Data flow languages

by WILLIAM B. ACKERMAN
Massachusetts Institute of Technology
Cambridge, Massachusetts

INTRODUCTION

There are several computer system architectures which have the goal of exploiting parallelism—multiprocessors, vector machines and array processors. For each of these architectures there have been attempts to design compilers to optimize programs written in conventional languages (e.g. "vectorizing" compilers for the FORTRAN language). There have also been new language designs to facilitate using these systems, such as Concurrent PASCAL for multiprocessors,[6] and languages that utilize the features of such systems directly, such as GLYPNIR for the Illiac IV array processor[19] and various "vectorizing" dialects of FORTRAN. These languages almost always make the multiprocessor, vector, or array properties of the computer visible to the programmer—that is, they are actually vehicles whereby the programmer helps the compiler uncover parallelism. Many of these languages or dialects are "unnatural" in that they closely reflect the behavior of the system for which they were designed, rather than reflecting the way programmers think about problem solutions.

Data flow computers also have the goal of taking advantage of parallelism. As will be seen below, the parallelism in a data flow computer is both microscopic (much more so than in a multiprocessor) and all-encompassing (much more so than in a vector processor). Like the other forms of parallel computer, data flow computers are best programmed in special languages. In fact, their need for such languages is stronger—most data flow designs would be extremely inefficient if programmed in conventional languages such as FORTRAN or PL/I. However, languages suitable for data flow computers can be very elegant. The language properties that a data flow computer requires are beneficial in their own right, and are very similar to some of the properties that are known to facilitate understandable and maintainable software, such as the absence of undisciplined control structures and module interactions. In fact, languages having many of these properties have been in existence since long before data flow computers were conceived. The principal property of a language suitable for data flow is *freedom from side effects*, which will be described below. The (pure) LISP language[20] is the best known example of a language without side effects. The connection between freedom from side effects and efficient parallel computation has been known for over ten years.[25]

To see why data flow computers require languages free of side effects, we must examine the nature of data flow computation and the nature of side-effects. A detailed description of the mechanism of data flow computers is beyond the scope of this paper. The interested reader is referred to References 2, 12, 15, 21, 22, 23.

There are three "data flow" languages that will be discussed in this paper. VAL[1] and ID[3] were developed by the data flow projects at the Massachusetts Institute of Technology and the University of California at Irvine, respectively. LUCID[4] was developed for program verification, not for programming data flow computers. It nevertheless is a suitable language for data flow computation.

Let us begin by examining a simple sequence of assignment statements written in a conventional programming language such as FORTRAN:

```
1  P=X+Y
2  Q=P/Y
3  R=X*P
4  S=R-Q
5  T=R*P
6  RESULT=S/T
```

A straightforward analysis of this program will show that many of these instructions can be executed concurrently, as long as certain constraints are met. These constraints can be represented by a graph (see Figure 1) in which nodes represent instructions and an arrow from one instruction to another means that the second may not be executed until the first has completed. So the permissible computation sequences include, among others:

```
(1,3,5,2,4,6)
(1,2,3,5,4,6)
(1, [2 and 3 simultaneously], [4 and 5 simultaneously], 6)
```

This type of analysis (commonly called data flow analysis, a term which long predates data flow computers) is frequently performed in two situations—at run-time in the arithmetic processing units of high performance conventional computers such as the IBM 360/91, and at compile time in optimizing compilers. In optimizing compilers, data

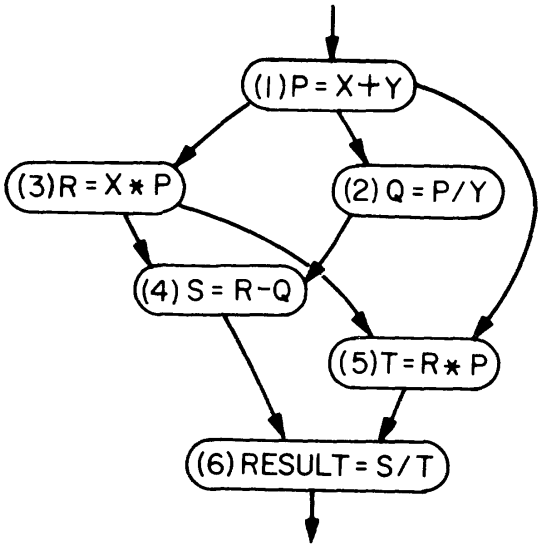

Figure 1

flow analysis yields improved utilization of temporary memory locations. For example, on a computer with high-speed general purpose or floating point registers, this program can be compiled to use the registers instead of core memory for P, Q, R, S and T, if it can be determined that they will not be used again. (This determination is very difficult, principally because of GO TOs, which is one of the reasons why it is very difficult to write optimizing compilers for languages such as FORTRAN.)

