NASA Contractor Report 172615

ICASE REPORT NO. 85-29

ICASE

THE BLAZE LANGUAGE: A PARALLEL LANGUAGE FOR SCIENTIFIC PROGRAMMING

Piyush Mehrotra

John Van Rosendale

Contracts No. NAS1-17070 and NAS1-17130

May 1985

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

Operated by the Universities Space Research Association

The BLAZE Language: A Parallel Language for Scientific Programming[1]

Piyush Mehrotra
Department of Computer Science, Purdue University

John Van Rosendale
Institute for Computer Applications in Science and Engineering

Abstract

Programming multiprocessor parallel architectures is a complex task. This paper describes a Pascal-like scientific programming language, Blaze, designed to simplify this task. Blaze contains array arithmetic, "forall" loops, and APL-style accumulation operators, which allow natural expression of fine grained parallelism. It also employs an applicative or functional procedure invocation mechanism, which makes it easy for compilers to extract coarse grained parallelism using machine specific program restructuring. Thus Blaze should allow one to achieve highly parallel execution on multiprocessor architectures, while still providing the user with conceptually sequential control flow.

A central goal in the design of Blaze is portability across a broad range of parallel architectures. The multiple levels of parallelism present in Blaze code, in principle, allows a compiler to extract the types of parallelism appropriate for the given architecture, while neglecting the remainder. This paper describes the features of Blaze, and shows how this language would be used in typical scientific programming.

---

[1]Research was supported by the National Aeronautics and Space Administration under NASA Contract Nos. NAS1-17070 and NAS1-17130 while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

1. Introduction

Designing software environments for parallel computers is a central issue in parallel computing research. For a variety of reasons, it seems to be far easier to program sequential computers than parallel machines. In particular, designing concurrent programs having multiple threads of control flow, has proven remarkably subtle. With parallel computing becoming the standard approach to large scale scientific computing, better programming methodologies are clearly essential.

One of the first questions to ask is whether the difficulty experienced in programming parallel architectures is inherent in parallel execution or is instead a reflection of the inadequacy of current software tools. We suspect the latter, but the question is still open. Another way of posing this question is to ask whether one can design programming environments which allow one to write correct and efficient parallel programs as easily as one currently writes sequential programs. The Blaze language is intended as a first step towards the creation of such environments.

1.1. The Blaze Language

Blaze is a parallel scientific programming language having a Pascal-like syntax and functional or applicative procedure calls. The scientific orientation of Blaze is reflected in a number of language features, such as the extensive array manipulation facilities, which are similar to those of Ada[1] and Fortran 8x[15]. Blaze is also, to a lesser extent, intended as a general purpose programming language containing extensive data structuring facilities and

structured flow control constructs. In particular, it contains records, lists, recursion, type definitions, and enumerated data types. Pointers are not provided, as pointers are virtually incompatible with the functional semantics of procedure invocation here. However, the list data type provides some of the same expressive power and is often much simpler to use.

Blaze is not a multi-tasking language. With the exception of the "forall" statement here, control flow in Blaze is entirely sequential. To be more precise, control flow is conceptually sequential, though programs will often be executed in asynchronous multi-tasking environments. The intention is to achieve highly parallel execution on a variety of SIMD and MIMD architectures, while shielding the user entirely 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. Such issues are the responsibility of the compiler and run-time environment.

Achieving the goal of hiding the parallel run-time environment from the user is largely a matter of compiler technology. With current compiler technology, it is very difficult to achieve highly parallel execution by automatic restructuring of conventional sequential languages. The structure of Blaze is designed to meet this problem. Thus, for example, functional procedure invocation is used here, both because it lends itself to a clean and elegant programming style, and because it greatly simplifies the compiler transformations

required to automatically extract parallelism. In this, and several other regards, we follow the lead of the researchers in data flow languages.

Though Blaze is not a research language in the usual sense, it is intended as a vehicle for several kinds of research. For example, Blaze is one of the first languages to combine functional semantics in procedure invocation with conventional imperative semantics within procedures. This idea has been used before, in the language proposed by Kessels[9] and in the Edinburgh language ML[6,12,13] but Blaze is the first language exploiting this idea intended for widespread practical application. Thus, while Blaze is in some ways virtually a data flow language, programming in Blaze feels almost like programming in Pascal or Modula. For programmers trained in conventional languages, Blaze programming will be natural, while programming in the data flow languages appears to require a significant adjustment.

Blaze is also intended as a research vehicle in compiler technology. Many people would argue that high performance on multiprocessor architectures requires the use of multi-tasking languages whose semantics closely reflects the underlying architecture. Whether the alternative approach here will succeed depends on the kind of parallelism encountered in actual programs, on architectural performance issues, and on the development of appropriate compiler technology.

Finally, Blaze is also a research vehicle in the sense that it is one of the few attempts in recent years to create a new scientific programming language. Fortran has become such a standard that it is difficult to seriously consider

other scientific languages. However, if an alternate language provides obviously useful features not available in Fortran, users might be willing to adopt such a language, especially if the alternate language yielded significantly faster execution. Exactly which language features are most useful in scientific programming is not well understood, and Blaze is, in a real sense, a probe into this issue.

1.2. Overview of Paper

This paper presents the Blaze programming language and describes preliminary work on its implementation. The design goals and central features of Blaze are discussed, and the paper raises two basic questions. First, how suitable is this language for scientific computation? Second, can Blaze programs be easily compiled into efficient executable code for parallel architectures?

The first question is addressed in the next three sections. Section two discusses the central issues in parallel programming and the way in which these issues affected the design goals of Blaze. Section three describes the principal features of Blaze in relative detail Following that, section four presents an extended example program taken from plasma physics From this example, one should be able to assess reasonably well the relative merits of Blaze as a scientific programming language.

The second question, on compilation of Blaze, is discussed briefly in section 5. On multiprocessor architectures, we hope to be able to automatically extract large amounts of parallelism from typical scientific code, but many research questions remain in this area.

2. Language Design

Given the hundreds of computer languages already in existence, one should not casually introduce a new one. However, the problem of learning how to use multiprocessor architectures is important enough to justify exploration of a number of alternative approaches to this problem. In our view, the design of programming languages appropriate for parallel architectures is one of the most promising approaches to this problem.

One prominent viewpoint is that compilers should be constructed which automatically restructure conventional languages for efficient execution on parallel architectures The difficulty with this viewpoint is that the semantics of conventional languages reflect the sequential Von Neumann architecture so strongly that the required program transformations are extremely difficult. For example, in Fortran subroutine invocation the aliasing effects obscure data dependencies and thus severely limit the compiler's ability to extract parallelism.

As with the data flow languages, Blaze is based on the view that a more rational approach is for programmers to meet compiler writers halfway, with languages reflecting the needs of both. Conventional languages, such as Fortran and Pascal, reflect the architecture, compiler technology, and programming needs at the time of their creation. As programming methodologies and computer architectures change, programming languages must evolve with them. Such evolution is seen in conventional languages, as Fortran 77 gives way to Fortran 8x, and Pascal to Ada and Modula. In Blaze, this natural evolution is carried somewhat further, with the language semantics driven by compiler and

architectural issues to an unusual degree. However, Blaze is not a radically new language and can be quickly mastered by programmers used to Pascal-like languages.

2.1. Advantages of Functional Procedure invocation

In most current languages, procedures are the principal means for structuring programs. Thus the method of passing parameters between procedures is one of the critical distinguishing characteristics of a language. Most languages allow two methods of accessing non-local data; parameters may be passed via argument lists, and there is also some sort of global variable referencing mechanism. For example, Fortran uses argument lists and common blocks. Similarly, Pascal uses argument lists, and each procedure may also access the variables of any procedure in which it is nested.

