Data Flow Computing
The VAL Language

Computation Structures Group Memo 188
January 1980

James R. McGraw
Lawrence Livermore Laboratory
Livermore, California

Prepared for submission and publication to ACM Transactions on Programming Languages and Systems

PREPRINT UCRL-83251

ABSTRACT

Data flow computing is a radical strategy for achieving high computational power by focusing many small processors on the execution of each program. A computation is represented by its data flow graph, which displays all available forms of concurrency. Each operator in a graph is scheduled for execution on one of the processors as soon as all of its operands are available. Software in this environment must promote the identification of concurrency in algorithms and its representation in data flow graphs.

This paper presents a detailed introduction to VAL, a language specifically designed for data flow computing. VAL stresses implicit concurrency, most of it possible because side-effects and aliasing are not representable. The salient language features are described and illustrated through examples taken from a complete VAL program for adaptive quadrature. An analysis of the language shows that VAL meets the basic needs for a data flow environment, but that substantial work still needs to be done in the areas of language translation and use.

KEYWORDS: data flow functional programming VAL concurrency multiprocessing

CR CATEGORIES: 4.22 4.32

CONTENTS

Data Flow Computing

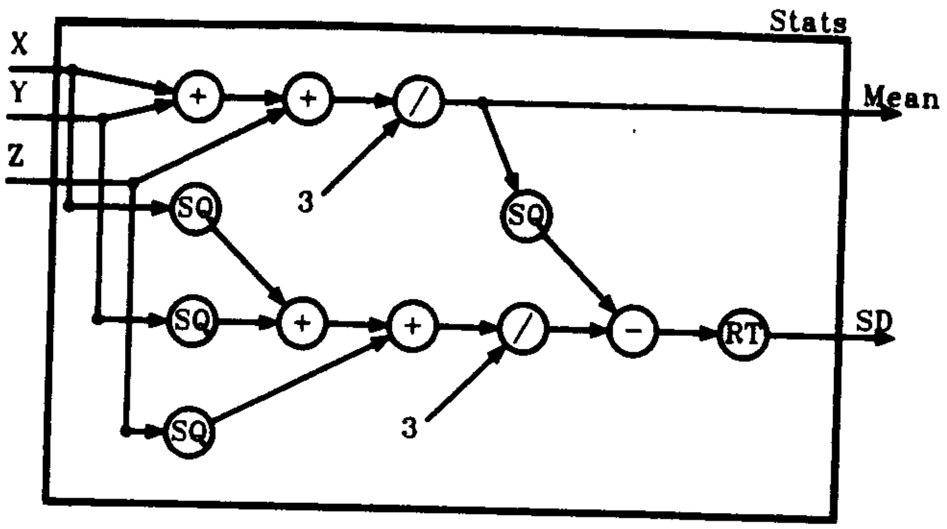
The VAL Language

As the use of computers expands in scientific areas, the demand for faster computers continues to increase at a surprising rate. As one example, physicists at Lawrence Livermore Laboratory have computations requiring gigaflop speed (one billion floating point operations per second) for reasonable response time. In the past, improved hardware technology has been responsible for most of the advances in speed; however, those gains have been slowing. Physical constraints (speed of light, power-cooling ratios, etc.) may soon limit the speed achievable by a single processor. One solution applies multiple processors to the execution of each task. Data flow computing attempts to carry out this strategy on a massive scale (*eg* 1000 processors per task). In order to achieve high concurrency, appropriate tools must be designed for describing algorithms and mapping them onto this unusual environment. VAL is a high-level language developed at MIT for exactly this purpose. This paper presents the basic features of the language through discussion and extensive examples.

Background

Highly computational tasks often contain substantial amounts of concurrency. At LLL the majority of these programs use very large, two-dimensional arrays in a cyclic set of instructions. In many cases, all new array values could be computed simultaneously, rather than stepping through one position at a time. To date, vectorization has been the most effective scheme for exploiting this concurrency. However, pipelining and independent multiprocessing forms of concurrency are also available in these programs, but neither the hardware nor the software exist to make it workable.

The data flow concept incorporates these forms of concurrency in one basic graph-oriented system. Every computation is represented by a data flow graph. The nodes of the graph represent operations, the directed arcs represent data paths. Execution of a graph is based solely on operand availability; each operation may begin execution as soon as all of its inputs are present. When an operation completes, the results are transmitted on the output arcs to the appropriate places. This view of task execution is a direct application of Petri Nets[22,23]. Figure 1 illustrates the graph representation of a simple VAL function for computing the mean and standard deviation of three inputs. The concurrency in this graph is limited only by the data dependencies inherent in the computation. If all inputs become available at once, four operations (plus, and three



```
**function** Stats (X,Y,Z: **real** **returns** **real**, **real**)
  **let**
      Mean **real** := (X + Y + Z) / 3;
      SD **real** := SQRT( (X² + Y² + Z²) / 3 - Mean² );
  **in**
      Mean , SD
  **endlet**
  **endfun**
```

Figure 1: A simple statistics function and its corresponding data flow graph.

squaring ops.) can begin, thus taking advantage of general concurrency available within the function. Vectorized concurrency is embedded in the execution in an unusual way in that the three squaring operations can proceed together. Finally, pipelining is possible if multiple executions of the function proceed together; separate sets of values can work their way through the graph at their own rate.

One major component of data flow research is addressing the issue of appropriate hardware for graph execution. Quite a few different hardware organizations are being proposed for data flow[9,12,13,20,21,24,28]. One feature common to all of these approaches (and probably essential for high performance) is that each operator is scheduled for execution on a particular processor by the <u>hardware</u> after all operands become available.

Software for data flow programming must help identify concurrency in algorithms (and their corresponding programs) and map that concurrency into graphs. A particular algorithm may have many different data flow graphs ranging from nearly linear to vastly concurrent. Data flow can only achieve ultra-high speeds if the graphs lean toward the latter. Many different software options are available, including writing pure data flow graphs[7,10,15], altering Fortran to generate graphs[14,20], using one of the newer concurrent languages[8,30], and (as always) defining a new language[1,4,24]. Regardless of the approach taken, the ultimate criterion for evaluating data flow software must be its effectiveness in generating highly-concurrent graphs for solving large problems.

The VAL language[1] is currently being developed by a group at MIT under the direction of Prof. J.B. Dennis. Their goal is a language suitable for the expression and translation of concurrency into data flow graphs. During the past two years Lawrence Livermore Laboratory has cooperated in the design and evaluation of VAL. Emphasis at LLL has been on the usefulness of VAL for current programming problems and its potential for future applications.

Overview of paper

This paper presents the design goals and major features of VAL, with examples and an analysis of its current status. To help illustrate use of the language, one sample program will be used throughout the text. Appendix I contains a complete program for adaptive quadrature[11]; all sample lines of VAL code are taken directly from this source and accompanied by function names and line numbers to help identify the context. The reader is advised to scan this appendix prior to continuing with the rest of this paper.

The next section of the paper presents the basic design goals for VAL. These goals had a strong influence on the basic structure of the language, which is detailed in Section 3. Section 4 discusses the implementation of the adaptive quadrature program, working from the basic algorithm through the design and including an analysis of its speed. Section 5 critiques the current strengths and weaknesses of VAL, and Section 6 identifies areas of continuing research for data flow software.

Design Principles

The ultimate goal of data flow software must be to help identify concurrency in algorithms and map as much as possible into the graphs. In response to this goal, two basic design principles have been followed in the development of VAL. First, the concurrency in a VAL program must be visible to both programmer and translator, but in an implicit way. Visible concurrency is critical becasue a programmer faced with a slow program must be able to distinguish between a poor algorithm and a poor translation. Implicit concurrency is critical because of the potentially large number of processors involved. The second design principle in VAL is to help programmers write correct programs. The complexity of a multiprocessing environment must not degrade the time and effort needed to arrive at a working program. This section examines these design principles in more detail, with a concentration on how they influenced VAL.

Implicit concurrency

The idea behind implicit concurrency is to get a programmer to write his code in such a way that very little effort is expended on the details of multi-processing. Unfortunately, the details cannot be completely hidden because the programmer must be able to understand what concurrency is possible (so later improvements can be made). These constraints argue for very simple rules for identifying concurrency; a programmer cannot be expected to remember odd or complex rules that a translator might need in order to handle "worst-case" programs. In VAL the rules for identifying concurrency are simple, but at the expense of requiring programmers to operate in a more constrained environment. Programmers write expressions and functions; there are no statements or subroutines available. This decision prevents the occurrance of any side-effects in the language. As a result, most concurrency is obvious by just examining a program.