In the graph representation, an instruction can be executed as soon as all the instructions with arrows pointing into it have completed. On a multiprocessor system, we could allocate a processor for each instruction, with appropriate instructions (such as semaphore operations[13]) to enforce the sequencing constraints, but execution would be hopelessly inefficient because the parallelism of this example is far too "fine grained" for a multiprocessor. The overhead in the process scheduling and in the *wait* and *signal* instructions would be many times greater than the execution time of the arithmetic operations. A data flow computer, on the other hand, is designed to execute algorithms with such a fine grain of parallelism efficiently. In these machines, parallelism is exploited at the level of individual instructions, as in the previous example, and at all coarser levels as well; in most programs there are typically many parts, often far removed from each other, at which computation may proceed simultaneously.

To exploit parallelism at all levels, the instruction sequencing constraints must be deducible from the program itself. Let us refer again to the previous program to see how this may be done.

The sequencing constraints in Figure 1 are given by arrows. It is not difficult to see that these arrows coincide with data transmission from one instruction to its successor through variables. In fact, the graph could be redrawn with the arrows labeled by the variables that they represent, as in Figure 2.

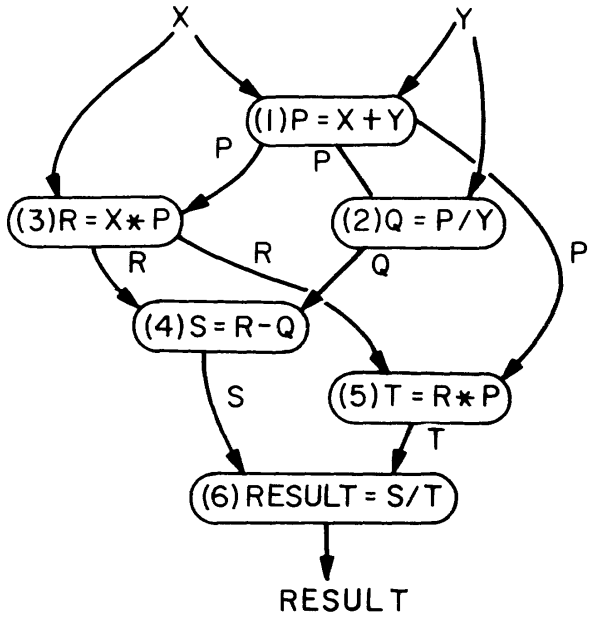

Figure 2

In a data flow computer, the machine level program is represented essentially in this form—a graph with pointers between nodes, the pointers representing both the flow of data and the sequencing constraints. Each instruction is kept in a hardware device (an extremely simple "processor") that is capable of "firing" or executing an instruction when all of the necessary data values have arrived, and sending the result to the processors that hold destination instructions.*

The programming language for a data flow computer must therefore satisfy two criteria:

1. It must be possible to deduce the data dependencies of the program operations.
2. The sequencing constraints must always be exactly the same as the data dependencies, so that the instruction firing rule can be based simply on the availability of data.**

There are two general properties of a language which make it possible to meet these criteria: locality of effect and freedom from side effects.

* Although the language concepts presented in this paper assume that the computer exploits parallelism at a microscopic level, not all "data flow" or "data driven" computers do so. Designs of data flow computers that exploit parallelism only at the subroutine level may be found in References 10 and 24.

** Not all designs for data flow computers accept the second of these criteria or its consequences. The LAU language [9] is intended for execution on a data flow computer, but it was designed to support data base updating and retrieval, so it has side effects on certain operations. The sequencing of these operations must therefore be constrained by means other than data dependencies, and so it does not satisfy the second criterion. The extra constraints in LAU are specified by path expressions [7] written into the source program.

LOCALITY OF EFFECT

Locality of effect means that instructions do not have unnecessary far-reaching data dependencies. For example, the FORTRAN program fragment given previously appears to use variables P, Q, R, S and T only as temporaries. A similar program fragment appearing elsewhere in the program might use the same temporaries for some unrelated computation. The logic of the program might be such that the two fragments could be executed concurrently were it not for this overuse of names. (Unfortunately, many conventional languages encourage this style of programming.) Any attempt to execute the program fragments concurrently would be impossible because of the apparent data dependencies arising from overuse of these temporaries, unless the compiler can deduce that the conflict is not real and remove it by using different sets of temporaries.

In languages such as FORTRAN and PL/I, this is not so easy to determine. A reference to a variable in one part of the program does not necessarily imply dependence on the value computed in another part—the variable might be overwritten before it is next read. Careful analysis is required to determine whether a variable is actually transmitting data or is "dead." This analysis is made much more difficult if unrestricted GO TOs or other undisciplined control structures are allowed.