In functional languages every procedure is a pure function. That is, there is no global referencing environment for a procedure, nor can procedures save values between invocations. Thus the entire effect of a procedure call in a functional language is the assignment of values to returned parameters. Functional programming is an inherently austere style of programming, conducive to concise, easy to understand programs. An important benefit of this programming style is that one can tell at a glance exactly which variables can be effected by a procedure.

Blaze is a functional language, both because the functional programming style appears to have clear benefits to the programmer, and because in parallel environments functional languages have the advantage that they make extraction

of procedure level parallelism much easier. For example, in the Fortran statement

$$Z = F(X) + G(X) + F(Y)$$

it may be possible to perform all three function invocations concurrently. However, it is difficult for a compiler to determine whether this is possible, since to do so it would have to check whether F alters its argument, and also examine all common blocks and save statements in functions F and G and in any functions or subroutines they in turn invoke.

Since Blaze is a functional language, two procedures can be executed concurrently whenever there are no data dependencies between their passed parameters This property dramatically simplifies construction of optimizing compilers for Blaze, since global data flow analysis is now trivial. With conventional languages, even if global data flow analysis is done, cases will arise where a compiler cannot tell with certainty whether it is safe to execute procedures concurrently. In such cases the compiler must make the safe but suboptimal decision, and require sequential execution.

2.2. Alternative Approaches to Parallel Programming

The Blaze language is one of a number of possible approaches to programming parallel architectures. Among the alternatives are the use of "vector" or SIMD languages There are also a variety of functional or applicative languages, such as Lisp, APL, FP, and the data flow languages Sisal, VAL, and Id Another alternative is the use of explicit multi-tasking

languages such as CSP, Ada, or Fortran 8x. Finally, one can use a conventional language such as Fortran or Pascal, relying on the compiler to extract parallelism.

Each of these approaches has advantages and disadvantages. The distinction between these approaches is largely a matter of how much of the responsibility for exploiting parallelism falls to the user, to the compiler, and to the run-time environment. With a vector language, or an explicit multi-tasking language, such as CSP, nearly all of the responsibility for exploiting parallel architectures falls to the user. The other extreme is the use of compilers to restructure conventional languages, where virtually all of the responsibility for exploiting the parallel architecture is left to the compiler. The use of functional or data flow language is an intermediate approach, where part of the responsibility falls to the user and part to the compiler and run-time environment.

A major attraction to the approach of using a conventional language, such as Fortran, with compilers that automatically restructure programs for parallel execution, is that this approach would allow one to transport the large body of existing programs to parallel architectures essentially without change. Partly for this reason, a substantial effort is being devoted to the development of such compilers. One of the major efforts is at University of Illinois, where D. Kuck and his colleagues, R. Kuhn, B. Leasure, D. Padua, M. Wolfe, and others, have been developing the Paraphrase and KAP program restructuring systems[10,11,14]. A related effort is underway at Rice University by K.

Kennedy, J. R. Allen, and others[3,8]. Finally F. Allen, J. Ferrante, K. Ottenstein, J. Warren, and others are carrying out similar research at IBM Yorktown Heights and Michigan Technical University[5,16].

One issue here is how well this approach will work as one moves to highly parallel architectures. There has been considerable success so far but many difficulties remain. A more serious issue is that the use of a conventional language gives the programmer no feedback on the amount of parallelism the compiler will extract. Such feedback can be critical in designing efficient parallel programs.

An alternate approach to the use of conventional languages and advanced compiler technology is the use of an explicit multi-tasking language. Such languages reflect the parallel run-time environment very strongly, perhaps too strongly. One disadvantage here is that multi-tasking programs tend to be much more difficult to write. The programmer must divide the program into a set of cooperating processes, with appropriate communication and synchronization between processes. This is complicated and also introduces the possibilities of dead-lock and non-determinate execution. Also, it is critical that the granularity of user defined processes match that required by the specific architecture. This gives the user the possibility of tuning his algorithm to the architecture, at the cost of a corresponding loss of portability.

The approach of using a functional or data flow language is intermediate between the approach of using a conventional language, with compilers that automatically extract parallelism, and the alternate approach of using an explicit

multi-tasking language. With a functional language, the programmer, compiler, and run-time environment share responsibility for achieving parallel execution. The functional or side-effect free semantics of procedure invocation eliminates the need for complex and expensive inter-procedural data flow analysis during compilation However, sophisticated compilation techniques are still required to split loops across processors, to deal with memory allocation, and to extract parallelism.

3. Language Features

In this section, we survey the principal features of the Blaze language. The first few subsections describe the data types available and the operations allowed on them. The next few subsections present the sequential and parallel control constructs of the language. Finally the input and output operations are discussed.

A Blaze program consists of a main procedure (begun with the reserved word **program**) and a sequence of other procedures. Blaze procedures can return zero, one, or more values. Thus they subsume the roles played by both functions and procedures in conventional languages.

Procedure invocation in Blaze is functional or applicative. Interpreted strictly, this would mean that the entire effect of calling a procedure would be the assignment of values to the parameters being returned. Blaze departs slightly from strictly functional procedure invocation by allowing procedures to read and write the standard input and output files. However, this is the only side effect occurring in procedure calls here. In particular, access to non-local

variables is not allowed.

Blaze does allow global constants and global type definitions. These create no unwanted side effects and are convenient. For example, array sizes can be set by global constants. Global constants and type definitions are given in a preamble before the program statement.

Unlike Pascal, where procedures may be nested, procedures here are all declared at the same 'level,' as in the C language. This is natural here, since procedures cannot access non-local variables. Thus the concept of nested procedures is not very meaningful here. In a planned future extension, Blaze will incorporate separate compilation units and some form of abstract data type, thus largely obviating the need for procedure nesting.

3.1. Elementary Data Types

Blaze contains elementary and structured data types similar to those found in other current languages. The elementary data types are integers, booleans, characters, and single and double precision floating point numbers. The language requires the explicit declaration of all variables, except loop indices, using a declaration syntax similar to that of Pascal:

```
var     i,j,k  : integer;
        flag   : boolean;
        x,y    : double;
```

Blaze also allows constants. A constant here is a variable whose value is set at declaration, and whose value may not be subsequently altered. The constant declaration syntax is:

```
const    n = 10;
         m = n-1;
         u = 3.14e-2;
         v = some_function(m,n);
```

As shown, the value of a constant can be given by any expression, including expressions involving procedure calls and thus may require run-time evaluation.

3.2. Structured Data Types

Blaze contains extensive facilities for constructing structured data types. The structured types available here are arrays, records, enumerated types, lists, and combinations of these types. The need for such structured data types is now widely appreciated. They are especially important in languages like Blaze, having functional procedure invocations and no global variables, since they make it possible to package several types of data in a single structure, obviating the need for long and awkward parameter lists.

Arrays

Arrays are the most important structured data type in scientific computation, and Blaze contains an extensive set of array manipulation facilities. The syntax for array declarations is similar to that of Pascal:

```
var      u,v,w  : array[1 .. N] of real;
         b0,b1  : array[1 .. 512, 1 .. 512] of boolean;
```

One can also declare arrays using type declarations:

```
type    vector    = array[1 .. N] of real;
        bit_array = array[1 .. 512, 1 .. 512] of boolean;

var     u,v,w   : vector;
        b0,b1   : bit_array;
```

In Blaze, one can access either single elements of arrays or rectangular subarrays. The syntax for accessing a single element is the same as the syntax in Pascal:

u[i]	*— the i'th element of u*
b0[i + 3, 2*j]	*— an element of array b0*

Rectangular subarrays can be accessed by using an index range or by using an asterisk to specify the entire range of an index. Examples are:

u[1 .. 10]	*— the first 10 elements of u*
b0[3, *]	*— the third row of array b0*
b0[*, 2 .. 5]	*— a rectangular slice of array b0*

In all cases, Blaze checks for out-of-range array accesses, though a compiler option allows this checking to be turned off.