The prohibition against side-effects in VAL plays an important role in achieving implicit concurrency. In data flow, concurrency is only permitted when the execution of two or more operations cannot affect each other's results. In a language with side-effects this property is often difficult to verify, so many potential concurrencies are sequenced to insure correctness. In VAL every operation is a function and the effect of each function is limited to returning values. Hence, a programmer can write expressions and functions that define his computation without having to explicitly state which ones can execute together. If one operation does not use output from another, no sequencing will be imposed.

The problem with eliminating side-effects is that most everyone's programming experience is tied to them. Assignment statements stand out as the principal example. The concept of updating memory (*ie* modifying variables) must be thrown away. Another well-known (but rarely recommended) language feature, aliasing, must also be banned. These two examples only begin the list of common features that must be avoided, but they illustrate the enormity of the situation. The key for a programmer to overcome this situation is adapting to a value-oriented environment. VAL uses values. In a value system, new values may be defined and then used in many places, but no existing value can ever be modified. While this approach may seem rash at first, there is precedent in Lisp[19], and in fact, the transition is not really that difficult.

Aids for system design

The other design principle embraced in VAL is to help programmers produce correct and time-efficient programs. The language should be clear and concise; all potential actions should be understood by the programmer. Hidden meanings and actions are avoided simply because the added complexity introduced by thinking about concurrency will cause more effort than before.

One major effect of this design principle was the selection of an algebraic-type syntax (*ie* infix operators in expressions, where possible). Looking at the previous comments, some variant of pure Lisp seems like a natural choice. However, the readability of Lisp drops off drastically when dealing with large arithmetic expressions and some standard control structures. Since most of the codes requiring the high performance of data flow involve extensive physics and large expression evaluations, a conventional algebraic language seems appropriate.

Other important influences of this design principle are visible in VAL's definition of legal programs and its handling of run-time errors. Scope rules and type-checking tend to be restrictive, and they must be tightly enforced. Responses to run-time errors are also well-defined. If an array subscript is out of bounds, or some computation causes an overflow, the language definition specifies precisely what actions will be taken by the language. This is in contrast to the standard Fortran view (which is shared by most languages) that the results are undefined and hence may be implemented in any way convenient for the compiler.

With this brief description of the design principles, the language features to be presented may be somewhat easier to follow. For further discussion of the motivation and reasoning behind VAL and other data-flow like languages, see Ackerman[2].

Language Features

This section surveys the major features of VAL, concentrating on the more interesting and unusual parts of the language. This presentation takes an informal approach to describing parts of the language, with extensive use of examples. Implicit concurrency resides in almost every feature of the language, and this fact is emphasized throughout the discussion. The principal areas of interest in VAL are: data types, values, basic expressions, parallel expressions, sequencing expressions, and error handling.

Data types

VAL data types are similar to those found in most structured languages. **Boolean**, **integer**, **real**, and **character** comprise the basic language-defined types. Each of these types carries an appropriate set of operators (*eg* +, -, &, |, ~ (not), >, =). New types may be constructed through **array**, **record**, and **oneof** (discriminated union) definitions. The principal differences between VAL types and types in most other languages are in the definition and use of constructed types, type-checking rules, and the existence of error values within every type.

VAL arrays are unusual because the array bounds are not treated as part of the static type; they are extra information associated with each individual array. An array's component type is the only type information associated with an array. Hence, array-structured types use the declarative form illustrated here:

```
    type Interval_list = array [ Interval ];                    (AQ,3)[1]
```

An interval_list is represented as an array of Intervals. Subscripts for arrays are strictly limited to integers. By eliminating the range information from the type, strong type-checking can be enforced without getting into the Pascal difficulty that arrays of different bounds cannot be passed to the same procedure. This option also increases possibilities for useful array operations, such as:

[1] This notation is used to indicate the location of the corresponding code in Appendix I. In this case the line is taken from the Adaptive_Quadrature function line 3.

1. shifting the origin:	**array_adjust**
2. adding elements at either end:	**array_addh**, **array_addl**
3. deleting elements at either end:	**array_remh**, **array_reml**
4. catenating two arrays:	||
5. merging two arrays:	**array_join**
6. setting array bounds:	**array_seth**, **array_setl**
7. testing array bounds:	**array_limh**, **array_liml**

The catenate operator is infix; all others are function invocations. Notice that general string manipulation is a special case of the above operations on arrays of characters.

Array construction in VAL takes an unusual form in order to improve possibilities for concurrency. All elements of an array can be specified simultaneously thus allowing all of the evaluations of array entries to proceed together. The syntax for array construction is an arbitrarily long list of ordered pairs, with the list enclosed in square brackets. The first of the pair is the index position in the array, the second is an expression representing the value to be held in that position. For example,

```
[ 1 : left_interval; 2 : right_interval ]        —       (BTI,16-17)
```

is an expression which creates an array with two elements. The values bound to the two identifiers become the values for the corresponding array locations. In this case, there is little concurrency (get the values bound to both identifiers) because the expressions are simple values. In general, however, all expressions inside the array construction can be computed simultaneously.

Record-constructed data types differ from most current approaches in both their static type properties and in their construction. The type of a VAL record is based solely on the constituent field names and their associated types. In particular, neither the order of definitions nor any type name bound to the structure influence the type matching process. Record construction is patterned after the array construction scheme which promotes concurrency.

```
record [  x_low : low;  Fx_low: lowv;                       (Int,6-9)
          x_high : high; Fx_high: highv  ]
```

The above expression builds a record with four fields. Each field name is followed by an expression representing the value to be entered in the record. As with arrays, all field values can be computed concurrently, which again in this case gains very little.

The final type constructor, **oneof**, permits discriminated union types as in CLU[17]. Values of this type can take on the appearance of different types at different times during execution. For example, in the adaptive quadrature algorithm, the result of analyzing an interval will either be two new intervals (the old one divided in two) or no intervals at all if the analysis is complete. Such a result type can be specified by:

```
type Result_info = oneof [ none : null;                    (AQ,4)
                           more : Interval_list ];
```

An instance of type Result_info can represent a value of type Interval_list (*ie* an array with two entries) or of type **null**. The tag fields (none, more) identify the constituent types. When an object gets a value, one of the tags must be specified along with the appropriate information. For example,

make Result_info [more: two_intervals] (BTI,18)

builds an object of type Result_info, whose constituent type is an Interval_list. In this case the identifier "two_intervals" has been previously bound to an array containing the two new intervals. The keyword **make** indicates that a value of the **oneof** type is to be constructed. Later, in the discussion of sequencing expressions, an operation (**tagcase**) will be introduced for getting at the information in a **oneof** value. As in the case of records, two **oneof** types are equivalent if the field names and the consitituent types are the same.

Within this framework for defining and using types, VAL imposes strong type-checking. In a function call, actual and formal parameters must have equivalent types. Automatic type conversions are <u>never</u> made by the language. Each language-defined function has specific types expected for each parameter and they must match. As an example, the add operator can take two **integers** or two **reals**, but it will not do mixed mode arithmetic. Similarly, when values are bound to identifiers, the type of the value must be equivalent to the declared type of the identifier (*ie* no **real** can be bound to an **integer** identifier). Rather than having implicit type coercion, the language provides functions that will do conversions for all of the reasonable cases. This approach was taken partly as an aid to programmers to make it completely clear where conversions occur in the program, and partly to simplify the language rules with respect to mismatched types. As an implementation note, all type checking defined in VAL can be done at compile-time.

The last unusual aspect of VAL's data types is the presence of special error values which are an integral part of every data type. Included among these values are **undef**, **pos_over**, **mis_elt**(missing array element), and **unknown**. They provide a simple mechanism for handling run-time errors without violating any type-correctness properties. The language also defines functions that test for the presence of these values, and all of the operators have well-defined propagation rules in the face of these values. More details on this subject are included in the section on error handling.

<u>Values</u>

The VAL programming philosophy is value-oriented, as opposed to the more traditional variable orientation. Most languages have concepts like "variables" and "memory updating", which imply that objects are mutable or modifiable. In VAL these concepts have been replaced by a value system where each object is immutable. The basis for this view resides in the target environment of data flow graphs. Once a value is computed by some operator and put on an output arc, the same value must be transmitted to all receiving operators. At first glance this orientation may seem very restrictive for programmers, possibly to the point of being unusable. That is, however, not true.

