A Data Flow Supercomputer

Computation Structures Group Memo 213
March 1982

Jack B. Dennis
Guang-Rong Gao
Kenneth W. Todd

The work reported in this paper was performed with support from the National Science Foundation and the U.S. Department of Energy.

A Data Flow Supercomputer[1]

Jack B. Dennis
Guang-Rong Gao
Kenneth W. Todd
MIT Laboratory for Computer Science
Cambridge, Massachusetts 02139

Computation Structures Group Memo 213

March 1982

Abstract

In data flow computers all instruction sequencing is controlled by the flow of data. Data flow computers promise efficient parallel computation limited in speed only by data dependencies in the calculation being performed. At the MIT Laboratory for Computer Science, the Computation Structures Group is working to prove feasibility of practical data flow supercomputers that can outperform conventional supercomputers.

Since data flow computers are very different in their structure and programming from conventional sequential computers, it is impossible to project their performance potential meaningfully except with reference to a specific computation. To establish the performance improvement that data flow computers offer, we have chosen to analyze a NASA benchmark program that implements a global weather model.

We discuss the structure of the weather code as expressed in VAL, a functional programming language developed by the Computation Structures Group expressly for high performance numerical computation, and discuss the corresponding machine-level program structure for efficient execution on a data flow supercomputer. On the basis of this analysis, we are able to calculate the number and types of hardware units required to build a data flow machine that will achieve a twenty-fold improvement in performance for the weather simulation application.

[1]The work reported in this paper was performed with support from the National Science Foundation, and the U. S. Department of Energy.

1. Introduction

Supercomputers are the highest performance machines available for automatic computation. They are used mainly in applications where physical phenomena are modeled. They simulate natural processes such as weather, earthquakes, and ocean currents for purposes of prediction. They are used to study human inventions that depend on physical processes, but where experimental study is uneconomical, difficult, or hazardous — such as the behavior of airfoils in a wind stream, the vibration of buildings and other structures, and the detonation of nuclear weapons.

The most successful supercomputers of the present day are conventional stored program computers based on sequential instruction execution. They are constructed using ECL (emitted-coupled transistor logic) gates, the fastest logic within the state of the art. To increase speed of computation beyond that attainable with the most straightforward architecture of the central processing unit, a variety of innovative concepts have been developed. The cache speeds up memory accesses; memory interleaving and instruction overlap architectures make it possible to process several instructions concurrently; and pipelined processing units provide efficient support for exploiting parallelism by means of vector operations.

In spite of these advances, our appetite for computing has not been satisfied. Scientists readily conceive of applications beyond the capacity of our most powerful computers. Studies in two dimensions show that computational modeling in three dimensions will yield new insights into the nature of physical processes such as turbulence in fluid flow or the interactions that take place in a hot plasma of ionized particles.

The amount of parallelism available for exploitation has been shown to be enormous, confounding earlier critics of parallel computation. Many applications take the form of computing successive states of a physical system represented by physical quantities on an Euclidean grid in two or three dimensions. In many cases, the new values for each grid point may be computed independently. Thus the degree of concurrency is often at least equal to the number of grid points.

In signal processing, the computation is typically a discrete model of an interconnection of modules each of which is kept busy processing successive samples of signal waveforms. The degree of concurrency is the number of modules multiplied by the average concurrency within the modules and may easily exceed a hundred.

If future supercomputers are to achieve a significant gain in performance over current designs, they will have to be able to process large numbers of basic operations (additions and multiplications) simultaneously. Thus these machines will have many functional units, many instruction decoders, and many independent memory units. To realize these high levels of concurrency in practice is a major challenge to computer system architects. No machine yet built has achieved close to one hundred-fold concurrency in operation. High levels of concurrency are most likely to be achieved using LSI parts in a "medium performance" technology such as NMOS or CMOS because use of a lower level of integration or a bipolar technology implies a larger physical complexity, size, and dissipation.

A design using LSI parts will have to use a large number of parts to yield an attractive gain in overall performance beyond existing supercomputers. Yet there must be only a few distinct part types so that the investment in their design and tooling is not exorbitant.

A proposal frequently suggested is to assemble a thousand microprocessor chips into a high performance machine. Unfortunately, there are many problems. First of all, memory and interconnection arrangements must be provided by means of additional chips. But an even more difficult problem is devising a suitable scheme for programming the communication of data between microprocessors without consuming nearly all execution cycles in executing the protocol. The cost of coordinating activities forces the use of long uninterrupted sequences of computation steps which will prove difficult for even a very smart compiler to uncover in ordinary programs.

2. Data Flow Computer Architecture

Data flow computer architecture is a fundamentally different way of organizing a computer system. There are some similarities with machines designed to execute conventional sequential programs: a data flow computer is a stored program computer, and a machine-level program consists of individual instructions whose execution constitutes the activity of the machine. Instructions call for conventional operations to be performed: the basic arithmetic operations on integers and floating point numbers and the usual comparison and Boolean operations. The difference is the manner in which instruction execution is controlled: a data flow computer has no program counter. Rather, each instruction is activated when it has received (as the results of other instructions) the data on which it is to operate.

In this article we consider a *static* concept of data flow architecture [9] in which the instructions of a machine level program are loaded into specific memory locations in the machine before computation begins, and only one instance of an instruction is active at a time. We have chosen the static architecture as the basis for data flow supercomputers because the hardware structures involved are relatively simple and suitable for exploiting integrated circuit technology at the level of current art. Other concepts of data driven instruction execution are being studied. [4, 13, 11, 8].

The structure of the envisioned data flow supercomputer is shown in Figure 1. Instructions of a data flow program are held in the local memory of the *processing elements.* Each processing element is equipped to recognize which of the instructions it holds have been enabled for execution by arrival of the operand values it needs. If an enabled instruction calls for a scalar arithmetic operation (a floating point addition or multiplication, for example), the instruction, including its operands, is sent to a *functional unit* capable of performing that operation. The *array memory* units are provided to hold arrays of data making up the (possibly very large) data base of the computation and are accessible through the *distribution routing network* from any of the processing elements. As will be explained in detail later, instructions that build array values or access elements of arrays are sent to the appropriate array memory unit through the *memory routing network.* Instruction execution in a functional unit or array memory unit yields result packets each of which consists of a data value and a destination field that specifies the target instruction for the result packet. The result packets are sent to processing elements that hold the target instructions through the *distribution routing network.* The way in which a data flow processing element handles result packets, identifies enabled instructions, and initiates