Blaze provides built-in primitives to determine the sizes of array variables. For example, the lower and upper limits for indices of an array variable can be determined as follows:

upper(*u*)	— *the upper limit of the one dimensional array u*
upper(b0[*,])	— *the upper limit of the first index of array b0*
lower(b0[, *])	— *the lower limit of the second index of array b0*

The asterisk here specifies the index position of interest and is optional for one dimensional arrays.

Records

Arrays form a homogeneous collection of elements, while records structures allow the programmer to specify heterogeneous elements in a single structure. For example, the following record structure can be used for the specification of the properties of a charged particle:

```
type    ion_species = (electron, proton, neutron, deuteron, alpha);

        charged_particle = record
                xpos, ypos      : real;
                charge          : integer;
                species         : ion_species;
        end;
```

The fields in a record may be of any previously declared type. Here, for example, *species* is of the enumerated type *ion_species*. This allows arrays, record structures, and lists to be fields of a record. Access to fields in records follows standard Pascal syntax:

```
var a : charged_particle;

        a.xpos          — the x position of the particle

        a.species       — the species of this particle
```

Blaze also allows tagged variant records, using syntax following that of Ada. For example, the above record could have been extended as follows:

```
type variant_charged_particle = record
        xpos, ypos : real;
        xvel, yvel : real;
        charge     : integer;

        case variant_tag : ion_species of

                when deuteron =>
                        proton_cross_section    : real;

                when alpha =>
                        ionization_level        : integer;

                when others =>
                        lifetime                : real;
        end;
end;
```

The choice **others** is used to specify the fields for the default case.

Lists

Blaze provides an extensible one-dimensional array called a list. The syntax for list declarations follows that of arrays:

```
var plasma : list of charged_particle;
```

The lower limit of a list variable is always one, while the upper limit can be changed dynamically. This can be done, for example, by concatenating another list or element of the same base type to the end of the original list. This will be described later in more detail. The elements of a list are accessed exactly as one accesses the elements of a one dimensional arrays, and one can use the built-in primitive **upper** to determine size of a list at any time.

3.3. Recursive Data Structures

There are no pointers in Blaze, but many of the data structures one would construct with pointers in other languages can be built with the lists here. We also allow recursive data structures. As in most languages, Blaze does not allow a record type to have itself as one of its fields. However, recursion is allowed in variant records:

```
type binary_tree_node = record
        case tag : (leaf, non_leaf) of
                when leaf     =>
                        data : integer;
                when non_leaf =>
                        l_child, r_child : binary_tree_node;
        end;
```

One can also use lists to achieve a similar effect:

```
type tree_node = record
                 data          : particle;
                 sub_tree      : list of tree_node;
end;
```

This type definition recursively defines a general n-ary tree, while the previous one defined a binary tree. Notice that these recursive definitions allow data structures of arbitrary size, but do not imply infinite storage. The recursion is terminated in the first case by setting the variant tag to "leaf," while in the second case, lists of size zero will terminate the recursion.

3.4. Expressions

Expressions in Blaze are similar to those in other high level languages, differing mainly in that the arithmetic operations are extended to allow array operations The assignment operator is also somewhat more general than usual, allowing assignment to occur to any type of compatible object. Thus arrays, records, and lists are permitted on the left side of assignment statements. These features are convenient and provide a natural source of low level parallelism.

Arithmetic Expressions

Automatic type conversions of arithmetic operands occur in Blaze, as in most languages. Integers are converted to reals when they occur together in dyadic arithmetic operations, and, similarly, single precision reals are converted to double precision when they occur together in expressions. The reverse conversions occur only in assignment. For example, assigning a real value to

an integer variable induces rounding.

Array Expressions

Because of the importance of arrays in scientific computation, Blaze contains extensive array manipulation features. Given arrays A and B which have the same base-type, the same number of dimensions, and the same index range in each dimension, one can perform the assignment:

$$A := B;$$

Similarly, Blaze allows the pointwise arithmetic operations on arrays:

$$A + B \quad A - B \quad A*B \quad A/B$$

In each case the result is the array produced by performing the given scalar operation on each corresponding pair of elements of the arrays A and B. This definition agrees with normal mathematical usage for addition and subtraction while this type of array multiplication is more commonly used in picture processing.

In order to admit the usual mathematical definitions of array multiplication, without violating this convention, we introduce the pound sign as a separate kind of array multiplication. The product $A\#B$ is defined only when A and B are one or two dimensional arrays with mathematically appropriate size. When A and B are two dimensional, it yields the usual matrix product, while when A and B are both one dimensional, it yields the

vector inner product. If one array is one dimensional and the other two dimensional, it gives their matrix-vector product.

Given the emphasis here on array operations, it is natural to permit automatic type conversion from scalars to arrays. For one thing, this makes it trivial to assign a single value to all elements of an array. As examples of type conversion of scalars to arrays, consider the code fragment:

```
var     A,B,C     : array[1 .. N, 1 .. N] of real;
        x,y       : real;

        . . .

        A := x;

        C := B#C - A + 1.0;
```

In each case, the scalar value is converted to a conforming array with all elements set equal to the given scalar, before the arithmetic operation

Automatic conversion of scalars to arrays is strictly one-way. Assigning an array value to a scalar variable is an error, an array value may be assigned only to an array variable matching in size and dimension.

Operations on Records

Just as with arrays, Blaze allows assignment between record variables of the same type.

```
var new_particle, old_particle : charged_particle;

. . .

new_particle := old_particle;
```

Since the fields of a record can be of any type, the operations of the appropriate type extend to individual fields of a record.

Operations on Lists

As noted earlier, lists are just extensible one-dimensional arrays. Thus all the operators that are valid for arrays are also valid for lists. In addition, lists can be dynamically extended by using the concatenation operator **cat**, as shown here:

```
var new_plasma, old_plasma, plasma : list of charged_particle;

. . .

plasma      := plasma cat old_plasma;
new_plasma  := old_plasma[15 .. 30];
```

Here, the list *plasma* is replaced by a new list consisting of the original *plasma* concatenated to the list *old_plasma*. To shorten a list, one can use a subrange, as shown, just as one can form rectangular subarrays of an array. Note that the list *old_plasma* is completely unaffected by either of these operations.

Accumulation Operators

Blaze contains special accumulation operators, which allow one to "accumulate" values onto a single variable. Thus to sum the elements of an array the following statement can be included in a loop.

sigma **sum=** *x[i];*

This accumulation operator is analogous to the += operator in C. The effect of the operator used here is to add the value of $x[i]$ to the value of σ and store the result back in σ. Thus when used in a sequential loop, the above statement is equivalent to the following:

sigma := sigma + x[i],

The semantics of accumulation operators in **forall** loops will be discussed later.

The Table 3.1 provides a list of accumulation operators available in Blaze. Eventually, when abstract data types are added to the language, facilities for user-defined accumulation operators will be provided.