Within the framework of value orientation, the concept of binding values to identifiers is still possible. The important point is to prevent identifiers from being used as variables, hence the rule:

> Once an identifier is bound to a value, that binding must remain in force for the entire scope of access to that identifier.

This is commonly referred to as the single-assignment rule[2,24]. The true effect of this rule is that of permitting defined constants. Each time a scope of access (block, function, etc.) is entered new bindings can be made which remain in force until that scope of access is completed (block exit, function return, etc.). Notice that on each scope entry the identifier-value bindings may be different, which distinguishes them from normal constants.

Array and record operations are most affected by the value orientation. Structured objects must be viewed as single values. Hence, arrays and records can never be modified. The only option is to build a new array or record that has the same values in all of the old positions, except for the particular element that is to be changed. Enforcement of this rule is accomplished by requiring that every identifier-value binding be made to a full identifier (not to a field or subscript position). So for example, no binding can begin with " a[i] := ... ". To compensate for these difficulties in building a structure, VAL provides other schemes for allowing arrays and records to be completely built (in parallel if possible) and then bound to an identifier for use. The simple array construction operator was illustrated earlier as part of the description on arrays; a more complex array construction will be introduced in the discussion of parallel expressions.

<u>Basic expressions and functions</u>

The VAL language contains expressions and completely functional operators. There are no "statements" in the conventional sense. Every active language feature (*ie* non-declarative) is function-oriented in that it uses values provided by the current execution environment and its sole effect is to produce a set of result values. The rules for construction and use of expressions provide most of the protection against side-effects and at the same time help identify many forms of implicit concurrency.

The simplest expressions in VAL look just like expressions in most other languages. Infix operators (*eg* +, -, *, and /) can be applied to identifiers to compute some result. Among other things, the identifiers can represent simple types (**integer**, **real**, etc.), structured types (subscripted arrays and record field-accessing), and function calls. In function calls all parameters must be passed by value; this insures that the <u>only</u> effect of the expression is to compute a result. The amount of concurrency in an expression depends on the expression. The standard operator precedences are enforced by VAL, but values for the two operands for an infix operator can always be computed in parallel. So in the expression:

```
    (right - mid) * (rightv + midv) * 0.5  +                    {CQ,18-19}
    (mid - left) * (midv + leftv) * 0.5
```

```
let                                                              (BTI,4-19)
   left_interval   :  Interval
                   := record [   x_low : left ;
                                Fx_low : leftv;
                                x_high : mid;
                               Fx_high : midv   ];

   right_interval  :  Interval
                   := record [   x_low : mid   ;
                                Fx_low : midv  ;
                                x_high : right  ;
                               Fx_high : rightv; ];

   two_intervals   :  Interval_list
                   := [ 1: left_interval;
                        2: right_interval ] ;

in  make  Result_info [ more: two_intervals ]

endlet
```

Figure 2: A **let-in** expression illustrating temporary environment expansion. It builds an array containing information on two intervals.

every operation inside parentheses can execute simultaneously.

This general notion of an expression is enhanced significantly in VAL, so that expressions and functions look and act very much alike. Normally, a function displays the following characteristics--it receives inputs through parameters, defines some environment for execution (*eg* makes new function definitions or creates new identifier/value bindings), and then computes some expression which is its result. The environment defined by the function is only available during the execution of the function and it disappears when the function returns. Several VAL expressions also have this ability to temporarily extend the current execution environment by creating new identifier/value bindings. As with functions, the scope of these bindings is limited to the expression that makes them.

Environment extension within an expression is specified in VAL by giving a list of identifier/value bindings in a header to the actual expression. Both the type and value must be specified for each identifier; these bindings then remain in force until the final expression is computed. Using this view, it should be clear that each binding must be made to a different identifier (*ie* single-assignment). The simplest example of an expression which can extend the execution environment is the **let-in** expression. The code in Figure 2 takes information on two integration intervals and builds a value of type Result_info which was described earlier. The expression in the **in** clause generates the final result. All of the bindings made at the beginning of this expression disappear as soon as the result value has been computed.

The header of an expression may contain substantial amounts of concurrency. All expressions cannot generally execute simultaneously, because an expression may use identifiers bound earlier in the header. However, the only limit to concurrency is the data dependencies. In Figure 2, both record structures can be constructed in parallel before they are entered in the array. While this amounts to very little concurrency, several examples in Appendix I illustrate situations where the concurrency is very large (*eg* the Compute_Quads function).

VAL functions and expressions have also been extended to allow them to return more than one value. In conventional languages a function can only return one value. If others are needed, they come back through the parameters or global data. Since both are side-effects (unexpressable in VAL) a function is permitted to specify a list of results that it will return on every call. The most common way of generating a list of results is by specifying a comma-delimited list of expressions (other forms will be introduced in the discussion of parallel expressions). A list of results can then be used in a program anywhere that a list of the particular type is expected, such as multiple binding definitions and parameters to another function call. The former is illustrated below. The Compute_Quads function returns two values: an area calculation and a list of intervals that need further processing.

```
        new_area    :  real,
        result_data :  Result_list                               (Int,12-14)
                    := Compute_Quads ( list );
```

Since the types and order of the return parameters match the types and order of the target identifiers, this is a valid use of multiple expressions.

Parallel expressions

One of the most important forms of concurrency available in VAL comes from the **forall** expression. In conventional languages, looping constructs are often employed to compute some information when in reality all of the loop "passes" could execute in parallel without interference. Sequencing is imposed only because the language and target machine operate sequentially. Array construction, where each entry's value can be computed simultaneously, is one example. Another example, where the "passes" are not completely independent, but can be organized for faster execution, is summing a long list of numbers. A binary tree structured evaluation of the partial sums permits the entire summing operation to be completed in log time instead of linear time. Both of these concurrencies are representable in VAL.

Parallel array construction is expressed in VAL by using **forall** with a **construct** clause. Figure 3 illustrates this usage. The body of the expression executes once for each element in the range of execution specified at the top of the expression. The **construct** clause implies that the value of the **forall** expression is an array; the indicies of the array are the range elements and each element value is determined by the expression following **construct**. In general, several ranges can be specified in the header. In this case the body will execute on the cross-product of the index values and the result will be an array of

```
forall i in [ array_liml(list), array_limh(list) ]
   { list of header definitions }                                (CQ.2-29)
   construct  if done then make Result_info [none:nil]
                       else Build_two_intervals( left, leftv,
                                                 mid, midv,
   endall              endif                     right, rightv )
```

Figure 3: **Forall** expression that generates all elements of an array simultaneously.

arrays. If more than one **construct** arm is specified, the **forall** becomes a multiple expression yielding a separate array for each arm.

Fast expression evaluation through binary tree structuring is possible in VAL through the use of an **eval** clause in **forall**. This is only possible when one associative, dyadic operator is to be applied to a sequence of values. The following piece of code, when inserted in Figure 3 directly above the existing **construct** clause, adds up areas for intervals with acceptable approximations.

```
eval plus  if done then  new_area
                   else  0.0     endif                      (CQ.22-23)
```

The operator specified immediately after **eval** becomes the associative operator on the list of values. The list of values comes from the expression following the operator—each execution in the **forall** range produces one value for the list. Notice that the result of an **eval** clause is one value, as opposed to one array in the case of the **construct**. Currently, only operations known to the language may be identified as the special operator (**plus**, **times**, **max**, **min**, **and**, and **or**). The **eval** arms may be used in exactly the same places as **construct**. They may have multiple occurances—each one yielding a single value as part of the multi-expression **forall**. **Eval** and **construct** arms may be interleaved in any order. In these situations the order of the result arms defines the order of the **forall** results.

Sequencing expressions

While most of VAL is designed to promote concurrency, sequential execution is enforced in some places. Sequencing is imposed either to insure logical correctness or to avoid initiating some operations whose results will never be usable. These situations can arise with all three VAL conditional expressions. **If-then-else** expressions permit basic selection of expression results. The **tagcase** expression provides a means of interrogating values having a **oneof** type, while maintaining type-correctness. The last expression, **for-iter**, implements loops that cannot execute in parallel because values produced in one pass must be used in the next. In all of these expressions, some conditional clause (**if** boolean expression, **tag** of a **oneof** type, or the loop restart test) controls activities that will follow. VAL imposes the explicit constraint that no result operation will begin execution until the conditional clause selects the appropriate action.

