilation to parallel architectures. There are a number of research projects focusing on the design of parallel languages which will hide most details of the parallel runtime environment. Examples of such projects include the Crystal [7], ParAlfl [10], VAL [16], SISAL [15], and BLAZE projects.

These language design projects, which are generally focusing on functional languages, are attempting to allow the programmer to concentrate on the *specification* of the algorithm rather than on its *implementation*. The goal has been to provide languages with simple and clean semantics, which make them easy for programmers to use, while also enabling the compiler to produce efficient executable code for parallel systems.

There are difficulties with this approach as well, though we tend to favor it over the other three approaches. One issue is that it is not yet clear whether these languages will enable programmers to extract the full potential of highly parallel architectures. All of the projects mentioned are in their infancy, and it is too early to declare any of them successful. Also, there is a problem with user acceptance. Programmers are, in general, reluctant to move to new a language, no matter how elegant, unless the benefits to doing so are overwhelming. Finally, none of these new languages mates well with existing languages, so it is difficult to combine the millions of lines of existing code with procedures or modules written in these newer languages.

1.2 Comparison of Approaches

In describing these four approaches to parallel programming, we have listed some of the advantages and disadvantages of each. Each of these approaches is being actively explored, because each offers advantages lacking in the others. However, in the longer term, we feel that the latter two approaches will come to dominate. The following are our reasons for this conjecture.

First, regarding explicit-tasking languages, such languages have been extremely useful in allowing users to experiment with parallel algorithm design. However, multiprocessor architectures are becoming increasingly complex, and now provide a variety of types of parallelism within the same system. As this happens, it is becoming increasingly unreasonable to expect programmers to

manage the variety of parallel resources available in a complex multiprocessor systems. Further, multiprocessors are changing from laboratory curiosities to everyday work-horses. As this happens, higher level programming environments will become essential.

Second, regarding direct compilation of current sequential languages to parallel architectures, this approach has a continuing role to play in allowing exploitation of "dusty deck" programs. However, as multiprocessors become standard, and as the number of processors in a high performance systems grows, the limitations of this approach will become apparent. As Kennedy and other experts have pointed out, compilers cannot do everything; the programmer must help.

The implication of all this is that in order to obtain maximum performance from existing code, those programs will have to be extensively "massaged" with interactive program restructuring tools and compilers. Heavily used "kernels" may be reprogrammed in new languages, but the bulk of these programs will remain essentially unchanged. Users will have to add various pragmas and compiler directives either directly to the source, or indirectly through the program transformation system. However, the programs themselves will change relatively little.

The situation for new programs is different. In this case, both of the last two approaches to parallel programming seem viable. One will either use one of the current sequential languages, together with a sophisticated interactive compiler, or one will write the program in a new high level parallel language. Both of these approaches are able to fully exploit the performance potential of highly parallel multiprocessors, and both are also relatively user-friendly, hiding most of the complex details of the parallel runtime environment from the programmer.

The choice between these two approaches is complex. Our research has focused on the design of new languages, though we fully appreciate the merits of interactive compilers and program restructuring systems. In the end we expect these two research directions to merge, resulting in the eventual creation of elegant and friendly parallel programming environments combining the best aspects of both approaches.

2 THE BLAZE PROJECT

The focus of our research in the last few years has been the BLAZE Programming Environment. BLAZE [17] is a new functional programming language for scientific applications. The intention with BLAZE is to achieve highly parallel execution on a variety of shared memory multiprocessor architectures, while shielding the user from the details of parallel execution. In particular, neither the program structure nor the execution results will in any way reflect the multiple threads of control flow which may be present during execution. Our point of view is that such issues should be the responsibility of the compiler and runtime

environment. In this section, we provide an overview of the BLAZE language and its compiler. We also briefly sketch our efforts in targeting compilers to nonshared memory machines.

2.1 The BLAZE Language

BLAZE is a parallel language for scientific programming with its roots in modern programming languages such as PASCAL, ADA, EUCLID, and MODULA 2. It contains extensive data structuring facilities and structured flow control constructs.

BLAZE is similar to data-flow languages in that it uses functional procedure invocation. That is, procedures in BLAZE act like functions which may return several values and operate without side-effects. In other words, they use value-result semantics for arguments and have no access to nonlocal variables. This simple semantics makes restructuring of BLAZE programs for parallel execution much simpler than analogous restructuring of conventional languages. In particular, whenever there are no dependencies between the input and output values of two procedure calls, they can be executed in parallel. For example, the calls to the two procedures F and G in the following program fragment can be executed in parallel.

```
a, b  :=  F( c, d);
x  :=  G( c, y);
```

With conventional languages, determining when two procedures can safely be executed in parallel requires a complex and expensive global analysis of the program.