3.5. Sequential Control Structures

The sequential control structures here are **if** statements, **case** statements, and several kinds of loop constructs. There is also a parallel control structure, the **forall** statement, discussed in the next section.

multiplication/division	**mult**
addition/subtraction	**sum**
maximum	**max**
minimum	**min**
logical and	**and**
logical or	**or**
list concatenation	**cat**

Table 3.1 : List of Accumulation Operators

If Statements

The **if** statement in Blaze is identical to that in Modula. The basic forms of this statement are:

```
if <boolean_condition> then
    <statement_list>
end;
```

and also:

```
if <boolean_condition> then
    <statement_list>
else
    <statement_list>
end;
```

One can string **if** tests together here with an **elsif** construct:

```
if x > 3 then
    y := 100;

elsif x > 2 then
    y := 10;

elsif x > 0 and x <= 2 then
    y := 2;

else
    panic();

end;
```

Case Statement

The **case** statement can also be used for multiway branching. The statement selects one of a number of alternative sequences of statements for execution based on the value of the expression provided in the **case** header as shown below:

```
var particle : variant_charged_particle;

  . . .

case particle.species of

        when deuteron =>
                <statement_list>

        when alpha    =>
                <statement_list>

        when electron | proton | neutron =>
                <statement_list>

        when others   =>
                <statement_list>
end;
```

The expression is evaluated and then the statement list specified by the **when** clause matching the expression is executed. Multiple choices can be specified as in the **when** clause for electrons, protons, and neutrons. The choice **others** can be used as a default choice for all cases not explicitly covered. After the execution of the appropriate statement list, control transfers to the statement after the **case** statement.

Loops

Blaze contains several loop constructs including **for** and **while** loops. There is also a parallel **forall** loop, described in the next subsection. The simplest loop in Blaze has the form:

```
loop
   <statement_list>
end;
```

This type of loop would run forever unless one or more **exit** statements were included within the loop body. Thus one could, for example, write a loop of the form:

```
loop
   <statement_list>

   exit when <boolean_condition>;

   <statement_list>
end;
```

One may also omit the **when** clause in an **exit** statement leaving an unconditional **exit**. An **exit** statement causes the enclosing loop to be terminated with control being transferred to the statement after the loop statement. Thus in situations where loops have been nested, an **exit** statement will terminate only the closest enclosing loop.

The other kinds of sequential loops in Blaze are constructed by preceding a **loop-end** block with a controlling clause. The two alternatives are **while** loops,

```
while <boolean_condition> loop
    <statement_list>
end;
```

and **for** loops:

```
for <loop_index> in <range_specification> loop
    <statement_list>
end;
```

In either case, exit statements can be included within the loop.

The loop indices in **for** loops can be integers, user-defined enumerated types, or the elements of lists or one dimensional arrays. A typical example using integer indices is:

```
for i in 1 .. 100 loop
        for k in 50 .. -50 by -5 loop
                <statement_list>
        end;
end;
```

The default increment is one, although alternate increments can be used, as shown. One can also include a boolean **where** condition which selects a subset of the index range over which the loop will execute:

```
for i in 0 .. 2000 by 2 where test(i) loop
    <statement_list>
end;
```

The effect of a **where** clause can also be achieved by an **if** statement, but the syntax here appears quite natural.

There is one important semantic difference between the **for** loop here and that in C or Pascal. In Blaze, the index variable of a **for** loop is local to the loop and is, in effect, declared by the loop header. Thus in the following code fragment:

```
i := 17;
for i in 1 .. 10 do
  <statement_list>
end;
k := i;
```

the value assigned to k would be seventeen, since the loop index i is a different variable than the previously declared variable i. This approach avoids trivial loop index declarations and also eliminates the possibility of unintended side effects which may occur when loop indices retain their value outside loops.

Aside from the simple discrete type of loop indexing, Blaze also allows indexing over lists and one dimensional arrays. An example of this type of loop is:

```
var     x_array : array[0 .. 100] of real;
        sigma   : real;

        . . .

        sigma := 0.0;

        for x in x_array loop

            sigma sum= x**2;

        end;
```

This **for** loop would compute the sum of the elements in array *x_array*, and similarly one can index over the elements in a list. Note that the loop header serves as the declaration of the loop index, and in this case the loop index is implicitly typed **real** since the array *x_array* here is an array or reals. The **by** and the **where** clauses are also allowed in this type of **for** loop.

3.6. Procedures

A procedure in Blaze begins with a header declaring a list of zero or more input parameters and a list of zero or more output parameters. Next, the types of these formal parameters are declared in a **param** section following the header. Then, the constants, types, and variables local to the procedure are then declared, and finally one gives the body of the procedure.

As a simple example of this syntax, consider the following procedure, which sums two two-dimensional arrays:

```
procedure array_sum(x,y) returns: z;
param          x,y : array[ , ] of real;
               z   : like(x);
begin
   forall i in range(x[*, ]) do
      forall j in range(x[ ,*]) do
         z[i,j] := x[i,j] + y[i,j];
      end;
   end;
end;
```

The size of input arrays need not be specified, though the base types and number of dimensions must be declared. For output arrays, one must declare the size as well. In this example, the built-in primitive **like** is used to create an array of the same size and base type as one of the input arrays. Other primitives available here are **lower**, **upper** and **range**, which give the lower bound, the upper bound and the range of a specified array variable as already explained.

Procedure Invocation

As noted earlier, procedures in Blaze subsume the role played by functions in languages like Pascal, Ada, and Modula. The example procedure *array_sum*, described above, could be invoked in the assignment statement:

a := array_sum(a,b);

This procedure has two input arrays and produces one array as output. Notice that there is no requirement that inputs and outputs be distinct, as here, where *a* occurs in both the input and output lists. Procedures, such as this, which return only one value, can be used in expressions, even if the value returned is an aggregate such as an array or record. It is only necessary that the type of the output is appropriate in the context of that expression.

One can also define procedures with more than one output value, but these cannot be invoked in expressions. An example of such a procedure is:

Ex, Ey := gradient (phi),

Procedure with no input parameters or no output parameters also make sense here, as in the procedure invocations:

write_arrays (x);

y := read_array(),

These procedures could be user defined procedures performing input and output on arrays.

Blaze is a strongly typed language, and thus the types of the input and output parameters in the procedure invocation must agree with the formal parameters in the procedure declaration. In the example above,

Ex, Ey := gradient (phi)

the values of the first output parameter of *gradient* is assigned to *Ex*, and the second is assigned to *Ey*. The types of these parameters must match with the types of *Ex* and *Ey* respectively. Most of this type checking is done at compile time, but checking that array sizes match can, in general, be done only at run-time. If the outputs of *gradient* were arrays, and the sizes of the output arrays produced did not match those of *Ex* and *Ey*, a run-time exception would be raised.

3.7. Parallel Constructs

The control constructs covered so far are similar to those in other structured programming languages such as Pascal and Ada. Blaze also contains an explicitly parallel construct, the **forall** loop. Syntactically a **forall** loop is similar to a **for** loop, as shown below:

forall *i* **in** *1 . 100* **do**

<statement_list>

end;

The **forall** header specifies an index variable and a range of values for this variable The body of the **forall** loop is the list of statements following the header. A copy of the body is invoked in parallel, for all values specified in the range (hence the name **forall**). In the above example, all one hundred invocations of the body would be performed concurrently.

The header of a **forall** loop is syntactically the same as the header of **for** loop. In particular, **by** and **where** clause are allowed in forall loops, and one can use elements of lists and one dimensional arrays as indices. A major difference between **for** loops and **forall** loops is that **forall** loop can contain variable declarations. This declares variables local to the loop body, with a separate copy of the variable for each loop invocation. For example, in the loop,

```
forall i in 1 .. 100 do
   var k  : integer;
            k := some_procedure (i)
            x[k] :=  y[k+1]
            . . .
end;
```

each loop invocation has its own copy of k, and there is no communication between loops.