The problem can be simplified by making every variable have a definite "scope," or region of the program in which it is active, and carefully restricting the entry to and exit from the blocks that constitute scopes. It is also helpful to deny procedures access to any data items that are not transmitted as arguments, though this is not really necessary if global variables are avoided and procedure definitions are carefully "block structured" as in PASCAL.

SIDE EFFECTS

Freedom from side effects is a necessary property to ensure that the data dependencies are the same as the sequencing constraints. It is much more difficult to achieve than locality of effect. This is because locality only requires superficial restrictions on the language, whereas freedom from side effects requires fundamental changes in the way the language's "virtual machine" processes data.

Side effects come in many forms—the most well known examples are procedures that modify variables in the calling program, as in the following PASCAL example:

```
procedure GETRS(X, Y:real);   (* RS is declared in
begin RS:=X*X+Y*Y              an outer block *)
end;
```

Absence of global or "common" variables and careful control of the scopes of variables make it possible for a compiler to prohibit this sort of thing, but a data flow computer imposes much stricter prohibitions against side effects—a procedure may not even modify its own arguments. In fact, in a sense nothing may ever be modified at all.

To determine what kind of prohibitions against side effects are needed to achieve concurrent computation, we must examine programs that manipulate structured data such as arrays or records, since the problem does not arise when only simple data values are used.

Consider the following procedure which modifies its arguments by a conventional "call by reference" mechanism. SORT2 is a procedure to sort two elements, J and J+1, of array A into ascending order by exchanging them if necessary.

```
    procedure SORT2(var A:array[1. .10] of real;
      J:integer);
    var T:real;
    begin if A[J]>A[J+1] then begin
                    T:=A[J];
                    A[J]:=A[J+1];
                    A[J+1]:=T;
                    end
    end;
(1) SORT2(AA, J);
(2) SORT2(AA, K);
(3) P:=AA[L];
```

Statements 1 and 2 might interfere with each other and with Statement 3. Since the values of J, K, and L are not known to the compiler, it must assume that the statements will conflict, and execute them in the exact order specified. Any attempt at parallel execution might result in the incorrect results, depending on J, K, L, and unpredictable fluctuations in timing.

A phenomenon known as "aliasing" makes the problem even more difficult. This occurs when different formal parameters to a procedure refer to the same actual parameter, that is, they are "aliases" of each other:

```
procedure SORTREAD(var A, B:array[1. .10] of real;
  I, J:integer);
begin SORT2(A, I);  (* SORT2 is defined above *)
  RESULT:=B[J];
end;
```

In this program it would appear that, since A and B are different arrays, the invocation of SORT2 and the reference to B[J] could proceed concurrently. However, if this were part of a larger program and SORTREAD were invoked in the statement

```
SORTREAD(Q,Q,M,N);
```

the arrays A and B would actually be the same. Languages such as FORTRAN and PL/I, in which external procedures are not available to the compiler when the calling program is being compiled, make the problem harder still. Facilities for manipulating data structures by pointers, such as the "pointer" data type in PL/I and the record manipulating operations of PASCAL make it possible for all of these problems to arise without using procedures.

Even if procedures and pointers are not used, the sequencing constraints may be far from clear, as in

```
(1) A[J]:=3;
(2) X:=A[K];
```

If the convention is made that any statement modifying any element of an array constitutes a "writing" of the array, then Statement 1 clearly passes array A to Statement 2. But then a statement such as the assignment in

```
for J:=1 to 10 do
  A[J+1]:=A[J]+1;
```

depends on itself!

The inescapable conclusion is that, if arrays and records exist as global objects in a memory and are manipulated by statements and passed as procedure parameters, it is virtually impossible to tell, when an array element is modified, what effects that modification may have elsewhere in the program.

One way to solve some of these problems is to use "call by value" instead of the more common "call by reference." This solves the aliasing problem and the problem of procedures modifying their arguments. In a "call by value" scheme, a procedure copies its arguments (even if they are arrays). This way it can never modify the actual argument in the calling program. Call by reference has traditionally been used instead of call by value because it is a more natural way of thinking about computation (and is more efficient) on von Neumann computers.

APPLICATIVE LANGUAGES

For data flow languages, a scheme is used which goes far beyond call by value: all arrays are *values* rather than *objects*, and are treated as such at all times, not just when being passed as procedure arguments. Arrays are not modified by subscripted assignment statements such as

```
A[J]:=S;
```

Instead, they are processed by *operators* which create new array values. The simplest operator to perform the nearest equivalent of modifying an array takes three arguments—an array, an index, and a new data value. The result of the operation is a new array, containing the given data value at the given index, and the same data as the original array at all other indices.