At the statement level, BLAZE differs from data-flow languages in that it uses traditional imperative semantics, rather than the "single assignment rule" used by data-flow languages. In BLAZE variables hold values that can be altered by assignment, just as in PASCAL, FORTRAN, and other conventional languages. By contrast, in data-flow languages "variables" represent values rather than storage locations. Since names are bound to values rather than storage cells in these languages, the value of a "variable" cannot be altered once it has been set. Hence the idea of the single assignment rule.

Using traditional semantics at the statement level makes BLAZE programming natural to programmers accustomed to conventional languages. More surprisingly, the single assignment rule of data-flow languages does little to help compilation for multiprocessor architectures and in some cases can severely hamper the compilation process. In particular, handling arrays and other large data structures in data-flow languages has proven quite awkward.

In addition to the functional procedure calling semantics, which allows procedures to be executed in parallel, there are two levels at which parallelism

can be explicitly expressed in BLAZE. First, BLAZE provides extensive array manipulating facilities similar to those in ADA and FORTRAN 8x. Given the right hardware, these array operations can be executed in parallel.

Second, BLAZE contains an explicit parallel loop construct, called a **forall** loop. This provides a mechanism for the programmer to specify low-level parallelism not associated with vectors and arrays. Consider, for example, the loop:

```
forall   j   in  1 .. N     do
    . . .
end;
```

In this **forall** loop, each of the N invocations of the body can run in parallel. The actual number of parallel threads of control at runtime depends on the number of processors available on the target machine.

Each invocation of the *forall* loop body is independent and cannot modify any variable which is being accessed by another invocation. The only interaction allowed between invocations is through reduction operators as shown below:

```
x  :=  0.0;

forall  i  in  1 .. 100  do
    x  +=  y[i];
    . . .
end;
```

In the above loop the values in the array y are summed across the loop invocations. Other reduction operators provided by the language include: ***=**, **max=**, and **cat=** where the last is used for concatenation of one-dimensional dynamic lists. Properly implemented, these operators can be executed in "log-time" on a sufficiently parallel machine.

Below we give a BLAZE procedure which performs the forward elimination phase of Gaussian elimination without pivoting.

```
procedure gauss (A,b)  returns : (A,b)

param A: array [ , ] of real;
      b: array [ ]   of real;

const N = upper(A,1);                -- upper bound of the first
                                     -- dimension of the array A

begin

    for  k  in  1 .. N  do                -- loop over pivot rows

        forall  i  in  k+1 .. N  do
           real   scale;                  -- local variable for
                                          -- each loop invocation

           scale :=  -A[i, k] / A[k,k]; -- compute scale factor

           A[i, k..N]  +=  scale * A[k, k..N];  -- update row
           b[i]  +=  scale * b[k];        -- modify data vector
        end;
    end;
end;
```

The outer loop is a sequential loop over all the rows of the array making each row the pivot row in turn. The inner loop is a parallel loop which updates all rows below the the pivot row. In this example, the reduction operator "+=" is used simply as a notational convenience. Since the variables on both sides of these "reductions" are local to the current loop invocation, no parallel "tree-sum" is implied.

Structure of the BLAZE Compiler

The structure of BLAZE compilers is dictated by our desire to target this language to a number of sequential and shared memory systems and by the necessity of performing extensive transformation and optimization during compilation. There is a machine independent front-end which performs, lexical analysis, parsing, and the first few phases of optimization and machine-independent transformations. After this, further optimization and code generation is performed in machine specific compiler back-ends.

BLAZE source programs are first translated into an intermediate form representing the control-dependence between the statements of the program [12]. Extensive data-flow analysis is then performed to augment the control-dependence graph with the data dependencies between the variables. These include flow-

dependencies, anti-dependencies, and output-dependencies, as described by Kuck et al [14].

The functional procedure invocation semantics of BLAZE makes data-flow analysis much simpler and more "accurate" than it is with conventional languages, since no inter-procedural analysis is required. Even when complete inter-procedural data-flow analysis is performed for conventional languages, the resulting information is imprecise, because language features such as pointers and common blocks frequently obscure data-flow information.