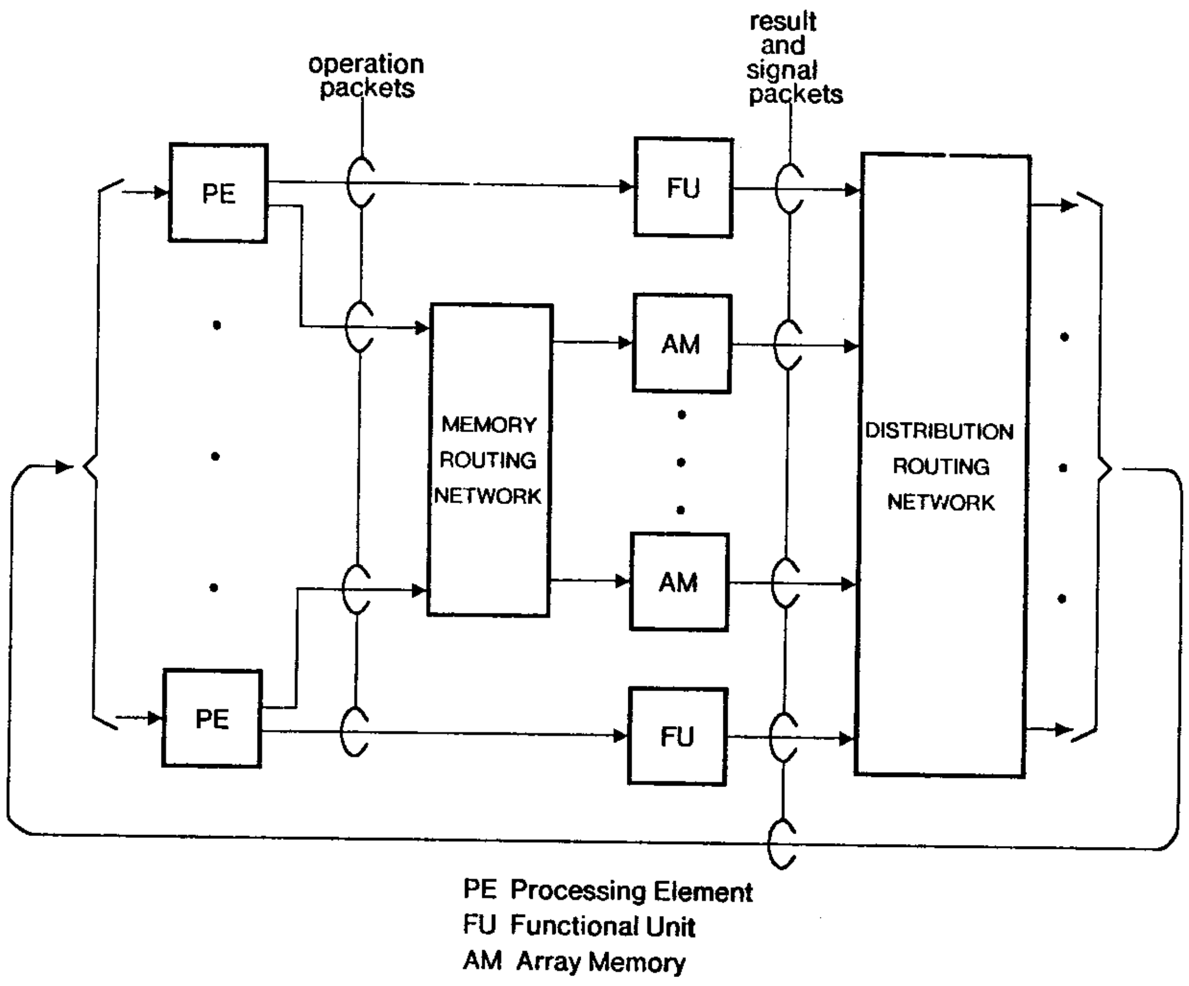


Figure 1. A Data Flow Supercomputer.

instruction execution has been described in several publications [7, 11, 9]. The use of packet routing networks in data flow computers is discussed in [9, 10] and analyses of efficient structures for routing networks have been done by Boughton [5].

Other instructions such as those calling for duplicating data values, for Boolean operations, and for simple tests, are performed within the processing element.

Because data flow computers are radically different from familiar sequential stored program computers, it is impossible to predict the performance of a data flow computer through a comparison of the speed and cost of its basic units with corresponding units of a conventional machine. Rather, it is necessary to choose specific computations for making a comparison. We have chosen a global weather model as the computational problem to be studied. Our objective is to show the reader the size of data flow supercomputer required to perform this computation at a rate substantially faster than a conventional supercomputer, and to argue that this performance can be achieved at considerably lower cost.

Before considering the problem of structuring a data flow machine program for the global weather model, we will start by considering simpler computations to illustrate the concepts and issues involved in deriving data flow machine code from programs expressed in a high-level programming language. In the examples we will use the programming language VAL to specify the intended computation.

3. The Programming Language VAL

The programming language VAL (*Value-Oriented Algorithmic Language*, [1]) is a functional language designed as a practical source language for writing programs for data flow computation. In VAL there are no control transfer statements or global variables, and in program modules arguments and results are clearly separated. The basic operations of the language, including operations on data structure values — records and arrays — are simple functions mapping operand values to results.

Three unique features of VAL are the iteration (**for**) construct, the **forall** construct, and the provisions for generating and propagating error values.

The iteration construct in VAL is designed to correspond to tail recursive function definitions. Only those recursive definitions that are expressible as tail recursions can be written in VAL. This includes all programs expressible using the **while** . . . **do** . . . **end** construct and program examples that seem to require a **break** or **exit** feature as well. Since the iteration construct is unusual, we illustrate it using the traditional factorial example:

```
function Factorial (n: integer returns integer)
    for  i: integer := 0;
         p: integer := 1;
    do   if i = n then p
         else
              iter  i := i + 1;
                    p := p * i
              enditer
         endif
    endfor
endfun
```

The body of the **for** construct is a conditional expression with two types of arms. The first arm, taken in the case where $n = i$, is simply an expression, giving the result returned by the iteration. The second arm, taken when $n \neq i$, is an **iter** arm which binds new values to the loop names i and p and indicates that the **for** body should be evaluated again using the new bindings.

The **forall** construct is provided for situations that require a group of similar independent computations to be performed where the computations may proceed concurrently. The construct has two forms: in one, the results of the individual computations are made components of an array which is the result value; in the other the individual results are combined using a specified associative operation such as addition.

In each data type of VAL *error values* are provided to represent the results of operations which cannot yield proper values of the result type. Such operations include arithmetic overflow and underflow, division by zero, and accessing an element outside the bounds of an array. The language definition includes rules that govern propagation of error values through subsequent operations.

Our next example of a VAL program is one we shall return to later to illustrate efficient machine code structure for a static data flow computer. The computation is the solution of the LaPlace problem on a rectangular grid having elements numbered from 0 through $m + 1$ in each row and numbered from 0 through $n + 1$ in each column. The program is given an initial data array and computes successive data arrays in which each interior element is the average of the four near-by elements of the previous data array. Each data array retains the initial values for the boundary elements. The VAL code for this computation is shown in Figure 2.

```
function Solve (
    A: data;             % initial grid values
    M, N: integer;       % grid size
    cnt: integer         % number of steps
    returns data)        % solution matrix
    type data = array[array[real]];
    for  X: data := A;
         k: integer := cnt;
    do   if k = 0 then X
         else
              let Y: data :=
                   forall i in [0, m + 1], j in [0, n + 1]
                   construct
                        if i = 0 | i = m + 1 | j = 0 | j = n + 1 then X[i, j]
                        else 0.25 *
                             (X[i, j − 1] + X[i, j + 1] + X[i − 1, j] + X[i + 1, j])
                        endif
                   endall;
              in   iter
                   X := Y;
                   k := k − 1;
                   enditer
              endlet
         endif
    endfor
endfun
```

Figure 2. The LaPlace solver in VAL.

