Parallel Computing 5 (1987) 339–361
North-Holland

The BLAZE language: A parallel language for scientific programming *

Piyush MEHROTRA
Department of Computer Science, Purdue University, West Lafayette, IN 47907, U.S.A.

John VAN ROSENDALE
Department of Computer Science, University of Utah, Salt Lake City, UT 84112, U.S.A.

Received October 1985

Abstract. Programming multiprocessor parallel architectures is a complex task. This paper describes a block-structured 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, allow 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.

Keywords. The BLAZE language, parallel programming language, multiprocessors, MIMD architectures, restructuring of conventional sequential languages, compiler.

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. In particular, designing concurrent programs having multiple threads of control flow has proven remarkably subtle. With parallel computing becoming the standard approach for 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.

* 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, U.S.A.

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 [2] and FORTRAN 8x [25]. BLAZE is also, to a lesser extent, intended as a general purpose programming language, and contains 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 and recursive records here provide some of the same expressive power and is often much simpler to use.

BLAZE is a parallel language but not a multi-tasking language. That is, with the exception of the 'forall' statement, 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 intension 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. 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 [16] and in the Edinburgh language ML [22], 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 [13] or MODULA [27]. 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 reflect 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 developments 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 the language 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 2 discusses the alternatives for specifying parallel algorithms and where BLAZE fits in the spectrum of available choices. Section 3 describes the principal features of BLAZE. Following that, Section 4 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. Alternative approaches to parallel programming

There are 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 Pure LISP [19], APL [9], FP [7], and the data flow languages SISAL [21], VAL [20], and ID [6]. Another alternative is the use of explicit multi-tasking languages such as CSP [11], ADA [2] or HEP-FORTRAN [3]. Finally, one can use a conventional language such as FORTRAN [1] or PASCAL [13], relying on the compiler to extract parallelism.

The distinction between these approaches is largely a matter of how the responsibility for exploiting parallelism is shared between the user, the compiler, and the run-time environment. With a vector language, or an explicit multi-tasking language, such as ADA, 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.

The most obvious approach to programming multiprocessor architectures is the use of an explicit multi-tasking language, such as ADA or MODULA. Such languages reflect the parallel run-time environment very strongly. A major disadvantage with these languages 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. Programs containing timing sensitive errors are remarkably difficult to correct.

The alternate approach of using a conventional language, such as FORTRAN, with compilers that automatically restructure programs for parallel execution has substantial advantages. This approach should allow one to transport the large body of existing programs to parallel architectures without change. Partly for this reason, a substantial research effort is being devoted to the development of such compilers. Some of the major efforts are at University of Illinois by D. Kuck and colleagues [17,18,24], at Rice University by K. Kennedy and colleagues [5,14] as well as at IBM Yorktown Heights and Michigan Technical University [10,26]. The main difficulty with this approach 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, aliasing effects occurring in FORTRAN obscure data dependencies and thus severely limit the compiler's ability to extract parallelism.

The approach of using a data flow language such as SISAL [21], VAL [20] or ID [6] is intermediate between the approach of using an explicit multi-tasking language and the alternate approach of using a conventional language with compilers that automatically extract parallelism. With a data flow language 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.

BLAZE is similar in concept to the data flow languages, reflecting the view that the most rational approach is for programmers to meet compiler writers halfway, with languages reflecting the needs of both. BLAZE uses functional procedure invocation both because the functional programming style appears to have clear benefits to the programmer, and because it makes extraction of procedure level parallelism much easier.

At the statement level BLAZE differs from data flow languages in that it uses traditional imperative semantics, rather than the single assignment rule. That is, a given variable can appear on the left-hand side of many assignment statements in BLAZE, while it must not appear more than once on the left-hand side of assignment statements in data flow languages. The use of imperative semantics within procedures makes BLAZE programming seem natural to programmers accustomed to conventional languages. Also, and perhaps surprisingly, this imperative semantics does not hamper compilation of BLAZE for either multiprocessors architectures or data flow machines. In fact there are a number of cases where the single assignment rule hampers effective compilation. For example, it is quite difficult to discover cases in data flow programs where storage for an array variable can be reused and the array 'updated in place'.