An extensive set of program transformation techniques has been developed over the years for automatic vectorization of sequential code, most notably by the research groups at University of Illinois [14] and at Rice University [5]. The BLAZE compiler builds up upon this body of knowledge in an attempt to generate code for multiprocessor architectures.

The underlying goal of this analysis and transformation phase is to expose the parallelism available in the program. However, in most cases, the inherent parallelism of the algorithm does not exactly match that of the target architecture. Thus, the next step in the transformation process is to map the algorithm parallelism onto the architecture at hand. The independent threads of control in the program are "bundled" into a set of concurrently executing processes, which can efficiently exploit the parallel architecture.

Implementations of the BLAZE compiler currently exist for Sequent, Alliant, and Butterfly multiprocessors systems. These have been implemented by students at Purdue University, Indiana University, and the University of Utah. These are experimental versions and take a relatively naive approach to implementing parallel constructs such as the `forall` loop. We are currently exploring alternate implementations and are beginning to study the effect that alternative implementations have on runtime performance [13].

2.2 Targeting Nonshared Memory Architectures

The BLAZE language is targeted primarily towards shared memory multiprocessors. While nonshared memory architectures having very high performance can be built, programming them is substantially harder than programming shared memory multiprocessors. This is primarily because issues such as data distribution and load balancing play a far more critical role on nonshared memory architectures than they due on shared memory machines. These issues make it very difficult for a compiler to automatically generate good code for nonshared memory machines. Instead, the user needs to explicitly specify data distributions, and must carefully plan load balancing strategies, in order to effectively utilize these machines.

The current approach to programming nonshared memory architectures is based on the use of explicit-tasking languages. Such languages seem to be ideally suited for some classes of algorithms, such as game tree searching and discrete event simulation, where the problem decomposes naturally into a sys-

tem of cooperating processes. However, for algorithms relying on synchronous manipulation of distributed data structures, such languages have proven quite awkward.

KALI [1] is a research language designed to simplify the problem of programming nonshared memory architectures. It provides the semantic power of message-passing languages, such as CSP and Occam, while also providing a set of novel features for specifying and manipulating distributed data structures. The goal is to allow the user to retain control over data distribution and other issues critical to efficient parallel execution, while leaving the complex details of data transmission to the compiler and runtime environment. In the next section we give an overview of the parallel constructs of KALI.

2.2.1 Overview of KALI

KALI is a high-level, object-oriented language for distributed memory architectures. A KALI program is a sequence of *cluster* specifications, followed by an optional list of procedures. Clusters are a form of "object" or "process." That is, each cluster encapsulates a data structure and has its own independent thread of control flow.

At program initiation, the unique cluster **main** begins execution. It may in turn dynamically create *instances* or *activations* of other clusters during program execution by sending **create** messages. Arbitrarily many instances of any cluster may be created, except for **main**, which has only one instance. Cluster instances do not share variables and can interact only via asynchronous message passing.

Clusters

There are two kinds of clusters, *sequential* clusters and *distributed* clusters. A *sequential* cluster is a process or object having a single thread of control flow. Multiple instances of a sequential cluster may execute concurrently, but each executes as a sequential process. A *distributed* cluster, by contrast, supports SPMD-style (Single Program Multiple Data) parallel execution within each instance of the cluster. Since sequential clusters are quite conventional, we will focus only on distributed clusters and on the data-parallel execution within them.

KALI assumes a nonshared memory architecture in which the programmer explicitly manages all critical resources. It further assumes that the architecture can support the idea of *processor arrays*, multidimensional arrays of physical processors, dynamically allocated by the user. This assumption is natural for hypercubes or mesh connected machines and can easily be accommodated on

[1] The name "KALI" is taken from one of the Hindu goddesses with multiple arms, suggesting the idea of parallel execution.

a variety of other architectures. At the time of its creation, each instance of a distributed cluster is allocated a processor array, on which it will execute.

Syntactically, a cluster specification has a single level of static nesting. There is a sequence of declarations at the beginning of the cluster specification declaring variables and constants visible to procedures within the cluster. For a distributed cluster, this sequence of declarations also contains a declaration of a processor array as shown below:

```
procs    P[np, np];
integer  np ~ 10;
```

These statements allocate a square array P of np^2 processors, where np is an integer constant between 1 and 10 dynamically chosen by the runtime system.