4. Data Flow Machine Language

A machine level program for a static data flow computer is a collection of *instruction cells* that are loaded into the processing elements of the machine. In illustration, Figure 3 shows a translation of the VAL definition

$$Ave\text{: real} := 0.25 * (A + B + C + D)$$

Each instruction cell is represented by a rectangular block divided into fields. The top field

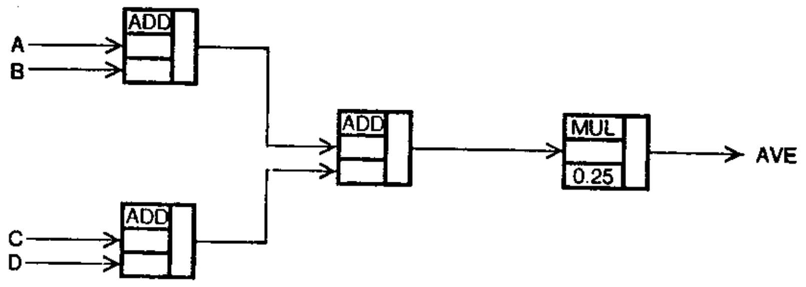


Figure 3. Data flow machine code for an arithmetic expression.

contains the operation code, and beneath it are one or two fields that provide space for operand values. At the right is space used for *destination* fields that specify the target instruction or instructions to which the result value is to be sent. A destination field consists of the address of the target instruction cell and an integer that specifies which operand of the target instruction is being delivered. In the figures we indicate destinations by drawing arcs from each instruction cell to its target cells, the arcs ending on the appropriate operand field.

An instruction cell is ready for execution once its operand fields have been filled. If it calls for a simple operation such as an identity or Boolean operation, it is processed by the processing element in which it resides. Otherwise the entire instruction cell is sent in the form of an *operation packet* to the appropriate functional unit (in the case of floating point arithmetic operations, for example) or to one of the array memories (in the case of array construction or access operations). In any case, instruction execution (the "firing" of an instruction cell) generates result values that are placed in the operand fields of target instruction cells.

When an instruction cell shows a value written in one of its operand fields, as in the MUL instruction cell of Figure 3, the value is a constant operand of the instruction and is unchanged from one execution of the instruction to the next.

The computation illustrated in Figure 3 shows a small degree of concurrency in that the pair of ADD instruction cells on the left may be in execution simultaneously. This we call *spatial* parallelism. A second form of concurrency is important in data flow computation and plays an essential role in the exposition to follow: *pipelining.* In conventional computer architecture, pipelining refers to a hardware functional unit designed so each operation passes through several stages of logic, and so several succeeding operations may be initiated before the result of the first operation is produced. In data flow computation, pipelining means arranging the machine code so successive computations may follow each other through one copy of the code, thus achieving more intensive use of instruction cells and supporting faster computation by the data flow machine.

In the static data flow architecture, an instruction cell must not fire again until all copies of the result sent out by the previous firing have been consumed. Otherwise operand values might be overwritten or, if this is prevented, deadlock could occur. To ensure that this does not happen, we use two types of destination arcs: *result arcs* and *signal arcs.* Result arcs specify transmission of the result value to target instruction cells. Signal arcs specify transmission of a *signal* to an instruction cell from which an operand value was received. The condition for firing an instruction cell now demands that all operand fields be filled and that one signal packet be received from each target instruction indicating that the last result computed has been consumed. This augmented firing condition is implemented by including two additional fields in each instruction cell: *signals needed* and *signal reset.* The integer in the signals needed field, is reduced by one for each receipt of a signal packet. The cell is ready for execution when this field becomes zero and all operands are present. The signal reset field is constant and provides the reset value for the signals needed field whenever the cell fires. Thus, if a cell sends its result to five other cells, its signal reset value should be five and each of the five cells should send a signal back to that cell when they fire.

Figure 4 shows the program of Figure 3 modified to support pipelined operation. The two new fields to implement signalling are shown just below the opcode field: signals needed on the left and signal reset on the right.

In most of our examples the machine code will be arranged to support pipelined operation in the manner just described with a signal being returned for each result value transmitted. To simplify the diagrams we will use an arc with a solid arrowhead as an abbreviation for the combination of a result arc and its associated signal arc.

Implementing the conditional expression of VAL requires some means for routing

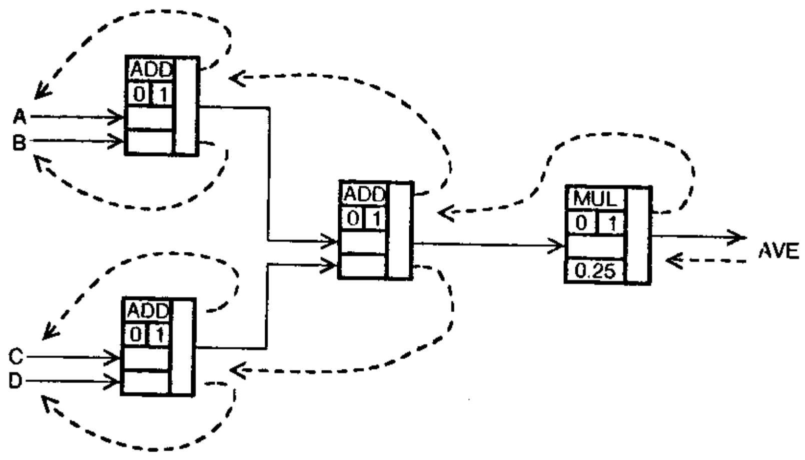
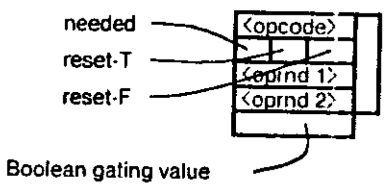


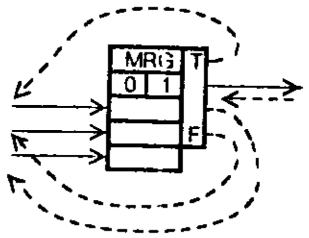
Figure 4. Pipelined form of data flow machine code.

values through alternative machine code paths, but this is not possible using the mechanisms so far introduced. Therefore, a means of switching result packets between alternative destinations is provided, as well as a special MERGE instruction. Any instruction cell may also perform a switching function by including an extra *control* operand of Boolean type, and tagging any of its result and signal arcs with either T for **true** or F for **false**. When the cell fires, a result or signal packet is sent for a tagged arc only if the tag matches the Boolean control operand. Since the number of result arcs tagged T and F are not generally equal, two signal reset fields are needed, one for use when the control value is **true** and one for use when it is **false**. An example of an instruction cell with this routing feature is shown in Figure 5a.