The **if** expression is akin to the standard **if** statement found in most languages--only in VAL each result clause must be an expression (or multiple expression). A Boolean expression provides selection between a **then** and **else** expression. To insure static type consistency, VAL requires that the expressions defined by each result option be type equivalent. If a programmer needs to return two different structures depending on the test condition, he can employ the **oneof** type constructor. An **if** expression with a **oneof** result can be found in Figure 3. The **construct** clause produces an array, where each element is of type Result_info. The **then** clause of the **if** produces a Result_info value having a "none" **tag**. The **else** clause uses another function which produces a Result_info value having the "more" **tag**. Hence, regardless of the result clause chosen at execution, the type of the result is certain to be Result_info.[2]

The **tagcase** expression permits access to information on an identifier whose type is **oneof** without violating type correctness. The strategy is to interrogate the **tag** field to discover the value's true type and then use a multi-way branch to select the appropriate action. It is illustrated below:

```
tagcase interval_data := result_data[loc]                    (BL,9-13)

        tag none : new_list

        tag more :   new_list
                  || [ 1:  interval_data[1]   ]
                  || [ 1:  interval_data[2]   ]

        endtag
```

This expression examines a value of type Result_info (namely, the value bound to result_data[loc]) and the result of the **tagcase** depends on the tag associated with that value. The assignment in the header is a semantic maneuver to preserve type correctness; in each **tag** arm the identifier "interval_data" takes on the type associated with the corresponding **tag** field. This permits the information within a **oneof** object to be accessed properly for each possible case. The result of the above **tagcase** is a list of intervals, either the existing list (if the result_data had no intervals), or the existing list with the two intervals catenated at the end (each new interval is first transformed into a one element array to establish type compatability for the catenate operator). Notice that as in the **if** expression, all arms of a **tagcase** must generate the same type of expression.

---

[2] An alternate solution is to have the **if** expression return an array of intervals. The **then** clause would return an array with zero elements and the **else** clause would contain two elements. This is legal since type checking does not involve array ranges.

```
        for
             area : real                                            (Int,2-26)
                  := 0.0 ;
             list : Interval_list
                  := { initial interval info. } ;
        do
             let
                  new_area        :  real,
                  result_data     :  Result_list
                                  := Compute_Quads ( list );

                  new_intervals   :  Interval_list
                                  := Build_list ( result_data );
             in
                  if array_size ( new_intervals ) = 0

                       then   area + new_area

                       else iter
                            area  := area + new_area;
                            list  := new_intervals;
                            enditer
                       endif
             endlet
        endfor
```

Figure 4: Illustration of the **for** loop.

The last major form of expression in VAL is **for-iter** which permits sequential looping. In this expression values can be transmitted from one pass through the loop to the next. This transmission can <u>only</u> occur by defining loop variables and then making assignments to them just prior to the beginning of the next pass. Notice that the conventional method of retaining information from one pass to the next, assigning to global variables, is not permitted in VAL. Within a **for** loop, objects can only be bound to identifiers defined <u>inside</u> the loop. The program segment in Figure 4, illustrates the use of **for**. This loop cycles on the following three steps:

1. Find intervals with acceptable area estimates
2. Organize remaining intervals in a new list
3. Stop when there are no more intervals

The loop parameters (declared in the heading) carry information from one pass to the next, namely the area accumulated so far, and the list of intervals requiring further processing. The body of **for** must be an expression. In this case, when all intervals are done, the value computed in the **then** clause will be returned as the final value of **for**. Iteration is allowed through an **iter** clause that can be used in place of any result expression. The assignments in the **iter** clause specify new bindings to be made to the loop parameters prior to reexecution.

In all of the conditional expressions concurrency is limited by the controlling boolean expression. Only the required arm of an **if** or **tagcase** expression will execute. Similarly, the next pass in a **for** loop will not begin until an **iter** clause is evaluated. This strategy for handling conditionals may cost some time in terms of execution speed (evaluation of result clauses cannot be overlapped with the controlling conditional) but it almost certainly saves in that only usable computations are begun, and hence there is no need for abort procedures.

Error handling

One unfortunate effect of having a language that emphasizes concurrency at the operation level is that it is difficult to stop a computation in midstream, since the program can be concurrently executing in many places. This problem requires a different error-handling mechanism from conventional languages. A VAL function can never abort in the middle of its calculation; it must either terminate generating legal type-correct results, or run forever (due to an infinte loop). This characteristic is possible because every data type contains error values in addition to the values commonly associated with the type. When an operator cannot carry out its assigned task (*eg* multiplication underflows, or accessing an undefined array element) it returns the appropriate error value. All operators are defined over the complete range of the appropriate types, so the error information can propagate through. Language-defined functions permit testing for the presence of error values so a programmer can choose to provide alternate results when an anticipated error arises.

This strategy is still only of marginal use when actually tracing back some error. It is unreasonable to insert code to test the validity of every operator, yet failing to do so may obscure where an error actually arises. One proposal (which is not yet officially in VAL) is to have the language define a special audit trail associated with each error value. This trail would provide information about where an error originated and what transformations it went through during propagation. Since this audit trail would constitute an unusual side-effect if it were accessible within the program, one constraint on this information is that it could only be dumped to some system file for later access. While this is not an ideal solution, it would permit reasonable testing without violating any language principles.

Adaptive Quadrature in VAL

Programming in VAL can be illustrated by examining its use in solving some problem. Adaptive quadrature has been selected because it contains several interesting levels of concurrency, the algorithm is relatively simple to understand, and the solution in VAL gives a balanced view of the language. Three basic parts of the programming process are of interest here. Algorithm selection requires a thorough understanding of how the algorithm works, and where the potential concurrencies and "slow spots" reside. Program design stresses correctly mapping the algorithm onto VAL, and at the same time insuring that the algorithm's concurrencies are still visible in the data flow graph. Finally, analysis of the program is critical for determining improvements and variations that would either favor more accurate or faster results.

Review of adaptive quadrature

Adaptive quadrature is an algorithm for computing the integral of some function, $\mathcal{F}$, over a specified interval. $\mathcal{F}$ is assumed to be continuous over the interval, however the only way to gather specific information on the function is to evaluate it at various points within the interval. Adaptive quadrature uses a simple approximating function, APPROX, (*eg* the trapazoidal rule) and a scheme for sub-dividing intervals (*eg* bisection). The algorithm begins by applying APPROX to the interval as a whole to get one integral approximation, and then sub-dividing the interval and applying APPROX again to each subinterval, summing the results, to get a second integral approximation. The two approximations are compared, and if sufficiently close, the latter one is taken as the integral of the function. Otherwise, the entire algorithm is applied independently to each sub-interval until acceptable results are found, and then all of the partial results are added for a final result.

The effectiveness of adaptive quadrature in computing an integral relies on its sparing use of calls to $\mathcal{F}$. In most applications, function calls are likely to be very expensive, often completely dominating the cost of the integration calculation. By treating each split interval independently, most intervals are likely to converge to a good approximation quickly—without using many function calls. Hence, the cost of function calls is only incurred where the information is likely to make substantial improvement in the approximation. For more information on this algorithm see de Boor[11].

One of the open questions regarding this algorithm is the choice of an acceptance criteria for deciding when a particular approximation should be taken. Rice[25,26,27] proposes a myriad of options that trade off accuracy for execution time, the principle differences being in the amount of information required in order to make the accept/reject decision. Simple information, like the interval's size and overall acceptable error tolerance, is sufficient to construct a converging solution; however, extra information, like error extimates for completed intervals, may lead to a faster solution.

The important concurrency in adaptive quadrature rests in the independent handling of each of the sub-intervals. Once the process has iterated through several sub-divisions there are many intervals that can be analyzed simultaneously. In particular, each interval analysis requires one invocation of $\mathcal{F}$ (at the midpoint, when using bisection and the trapazoidal rule) so all of those function calls can execute concurrently. Since these calls dominate the cost of execution, they are almost certain to control overall speed of this algorithm on a multiprocessor.