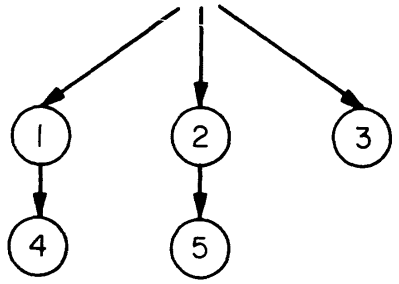

Figure 3

In the VAL language,[1] the syntax for this elementary operator is

```
A[J:S]
```

In the ID language,[3] it is

```
A+[J]S
```

This operation *does not modify* its argument. Hence, in this VAL program:

```
(1) B:=A[J:S];
(2) C:=A[K:T];
(3) P:=A[L];
(4) Q:=B[M];
(5) R:=C[N];
```

Statements 1 and 2 do not interfere with each other or with array A. Statement 3 may be executed immediately, whether 1 and/or 2 have completed or not, since they would have no effect on Statement 3 anyway. Statement 4 can be executed as soon as Statement 1 completes, whether Statement 2 has completed or not. The sequencing constraints are shown in Figure 3.

Note that this situation is similar to the one in the simple FORTRAN example of the first section. The sequencing constraints are exactly the same as the data dependencies, which is the property we seek for data flow.

An operator-based handling of arrays and records automatically accomplishes call by value. As in a call by value scheme, a routine such as SORT2 would not accomplish its purpose. SORT2 must return the new array as its value. It could be written in VAL (omitting type declarations) as:

```
function SORT2(A, J)
  if A[J]<A[J+1] then A
    else (A[J:A[J+1]])[J+1:A[J]]     % No temporary needed during this
  end                                % exchange because A is not modified.
end
```

Note that the array construction operator may be composed with itself and with other operators in the same way as arithmetic operators.

In conventional languages, procedures do most, if not all, of their work through side effects—a procedure might be designed to alter dozens of variables in the calling program.

Functions, on the other hand, are typically limited to returning a single value, which typically may not be an array or record. To make functions as powerful and flexible as procedures in conventional languages, applicative languages often allow functions to return several values, or entire records or arrays, or both. If a function returns several values, its call can be used in a "multiple assignment" such as

```
X,Y,Z:=FUNC(P,Q,R)  % FUNC returns 3 values
```

Treating arrays and records as values instead of objects is perhaps the most profound difference in the way people must think when writing programs in data flow languages instead of conventional languages. The customary view is of arrays as objects residing in static locations of memory, and being manipulated by statements that are executed in some sequence. As we have seen, this view is incompatible with detection of parallelism among the statements. The correct view for data flow is of arrays and records as values manipulated by operations just as simple values (integers, reals) are. Then the parallelism among the operators can be deduced from the data dependencies just as for simple values.

The value-oriented approach to arrays is confusing to some at first, but it need not be. An integer array should be thought of as a string of integers, just as an integer can be thought of as a string of digits. If J has the value 31416, the statement

```
K:=J-400;
```

leaves K equal to 31016; no programmer would expect the value of J to be affected. If A is an array with elements [3,1,4,1,6], the statement

```
B:=A[3:0];
```

is completely analogous; it leaves B with elements [3,1,0,1,6] and of course does not change A.

Languages which do all processing by means of operators applied to values are called *applicative* languages, and are thus the natural languages for data flow computation. The earliest well known applicative language implemented on a computer is LISP.[20] (It is applicative only if RPLACA, RPLACD, and all other functions with side effects are avoided; this subset of the language is often called "pure" LISP.) The connection between applicative languages and the detection of parallelism has been reproted by Tesler and Enea[25] and by Friedman and Wise.[14] The development of LISP and other applicative languages, and the Tesler/Enea paper, all predate the data flow computer concept by several years. The concept of computation by applicative evaluation of expressions goes back to the invention of the lambda calculus in 1941.[8]

EFFICIENCY CONSIDERATIONS IN APPLICATIVE LANGUAGES

Applicative languages have recently given rise to controversy concerning their time efficiency in practical situations. It is claimed that many algorithms cannot be executed as efficiently in applicative languages as in conventional statement-oriented languages. The issue is only one of exploiting parallelism, not any fundamental limitation in the computing power of applicative systems. This is because any program written for a conventional von Neumann computer can be rewritten in an applicative language by treating the entire memory space as one array. Statements that manipulate the memory then become operations on that array. The array must be passed from each operation to the next, so execution in such an applicative system must be strictly sequential—no parallelism can be exploited. However, in the original program written in a conventional language, a knowledgeable programmer might be able to explicitly specify parallelism, making the program run more efficiently than in the applicative system.