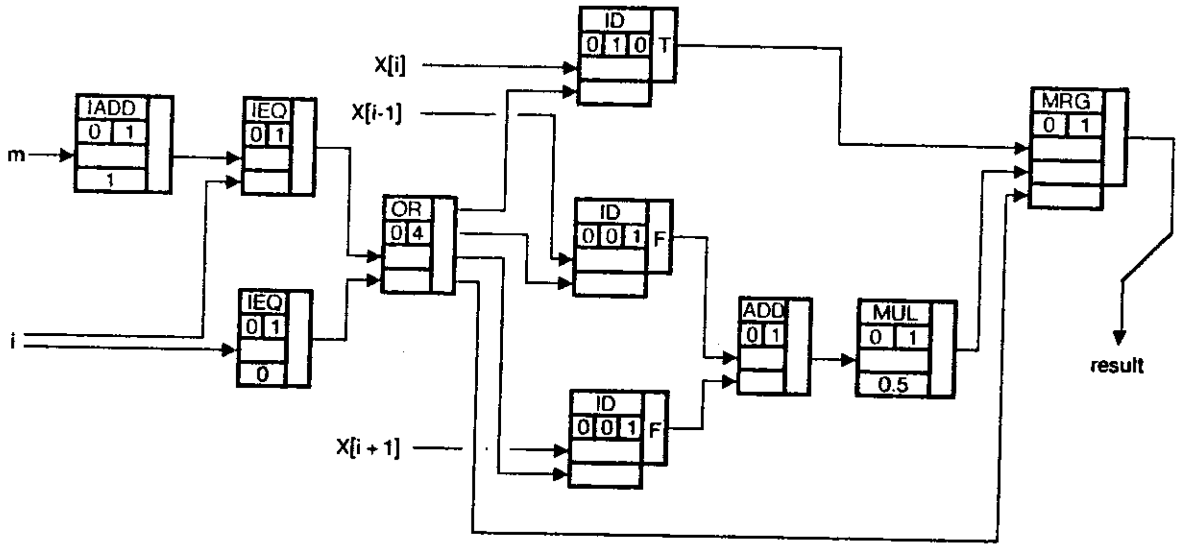
A MERGE instruction is used to select which one of two sources of values is to supply the next value for transmission as a result packet. As shown in Figure 5b, a MERGE instruction cell has two operand fields for data values and a third operand field for a Boolean control value. If the control value is **true** then the first operand is consumed and becomes the result value, leaving the second operand (if present) untouched. Similarly, if the control value is **false** then the second operand is the one used, and the first one is left alone. Because only one of the data operands is needed, the ready condition for a MERGE instruction is unusual: the signals needed field must be zero, the control operand must have arrived, and the first or second data operand must have arrived, according as the control value is **true** or **false**.



(a) Instruction cell with gating capability.



(b) Merge instruction cell.



(c) Pipelined machine code.

Figure 5. Data flow machine code for a conditional expression.

As an example, consider the following VAL conditional expression:

if $i = 0 \mid i = m + 1$
then $X[i]$
else $(X[i - 1] + X[i + 1]) / 2.0$
endif

One possible translation for it is given in Figure 5c. After calculating the predicate, its value is used to route either the value $X[i]$ to the MERGE instruction cell or the values $X[i - 1]$ and $X[i + 1]$ to the cells that calculate their average and send that value to the MERGE cell. The predicate is also used by the MERGE instruction to specify which operand is to be sent as the result packet. Note that we have extended our abbreviations for destination arcs. A tag of T or F on an abbreviated arc means that the result arc it replaced was similarly tagged. An abbreviated arc pointing to the first (second) operand field of a

MERGE cell means that the signal arc it represents must be tagged T (F).

5. Data Flow Computation with Arrays

Numerical computation usually involves operations on arrays of data — sequences of values selected by integer index values

$$X[1], X[2], \ldots, X[i]$$

In conventional computers one thinks of *transforming* array values by making successive replacements of element values in the array. In data flow computation, a different view is required [2]: a computation *constructs* an array value from other array values, scalar quantities and constants.

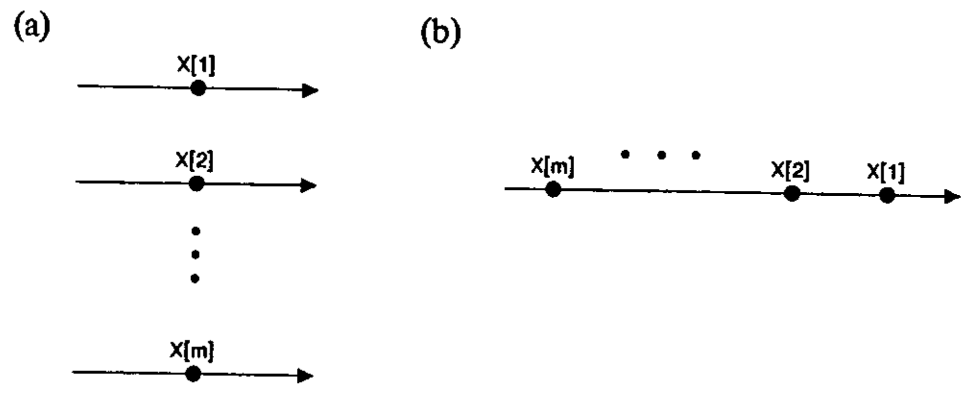


Figure 6. Two data flow representations for an array value.

In the static data flow architecture we regard an array value as either the set of result values conveyed at some moment by a certain group of destination arcs (as in Figure 6a), or as the sequence of values carried by result packets on a single arc at successive moments (as in Figure 6b). These two representations for array values show the basic space/time tradeoff in structuring machine code for efficient operation on a static data flow computer.

In VAL the basic means of array creation is the **forall** expression:

```
X: array[real] :=
    forall i in [0, m + 1]
    construct
        if i = 0 | i = m + 1 then A[i]
        else 0.5 * (A[i − 1] + A[i + 1])
        endif
    endall
```