The programmer controls the distribution of the data structures across the cluster's processor array. KALI currently supports only distributed arrays, though other distributed data structures will be allowed in future versions of the language. Arrays 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 on each of the processors in the processor array.

Data Distribution Primitives

Each dimension of a data array can be distributed across processors in one of two patterns, or can be left undistributed. The distribution patterns are **block** and **cyclic**. With a **block** distribution, each processor contains a contiguous block of elements of the array. Conversely, with a **cyclic** distribution, the array elements are distributed in a round-robin fashion across the processors. As an example, consider the following declarations:

```
procs    P[np];
integer  np ~ 10;
real     A[100]  dist [block];
real     B[100]  dist [cyclic];
```

Here, P is an array of up to ten processors, and A and B are vectors having 100 elements. Assuming for simplicity that P contains exactly 10 processors, the subvector $A[1..10]$ would be assigned to processor $P[1]$, $A[11..20]$ would be assigned to processor $P[2]$, and so on. By contrast, with the **cyclic** distribution, processor $P[1]$ will have elements 1, 11, 21, ..., 91 of the vector B, processor $P[2]$ will have elements 2, 12, 22, ..., 92 of B, and so on.

The number of dimensions of an array that are distributed must match the number of dimensions of the underlying processor array. Hyphens are used to indicate dimensions of data arrays which are not to be distributed. Consider, for example, the following declarations:

```
procs     P[np];
integer   np = 10;
integer   C[100, 100]  dist [block, -],
          D[100, 100]  dist [-, block];
```

In this case, the row dimension of C is broken into blocks. Thus, each processor in the processor array P contains a group of rows of C, and each column is distributed across all processors in P. Conversely, in the case of the array D, each processor contains a group of columns, and the rows of D are split across processors.

Forall Loops

Data-parallel computation on distributed data structures is specified via **forall** loops. The **forall** loop header consists of a range specification and an **on** clause. The range specification specifies the number of invocations of the loop body, while the **on** clause specifies the processor on which each loop is to be executed. The most elementary way of doing this is to simply specify the processor explicitly:

```
procs   P[10];
real    A[100]  dist [block];
  . . .

forall  i  in 1 : 10 on P[i] do
     . . .
end;
```

Here the ith processor executes the ith loop invocation.

More generally, the execution of a **forall** loop can be tied to a distributed data structure through the use of a **proc** primitive in the **on** clause. Given an element of a distributed data structure, **proc** returns the processor on which it resides. This allows one to specify that a loop invocation be executed on the processor containing certain data, avoiding the necessity of messy index calculations. In the program fragment below, 100 loop invocations are performed, with the ith invocation executed on the processor owning the ith element of the vector A.

```
procs P[10];
real A[100] dist [block];
  . . .

forall  i  in  1 : 100  on  proc( A[i] )  do
     . . .
end;
```

The compiler will strip-mine the above loop and convert it into a system of cooperating processes, one per processor. Each process will contain a sequential loop running over the elements of the distributed vector A local to that processor.

Data Movement

Each invocation of a **forall** loop can directly access only those data elements local to the processor executing that loop invocation; nonlocal parts of the data structure cannot be implicitly accessed. In KALI, access to nonlocal data must be explicitly specified via a set of high level primitives provided by the language. There are five communication primitives for data movement within a cluster: **expand**, <- (send), **nbr**, **fetch**, and **reply**. Though KALI requires the user to manage communication within a cluster, these are relatively high-level primitives. The compiler translates these primitives into the system of sends, receives, and synchronization barriers that will actually be executed.

The first two of these primitives are used for sending data to other processors. **Expand** is used to broadcast data. It takes as argument data local to a processor and broadcasts it to all processors in the processor array. Thus, in the following example, the processor owning the element $A[j]$ broadcasts it to all others.

```
forall  i  in  1 : N  on  proc( A[i] )  do
    real  x;
    x :=  expand ( A[j] );
          . . .
end;
```

Since x is declared within the loop, each loop invocation has its own copy of x.

The other form of "send" is denoted by an arrow: <-. It is used in place of the normal assignment operator := to send data to a remote place. For example, consider the following program fragment:

```
forall  i  in  1 : 100  on  proc( A[i] )  do
     B[ f(i) ]  <-  A[i] ;
          . . .
end;
```

Here the values in the array A are "permuted" to form the array B.