By using a **forall** loop instead of a **for** loop, the programmer is asserting that the loop invocations are to be executed concurrently. Consider, for example, the following **forall** loop:

```
forall i in 1 .. 100 do
    x[ f(i) ] := x[i];
end;
```

If the procedure f generates a permutation, then each invocation of the body

would assign a value to a different element of array x, satisfying the criterion of independence. On the other hand, if the procedure f is such that it returns the same value for the two different i values, then two invocations of the loop body would assign to the same element of x. As a compiler option such occurrence could be ignored or could raise a run time exception. In the latter case, one can elect to have a warning message printed out or to halt execution.

Each of the loop invocations in a **forall** loop is analogous to a procedure call, in the sense that it has similar copy-in copy-out semantics. In the above example, the values of x accessed on the right hand side of the assignment are the old values of the x array regardless of the order in which the loop invocations may be executed. Thus the array x is, in effect, 'copied into' each invocation of the **forall** loop, and then the changes to x are 'copied out' and used to modify x after the execution of all loop invocations.

Accumulation operators can be used to obtain information from all invocations of a **forall** loop. For example, in the following loop,

```
var x : real;
        x := 0.0;
        forall i in 1 .. 100 do
            x sum= y[i];
        end;
```

the values in the array y are summed across the loop invocations. This does not raise a run-time exception, though direct assignment to x would have.

The order in which the associative accumulation operations are performed is not specified as part of the language definition. Thus, given sufficiently many processors, this accumulation can be done with a fan-in tree in logarithmic time.

3.8. Input-Output

The current version of Blaze provides a simple set of input-output facilities. Blaze procedures may read and write only to the standard input and output files. This is done with **read** and **write** procedure calls, which are syntactically similar to the 'scanf' and 'printf' statements of C.

variable_list := **read** (*"format_string"*);

write (*"format_string"*, *variable_list*);

The **read** procedure requires a format string as parameter (as in C) and returns the values read from standard input. These values are then assigned to the variables on the left side of the statement. The number of values and their types are determined via the format string. Similarly, for the **write** statement, the *format_string* specifies the format in which the values of the variables are to be printed. The format notations used are the same as those used in C.

There are a number of subtle difficulties in providing input-output facilities in parallel environments. At the language level, input and output are side-effects and thus contrary to the spirit of functional procedure invocation here.

There is also a determinacy question. For example, what is the order of I/O in **forall** loops? Finally, there are several problems that arise during implementation.

The determinacy question is treated here by observing that all control constructs in Blaze can be viewed as having an implied sequential order. For example, in the code fragment

$y := f(x);$

$z := h(g(x));$

the implied sequential order is that f is executed first, then g, and finally h. Though the actual order of execution might differ from this, all input and output would be done as if this sequential order had been followed at run-time. Similarly in **forall** loops, we take the implied sequential order as being that the loop invocations occur in sequence, with each invocation completing before the next begins, exactly as with **for** loops.

Good input-output facilities are critical to a language's usability, especially during initial program development. For this reason, the input-output facilities were included here, though they raise severe implementation difficulties. These difficulties are greatest in handling input. For output, each procedure creates blocks of output data, which get concatenated together to generate the output stream. This operation is highly parallel, and at least conceptually straightforward. Treating input well is more subtle, and in the worst case **read** statements may sequentialize program execution.

3.9. Status of the Language

The features described here characterize the Blaze language, as it currently exists. However, as experience is gained in the use of this language, it will necessarily evolve. Some of this evolution is predictable, such as inclusion of features like separate compilation units, exception handlers, and abstract data types. Inclusion of more flexible data structures than lists and of general file input-output would also be desirable, but raises complicated research issues.

The most interesting language question here is whether the model of computation embedded in this language is an adequate model of parallel computation. It is our view that determinate execution is essential, if one wishes parallel computation to be easy to describe and understand. However, by restricting the language to determinate execution, we have restricted the class of algorithms which can be described in this language. There is, for example, no means of expressing the asynchronous relaxation algorithm of Baudet[4], in Blaze. As experience in parallel computation grows, it may be necessary to widen the computational model here, allowing a certain amount of carefully controlled indeterminacy in the language.

4. A Plasma Simulation Example

One way to assess the relative merits of a programming language is to look at examples. In this section we consider a comparatively extensive numerical program, a plasma simulation code based on the particle-in-cell method. This type of simulation program is routinely used to model plasmas for controlled nuclear fusion, and similar programs are used to study the motions of stars and galaxies[7]. This program makes an interesting example here, since it contains two distinct phases of computation, one devoted to numerical linear algebra, and the other devoted to data structure manipulation. Moreover, the kinds of data structures and operations in this example are quite typical of those required in many large scientific programs.

4.1. Description of Particle-in-Cell Program

A plasma is a high temperature gas consisting of free electrons and positively ionized molecules. The motion of the particles (electrons and positive ions) is governed by Newton's laws of motion. For simplicity, the only forces we consider here are the electrostatic forces between the charged particles.

The particle-in-cell computation consists of a sequence of time steps, each time step composed of two basic phases: a *field computation* phase, and a *particle push* phase. In the *field computation* phase, given the plasma charge distribution, one solves a set of finite difference equations to determine the global electric field. In the *particle push* phase, given the global electrical field, one computes the force on each particle, integrates the particle motions over a small time interval, and recomputes the global charge distribution. Both phases

involve numerical operations, but the *particle push* phase also involves simple data structure operations, demonstrating the utility of Blaze in this arena.

There are a variety of data structures which can be used to keep track of the collection of particles in the particle-in-cell program. One of these is a two dimensional array representing a grid of rectangular cells. Associated with each cell is the list of all particles currently in that cell. Type declarations for this data structure are given in Listing 4.1. This particular data structure is especially useful if one wishes to extract information on particle collision probabilities or collision velocities, as one might in controlled fusion studies.

```
—          type declarations for particles          —

const nx = 64;
      ny = 64;

type ion_species = (electron, proton, deuteron, alpha);

      charged_particle = record
        xpos, ypos : real;    — position of particle
        xvel, yvel : real;    — velocity components of particle
        charge     : real;    — electrical charge of particle
        mass       : real;
        species    : ion_species;
      end;

      ion_list = list of charged_particle;

      plasma_cloud = array[0 .. nx, 0 .. ny] of ion_list;
```

Listing 4.1 Data Types for Charged Particles

```
—                main program for PIC computation
```

```
program particle_in_cell;

const tol      = 1.0e-3;
      max_time = 1000;

var   phi,sigma,Ex,Ey : array[0 .. nx,0 .. ny] of real;
      plasma          : plasma_cloud;

begin

   plasma := create_plasma;                    — generate particles, setting their
                                               — intial positions and velocities and

   for time in 0 .. max_time loop

      if (time % 25 = 0) then                  — perform output of current plasma
         graphics_output(plasma);              — configuration every 25 time steps
      end;


      sigma := charge_dist(plasma);            — compute charge distribution
                                               — by summing over all particles

      phi := multigrid(phi,sigma,tol);         — solve Poisson's equation for
                                               — the electric potential phi


      Ex,Ey := gradient(phi);                  — take the gradient of phi to
                                               — compute the electric field

      plasma := part_push(plasma,Ex,Ey);       — now advance the particles
                                               — along their trajectories

   end;
end;                          —— particle–in–cell main program ——
```

Listing 4.2 Main Procedure for Particle-in-Cell Program

Given these type declarations, one can program the main procedure of the *particle–in–cell* program as shown in Listing 4.2. One of the interesting things to note here is that this procedure would have been relatively easy to understand even if the comments had been omitted. The applicative procedure calling mechanism, shared by Blaze and the data flow languages, seems to go a long way toward clarifying code. For example, procedure *gradient* takes array ϕ as input and has as output arrays *Ex* and *Ey*. This would be apparent, even if one did not have the slightest conception of the purpose of this procedure. By contrast, in languages like Ada and Pascal, one would need to find the procedure header to determine which parameters were altered by a procedure, while in Fortran, the entire body of the subroutine would have to be scanned to determine this.