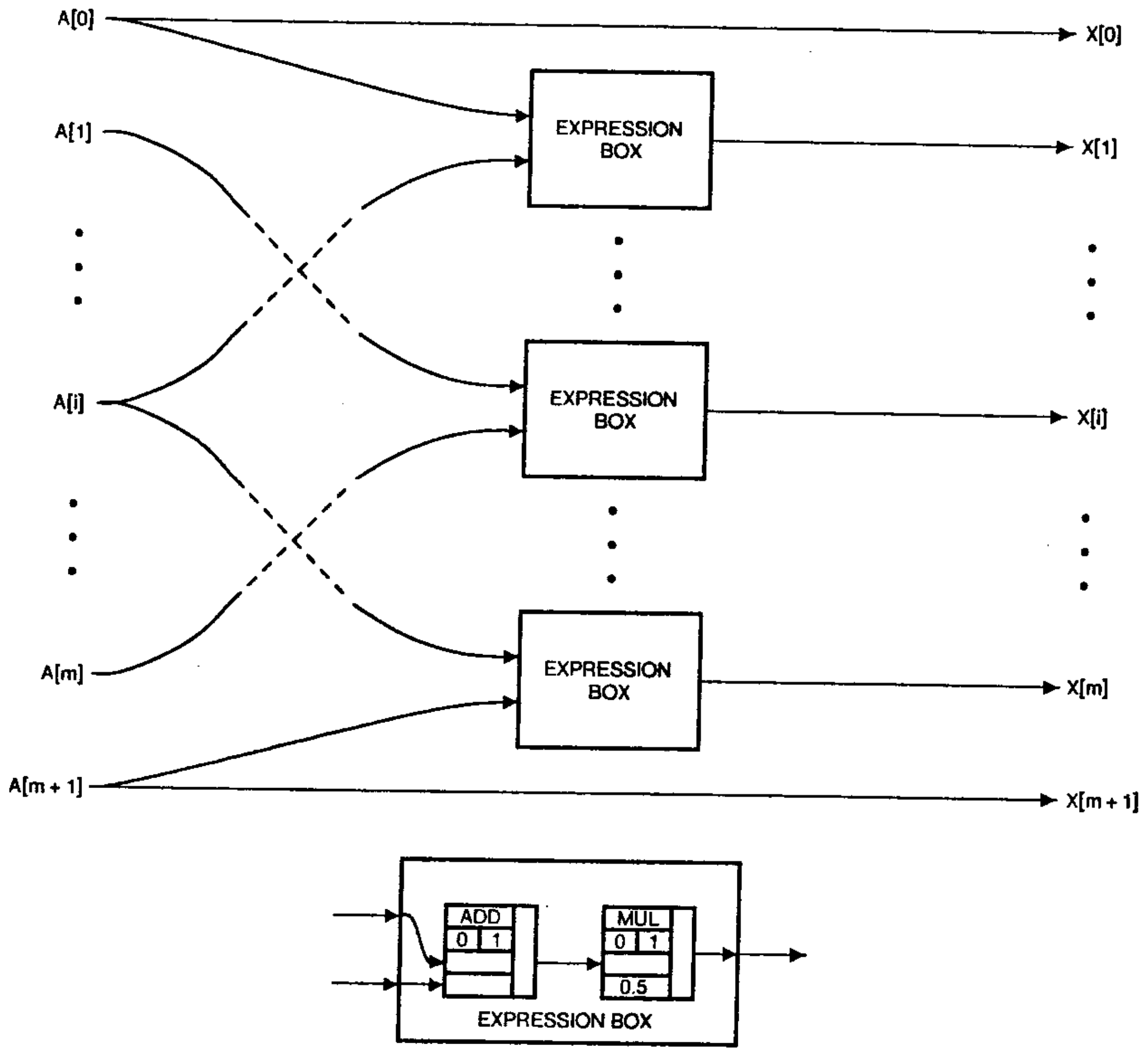


Figure 7. Parallel computation of an array value.

Depending on which representation of the array *X* is chosen, one is led to two different machine code structures for the above **forall** expression. The first choice consists of a copy of the program body for each array element, as shown in Figure 7. Since the value of the **forall** index i is fixed in each copy the conditional vanishes and each copy reduces to one of the two arms of the conditional expression. Of course, this construction can only be done if the dimension m is known when the machine code is loaded into processing elements.

The second possibility is to generate the array elements by operating the body of the **forall** construct in pipeline fashion. The code for this is shown in Figure 8. We suppose the given array A is represented by the sequence of result packets arriving over the arc labeled A. The three ID instruction cells each select a sequence of elements from A according to the sequences of control values supplied:

T . . . T F F	$A[0], A[1], \ldots, A[m-2], A[m-1]$
F T . . . T F	$A[1], A[2], \ldots, A[m-1], A[m]$
F F T . . . T	$A[2], A[3], \ldots, A[m], \quad A[m+1]$

It is easily seen that these are exactly the right sequences of values that must be presented to the two arms of the conditional expression that forms the body of the **forall**, so that the resulting elements of X arrive at the MERGE cell in the right order. The two groups of cascaded ID cells in the diagram are inserted to equalize path lengths through the code, allowing pipeline operation at the maximum rate.

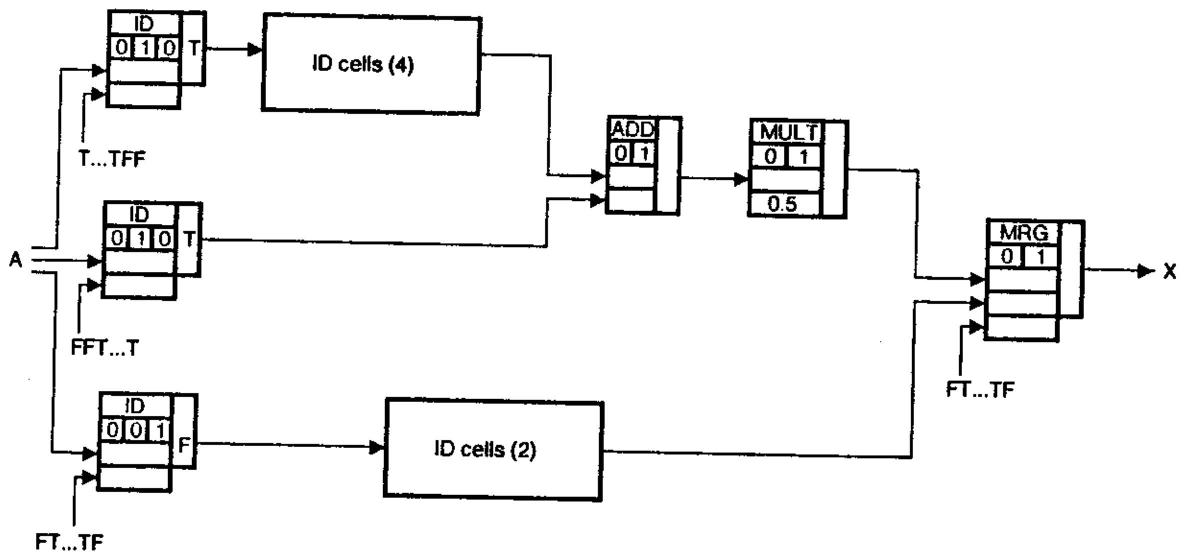


Figure 8. Pipelined computation of array values.