Program design

The goal in implementing an adaptive quadrature program is to arrange the computation so that all of the intervals can be operated upon concurrently. One approach for achieving this goal is to use recursion. As the interval splits occur, the independent function invocations can proceed simultaneously. Unfortunately, VAL does not currently permit recursion so another design scheme must be used.[3] The design implemented in Appendix I uses a list structure (represented in VAL as an array) to keep track of the intervals that require further processing. This structure was proposed by Rice[25] with the idea that in a multiprocessor environment the processors would be constantly taking intervals from the front of the list, and adding the split intervals to the rear. This implementation differs in that it processes the entire list simultaneously, and then constructs a new list comprising all of the split intervals. The issue of an appropriate interval acceptance criterion is temporarily side-stepped by having the user supply a stopping condition routine in addition to $\mathcal{F}$.

Concurrent processing of all intervals on the list is accomplished with a **forall** expression that returns an array of results (one result for each interval processed). The code for this part of the program is in the Compute_Quads function. The range of the **forall** is set to the length of the list and the body computes the interval analysis for one interval (which includes invocation of the $\mathcal{F}$ function). The result of each interval analysis is returned in an array entry, as specified in a **construct** clause. The concurrency of the algorithm is obvious because the critically expensive function call is embedded in the parallel expression feature.

The only weakness of this solution is the return of the new list of intervals for the next pass. From the dividing nature of the algorithm we know that the new list may contain anywhere from zero to twice the number of intervals that were input. Unfortunately, the **construct** arm of a **forall**

---

[3] This language constraint may change in the future. There will be further discussion of this point later in the paper.

must return an array that has exactly the range of the loop control identifier (*ie* the same length as the input array). To compensate, elements of the output array are **oneof** structures whose contents can vary between array elements. Each entry has either no intervals or two new intervals. Since this structure is not the same as the list of intervals input to the **forall**, a restructuring step must take place. This is done in Integrate by calling Build_list.

Program analysis

This solution has both good and bad points when considering concurrency. The program does permit many simultaneous evaluations of the target function. Since that part is assumed to be the most expensive step in the operation, that should have a very good impact on the execution time. On the negative side, the list restructuring (in Build_list) is a sequential operation and hence, time-consuming. One possible remedy for the future is to add another form of result generator to the **forall**, one which would permit the construction of arbitrary length arrays in parallel. For now though, we must settle for the position that the sequential portion of the program, while inelegant, is unlikely to dominate the cost of the computation.

Another important issue in analysis is how the various interval acceptance criteria proposed by Rice would affect the concurrency of the program. Rather than discussing each option in detail, several principles will be stated that should help the reader understand the nature and use of VAL more clearly. The primary concern is to identify those pieces of information that can and cannot be passed to some "Stopping Condition" function. As currently structured, the decisions about interval splitting are made during the processing of each interval--thus they are done in parallel. At that point, only information on the current interval and information from the previous passes can be used. For example, instead of providing the interval's size, any or all of the following information could be passed (assuming appropriate changes in the code to accumulate these statistics):

1. interval endpoints
2. partial area accumulation as of last pass
3. partial error estimate as of last pass
4. number of intervals remaining in the list

These are some types of information that could be useful in designing different stopping conditions. Absent from this list is any information about analysis on the current pass through the intervals. It may be possible that almost all of the intervals in the current pass get very accurate estimates and so the few remaining ones can afford to be less precise. That information will not be available until the next pass, because the analysis of all intervals is concurrent.

An alternate solution design would permit information about the current pass results to be available immediately, but at the expense of some speed at execution. The decision on whether to split an interval or quit could be moved to the Build_list routine. This routine is already sequentially oriented, so it would not drastically change the program. The **for** loop could be changed to include some loop variables that keep information on the latest error and area estimates. The cost of this approach is that the decisions on whether or not to split each interval would now be done sequentially rather than simultaneously. Notice, however, that this cost is directly related to the type of algorithm that is desired. A stopping condition that needs absolutely current information about the state of the approximation imposes more constraints on the solution.

In summary, one of the main concerns of a programmer using VAL will be in understanding the timing constraints of potential solution candidates and finding an implementation that adds as little as necessary. In the adaptive quadrature problem, the critical concurrency must be the multiple function evaluations. The proposed solution permits this concurrency with only a modest amount of effort. The program variations to accomodate different stopping conditions can be done with the corresponding concurrencies available to each choice. The one sequencing constraint introduced by programming in VAL is the list restructuring at the end of each pass.

Language Critique

This section gives a detailed analysis of the VAL language. Its principal strength, relative to other language candidates, is VAL's ability to display large amounts of general concurrency in an implicit way. Possible side benefits may accrue from research in functional programming; most of the verification techniques discussed in that environment are directly applicable to VAL. On the negative side, the language design is not yet complete. Some extensions must be added, the critical one being I/O facilities. Other facilities would be beneficial to include--among them recursion.

Strengths

Implicit concurrency stands as the most significant aspect of VAL in relation to other language options for a data flow environment. Languages having explicit concurrency, like Concurrent Pascal[8] and Modula[30], have two serious drawbacks when considered for data flow. First, concurrency at the individual operator level requires far too much effort for a programmer to express. In VAL this concurrency requires no extra effort. As an illustration, consider the Compute_Quads function in Appendix I. This function abounds with operator level concurrency in every expression. In order to express this concurrency in Concurrent Pascal or Modula would require a very large number of process definitions--without question an unreasonable strategy. The second drawback to languages with explicit concurrency is that they require explicit synchronization. Again, refer back to the Compute_Quads function. The end of the **forall** expression requires merging of information from all of the different range evaluations into one array and also into one sum. This is actually a massive amount of process interaction, and would require liberal use of semaphores or monitors in any current languages. VAL has no need for synchronization tools because the interaction is implicit, thus saving programmers from the tedious and often difficult chore of defining correct interactions.

Another language option for data flow is to take standard Fortran and write a new compiler that generates appropriate graphs. The problem with this form of "implicit" concurrency is that both programmer and compiler would have difficulty identifying concurrency. The abundant side-effect possibilities would force even a smart compiler to pass over certain concurrent operations because their independence could not be assured. VAL avoids the problem because side-effects are not expressable.

A final data flow language option is to employ one of the existing languages used for multiprocessing. LRLtran[18] is Lawrence Livermore Laboratory's version of Fortran that compiles code for the CDC-7600, CDC Star, and Cray-I. Another candidate is Glypnir[16], which compiles code for the Illiac IV. These languages suffer one major drawback in that they concentrate on vector-type concurrency to the exclusion of all other forms.[4] VAL permits vector concurrency, in fact, strongly encourages it through the **forall** expression, but not to the exclusion of other forms. All forms of concurrency can be exploited at the same time, without special notation or effort on the part of the programmer.

Another strength of VAL is its strong resemblance to applicative (or functional) programming systems, such as the one proposed by Backus[6]. Applicative systems stress programming with true mathematical functions. Some early results show possibilities for proving properties about programs, which may lead to correctness and/or verification schemes. The functional nature of VAL programs contains many of the same properties as those emphasized in applicative systems. The main difference between the two approaches at this time is that VAL explicitly considers data and data types as part of the language—applicative systems do not. In principle, this difference means that functions in applicative systems are defined over the complete universe of inputs while VAL functions are only defined over the domain of the input types. In practice, it means that we need to limit analysis of of VAL functions to situations that are known to be type-correct, which is not at all unreasonable.

Weaknesses

The weaknesses of VAL tend to fall into two categories. The first, and more serious, are those shortcomings for which a good answer is necessary but currently missing. Principal among these deficiencies is the lack of I/O. The remaining weaknesses involve limitations in the language which were originally imposed to avoid possible problems or conflicts later. Some of those limitations may not be necessary and yet they constrain useful programs (*eg* no recursion).

The lack of I/O facilities in VAL is currently the most disturbing problem. The difficulty with adding I/O is its inherent side-effect nature. Output routines do not have a value, they modify some external environment. Input routines use an external environment to determine their results, so they are not functions either. The problem can best be illustrated in VAL by considering the adaptive quadrature program. What would happen if the Evaluate_function routine provided by a user invoked either input or output? Since the solution is designed to have many calls simultaneously executing, what order (if any) could be imposed on access to the external environment? The one problem with having completely functional environment is embedding it in a non-functional computing world.