Consider the conventional program

```
(1) A[J]:=S
(2) A[K]:=T;
(3) P:=A[M];
(4) Q:=A[N];
```

If the programmer knows that the set of indices for the array A can be divided into two disjoint sets, with J and M in one set and K and N in the other, then Statements 1 and 2 could be executed simultaneously, 3 would only need to follow 1, and 4 would only need to follow 2. If the programmer has the ability to control parallelism explicitly (say, by semaphores, monitors, or path expressions), he could specify exactly those constraints. This is not possible in a data flow language without some additional mechanism. However, the problem can often be avoided by dividing the array into separate parts, manipulating them separately, and combining them only when necessary. For example, suppose array A1 contains only those elements that J and M are known to index, and A2 contains those that K and N index. Then

```
B1:=A1[J:S];
B2:=A2[K:T];
P:=B1[M];
Q:=B2[N];
```

exploits the parallelism exactly. There are other ways of overcoming the tendency for arrays to limit parallelism, which will be discussed later. The question of how to exploit parallelism in applicative languages for a wide variety of programming problems is an active area of research.

DEFINITIONAL LANGUAGES AND THE SINGLE ASSIGNMENT RULE

Having accepted an applicative programming style and a value-oriented rather than object-oriented execution model, we next examine the implications of this style upon the meaning of assignment statements.

Except in iterations, (which will be discussed later), an assignment statement has no effect except to provide a value

and bind that value to the name appearing on its left hand side. The result of the assignment is accessible only in subsequent expressions in which that name appears. If the language uses blocks in which all variables are local to the block for which they are declared, the places where a variable is used can be determined by inspection. If the expression on the right hand side of an assignment is substituted for the variable on the left hand side in all places where that variable appears within its scope, the resultant program will be completely equivalent.

```
S:=X+Y;
D:=3*S;          is equivalent to  D:=3*(X+Y);
E:=S/2+F(S);                       E:=(X+Y)/2+F(X+Y);
```

(The program on the left is clearly more efficient, requiring only two additions instead of four. We are not proposing that the substitution be made in practice.)

Now this situation is the same as a system of mathematical equations. If a system of equations contains "S=X+Y", it is clear that "3*S" and "3*(X+Y)" are equivalent. Hence the statement

```
S:=X+Y;
```

means the same thing in the program that the equation "S=X+Y" means in a system of equations, namely that, in the scope of these variables, S is the sum of X and Y. The correspondence can be thought of as holding for all time; it is not necessary to think of the statements as being executed at particular instants. (In fact, perhaps the word "variable" is an inappropriate term.) Of course, the addition of X and Y to form S must take place before any of the operations that use S can be performed, but the programmer doesn't need to be directly concerned with this.

Hence the statement S:=X+Y should be thought of as a *definition*, not an assignment. Languages which use this interpretation of assignments are called *definitional languages* as opposed to conventional *imperative languages*. Such languages are well suited to program verification because the assertions one makes in proving correctness are exactly the same as the definitions appearing in the program itself. In conventional languages one must follow the flow of control to determine *where* in the program text assertions such as "S=X+Y" are true, because the variables S, X, and Y can be changed many times. Assertions must therefore be associated with points in the program.

In a definitional language the situation is extremely simple: If a program block contains the statement

```
S:=X+Y;
```

then the assertion S=X+Y is true. Of course, care must be taken to prevent circular definitions such as

```
X:=Y;
Y:=X+3;
```

We can either have the compiler allow definitions to appear in any order as long as there is some consistent order, or we can require them to appear in a consistent order. A consistent order is one in which no name is referred to (on the right hand side of a definition) before it is defined. This condition is easily checked by a compiler. So the actual proof rule is: If a program block contains the statement

```
S:=X+Y;
```

and the program compiles correctly, then S=X+Y is true. Strictly speaking, it is only true in the statements after the one defining S, but, since S does not appear in any earlier statements, the assertion can be treated as being true throughout the block.

The power of definitional languages for program verification is well known outside of the data flow field. LUCID[4] is an example of an applicative definitional language designed expressly for ease of program verification.

There is a problem that could ruin the elegance of definitional languages—multiple definition of the same name. Definitional languages almost invariably obey the *single assignment rule*: A name may appear on the left hand side of an assignment (definition) only once within its scope. The single assignment rule prevents program constructs which imply mathematical abominations, such as

```
J:=J+1;
```

Since the appearance of J on the right hand side precedes the definition of J, it implies an inconsistent statement sequencing that the compiler would diagnose. The prevention of such abominations is of course necessary if the definitions in the program are to be carried directly into assertions used in proving correctness, because the assertion "J=J+1" is absurd.