4.2. Field Computation Phase

We look next at the programming of the *field computation* phase, a routine numerical computation. The electro-static force on all particles in the plasma can be easily determined, if we know the electric potential $phi(x,y)$, which is given by the Poisson equation

$$\Delta \phi = \sigma,$$

where $\sigma(x,y)$ is the spatial charge density. This equation is one of the simplest partial differential equations. Using finite difference techniques, one can convert this equation to an analogous numerical linear algebra problem

$$A \phi = \sigma,$$

where A is a spare matrix, σ is now a known vector representing the charge distribution, and ϕ is now an unknown vector corresponding to the electric potential.

One of the best methods for solving this linear system is the multigrid method. A multigrid algorithm is generally programmed as a driver procedure, which treats storage allocation and overall control flow, and a small number of kernel procedures, which perform required numerical operations. As an example of such a kernel procedure, we consider procedure *project*, given in Listing 4.3. It takes as input an array corresponding to a given grid, and produces smaller array having about one fourth as many values, corresponding to a coarser grid

This procedure, written in Blaze, differs little from what it would look like in most other imperative high level languages, and is practically a verbatim translation of the corresponding Fortran code. Now suppose we have a family of numerical kernel procedures, such as the *project* procedure just given. It is an easy task in Blaze to combine these kernel procedures into a multigrid Poisson solver. The driver procedure *multigrid* (Listing 4.4) does just that.

The kernel procedure *smooth* here performs a simple "point" iteration, which would alone suffice to solve the linear system, but would be inefficient. Instead, after a few iterations, we recursively call procedure *multigrid*, if possible, to accelerate the iterative solution process. This process is repeated until the error tolerance is met.

```
procedure project(res)  returns b;

param res : array[ , ] of real;
      b   : array[0 .. mc,0 .. nc] of real;

const m  = upper(res[*, ]);        mc = m/2;
      n  = upper(res[ ,*]);        nc = n/2;

      c0 = 0.25 ;  c1 = 0.5 ;   c2 = 1.0;

begin

   b := 0;                          — initialize array b to zero

   forall ic in 1  .  mc–1 do        — loop over all interior
      forall jc in 1  ..  nc–1 do    — points of array b

         var  im,i,ip : integer;
              jm,j,jp : integer;

         i  := 2*ic;       j  := 2*jc;
         ip := i + 1;      jp := j + 1;
         im := i – 1;      jm := j – 1;

         —— compute value at an interior coarse grid point as a
         —— weighted average of nine corresponding fine grid values

         b[ic,jc] := c0 * res[im,jp]  +  c1 * res[i,jp]  +  c0 * res[ip,jp]
                   +  c1 * res[im, j]  +  c2 * res[i, j]  +  c1 * res[ip, j]
                   +  c0 * res[im,jm]  +  c1 * res[i,jm]  +  c0 * res[ip,jm];
      end;
   end;

end;   —— procedure project ——
```

Listing 4.3 Multigrid Kernel Routine Project

```
procedure multigrid(phi,sigma,tol) returns phi;

param phi,sigma : array[ , ] of real;
      tol       : real;

const m = upper(phi[*, ]);        mc = m/2;
      n = upper(phi[ ,*]);        nc = n/2;

      n_iter = 4;

var   v,b : array[0   mc,0 .. nc] of real;
      res : array[0 .. m, 0 .. n ] of real;

begin

  loop          ———————— begin outer loop ————————

    phi := smooth(phi,sigma,n_iter);        — perform n_iter iterations

    res := residual(phi,sigma);             — compute residual and test
    exit when( square_norm(res) < tol );    — for adequate convergence

    if ((m > 1) and (n > 1)) then           — test grid size

      b   := project(res);                  — interpolate residual

      v   := 0;                                  — solve coarse grid problem
      phi := phi + inject(multigrid(v,b,tol)); — and correct phi

    end;

  end;
end;    ———————— procedure multigrid ————————
```

Listing 4.4 Driver for Multigrid Poisson Solver

4.3. Particle Push Phase

Unlike the first phase of the particle-in-cell program, which dealt mainly with arrays, the second phase deals with data structures representing collections

of charged particles. As shown in Listing 4.5, the particle push phase may be written as calls to two kernel procedures. Procedure *advance* performs numerical computation, updating the particle's velocities and spatial positions, while procedure *reshuffle* realigns the data structure. Listing 4.6 gives the code for procedure *reshuffle*. Note that the accumulation operator, **cat=**, allows natural expression of the parallelism here.

5. Implementation

The extensive example program just described shows that Blaze is about as expressive and natural for this type of programming as any current language. This is not surprising, since the sequential constructs in Blaze have been strongly influenced by these languages. The more important issue here is compilation for parallel architectures, which is the central goal of the Blaze

---

```
procedure part_push(plasma,Ex,Ey) returns plasma;

param     plasma    : array[ , ] of ion_list;
          Ex,Ey     : array[ , ] of real;

begin
                                  — advance particles numerically, updating
  plasma := advance(plasma,Ex,Ey); — all positions and velocity fields

                                  — move records of particles that have
  plasma := reshuffle(plasma);    — moved out of their mesh cells, placing
                                  — them in their new cells list
end;
```

Listing 4.5 Procedure for Particle Push with Grid Data Structure

```
procedure reshuffle(plasma) returns: new_plasma;

param       plasma     : array[ , ] of ion_list;
            new_plasma : like(plasma);

const nx = upper( plasma[*, ] );
      ny = upper( plasma[ ,*] );

begin

  forall i in 0 .. nx do
    forall j in 0 .. ny do

      forall ion in range(output_plasma[ ,*]) do

        var inew,jnew : integer;

        inew := round( nx*ion.xpos);
        jnew := round( ny*ion.ypos);

        new_plasma[inew,jnew]  cat=  ion;

      end;

    end;
  end;

end;  —— procedure reshuffle ——
```

Listing 4.6 Procedure to Maintain Plasma Data Structure

language. The Blaze programming language is intended as a scientific programming language for parallel computers. Thus the question of how well Blaze programs will execute on parallel computers is critical. This section considers some of the research issues arising in mapping Blaze programs to parallel run-time environments.

5.1. Structure of the Compiler

The structure of compilers for Blaze is dictated by our desire to target this language to a number of sequential and parallel architectures, and by the necessity of performing extensive optimization during compilation. Features such as the functional procedure invocations and **forall** statements simplify compiler optimization and permit the restructuring transformations required for multiprocessor architectures. However, these same features will lead to inefficient execution if compiler optimization is omitted. Thus optimization during compilation is essential here.