The only answer to this problem in the current version of VAL is to treat I/O files as standard data objects; the input file must be completely present at program initiation. All I/O functions would have to be given a

[4] This weakness is not limiting in their current environment because the machines mentioned only have the power to exploit vector concurrency.

file (*ie* a record of values) to be operated on at every invocation, and return a different file on completion. Thus **read** and **write** would effectively become pop and push operations on files structured as stacks. This solution works because the "files" are not global data that are updated; simultaneous **writes** would create multiple files, each having a different final entry. Programmers would then have to deal with the problem. This solution is unacceptable because it disallows any interaction with the program during execution.

Another weakness in the current definition of VAL centers on the use of **eval** clauses in a **forall** expression. Currently only six language-defined operators can be used as the result merging operation. Other operations might be very useful in that position. If catenation were permitted, the adaptive quadrature solution could be shortened (and sped up) by having Compute_Quads build a new interval list directly, rather than use Build_List. One could also envision other operations users would need to write, that would be beneficial. In general, user-defined functions in the **eval** position are prohibited because associativity cannot be assured. Without this property, tree-evaluations can produce unexpected results. An interesting anomaly exists in that two of the six acceptable operations (**plus** and **times**) are not usually viewed as being associative in finite precision arithmetic. Prescaling the operands may circumvent this problem, but at a substantial cost in time (namely, a tree-structured scaling operation).

The remaining weaknesses of VAL are related to unduly constraining the language in several places. Probably the most important of these is the prohibition against recursion. Recursion in a conventional language is a powerful tool that can greatly simplify the presentation of some algorithms. In a parallel processing language the advantages are even greater because simultaneous recursive calls can display substantial concurrency.

The recursive version of Integrate shown in Appendix II could replace four functions (Integrate, Compute_Quads, Build_list, and Make_two_intervals) in Adaptive_Quadrature and still compute the same function. The obvious advantage of this version is that it is substantially shorter than the previous version while specifying the same computation. Even more significant, this version contains more potential concurrency. The original version required that all work on a particular interval list be finished before the next pass began. In the recursive version there is no such restriction, and there is no slow spot in the calculation due to a linear sequencing of operations. In fairness, however, it is important to notice that in a recursive version there are more limitations on the type of information that could be given to a Stopping_Condition function. Only information on the current interval (*eg* size, bounds, and area estimates) and initial constraints (*eg* desired error bound for the entire problem) can be used in the recursive version.

Two other possibly unnecessary language constraints are worth mentioning. A VAL function can only invoke functions which it has defined and functions declared **external**. This rule helps insure that function blocks are logically self-contained and thus easily transported to different environments. However, it hampers program design in that if some utility function is needed by two other functions, the utility must be promoted to the highest access level—**external**. For this reason, the Adaptive Quadrature program is not quite legal. For readability, that program has all functions declared at the outermost scope level. With this structure the functions should not be able to call each other. A simple program restructuring could nest the Adaptive Quadrature functions into an acceptable form, so the issue is only one of asthetics and convenience. The other annoying constraint is that functions cannot be passed as parameters. In conventional languages such a feature is difficult to implement due to environment control issues; but the VAL environment creates no such problems. Function parameters would improve the adaptive quadrature program by allowing the two user-defined functions to be passed rather than defined globally, thus permitting users to name their functions more appropriately.

Continuing Research

Research in the area of data flow computing, including work on data flow languages, is still in its very early stages. The work reported in this paper represents some current ideas about how to communicate concurrent algorithms. VAL has proved useful in writing some commonly used applications programs. However, there is much language work that still needs to be done. Three different directions need to be pursued in some detail: language modifications needed to address current weaknesses, compilation techniques for optimizing translation to data flow graphs, and programming techniques for using the language to its fullest.

Language improvements

Many of the current weaknesses in VAL can be handled by relatively minor adjustments to the language definition. These points have already been made and therefore will not be repeated here. The one major problem in VAL as it is currently defined is the lack of I/O facilities, a problem which is not easily remedied. The most promising solution for adding I/O facilities to VAL is the stream concept. This facility has been suggested for use in data flow languages[4,29], but the details have yet to be worked out thoroughly.

A stream is a sequence of values. The only stream oprations are **append** to the end of a sequence and **remove** from the front of a sequence. Because these operations only deal with the ends, a function could operate on a stream without having the entire sequence available. Normally streams would be used in a data flow language to set up a "pipeline" between two functions. The first function would build a list by appending results to a specific stream. The second function could then remove the results as they are generated and continue processing. Because the second function need not have the entire stream defined before it begins using it, the second function can begin execution as soon as any element is put in the stream. This implies that both functions can operate simultaneously with the receiver function limited in speed by the rate that the sender function generates values.

Streams may be a reasonable way to treat I/O. The language could define two streams (one for input and one for output) that would be connected to the outside world. Since the values for a stream need not be known until requested, the interface to the outside world could put the correct value on the input line, once it sees which information is being requested. Output could be transmitted directly out, not waiting for the end of the program to transmit everything. While this approach seems feasible, many details still need to be settled. In particular, how are streams defined, connected, and disconnected from functions as they are defined in VAL? Also, secondary storage reintroduces the problem of side-effects, now possible by writing to "memory" and then reading it later. How can the needs of users to do I/O be reconciled with the fact that it is an inherent side-effect?

Compilation techniques

A second area where substantial work needs to be done is the development of compilation techniques for translating into data flow graphs. Many of the features in VAL can be translated directly into graphs with very little effort. However, there are a few constructs which pose problems for even simple translation. Completely open is the area of optimizing techniques for graph translation.

One of the trickiest questions with respect to compiling into graphs is—what form should the graphs take? In particular, must the complete graph be defined and created by a compiler? If the graph must be completely specified at compile-time, then some concurrency in a program may become sequential in the graph. Take for example, the **forall** expression. If the range of the expression is only computable at run-time, how can the compiler know how many replications of the expression header to make? Similarly, if two invocations of a function can take place in parallel, how does one get multiple copies of the graph? Both of these situations arise in the adaptive quadrature program. The highly critical **forall** expression in Compute_Quads has a range that varies at run-time with totally unpredictable behaviour. Similarly, the number of potentially concurrent calls to Evaluate_Function is not known until execution. With static graphs, some sort of "reentrant" graph concept may be necessary.

If the complete graph need not be specified by the compiler, then some notation must be developed for describing how pieces of a graph can be constructed during execution of the graph. This appears to be non-trivial. A major problem with graphs that can grow at run-time is the potential for deadlock. There is certain to be some upper limit (albeit very large) on the amount of space a data flow computer will have for holding a graph. If a program yields a large amount of concurrency by using language features that replicate graph portions, it is conceivable that the program could attempt to expand past the limits of the space, and hence deadlock. These arguments suggest that a compiler for VAL would have to have some extra help from the programmer above and beyond the program input. This interaction would help the compiler generate code that would get as much concurrency as possible while controlling the use of graph space.

Optimizing compilers for a data flow language pose even more problems. A VAL program may contain some forms of concurrency that are difficult to reflect in a graph. Array usage is a case in point. An array may be built in one **forall** expression and then immediately be used inside another. From a data and control dependency viewpoint, as each element of the array is computed, the corresponding range index of the following **forall** may be clear to execute. If the expressions are compiled so that the entire array is built in one expression and then handed over to the next, that concurrency is lost. Since many number-crunching applications have exactly this type of structure throughout the program, such a loss could be significant. This point is discussed in more detail by Woodruff[31].

Programming techniques

In order to make data flow a truly useful computing approach, better programming techniques must be developed. Currently several efforts are underway to take existing production codes at Lawrence Livermore Laboratory and translate them into equivalent VAL programs. Such work is a start toward understanding the best ways for using VAL. We need to know more about the kinds of concurrent algorithms that are easily done in VAL, and more important, those that are not. Work also needs to be done on developing new concurrent algorithms for solving problems, rather than taking the old approaches and mapping them onto a new programming language. Finally, we need to consider the impact of concurrent languages on the code testing process. Until program verification becomes a common practice, debugging is a fact of life. In a highly concurrent environment it may be sheer torture.

Every time a programmer is introduced to a new language a substantial amount of effort must be made to get him into the appropriate "mind-set" in order to use it well. Part of this effort is learning how to reason about concurrent algorithms and understanding how to convey that information through the compiler. Since speed of execution is critical, it is important to be able to identify the "slow" spots in a program and recode them for better performance. Right now we know very little about general techniques for doing this kind of analysis.