It is not actually necessary that a data flow language conform to the single assignment rule. A data flow language with multiple assignments could be designed in which the scope of a variable extends only from its definition to the next definition of the same variable. The next definition in effect introduces a new variable that simply happens to have the same name. A program written in such a way can be easily transformed into one obeying the single assignment rule: simply choose a new name for any redefined variable, and change all subsequent references to the new name. However, the advantages of single assignment languages, namely, clarity and ease of verification, generally outweigh the "convenience" of re-using the same name.

ITERATIONS

There remains one area in which statements such as

```
I:=I-1;
```

or

```
A:=A[J:X+Y];
```

seem to be necessary, explicitly or implicitly, in conven-

tional languages, and that is in iterations. The technique of renaming variables to make a program conform to the single assignment rule only works for straight-line programs: If a statement appears in a loop, renaming its variables will not preserve the programmer's intentions. For example:

```
for I:=1 to 10 do      cannot be      for I:=1 to 10 do
  J:=J+1;           transformed to      J1:=J+1;
```

If the language allows general GO TOs, with the resulting possibility of complex and unstructured loops, the problem is indeed difficult. However, data flow languages generally have no GO TO statement, and require loops to be created only by specific program structures such as the "while. . .do. . ." and similar statements found in PL/I and PASCAL. This makes the problem easy to solve, and makes it possible to give iterations a very simple and straightforward meaning.

To develop the data flow equivalent of a "while. . .do. . ." type of iteration, we must consider what the "do" part of such a structure contains. Since there are no side effects, the only state information in an iteration is the bindings of the loop variables, and the only activity that can take place is the redefinition of those variables through functional operators. An iteration therefore consists of

1. Definitions of the initial values of the loop variables.
2. A predicate to determine whether, for any given values of the loop variables, the loop is to terminate or to cycle again.
3. If it is to terminate, some expression giving the value(s) to be returned. These values typically depend on the current values of the loop variables.
4. If it is to cycle again, some expressions giving the new values to be assigned to the loop variables. These also typically depend on the current values of the loop variables.

An iteration to compute the factorial of N could be written (omitting type declarations) in VAL as follows:

```
for I, J:=N, 1;                % Give loop variables I and
                               % J initial
                               % values N and 1
                               % respectively. I will
                               % count downward. J will
                               % keep
                               % accumulated product.
do if I=0                      % Decide whether to
                               % terminate.
  then J                       % Yes, final result is current
                               % J.
  else iter I, J:=I-1, J*I;
                               % No, compute new values
                               % of I and J,
end                            % and cycle again.
```

It could be written in ID as follows:

```
initial I←N; J←1
while I≠0 do
  new I←I-1;
  new J←J*I;
return J
```

Its representation in LUCID is similar to that in ID.

Although the values of the loop variables do change, they change only between one iteration cycle and the next. The single assignment rule, with its prohibition against things like "I=I-1" is still in force within any one cycle. All redefinitions take place precisely at the boundary between iteration cycles (though they need not actually occur simultaneously). This is enforced in VAL by allowing redefinitions only after the word *iter*, which is the command to begin a new iteration cycle. In ID and LUCID, the "new" values become the "current" values at the boundary between cycles.

Since the single assignment rule is obeyed, and names have single values, within any one iteration cycle the mathematical simplicity of assertions about values still exists. The assertions typically take the form

"In any cycle, S=X+Y"

Assertions used in proving correctness of an iteration are usually proved inductively. Because assertions take such a simple form, such proofs are usually simpler than in conventional languages. For example, the assertion

$I \geq 0$ and $J*(I!)=N!$

is used to prove correctness of the previous program. The basis of the induction is that it is true for the initial values I=N and J=1. (We assume $N \geq 0$.) The induction step is that, if another cycle is started with the values I-1 and J*I, they will obey the assertion, that is,

$I-1 \geq 0$ and $(J*I)*((I-1)!)=N!$

which is clearly true if we observe that a new cycle will only be started if I>0 and hence $I-1 \geq 0$.

ERRORS AND EXCEPTIONAL CONDITIONS

Locality of effect requires that errors such as arithmetic overflow be handled by *error values* rather than by program interruptions or manipulation of global status flags. If an error occurs in an operation, that fact must be transmitted to the destinations of that operation and nowhere else. This can be easily accomplished by enlarging the set of values to include error values such as overflow, underflow, or zero-divide.

If the intention is to abort the computation when an error occurs, this can be achieved by making the error values *propagate*—if an argument to an arithmetic operation is an

error, the result is an error. When an error propagates to the end of an iteration body, that iteration always terminates rather than cycling again. In this way the entire computation will come to a stop quickly, yielding an error value as the result. If the computer keeps a record of every error generation and propagation, that record will provide a detailed trace of when and where the error occurred, and what iterations and procedures were active.