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 structures 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. 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. 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 [15] language. This is natural, since procedures in BLAZE cannot access non-local variables, so the concept of procedure nesting is not meaningful here. In a planned future extension, BLAZE will incorporate separate compilation units and some form of abstract data type, 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. Thus a constant declared in a procedure will retain the same value for the duration of the procedure invocation. 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 expression involving procedure calls, and passed parameters.

3.2. Structured data types

BLAZE contains extensive facilities for constructing structured data types. The structured types available here are arrays, lists, records, enumerated types 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.

3.2.1. 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:

```
type  vector = array[1..N] of real;
      bit_array = array[1..512, 1..512] of boolean;
var   table   : array[1..N, 1..N] of integer;
      u, v, w: vector;
      b0, b1  : bit_array;
```

In BLAZE, one can access either single elements of arrays or rectangular subarrays. Rectangular subarrays can be accessed by using an index range or by using an asterisk to specify the entire range of an index:

```
u[i]       - the ith element of u,
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.

As in ADA and MODULA, 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($b0$, 1)	– the upper limit of the first index of array $b0$,
lower($b0$, 2)	– the lower limit of the second index of array $b0$.

3.2.2. Lists

Array size in BLAZE are determined at the beginning of execution of the procedure in which they are declared and are fixed for the duration of the procedure in which they are declared. BLAZE also provides a one-dimensional array, called a list, whose size can vary dynamically. The syntax for list declarations follows that of arrays:

```
var string: list of char;
```

The lower index limit of a list variable is always one, while the upper limit can change dynamically. For example, one can concatenate a list onto the end of another list. The elements of a list are accessed exactly as one accesses the elements of a one-dimensional array and one can use the built-in primitive upper to determine the size of a list at any time.

Lists and one-dimensional arrays are closely related data structures. They are distinguished in BLAZE largely for practical reasons. Though there is a clear need for dynamic data structures, as provided by lists, the overhead can be quite high. In cases where dynamic changes of size do not occur, arrays provide equivalent features at significantly lower cost.

3.3. Records

Arrays and lists are homogeneous collections of elements, while record structures allow the programmer to group heterogeneous elements in a single structure. For example, the following record structure can be used to specify the properties of a charged particle:

```
type ion_species = (electron, proton, neutron, deuteron, alpha);
     charged_particle = record
       xpos, ypos: real;
       charge    : integer;
       case variant_tag: ion_species of
         when alpha =>
           ionization_level: integer;
         when electron | proton =>
           cross_section: real;
         when others =>
           lifetime: real;
       end;
     end;
```

The fields in a record may be of any previously declared type, and in particular may be arrays, record, or lists. BLAZE also allows tagged variant records, as shown above. Access to fields in records follows standard ADA syntax.

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. BLAZE also allows recursive variant records. The following example provides a recursive definition of a binary tree:

```
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;
      end;
```

Recursion is also allowed via lists:

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

The above 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 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 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.

3.4.1. 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. An extensive set of built-in procedures for conversions is also provided.

3.4.2. 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 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      : real;
    ...
    A := B + x;
    C := B#C - A + 1.0;
```

In each of these cases, the scalar value is converted to a conforming array before the arithmetic operation is performed.

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.

3.4.3. Operations on lists

As noted earlier, lists are just dynamic 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.

3.4.4. 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 *sigma* and store the result back in *sigma*. Thus when used in a sequential loop, the above statement is equivalent to the following:

```
sigma := sigma + x[i];
```

Table 1
List of accumulation operators

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

The semantics of accumulation operators in **forall** loops will be discussed later. Table 1 provides a list of the accumulation operators available in BLAZE. User defined accumulation operators are not currently allowed.