The next two primitives, **nbr** and **fetch**, provide software simulation of shared memory semantics. **Nbr** is used for fetching data from adjacent processors in the processor array. This primitive is useful in a variety of numerical applications, such as relaxation schemes and "smoothing" algorithms, where the value at a point is computed from a set of neighboring values. As an example, consider the program segment:

```
forall  i  in  1 : 100  on  proc( A[i] )  do
     A[i] :=  A[i]  +  nbr( A[i-1] )  +  nbr( A[i+1] );
          . . .
end;
```

If A is distributed **block**, most of the accesses to $A[i-1]$ and $A[i+1]$ will be "local." However, communication is required at block boundaries, so the **nbr** primitive is needed.

The **fetch** primitive is more general and can be used to access data from anywhere in the processor array.

```
forall  i  in  1 : 100  on  proc(A[i])  do
     real x;
     x  :=  fetch ( A[f(i)] );
          . . .
end;
```

Since this primitive amounts to direct software simulation of shared memory, it should be used with great caution; the overhead involved is likely to be quite high. On a non-shared memory architecture, the compiler must translate each fetch request into a system of sends and receives. In this example, if the function f is not a permutation, each processor would have to field an arbitrary number of fetch requests. In general processors must busy-wait for fetch requests until all outstanding fetches have been answered, before continuing on to the next computation.

In order to make this approach tractable, the runtime environment must be designed so that communication between processors occurs in "phases." Between communication phases, processors execute sequentially, without interference from other processors. A communication phase occurs whenever processors synchronize. Synchronization occurs:

a. At the end of **forall** loops,

b. After any of the above communication operators, and

c. When induced by a **reply** statements.

Whenever a processor encounters any of these synchronization points it blocks and handles pending fetch and send requests. The semantics of these synchronization constructs is subtle, but fortunately has no effect program correctness. If the synchronization is handled badly, only performance bugs results, not incorrect results or dead-lock.

Comparison of KALI and BLAZE

We have given above a brief overview of KALI, concentrating on its most novel features. KALI was not designed to be an "elegant" language, in the sense of Modula, BLAZE, or Sisal. Rather, KALI was driven by the needs of programmers trying to map numerical algorithms to nonshared memory architectures. After one has seen programmers struggling enough times with the index arithmetic needed to implement **cyclic** and **block** distributions by hand, it becomes apparent that language features such as those given here are clearly needed for nonshared memory architectures.

KALI has been designed only recently and is not yet implemented. Thus, it is impossible to estimate accurately the cost of the communication primitives described. However, in many cases we can guarantee that the KALI implementation will execute as fast as the more laborious message-passing code. Consider for example the forward elimination phase of Gaussian elimination as shown below:

```
procedure  gauss(A, B, P)  returns(A, B);
procs    P[np];
real     A[n, n]  dist[cyclic, -];
         B[n]     dist[cyclic];
begin

    for  k  in  1:n  do              -- loop over pivot rows
        forall  i  in  k+1:n  on  proc(A[i, -])  do
           real   PivRow[n],  scale;

           PivRow[k:n] :=  expand( A[k, k:n] );
                                    -- broadcast pivot row

           scale       :=  -A[i, k] / PivRow[k];

           A[i, k:n]  +=  scale*PivRow[k:n];
           B[i]       +=  scale*expand(B[k]);
        end;
    end;
end;
```

This *gauss* procedure, as expressed in KALI, will perform as well as the analogous CSP or Occam procedure, since the output of the restructuring phase of the compiler is virtually identical to the analogous Occam procedure. Despite this, the above KALI procedure is shorter and much simpler than the analogous Occam code. In fact, it closely resembles the analogous BLAZE procedure given before. The precise differences are:

1. The processor array is explicitly present here.
2. Data distributions to accomplish load-balancing have been specified here.
3. Interprocessor communication was specified by the **expand** primitive.

However, despite this syntactic resemblance between KALI and BLAZE, one should not overlook the dramatic semantic differences. We do not want to suggest that KALI procedures like this are as easy to write as comparable BLAZE code; there are subtleties and land-mines here not present in analogous shared memory code. We are merely arguing that this is a moderately high level way to specify algorithms for nonshared memory architectures, while retaining the full performance potential of these architectures.

3 CONCLUSIONS

Software technology has not kept pace with hardware technology in the domain of parallel processing. One of the major problems facing users of multiprocessor systems is the lack of adequate software tools. Until this problem is resolved, we will fail to effectively utilize parallel architectures on most problems.