The general structure of Blaze compilers is shown in Figure 5.1. The lexical analysis, parsing, and first few phases of optimization can be performed

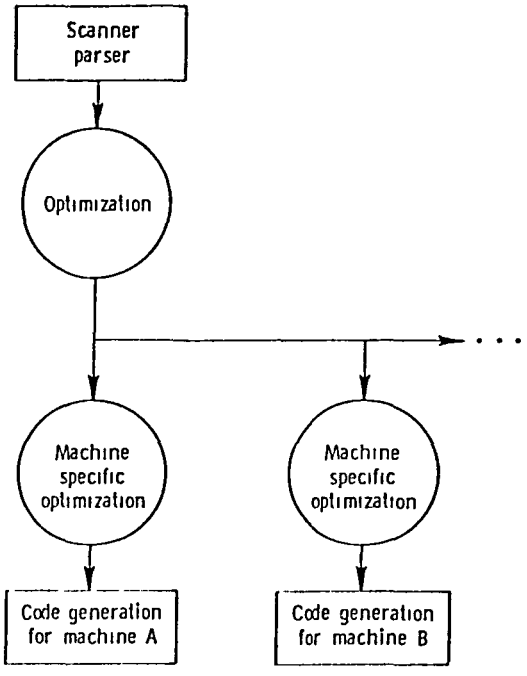

Figure 5.1 General Structure of Compiler

in a machine independent compiler front end, as shown. After this, further optimization and code generation is performed in machine specific compiler back-ends.

The intermediate form used in the compiler front end is a type of control flow graph. Simple examples of this intermediate form are shown in Figure 5.2. A variety of types of data flow analyses and optimizations can be performed on this intermediate form, including use-definition and definition-use chaining, live variable analysis, and dominator and post-dominator computation.

The absence of "goto" statements and the type of exit statement employed imply that the flow graphs here will be reducible[2]. In consequence, data flow analysis here is extremely fast. More importantly, because of the functional procedure calls, data flow analysis here generates precise information. Even when complete inter-procedural data flow analysis is performed for conventional languages, the resulting information is imprecise, because language features like pointers and common blocks often obscure data flow information.

5.2. Sequential Computers

Implementation of Blaze on sequential computers is not especially difficult, since Blaze contains a relatively modest set of features. In most respects its implementation is similar to that of Pascal, C, and similar languages supporting static type checking and stack based run-time environments, though there are important differences. In Blaze, as in Algol 60, the size of arrays can be determined, in general, only at run time. This, together with some aspects of the parameter passing mechanism, complicates stack allocation here. The run-

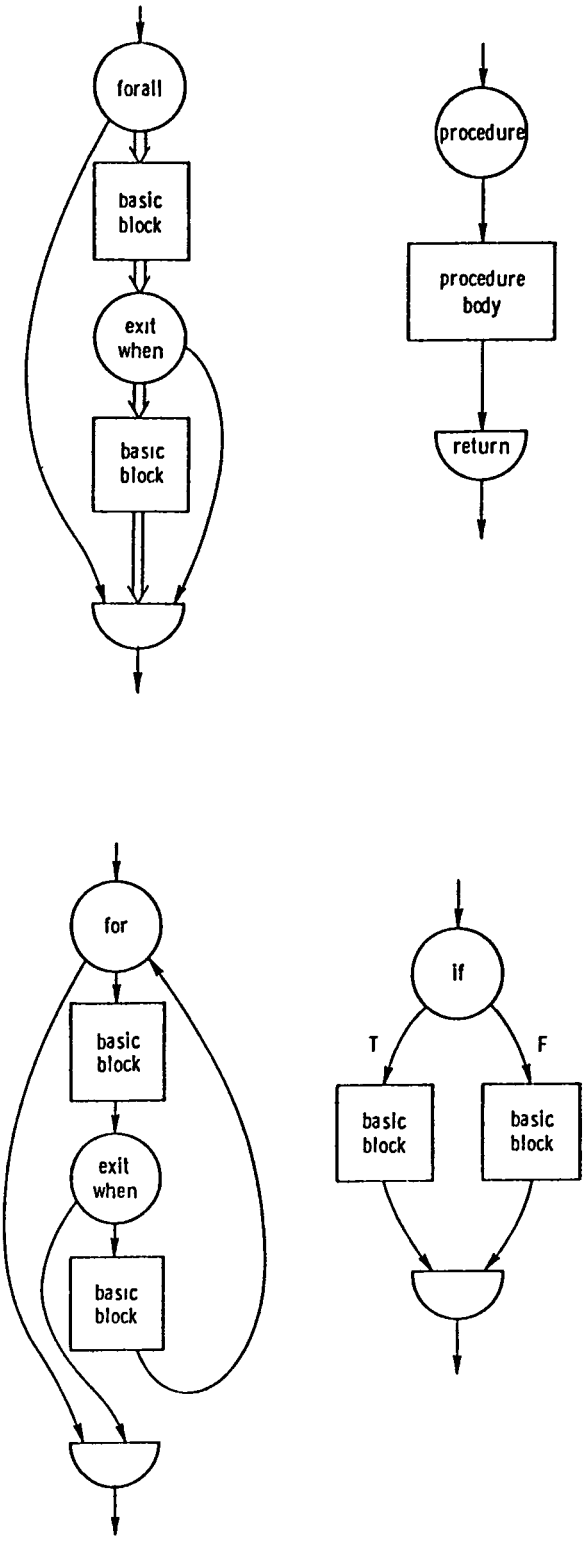

Figure 5.2 Intermediate Form

time environment is also complicated by the list data type, which requires heap allocation and linked list access structures.

The most critical issues in sequential environments are parameter transmission and storage allocation. Copy-in, copy-out semantics here has the well known disadvantage of requiring time consuming copying of arrays and record structures. These disadvantages, especially important on sequential computers, can be minimized through careful design of the compiler and run-time environment.

Our basic approach to this problem is to maintain conceptually copy-in copy-out parameter transmission, while letting the compiler substitute "by-reference" parameter transmission, whenever the effect seen by the user would be identical. Transforming from copy-in, copy-out parameter transmission to "by-reference" transmission is relatively straight-forward here, partly because the functional procedure calls eliminate most of the problems with aliasing. This transformation can be done using a system of run-time flags, which pass the data dependency context surrounding the procedure call. In the majority of cases, we should be able to avoid copying of arrays, lists, and records.

5.3. Parallel Computers

The Blaze language is designed to facilitate the compiler transformations needed to map sequential languages to parallel architectures. This subsection sketches the most critical implementation issues involved in mapping Blaze to some of the anticipated target architectures.

From the point of view of vector processors, Blaze is much like the proposed new Fortran, Fortran 8x. In both Blaze and Fortran 8x there are features to express low level parallelism, such as array arithmetic and **forall** statements. These features greatly enhance the compiler's ability to exploit vector hardware. For example, the Blaze array assignment

$$A := B;$$

will yield much better executable code than the corresponding nested do loops in Fortran 77. However, the usual vectorization issues, such as loop restructuring, treatment of "if" statements, and treatment of recurrences, still remain and must be dealt with, since the explicit parallel constructs available in Blaze and in other languages are not expressive enough for all programming tasks.

A multiprocessor is a parallel architecture in which there are a number of processors, each executing its own instruction stream. In restructuring Blaze programs for execution on a multiprocessor, our principal goal is to exploit the fine grained parallelism within loops. Scientific programs contain a variety of kinds of loops, which are often nested several levels deep and can be quite complex. In the first Blaze compiler for a multiprocessor, the kinds of "loop" parallelism we are attempting to exploit are:

vector parallelism

parallel loops containing indirect addressing
(i.e. scatter/gather operations)

loops containing accumulation operators

loops containing procedure calls

One possible scheme for memory allocation here is to assume that there is a control processor, with the other processors working in a quasi-SIMD mode, as slaves to this controller. Large data structures, such as arrays and streams, would be distributed across memory, with each processor having conflict free access to its own slice of the data structure.