3.5. Sequential control structures

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

3.5.1. Branching statements

The **if** and **case** statements in BLAZE are similar to that in MODULA or ADA. A typical example is

```
if x > 3 then
   j := 100,
elsif x > 2 then
   y := 10;
elsif x > 0 and x <= 2 then
   y := 2;
else
   panic( );
end;
```

Note that a sequence of **if** tests can be strung together as a sequence of **elsif** clauses.

Multiway branching can also be performed by a **case** statement:

```
var particle: variant_charged_particle;
...
case particle species of
   when alpha =>
      ⟨statement_list⟩
   when electron | proton | neutron =>
      ⟨statement_list⟩
   when others =>
      ⟨statement_list⟩
end;
```

A single **when** can have multiple choices as in the case of electrons, protons, and neutrons in this example. The choice **others** can be used as a default choice for all cases not explicitly covered.

3.5.2. Loops

BLAZE contains several loop constructs including **for** and **while** loops. The simplest loop in BLAZE has the form:

```
loop
  ⟨statement_list⟩
  exit when ⟨boolean_condition⟩;
  ⟨statement_list⟩
end;
```

Here the loop is executed until the condition in the exit statement becomes true. 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. One may also omit the when clause in an exit statement leaving an unconditional exit. A simple loop with no exit statement will run forever.

The other kinds of sequential loops in BLAZE are constructed by preceding a loop-end block with a controlling while or for clause. The while loop is similar to the one in PASCAL, while the for construct is slightly more flexible. A typical example of a for loop is

```
for particle_type in electron..alpha loop
  for k in 50..−50 by −5 loop
    ⟨statement_list⟩
  end;
end;
```

The loop indices in for loops can be integers or user-defined enumerated types as shown. They can also be the elements of lists or one-dimensional arrays.

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. 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.

As noted, 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*. In this case the loop index is implicitly typed real, since *x_array* is an array of reals.

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 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
  for i in range (x, 1) do
    for j in range (x, 2) 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 sizes 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.

3.6.1. 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, 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:

$$Ex, Ey := gradient(phi);$$

Procedures 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 in BLAZE is done at compile time, but checking that array sizes match will, in general, be performed 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 one 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;
```

However, with a **forall** loop, a copy of the loop body is invoked in parallel, for each index value specified in the header. In the above example, all one hundred invocations of the body would be performed concurrently.

The header for a **forall** loop is syntactically the same as the header of **for** loop. In particular, a **by** clause is allowed in a **forall** loop, and one can index over the elements of a list or a one-dimensional array. A major difference between **for** loops and **forall** loops is that **forall** loops can contain local 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 in such situations one can elect to have a warning message printed 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 array x regardless of the order in which the loop invocations are 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.

In BLAZE, the order in which the associative accumulation operations are performed is not specified as part of the language definition. Instead, the compiler can select any appropriate order. Given sufficiently many processors, the above summation would be done as a fan-in tree in logarithmic time. Note that though the compiler may rearrange the order of such associative arithmetic operators, the order is decided statically at compile-time. Thus the actual order of execution of a BLAZE program on a multiprocessor architecture cannot affect the results of arithmetic operations.

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 'print' statements of C.

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

The **read** and **write** procedures require format strings as parameter (as in C). These format strings specify the number and types of the variables returned by **read** statements and the format in which the values are printed by **write** statements. 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. Also, input and output statements have side-effects and are thus contrary to the spirit of the functional procedure invocation used here. However, good input/output facilities are critical to a language's usability, especially during initial program development, and for this reason were included here, despite the difficulties raised.

The first issue arising is that of determinate execution. For example, what is the order of I/O in **forall** loops? Input/output is made determinate 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.

There are implementation difficulties associated with handling both input and output, but these difficulties are much greater for input. For example, read statements occurring in loops will generally force sequential execution of part of a program. Input is generally less common than output, and in some cases one may be able to perform input in parallel, but there remain important and unresolved implementation issues here.

By contrast, implementing parallel output is fairly straightforward. Each procedure creates a block of output data, which will be concatenated onto the output stream. This operation is highly parallel, and at least conceptually simple, especially in batch environments. The issues involved in performing parallel output in interactive environments are somewhat more subtle, but there appear to be no critical problems.

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 data structures more flexible than lists and recursive records and inclusion of more general 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 programs 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. For example, there is no means of expressing the asynchronous relaxation algorithm of Baudet [8] 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.

At this writing, a BLAZE compiler for VAX architectures running Unix exists, and will be distributed to interested researchers. Compilers for several parallel architectures are in progress, and experience with them will be reported elsewhere. Section 5 describes some of the implementation issues involved in more detail.

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 [12]. 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.

4.1. Description of particle-in-cell program

A plasma is a high temperature gas consisting of free electrons and positively ionized molecules. 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 Fig. 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;
```