6. Machine Code for the LaPlace Solver

The LaPlace solver in VAL is an iteration in which each cycle calls for the construction of a two-dimensional array Y as a new value for the loop name X. Let us rewrite the **forall** expression, which is the heart of the program, to exhibit its nested structure:

```
Y :=
    forall i in [0, m + 1]
    construct
        if i = 0 | i = m + 1 then X[i]
        else

            forall j in [0, n + 1]
            construct
                if j = 0 | j = n + 1 then X[i, j]
                else 0.25 *
                    (X[i, j - 1] + X[i, j + 1] + X[i - 1, j] + X[i + 1, j])
                endif
            endall

        endif
    endall;
```

Several possibilities are available for structuring the corresponding data flow machine code, according to how we choose to represent the array X. We could employ spatial parallelism in either or both dimensions of the array. However, since we wish to prepare for our analysis of the global weather model, it is most instructive to support pipelined computation by using the serial packet form in both dimensions of the data array.

Our data flow machine code for the LaPlace solver is shown in Figures 9 through 11. Let us explain it by proceeding outward from its kernel. The code in Figure 9 implements the inner **forall** construct which computes rows of the new data array for $i = 1, \ldots, m$, and has the same structure as that shown in Figure 8.

Stepping out one level, the similar code structure shown in Figure 10 implements the outer **forall** construct. The difference is that the buffering of array elements which could be done by a few ID cells for the inner computation turns into a requirement for several long FIFO buffers, each able to hold a complete row of the data array. (We will discuss

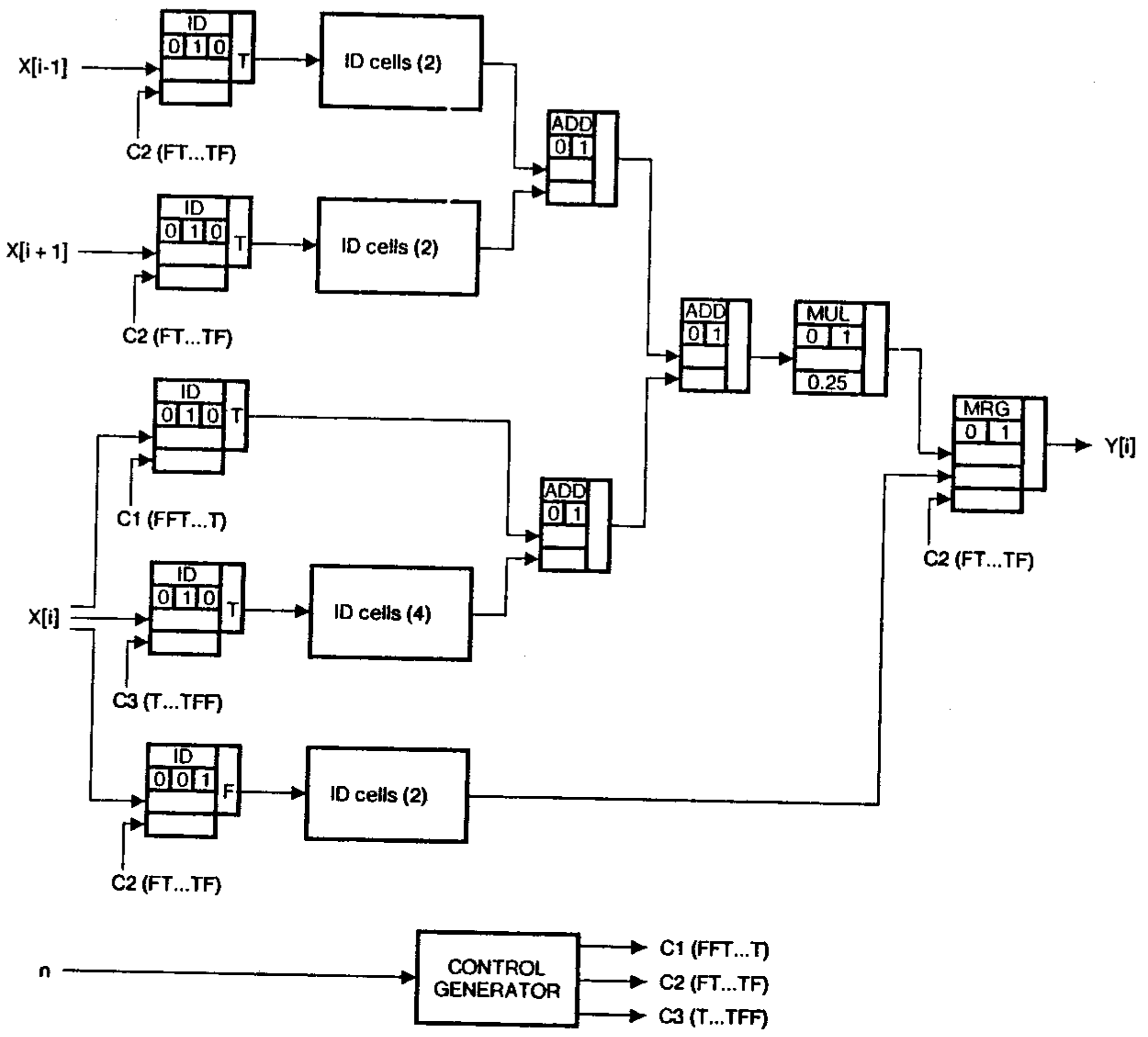


Figure 9. The kernel of the LaPlace Solver.

alternative implementations of FIFO buffers in the following section.) Also, generating the necessary control sequences is somewhat more complicated.

Finally the code structure that implements the iteration construct is shown in Figure 11. This consists of the body and a switching arrangement to feed in the initial data array A, to circulate the successive data arrays computed by the body, and then send out the final result. An important element in this figure is the FIFO buffer, which must have sufficient capacity to hold the complete data array of $(m + 2) \times (n + 2)$ elements. This is the data base of the problem and since it could be very large, should properly be stored in the array memories of the data flow supercomputer. How this may be done is our next topic.