In the area of finding new concurrent algorithms, very little is known. Arvind and Bryant[3] studied one LLL code for solving partial differential equations and found that while several parts contained massive concurrency, the overall time-limiting factor is a very sequential matrix inversion algorithm. A more concurrent algorithm (even one that requires substantially more computational effort) could remove this limitation and yield a much faster program. This is just one example of where more research on concurrent algorithms could be put to use.

Finally, there are practicalities. Debugging has often been considered one of the hardest and most time consuming portions of a programming project. If it is so difficult in a sequential environment, how much worse will it be in a parallel one? To make matters worse, some common debugging tools are probably not available. It is unclear how one could breakpoint a VAL program during execution and make any sense out of the current state because the current state may be active in literally thousands of places. While developing useful debug tools for this environment may not be as theoretically interesting, it would certainly be extremely useful.

Summary

Many of the recent efforts to increase the effective speed of computers have centered around the idea of multiprocessing, particularly on programs that currently take several machine hours to run. Unfortunately, current languages still impose a sequential mind-set on the programmers. Concurrency is treated as a special case that requires significantly more effort from the programmer both in terms of logical understanding and actual code to represent it. The basic principle of data flow computing is that most programs have a vast amount of concurrency embedded in them (mostly at the individual operation level) that could be exploited on hardware that has a large number of processors. In this sort of environment, current languages are inappropriate because they discourage concurrency in programs and even hamper thinking in those terms.

The VAL language provides a hospitable environment for dealing with concurrency. A VAL program automatically executes to its full level of concurrency, which is limited only by data dependencies, control dependencies, and the number of available processors. The data dependency limits are imposed by the algorithm employed in a program. The control dependency limits help avoid wasting processor usage on operations whose results are not necessary. In VAL, concurrency is the rule, and sequentiality is the exception. Programmers are encouraged to think about their algorithms and identify portions that can execute simultaneously. Once those sections are known, very little effort is needed to display the concurrency in the code.

One of the major strengths of VAL is its ability to represent many different types of concurrency. Vectorizing, pipelining, and independent operator execution are equally available for use. A programmer need not tune his program for one specific type of concurrency to the exclusion of all others. Many forms can be used simultaneously. This freedom should also be of great value in the development of new algorithms for solving old problems. It is interesting to note that the basic design principles behind VAL are exactly the same as those advocated by researchers in the areas of semantics and program verification. VAL prohibits <u>all</u> forms of side-effects and aliasing, insisting on functional program features. The motivation for this approach was to insure fast and maximal identification of operations that can safely proceed concurrently. While verification was not an explicit design goal in VAL, its functional characteristics may be very amenable to the current program proving techniques.

The restraining caution in this work is that <u>all</u> data flow research is in its infancy. VAL will require substantially more work on its definition, translation, and techniques for use before it will be a viable

production-level tool. Work is currently in progress at MIT to address the remaining language definition problems. At Lawrence Livermore Laboratory the emphasis has been on the techniques for use. By taking current production codes that dominate the computing time, and translating them into VAL we hope to develop a better understanding of both the physics problems and methods for solving them in a data flow environment. The remaining area, translation of VAL into data flow graphs, may turn out to be one of the more difficult problems. Basic translation of VAL into graphs is possible; however, finding good optimizing schemes may be well in the future.

Appendix I: Adaptive Quadrature Program

```
function Adaptive_Quadrature ( low,high : real returns real )

  % Adaptive Quadrature computes the integral of a function, F,
  % on the range: low ... high ( the inputs). The user must provide
  % two functions. Their characteristics are described in the external
  % declarations below. Most of the design and implementation
  % for this program are discussed in the paper under the section:
  % Adaptive Quadrature in VAL. Comments at the top of each function
  % explain the operations performed within that function.


  type Interval = record [ x_low, Fx_low, x_high, Fx_high: real];
       % An interval is represented by its endpoints and the values of
       % the function at those endpoints.

  type Interval_list = array [ Interval ];
       % A list of intervals is represented by an array of intervals.
       % The list always begins at the 1 index position and extends
       % as high as necessary.

  type Result_info = oneof[ none: null; more: Interval_list ];
       % The result of analyzing an interval may be that no new
       % intervals are generated (none tag) or two new intervals
       % are generated (more tag, list has 2 entries).

  type Result_list = array [ Result_info ];
       % A result list holds all results from the processing of one
       % interval list. Notice that each array entry is a oneof
       % type and the tags need not match between entries.

  external Evaluate_Function ( x:real returns real ) ;
       % This must be a user provided function for computing F(x)
       % where x will be on the range:  low ≤ x ≤ high .

  external Stopping_Condition ( area1,area2,interval_width : real
                                        returns boolean            );
       % This must be a user defined function for deciding if
       % "area1" and "area2" are close enough to permit the latter
       % to be used as the approximation for an interval of width
       % "interval_width". The program could be easily modified
       % to allow this function access to other information.
```

```
 %  To improve readability, each function is presented separately, at the
 %  same syntatic nesting level.  This structure is not legal in VAL
 %  because functions are not permitted to call other functions declared
 %  at the same level.  These functions could easily be reorganized into
 %  a correct VAL program, if necessary.

   function Integrate(low,lowv,high,highv: real returns real)

        %  The algorithm for AQ is implemented by keeping:
        %
        %       1. a list of intervals for which an acceptable
        %          approximation has NOT been found,  and
        %
        %       2. a running sum of areas for intervals with
        %          acceptable approximations.
        %
        %  Execution proceeds in the following cycle :
        %
        %      1. Compute a new approximation for each interval on list
        %            =>  if acceptable, accumulate area
        %            =>  if not, divide interval into two intervals
        %      2. Restructure unresolved intervals into a new list
        %      3. If no intervals on list, return total area
        %      4. Repeat cycle with new list

      for
           area :  real
                := 0.0 ;
           list : Interval_list
                :=  [ 1: record [ x_low    : low;
                                  Fx_low   : lowv;
                                  x_high   : high;
      do                          Fx_high  : highv ]    ]
           let
                new_area         :  real,
                result_data      :  Result_list
                                 := Compute_Quads ( list );

                new_intervals :  Interval_list
                              := Build_list ( result_data );
           in
                if array_size ( new_intervals ) = 0

                     then    area + new_area

                     else iter
                          area  := area + new_area;
                          list  := new_intervals;
                          enditer
                     endif
           endlet
      endfor
endfun
```

```
function Compute_Quads ( list : Interval_list returns real, Result_list )
    %       Compute Quads receives a list of intervals for which
    %  acceptable area approximations have not been found.
    %  For each interval ( all are done in parallel ) its
    %  midpoint and function value at the midpoint are computed.
    %  The trapazoidal rule is applied separately to each half
    %  interval, and once to the interval as a whole.  If the
    %  two approximations are acceptable ( as defined by the
    %  Stopping Condition ), that area is added into the other
    %  acceptable areas for this list.  Insufficient approximations
    %  return two sub-intervals to be operated upon later.
    %       The initial assignments below ( left -- rightv )
    %  are purely for readability.  Areas are only accumulated
    %  for acceptable approximations ( done in "eval plus " ).
    %  Due to strong type checking, the list of results must be
    %  elements of a oneof type representing either:
    %
    %        1. no new intervals    ( "none" option )
    %    or  2. two new intervals   =>  "more" option .

    forall i in [ array_liml(list),array_limh(list) ]
          left        : real
                      := list[i].x_low;
          leftv       : real
                      := list[i].Fx_low;
          right       : real
                      := list[i].x_high;
          rightv      : real
                      := list[i].Fx_high;
          mid         : real
                      := (left + right) / 2.0;
          midv        : real
                      := Evaluate_Function(mid);
          old_area    : real
                      := (right - left) * (rightv + leftv) * 0.5 ;
          new_area    : real
                      := (right - mid) * (rightv + midv) * 0.5 +
                         (mid - left) * (midv + leftv) * 0.5 ;
          done        : boolean
                      := Stopping_Condition(old_area,new_area,right-left);
       eval plus   if done then   new_area
                           else  0.0         endif

       construct   if done then make Result_info [ none:nil ]
                           else Build_two_intervals(left,  leftv,
                                                    mid,   midv,
                                                    right,rightv )
                           endif
       endall
endfun
```

```
function Build_list (result_data: Result_list returns Interval_list )
    % Build_list takes a list where each element is
    % either :
    %
    %     1. empty,     or
    %
    %     2. a pair of interval descriptions
    %.
    % and returns a list whose elements are the
    % non-empty intervals from the input.

    % This implementation examines each of the input elements
    % sequentially and for each pair of intervals found, it
    % catenates them one at a time to the list that will
    % eventually be returned.  The actual catenate operation looks
    % unusual because each element must be converted to a one element
    % array in order to insure type-correctness for "||", which
    % requires that all operands be arrays.

    for

        new_list : Interval_list := empty [ Interval_list ] ;

        loc : integer    := array_liml ( result_data );
    do

        if loc > array_limh ( result_data )

           then new_list

           else iter
                new_list := tagcase interval_data := result_data[loc]

                            tag none:   new_list

                            tag more:   new_list
                                     || [ 1: interval_data[1]  ]
                                     || [ 1: interval_data[2]  ]
                            endtag
                loc := loc + 1;
                enditer
           endif
    endfor
endfun
```

```
   function Build_two_intervals(left,leftv,mid,midv,
                                right,rightv : real
                        returns Result_info        )

      % This is a utility function which takes info
      % on the analysis of an interval and builds a
      % list consisting of the two subintervals.

      let
          left_interval  : Interval
                         := record [ x_low : left;
                                     Fx_low : leftv;
                                     x_high : mid;
                                     Fx_high : midv ] ;

          right_interval : Interval
                         := record [ x_low : mid;
                                     Fx_low : midv;
                                     x_high : right;
                                     Fx_high : rightv ];

          two_intervals :  Interval_list
                        := [ 1 : left_interval;
                             2 : right_interval ];

      in   make Result_info [ more : two_intervals ]

      endlet
   endfun




   %   The body of the main procedure is simply responsible
   %   for computing the value of the function at the end
   %   points and then invoke the integrate routine to complete
   %   the work.


   Integrate(low,Evaluate_Function(low),high,Evaluate_Function(high))

   endfun     %  end of Adaptive_Quadrature
```