In this paper we have briefly sketched the several directions that are being explored to provide portable programming environments for parallel machines. As architectures become more complex, tools such as the languages and compilers described will have to assume a greater role in making these architectures programmable and useful. Efficient utilization of parallel architectures requires a combination of good language design and advanced compiler technology.

Acknowledgement

The BLAZE project has been a joint research project with John Van Rosendale of University of Utah, currently serving as visiting faculty at Argonne National Laboratory. The author gratefully acknowledges his collaboration in this project and his helpful comments regarding this paper.

References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[2] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. *An Overview of the PTRAN Analysis System for Multiprocessing.* Research Report RC 13115 #56866, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1987.

[3] J. R. Allen, D. Callahan, and K. Kennedy. A parallel programming environment. *IEEE Software*, 2(4):21–29, July 1985.

[4] J. R. Allen, D. Callahan, and K. Kennedy. *Program Transformations for Parallel Machines.* Technical Report TR85-20, Department of Computer Science, Rice University, Houston, TX, March 1985.

[5] R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[6] J. Boyle, R. Butler, T. Disz, B. Glickfield, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors.* Holt, RineHart and Winston, Inc., 1987.

[7] M. C. Chen. *Very-high-Level Programming in Crystal.* Research Report YALEU/DCS/RR-506, Department of Computer Science, Yale University, New Haven, Ct., December 1986.

[8] H. Dietz and D. Klappholz. Refined FORTRAN: another sequential language for parallel programming. In K. Hwang, S. M. Jacobs, and E. E. Swartzlander, editors, *Proceedings of the 1986 International Conference on Parallel Processing*, pages 184–191, IEEE Computer Society Press, August 1986.

[9] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[10] P. Hudak. Parafunctional programming. *IEEE Computer*, 19(8):60–71, August 1986.

[11] H. F. Jordan. *The Force.* Report, Dept. of Electrical and Computer Engineering, University of Colorado, Boulder, Co., January 1987.

[12] Charles Koelbel and Piyush Mehrotra. *The BIF Data Structures User's Manual.* in preparation, Purdue University, West Lafayette, IN, 1988.

[13] Charles Koelbel, Piyush Mehrotra, and John Van Rosendale. Semi-automatic domain decomposition in BLAZE. In Sartaj K. Sahni, editor, *Proceedings of the 1987 International Conference on Parallel Processing*, pages 521–524, Pennsylvania State University Press, August 1987.

[14] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[15] J. McGraw, S. Skedzielewski, S. Allan, D. Gale, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee. *SISAL: Streams and Iterations in a Single-Assignment Language: Reference Manual Ver. 1.2.* Manual M-146, Rev. 1, Lawrence Livermore National Labs, Livermore, Ca., 1985.

[16] James R. McGraw. The VAL language: description and analysis. *ACM Transactions on Programming Languages and Systems*, 4(1):44–82, January 1982.

[17] Piyush Mehrotra and John Van Rosendale. The BLAZE language: a parallel language for scientific programming. *Parallel Computing*, 5(3):339–361, November 1987.

[18] David A. Padua, David J. Kuck, and Duncan H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, C-29(9):763–776, September 1980.

[19] D. Pountain. *A Tutorial Introduction to Occam Programming.* Inmos, Colorado Springs, Colo., 1986.

[20] T. W. Pratt. Pisces: an environment for parallel scientific computation. *IEEE Software*, 2(4):7–20, July 1985.

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181648 ICASE Report No. 88-21		

4. Title and Subtitle	5. Report Date
PROGRAMMING PARALLEL ARCHITECTURES: THE BLAZE FAMILY OF LANGUAGES	March 1988
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Piyush Mehrotra	88-21
	10. Work Unit No.
	505-90-21-01

9. Performing Organization Name and Address	11. Contract or Grant No.
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	NAS1-18107
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Contractor Report
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

Submitted to the Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing

16. Abstract

Programming multiprocessor architectures is a critical research issue. This paper gives an overview of the various approaches to programming these architectures that are currently being explored. We argue that two of these approaches, interactive programming environments and functional parallel languages, are particularly attractive, since they remove much of the burden of exploiting parallel architectures from the user.

This paper also describes recent work by the author in the design of parallel languages. Research on languages for both shared and nonshared memory multiprocessors is described, as well as the relations of this work to other current language research projects.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
programming languages, parallel computers	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	20	A02