If the intention is to correct an error when it occurs (perhaps keeping a list of such errors in some array), that can be accomplished through operations that test for errors. For example, a program to set Z to the quotient of X and Y, or to zero if an error occurs, could be written in VAL as follows:

```
ZZ:=X/Y;
Z:=if is_error(ZZ) then 0 else ZZ end;
```

METHODS OF OBTAINING MAXIMUM PARALLELISM

To achieve the greatest parallelism, it is necessary that computations not be performed sequentially unless necessary. The iteration constructs described previously imply a sequential execution of the various cycles. If the values of the iteration variables in one cycle depend on those of the previous cycles (as they do in the factorial example given previously), nothing can be done about it, although a data flow computer can often execute *part* of a cycle before the previous one has completed. If the values in one cycle do *not* depend on those of the previous cycles, the cycles can be performed in parallel. In VAL this is accomplished with a *forall* program construct which does not allow one cycle to depend on the others, and directs the computer to perform all cycles simultaneously. In ID the same effect is achieved automatically by tagging the values of the iteration variables with their cycle number, and allowing them to be processed out of sequence, or simultaneously, whenever they do not depend on each other.

Another potential "bottleneck" in data flow computation is the requirement that all elements of an array be computed before any element of that array may be accessed. If function "F" creates an array value by filling the array one element at a time, and then passes the array to "G," which reads the elements one at a time, G cannot begin until F completes. In many instances this delay is unnecessary, and various techniques have been proposed for eliminating it without departing from the principle that the sequencing constraints are exactly the data dependencies. One method, mentioned in the fifth section, is to explicitly divide the array into pieces, or use separate data items instead of an array. This method is quite general, but it requires specific calculation by the programmer of which parts of the array are needed at which time.

Streams

A method of overcoming the array bottleneck is the use of "streams."[3,11,26] A stream may be thought of as an array that is fragmented in time and is processed one element at a time.

In the previous example of F creating an array one element at a time and passing it to G, a stream would be the natural way to do this. G would receive each element as soon as F created it, so G would be processing the N^{th} element while F computes the $N+1^{st}$, resulting in parallel "pipelined" computation of F and G.

The constraint that stream elements be created and consumed in strict sequence may be enforced by placing some restrictions on the source program to prevent "random access." A program to manipulate streams may be written in a recursive style,[11,26] in which a stream is treated like a list in LISP, or in an iterative style[3] in which the rebinding of an iteration variable denoting a stream causes that stream to advance to the next element. Either method enforces the sequencing constraint if certain rules are followed regarding the permissible recursions or iterations.

When streams are viewed not as temporally separated arrays but as sequences of data items, functions that process streams have a few interesting and useful properties that pure mathematical functions do not have: they can emit more (or fewer) outputs than their inputs, and they can exhibit "memory" from one element to another. This makes stream functions suitable for "on-line" applications such as updating a data base. Stream functions are also useful for operations normally performed by coroutines, such as a function to remove all ⟨newline⟩ characters from its input, or insert a ⟨newline⟩ after every 80 characters. A stream function is equivalent to a coroutine that communicates by transmission and reception of data values. A data flow program using streams is a network of parallel communicating coroutines, a computational model that has been of some theoretical interest in the last few years.[16–18]

Streams and input/output operations

Streams form a natural mechanism for handling input and output. By treating the sequence of characters read in from an external medium as a stream, it is possible for a program to operate on the data as it is read in. The program can generate an output stream, which is printed as it is produced.

CONCLUSION

There have recently been calls for the abandonment of the traditional "von Neumann" computer architecture as a good way of realizing the enormous potential of VLSI technology.[5] There has also been widespread recognition that proper language design is essential if the high cost of software is to be brought under control, and that most existing languages are seriously deficient in this area.

Fortunately, the implications of these trends for language design are similar—languages must avoid an execution model (the "von Neumann" model) that involves a global memory whose state is manipulated by sequential execution of commands. Such a global memory makes realization of

the potential of VLSI technology difficult because it creates a "bottleneck" between the computer's control unit and the memory. Languages that use a global memory in their execution model also exacerbate the software problem by allowing program modules to interact with each other in ways that are difficult to understand, rather than through simple transmission of argument and result values. Future language designs based on concepts of applicative programming should be able to help control the high cost of software and to meet the needs of future computer designs.