Appendix II: Recursive AQ

```
function Integrate_2 (left,leftv,right,rightv: real returns real)
    %  If VAL were to allow recursion, this function could replace four
    %  functions in the program found in Appendix I:  Integrate,
    %  Compute_Quads, Build_list, and Build_two_intervals.
    %  It is computationally equivalent to the previous version,
    %  but only because the first one did not use any information
    %  about error estimates for completed intervals.  That type
    %  of information would not be available in a recursive
    %  algorithm.  On the other hand this version is potentially faster
    %  because there is no constraint to finish one list of intervals
    %  before starting the next one.

    let
         mid      :  real
                  := (left + right) * 0.5;
         midv     :  real
                  := Evaluate_Function(mid);
         old_area:   real
                  := (right - left) * (rightv + leftv) * 0.5;
         new area:   real
                  := (right - mid) * (rightv + midv)  * 0.5  +
                       (mid  - left) * (midv  + leftv)  * 0.5    ;
         done     :  boolean
                  := Stopping_Condition(old_area, new_area, right - left);
   in
         if done then     new_area
                 else     Integrate_2( left, leftv, mid, midv) +
                          Integrate_2( mid, midv, right, rightv)
                 endif
   endlet
endfun
```

References

[1] Ackerman, W.B., and J.B. Dennis. "VAL — A Value-Oriented Algorithmic Language : Preliminary Reference Manual", Computation Structures Group, TR-218, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, June 1979.

[2] Ackerman, W.B. "Data Flow Languages", *AFIPS Conference Proceedings*, Vol. 48, New York City, June 1979.

[3] Arvind, and R.E. Bryant. "Parallel Computers for Partial Differential Equation Simulation", Computation Structures Group Memo 178, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, June 1979.

[4] Arvind, K.P. Gostelow, and W. Plouffe. The (Preliminary) Id Report Department of Information and Computer Science (TR 114a), University of California at Irvine, Irvine, California, May 1978.

[5] Ashcroft, E.A. and W.W. Wadge. "Lucid, a Nonprocedural Language with Iteration", *Comm. ACM* 20,7(July 1977), 519-526.

[6] Backus, J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *Comm. ACM* 21,8(August 1978), 613-641.

[7] Boekelheid, K. "A High-level, Graphical, Data-driven Language", *Proc. of the Workshop on Data Driven Languages and Machines*, J.C. Syre, Ed., Toulouse, France, Feb. 12-13, 1979.

[8] Brinch-Hansen, P. "The Programming Language Concurrent Pascal", *IEEE Transactions on Software Engineering SE-1*, 2(June 1975), 199-207.

[9] Davis, A. "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 6,7 (April 1978),210-215.

[10] Davis, A. "DDNs—A Low Level Programming Schema for Fully Distributed Systems", *Proc. of the Workshop on Data Driven Languages and Machines*, J.C. Syre, Ed., Toulouse, France, Feb. 12-13, 1979.

[11] de Boor, C. "On writing an automatic integration algorithm", *Mathematical Software*, J.R. Rice, Ed., Academic Press, New York, 1971, 201-209.

[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] Gurd, J., I. Watson, and J. Glauert. "A Multilayered Data Flow Computer Architecture", Department of Computer Science, University of Manchester, Manchester, England, July 1978.

[14] Kuck, David J. "Parallel Processing of Ordinary Programs", *Advances in Computers*, Vol. 15, Academic Press, 1976, p. 119-179.

[15] Landry, S., and B. Shriver. "A Data Flow Simulation Research Environment", *Proc. of the Workshop on Data Driven Languages and Machines*, J.C. Syre, Ed., Toulouse, France, Feb. 12-13, 1979.

[16] Lawrie, D.H., T. Layman, D. Baer, and J.M. Randal. "Glypnir—A Programming Language for the Illiac IV", *Comm. ACM* 18,3 (March 1975), 157-164.

[17] Liskov, B.H., et.al. "CLU Reference Manual", Computation Structures Group (Memo 161), Laboratory for Computer Science, MIT, Cambridge, Mass., July 1978.

[18] Martin, J.T., R.G. Zwakenberg, and S.V. Solbeck. "LRLTRAN Language Used with the CHAT and STAR Compilers", Livermore Time-Sharing System, Chapter 207, Lawrence Livermore Laboratory, Edition 4 - December 11, 1974.

[19] McCarthy, J., et.al. *Lisp 1.5 Programmmer's Manual*, MIT Press, 1966.

[20] Oxley, Don, Texas Instruments Inc., Austin, Texas, private communication on the Distributed Data Processor, March 30, 1979.

[21] Patil, S.S., R.M. Keller, and G. Lindstrom. "An Architecture for a Loosely-coupled Parallel Processor", Department of Computer Science (UUCS-78-105), University of Utah, Salt Lake City, Utah, July 1978.

[22] Petri, C.A. "Concepts of Net Theory", *Proc. Symp. and Summer School on Mathematical Foundations of Computer Science*, High Tatras, Sept. 3-8, 1973, Math. Inst. Slovak Academy of Science, 1973, 137-146.

[23] Peterson, J.L. "Petri Nets", *ACM Computing Surveys* 9,3 (Sept. 1977), 223-252.

[24] Plas, A., D. Compte, 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, 293-303.

[25] Rice, J.R. "A Metalgorithm for Adaptive Quadrature", *Journal of the ACM* 22, 1975, 61-82.

[26] Rice, J.R. "Parallel Algorithms for Adaptive Quadrature - Convergence", *Proc. IFIP Congress* 74, Stockholm. Amsterdam: North Holland, 1974, 600-604.

[27] Rice, J.R. "Parallel algorithms for adaptive quadrature III- Program Correctness", *ACM Trans. Math. Software* 2,1 (March 1976), 1-30.

[28] Treleaven, P.C., et. al. "The Design of Highly Concurrent Computing Systems", Computing Laboratory (TR 126), University of Newcastle upon Tyne, Newcastle upon Tyne, England, July 1978.

[29] Weng, K.S. "Stream-Oriented Computation in Recursive Data Flow Schemas", Laboratory for Computer Science (TM-68) MIT, Cambridge, Massachusetts, October 1975.

[30] Wirth, N. "Modula: A Language for Modular Multi-programming", *Software Practices and Experience*, Vol. 7, Jan. 1977, 3-35.

[31] Woodruff, J.P. "Scientific Application Coding in the Context of Data Flow", Lawrence Livermore Laboratory, UCRL-81922, Dec. 15, 1978.

Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore Laboratory under contract number W-7405-ENG-48, and BES grant AK-01-04.