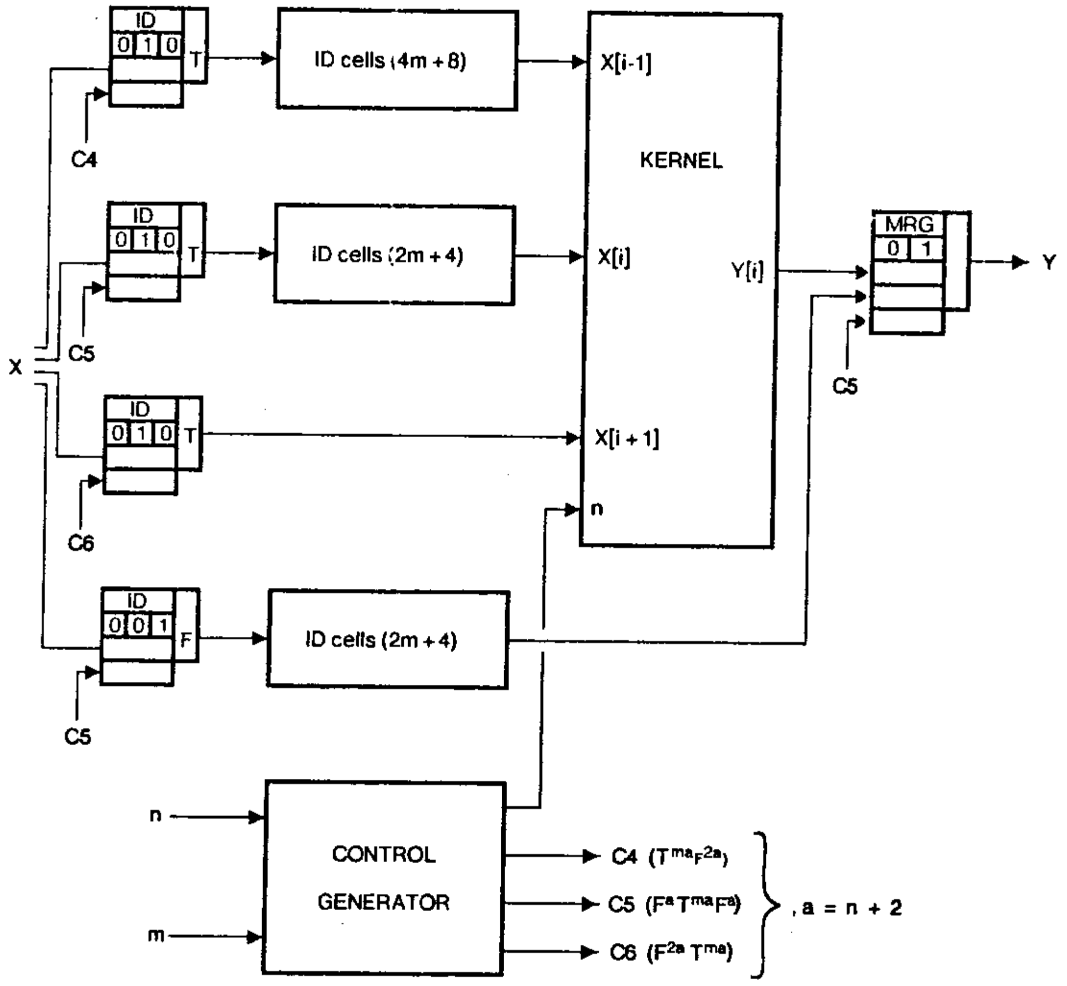


Figure 10. The body of the LaPlace Solver.

7. FIFO Buffering

There are many ways of implementing a FIFO buffer on a data flow machine, two of which we describe here. In the first, the FIFO buffer is built by cascading identity instruction cells as shown in Figure 12a. Data packets enter at the left end and flow out from the right end. Assuming each instruction sends an acknowledge packet to its predecessor, the capacity of such a buffer is one half the number of ID cells. The advantage of this scheme is in its simplicity, but the number of instruction cells, the number of cell firings per packet passed through, and the time required for a packet to pass through, all grow linearly with the capacity of the buffer. This implementation is appropriate in cases where only a few values must be held.

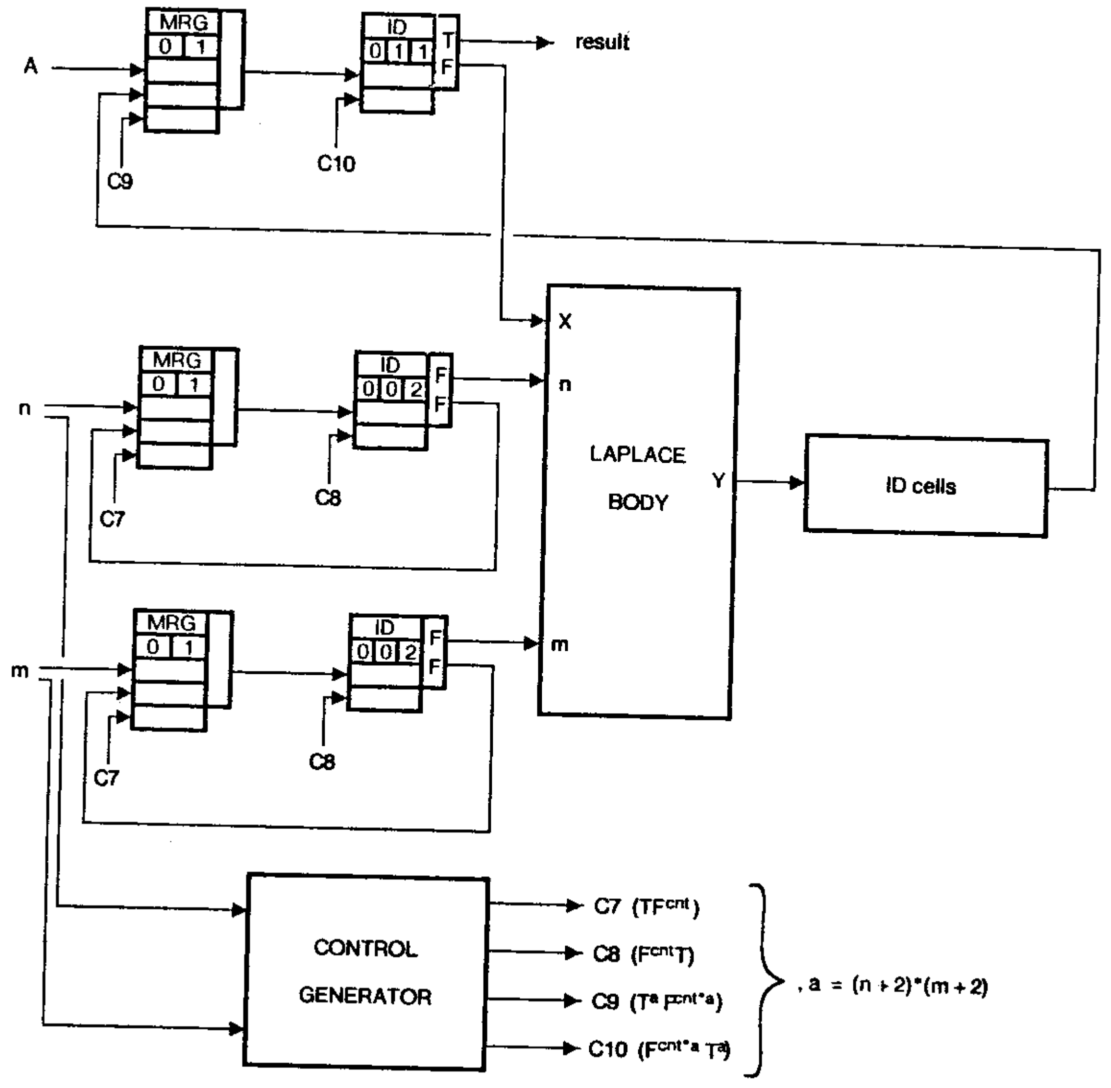
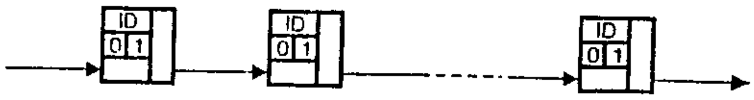
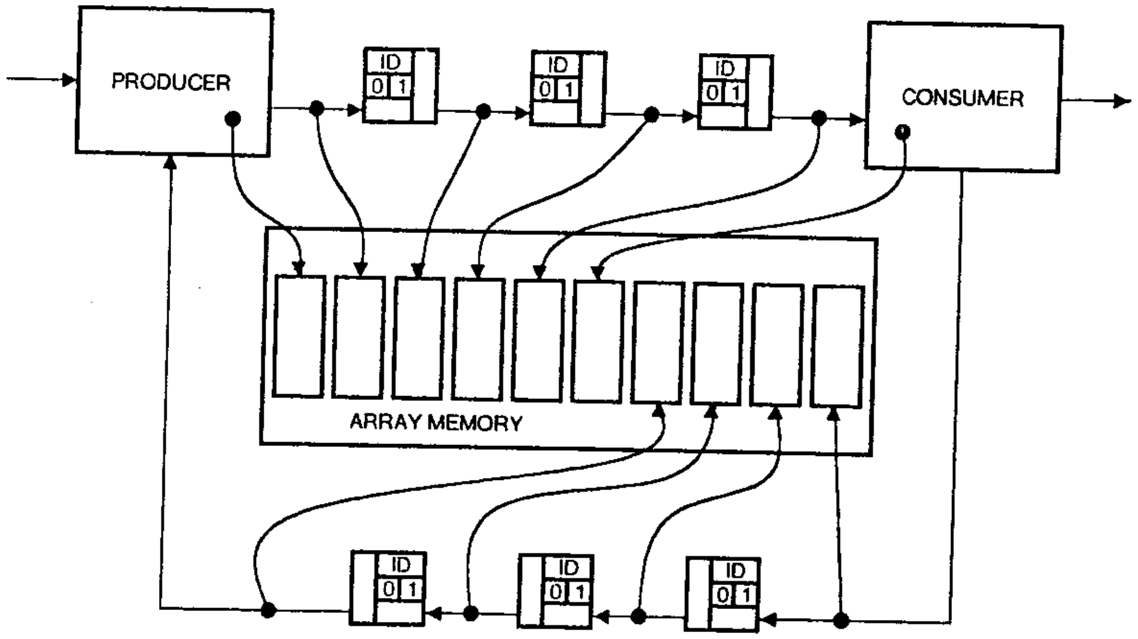