REFERENCES

1. Ackerman, W. B., and J. B. Dennis, "VAL—A Value-Oriented Algorithmic Language: Preliminary Reference Manual," Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Massachusetts.
2. Arvind, and K. P. Gostelow, *Dataflow Computer Architecture: Research and Goals*, Department of Information and Computer Science (TR 113), University of California-Irvine, Irvine, California, February 1978.
3. Arvind, K. P. Gostelow and W. Plouffe, *The (Preliminary) Id Report*, Department of Information and Computer Science (TR 114), University of California-Irvine, Irvine, California, May 1978.
4. Ashcroft, E. A., and W. W. Wadge, "Lucid, a Nonprocedural Language with Iteration," *Communications of the ACM*, Vol. 20, No. 7, July 1977, pp. 519-526.
5. Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM* Vol. 21, No. 8, August 1978, pp. 613-641.
6. Brinch-Hansen, P., "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975, pp. 199-207.
7. Campbell, R. H., "Path Expressions: A Technique for Specifying Process Synchronization," Department of Computer Science (Report UIUCDCS-R-77-863), University of Illinois at Urbana-Champaign, Urbana, Ill., 1977.
8. Church, A., "The Calculi of Lambda-Conversion," *Annals of Mathematical Studies*, Vol. 6, Princeton University Press, 1951.
9. Comte, D., G. Durrieu, O. Gelly, A. Plas and J. C. Syre, "Parallelism, Control and Synchronization Expressions in a Single Assignment Language," *SIGPLAN Notices*, Vol. 13, No. 1, January 1978, pp. 25-33.
10. Davis, A. L., "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine," *Proceedings of the Fifth Annual Symposium on Computer Architecture, Computer Architecture News*, Vol. 6, No. 7, April 1978, pp. 210-215.
11. Dennis, J. B., "A Language Design for Structured Concurrency," *Design and Implementation of Programming Languages: Proceedings of a DoD-Sponsored Workshop* (J. H. Williams and D. A. Fisher, Eds.), *Lecture Notes in Computer Science*, Vol. 54, October 1976. Also, Computation Structures Group, Note 28-1, February 1977, Laboratory for Computer Science, MIT, Cambridge, Massachusetts.
12. Dennis, J. B., D. P. Misunas and C. K. C. Leung, "A Highly Parallel Processor Using a Data Flow Machine Language," Computation Structures Group, Memo 134, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, January 1977. To appear in *IEEE Transactions on Computers*.
13. Dijkstra, E. W., "Cooperating Sequential Processes," *Programming Languages* (F. Genuys, Ed.), Academic Press, New York, New York, 1968.
14. Friedman, D. P., and D. S. Wise, "The Impact of Applicative Programming on Multiprocessing," *Proceedings of the 1976 International Conference on Parallel Processing* (P. H. Enslow, Ed.), August 1976, pp. 263-272.
15. Gurd, J., I. Watson and J. Glauert, "A Multilayered Data Flow Computer Architecture," Department of Computer Science, University of Manchester, Manchester, England, July 1978.
16. Hoare, C. A. R., "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21, No. 8, August 1978, pp. 666-677.
17. Kahn, G., "The Semantics of a Simple Language for Parallel Programming," *Information Processing 74: Proceedings of the IFIP Congress 74* (J. L. Rosenfeld, Ed.), 1974, pp. 471-475.
18. Kahn, G., and D. MacQueen, "Coroutines and Networks of Parallel Processes," *Information Processing 77: Proceedings of IFIP Congress 77* (B. Gilchrist, Ed.), August 1977, pp. 993-998.
19. Lawrie, D. H., "GLYPNIR Programming Manual," ILLIAC IV Document no. 232, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Ill., August 1970.
20. McCarthy, J., et. al., "LISP 1.5 Programmer's Manual," MIT Press, 1966.
21. Miranker, G. S., "Implementation of Procedures on a Class of Data Flow Procedures," *Proceedings of the 1977 International Conference on Parallel Processing* (J. L. Baer, Ed.), August 1977, pp. 77-86.
22. Patil, S. S., R. M. Keller and G. Lindstrom, "An Architecture for a Loosely-Coupled Parallel Processor (Draft)," Department of Computer Science (UUCS-78-105), University of Utah, Salt Lake City, Utah, July 1978.
23. Plas, A., D. Comte, O. Gelly, and J. C. Syre, "LAU System Architecture: A Parallel Data Driven Processor Based on Single Assignment," *Proceedings of the 1976 International Conference on Parallel Processing* (P. H. Enslow, Ed.), August 1976, pp. 293-302.
24. Rumbaugh, J. E., "A Data Flow Multiprocessor," *IEEE Transactions on Computers*, Vol. C-26, No. 2, February 1977, pp. 138-146.
25. Tesler, L. G., and H. J. Enea, "A Language Design for Concurrent Processes," *Proceedings of the AFIPS Conference*, Vol. 32, 1968, pp. 403-408.
26. Weng, K.-S., *Stream-Oriented Computation in Recursive Data Flow Schemas*, Laboratory for Computer Science (TM-68), MIT, Cambridge, Massachusetts, October 1975.