To allow procedure invocation within loops, the controller can free the "slave" processors to run in MIMD-mode, each allocating storage on its own private activation stack. Though this is a simple idea, and exploiting this type of parallelism is obviously important, tricky load balancing issues may arise if the execution times of the procedures invoked vary widely.

Blaze differs from the data flow languages primarily in not using the single assignment rule, in which a variable name can occur on the left side of an assignment only once. However, from the point of view of the compiler writer, this is a minor issue, since it is trivial to achieve the effect of the single assignment rule via variable renaming. Thus it should be easy to implement Blaze on dynamic data flow architectures, though the use of recursion and dynamic arrays here prohibits its use on static data flow machines.

6. Summary

The programming language Blaze is one response to the important problem of providing software interfaces to parallel computer architectures. With Blaze, responsibility for using parallel architectures falls equally on programmers, compiler writers, and computer architects. The programmer is given the responsibility of creating fast parallel algorithms and must also make the slight adjustment of programming them in an unfamiliar language. The compiler writer must design optimizing compilers to restructure programs so they will execute efficiently on parallel architectures. And finally the computer architect has the responsibility of constructing parallel architectures which are sufficiently elegant and simple that the compiler writer's task is tractable. This seems an equatable distribution of responsibility and should give this enterprise a reasonable chance of success.

Several characteristics of the Blaze language distinguish it from competing parallel languages. First, despite the significant changes in semantics entailed by the use of functional procedure invocation, Blaze is syntactically close to Pascal and similar conventional languages. Programmers are not faced with the prospect of learning to create complex task systems, or with learning to write programs using the single-assignment rule. Global variables are absent here, but good programmers often try to avoid them anyway.

A second characteristic distinguishing Blaze, especially from the data flow languages, is that in Blaze the user retains substantial control over memory allocation. An optimizing compiler may create copies of a data structure when

this is necessary for parallel execution, but replication of data structures is done parsimoniously. This is in contrast to the situation with data flow languages, where the programmer relinquishes all control of memory allocation and provides a new name for each new value, leaving it entirely to the compiler to discover cases where differently named objects can reuse the same storage.

The powerful **forall** statement and accumulation operators also distinguish Blaze from alternate parallel languages. Like the data flow languages, Blaze yields determinate execution. However this is achieved here by run-time checks on the execution, rather than by a restricted syntax which prevents one from writing indeterminate programs. Our approach gives the programmer considerably more flexibility, though it does complicate the run time environment substantially.

Despite these differences between Blaze and other parallel languages, there is a great deal in common as well. In particular, Blaze shares many features with Sisal and VAL. As with these data flow languages, Blaze code tends to be naturally short and elegant, due in part to the Pascal-style data structures here, and in part the functional procedure invocation semantics. Functional procedure invocation is clearly conducive to a clean and understandable programming style.

Like the data flow languages, Blaze gives the user sequential control flow and determinate execution. This feature sharply delineates Blaze from multi-tasking languages like Ada and provides its central attraction. We share with the data flow community the view that determinate execution is the central

issue in making parallel architectures readily usable by non-expert programmers.

7. References

1. *Reference Manual for the Ada Programming Language*, U. S. Department of Defense, Draft Revised MIL-STD 1815 (1982).
2. Aho, A. V. and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley Pub. Co., Reading Mass. (1977).
3. Allen, J. R., K Kennedy, C. Porterfield, and J. Warren, "Conversion of Control Dependence to Data Dependence," *Conference Record of Tenth Annual ACM Symposium on Principles of Programming Languages*, (January 1983).
4. Baudet, G., "Asynchronous Iterative Methods for Multiprocessors," *Journal of the ACM* **25** pp. 226-244 (1978).
5. Ferrante, J, K. Ottenstein, and J. Warren, "The Program Dependence Graph and Its Uses in Optimization," IBM Technical Report RC 10208 (August 1983)
6. Gordon, M. J, A. J. Milner, and C. P. Wadsworth, "Edinburgh LCF: A mechanised Logic of Computation," in *Lecture Notes in Computer Science 78*, ed J. Hartmanis,Springer-Verlag, NY (1979).
7. Hockney, R. W. and J. W. Eastwood, *Computer Simulation Using Particles*, McGraw-Hill, New York (1981).
8. Kennedy, K., "Automatic Translation of Fortran Programs to Vector Form," Rice Technical Report 476-029-4, Rice University (October 1980.).
9. Kessels, J. L. W., "A Conceptual Framework for a Nonprocedural Programming Language," *Communications of the ACM* **20** (12) pp. 906-913 (1977)
10. Kuck, D. J., *The Structure of Computers and Computation, Volume I*, John Wiley and Sons, New York (1978).
11. Kuck, D. J., R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Conference Record of Eighth Annual ACM Symposium on Principles of Programming Languages*, (January 1981).
12. Milner, R., "A Theory of Type Polymorphism in Programming," *J. Computer and System Sciences*, (17) pp. 348-375 (1978).
13. Milner, R., "A Proposal for Standard ML," Report CSR-157-83, Dept. of Computer Science, Univ. of Edinburgh (1983).
14. Padua, D. A., D. J. Kuck, and D. H. Lawrie, "High Speed Multiprocessing and Compilation Techniques," *Special Issue on Parallel Processing, IEEE Transactions on Computers* **C-29** (9) pp. 763-776 (September 1980)

15. Wagener, J. L., "Status of Work Toward Revision of Programming Language Fortran," *SIGNUM Newsletter* **19** (3)(July 1984).

16. Warren, J., "A hierarchical basis for reordering transformations," *Conference Record of Tenth Annual ACM Symposium on Principles of Programming Languages*, (January 1983).

1 Report No NASA CR-172615 ICASE Report No. 85-29	2 Government Accession No	3 Recipient's Catalog No
4 Title and Subtitle The BLAZE Language: A Parallel Language for Scientific Programming		5 Report Date May 1985 6 Performing Organization Code
7 Author(s) Piyush Mehrotra and John Van Rosendale		8 Performing Organization Report No 85-29 10 Work Unit No
9 Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665		11 Contract or Grant No NAS1-17070; NAS1-17130 13 Type of Report and Period Covered Contractor Report
12 Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		14 Sponsoring Agency Code 505-31-83-01
15 Supplementary Notes Langley Technical Monitor: J. C. South, Jr. Final Report		Submitted to Parallel Computing

16 Abstract

Programming multiprocessor parallel architectures is a complex task. This paper describes a Pascal-like scientific programming language, Blaze, designed to simplify this task. Blaze contains array arithmetic, "forall" loops, and APL-style accumulation operators, which allow natural expression of fine grained parallelism. It also employs an applicative or functional procedure invocation mechanism, which makes it easy for compilers to extract coarse grained parallelism using machine specific program restructuring. Thus Blaze should allow one to achieve highly parallel execution on multiprocessor architectures, while still providing the user with conceptually sequential control flow.

A central goal in the design of Blaze is portability across a broad range of parallel architectures. The multiple levels of parallelism present in Blaze code, in principle, allows a compiler to extract the types of parallelism appropriate for the given architecture, while neglecting the remainder. This paper describes the features of Blaze, and shows how this language would be used in typical scientific programming.

17 Key Words (Suggested by Author(s)) parallel programming, applicative languages	18 Distribution Statement 61 - Computer Programming & Software Unclassified - Unlimited		
19 Security Classif (of this report) Unclassified	20 Security Classif (of this page) Unclassified	21 No of Pages 57	22 Price A04

N-305

For sale by the National Technical Information Service, Springfield, Virginia 22161

End of Document