Fig. 1. Data types of charged particles.

Given these type declarations, one can program the main procedure of the *particle-in-cell* program as shown in Fig. 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, seem to go a long way toward clarifying code. For example, procedure *gradient* takes array ϕ as

```
- 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
                                - initial positions and velocities
  phi := 0.0;                   - initialize the electric potential
  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
```

Fig. 2. Main procedure for particle-in-cell program.

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 *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

```
procedure project(res) returns: b;
param   res: array [,] of real;
        b: array [0..mc, 0..nc] of real;
const   m = upper(res, 1);      mc = m/2;
        n = upper(res, 2);      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
```

Fig. 3. Multigrid kernel routine project.

```
procedure multigrid(phi, sigma, tol) returns: phi;
param   phi, sigma: array [,] of real;
        tol: real;
const   m = upper(phi, 1);      mc = m/2;
        n = upper(phi, 2);      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
```

Fig. 4. Driver for multigrid Poisson solver.

overall control flow, and a small number of kernel procedures, which perform the required numerical operations. As an example of such a kernel procedure, we consider procedure *project*, given in Fig. 3. It takes as input an array corresponding to a given grid, and produces a 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* (Fig. 4) does just that.

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

```
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);          - left their mesh cells into
                                        - their new cell lists
end;
```

Fig. 5. Procedure for particle push with gird data structure.

```
procedure reshuffle(plasma) returns: new_plasma;
param   plasma: array [,] of ion_list;
        new_plasma: like(plasma);
const   nx = upper(plasma, 1);
        ny = upper(plasma, 2);
begin
  forall i in 0..nx do
    forall j in 0..ny do
      forall ion in range(plasma, 2) 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
```

Fig. 6. Procedure to maintain plasma data structure.

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 Fig. 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. Figure 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 described in the last section shows that BLAZE is 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 other languages. The more important issue here is compilation for parallel architectures, which is the central goal of the BLAZE language. 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 BLAZE compilers is dictated by our desire to target this language to a number of sequential and parallel architectures, and by the necessity of performing extensive transformation and 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.

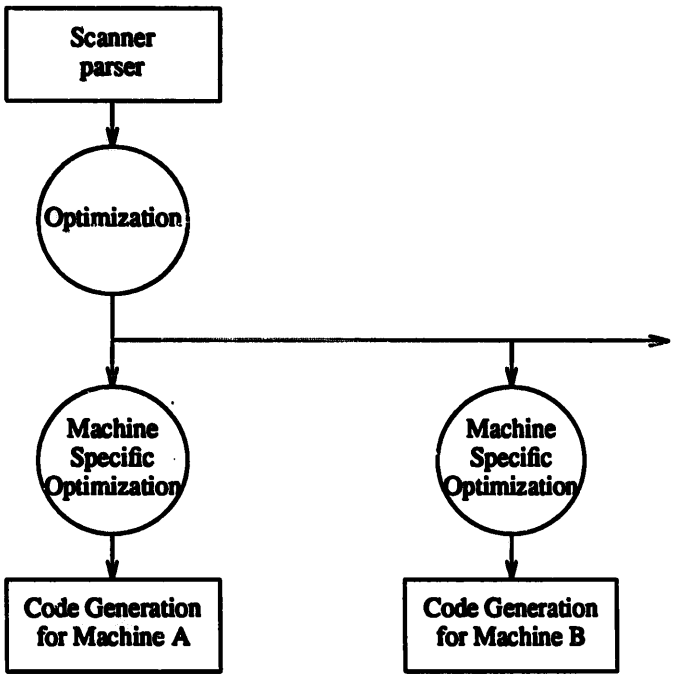

Fig. 7. General structure of compiler.

The general structure of BLAZE compilers is shown in Fig. 7. The lexical analysis, parsing, and first few phases of optimization can be performed 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 Fig. 8. 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 [4]. 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 interprocedural 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 [23], 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-time environment is also complicated by the list data type, which requires heap allocation and linked list access structures.

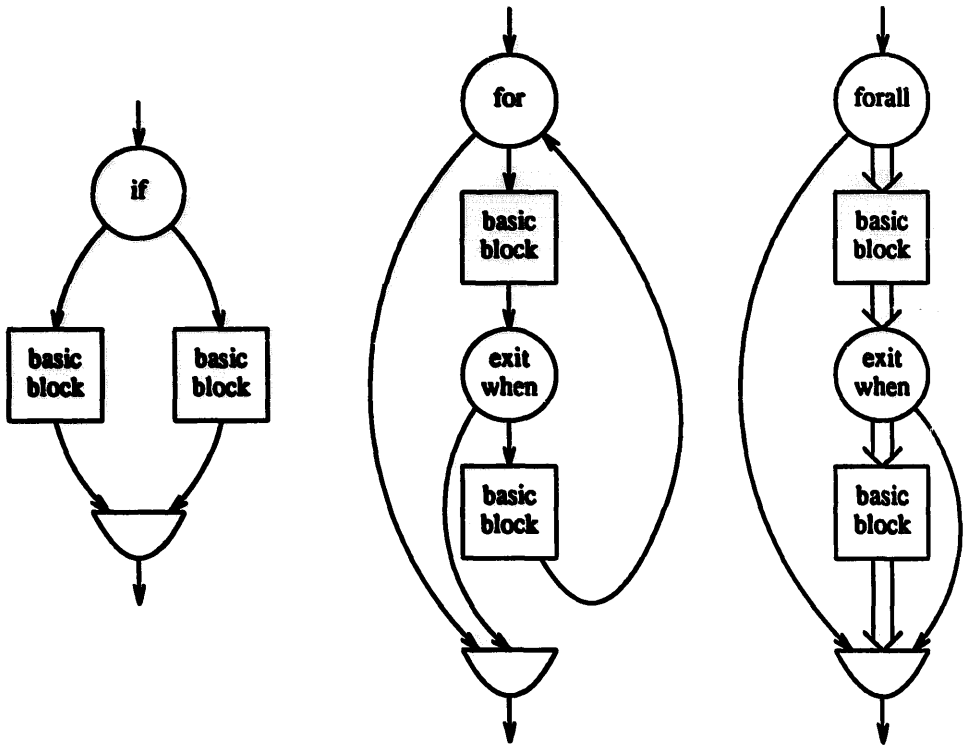

Fig. 8. Intermediate form.

The most critical issues in sequential environments are parameter transmission and storage allocation. Copy-in, copy-out semantics 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 straightforward 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 languages 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

$$A := B;$$

should 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 cannot express all forms of parallelism.

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 many kinds of loops, which are often nested several levels deep and can be quite complex. In the first BLAZE compiler for a multiprocessor, the types 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 scheme, tricky load balancing issues may arise if the execution times of the procedures invoked vary widely.

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 then 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 other block-structured 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 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 to 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 is its principal 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.

References

[1] American National Standard Programming Language FORTRAN, ANS X3.9-1978, American National Standards Institute, New York, 1978.

[2] Reference Manual for the Ada Programming Language, U.S. Department of Defense, Draft Revised MIL-STD 1815, 1982.

[3] HEP FORTRAN 77 User's Guide, Publication No. 9000006, Denelcor Inc., Aurora, CO, 1982.

[4] A.V. Aho and J.D. Ullmann, *Principles of Compiler Design* (Addison-Wesley, Reading, MA, 1977).

[5] J.R. Allen, K. Kennedy, C. Porterfield and J. Warren, Conversion of control dependence to data dependence, *Conference Record 10th Annual ACM Symposium on Principles of Programming Languages* (1983).

[6] Arvind, K.P. Gostelow and W.E. Plouffe, An asynchronous programming language and computing machine, TR-114a, Department of ICS, University of California, Irvine, 1978.

[7] J. Backus, Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs, *Comm. ACM* **21** (8) (1978) 613–641.

[8] G. Baudet, Asynchronous iterative methods for multiprocessors, *J. ACM* **25** (1978) 226–244.

[9] A.D. Falkoff and K.E. Iverson, The design of APL, *IBM J. Res. Development* **17** (4) (1973) 324–334.

[10] J. Ferante, K. Ottenstein and J. Warren, The program dependence graph and its uses in optimization, IBM Technical Report RC 10208, 1983.

[11] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* **21** (8) (1978) 666–677.

[12] R.W. Hockney and J.W. Eastwood, *Computer Simulation Using Particles* (McGraw-Hill, New York, 1981).

[13] K. Jensen and N. Wirth, PASCAL User's Manual and Report, 1976.

[14] K. Kennedy, Automatic translation of Fortran programs to vector form, Rice Technical Report 476-029-4, Rice University, 1980.

[15] B.W. Kernighan and L.L. Cherry, A system for typesetting mathematics, *Comm. ACM* **18** (1975) 151–157.

[16] J.L.W. Kessels, A conceptual framework for a nonprocedural programming language, *Comm. ACM* **20** (12) (1977) 906–913.

[17] D.J. Kuck, *The Structure of Computers and Computation, Vol. 1* (Wiley, New York, 1978).

[18] D.J. Kuck, R.H. Kuhn, B. Leasure, D.A. Padua and M. Wolfe, Dependence graphs and compiler optimizations, *Conference Record 8th Annual ACM Symposium on Principles of Programming Languages* (9181).

[19] J. McCarthy, Recursive functions of symbolic expressions and their computation by machines, *Comm. ACM* **3** (4) (1960) 184–195.

[20] J. McGraw, The VAL language: Description and analysis, *ACM Trans. Programming Languages and Systems* **4** (1) (1982) 44–82.

[21] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes and P. Hohensee, SISAL: Streams and iterations in a single-assignment language, Language Reference Manual Version 1.1, Lawrence Livermore National Laboratory Report M-146, 1983.

[22] R. Milner, A proposal for standard ML, Report CSR-157-83, Department of Computer Science, University of Edinburgh, 1983.
[23] P. Naur, Revised report on the algorithmic language ALGOL 60, *Comm. ACM* **6** (1) (1963) 1–20.
[24] D.A. Padua, D.J. Kuck and D.H. Lawrie, High speed multiprocessing and compilation techniques, Special Issue on Parallel Processing, *IEEE Trans. Comput.* **29** (9) (1980) 763–776.
[25] J.L. Wagener, Status of work toward revision of programming language Fortran, *SIGNUM Newslett.* **19** (3) (1984).
[26] J. Warren, A hierarchical basis for reordering transformations, *Conference Record 10th Annual ACM Symposium on Principles of Programming Languages* (1983).
[27] N. Wirth, Modula: A language for modular multiprogramming, *Software – Practice and Engineering* **7** (1979) 3–35.