Figure 11. Controller for the LaPlace Solver body.

A second way to implement a FIFO queues uses the array memories for storage. Suppose a buffer is needed to hold p rows forming a two-dimensional array of data. We allocate p blocks of storage in array memory modules, each block being large enough to hold the elements of one row in contiguous locations. Two FIFOs are used to coordinate queuing and dequeuing by the producer and consumer, as shown in Figure 12b. One queue is initially empty; during computation it holds descriptors of those storage blocks (rows of data) that have been produced but not consumed. The second FIFO is filled initially with p descriptors of empty storage blocks. The producer fills successive locations of an empty block with data values from its input stream and enters it in the busy block



(a) Cascade of pipelined ID cells.



(b) Buffer implementation using storage blocks in array memories.

Figure 12. Two schemes for implementing FIFO buffers.

queue. The consumer reads values from a filled block, sends them in its output stream, and returns the emptied block to the free block queue.

The advantage of this scheme is that very large buffers can be implemented using the denser and more economical storage in the array memories while using only a few instruction cells. In addition, the number of instruction cells fired per value buffered is small and independent of the number of elements per row. The disadvantage is the inclusion of array memories and a second routing network, making a more elaborate machine.

To store and retrieve data in blocks in the array memory modules as required in the producer and consumer program modules, we introduce three instructions: INDEX, WRITE, and READ. The index instruction takes as operands two values—a pointer to a block within the array memory and an index within that block of an element. It produces as a result the absolute address in the array memory of that element which is used by both write and read instructions. The write takes both the address and a value and stores the value in the array memory at that address; the read simply takes the address and returns the contents of that memory location.

8. The Global Weather Model

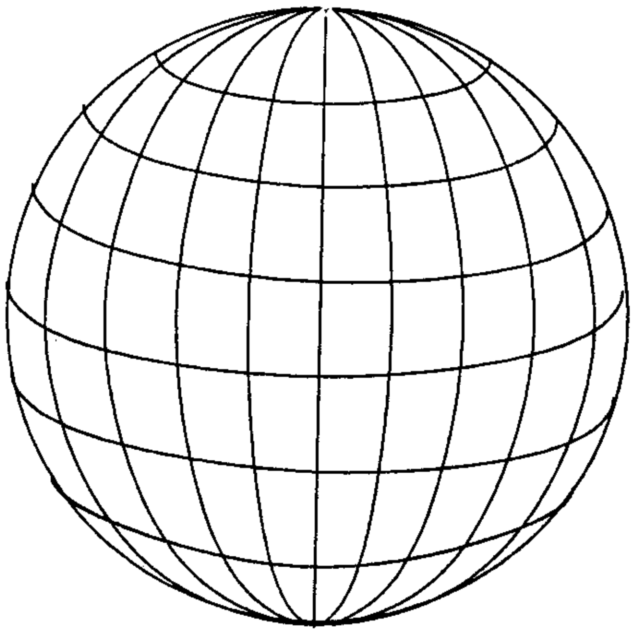


Figure 13. Global grid for a numerical weather model.

The weather, the behavior of the earth's atmosphere, is governed by physical laws expressed as a set of partial differential equations in which the most important variables are the wind (the motion of air parallel to the earth's surface), air temperature, water content, and atmospheric pressure [16, 14]. The objective of a numerical weather model is to project the state of the global atmosphere at some future time on the basis of current and historic observations of the weather. This is done by approximating the differential equations by a system of difference equations in which the physical quantities are described only at points on a three-dimensional grid over the surface of the earth, as illustrated in Figure 13. In the horizontal plane, the grid of points is defined by 144 meridian circles, equally spaced around the equator, and 87 parallels of latitude (including the south and north poles). In the vertical direction, the grid has nine layers. Thus the state of the atmosphere may be described by a group of three-dimensional arrays, one for each physical variable. In VAL this data base would be declared as follows:

```
type State = record [
    U: Grid;                    % Wind: longitudinal
    V: Grid;                    % Wind: latitudinal
    T: Grid;                    % Temperature
    H: Grid;                    % Water content
    P: Plane                    % Surface pressure];

type Grid = array [Plane];      % index set is {1, . . . , p} where p = 9

type Plane = array [Line];      % index set is {1, . . . , n} where n = 87

type Line = array [real];       % index set is {1, . . . , m} where m = 144
```

A value of data type *Plane* is an array of 87 $\times$ 144 real numbers that represent some state variable for one of the nine levels of the atmosphere; a value of data type *Line* is a sequence of 144 real numbers corresponding to the grid points at one latitude at a particular atmospheric level. Note that surface pressure is represented by just a two-dimensional array.

For each time step, the data base describing the atmospheric state at time $t + \Delta t$ is computed from the data base for time t. The value of each physical quantity at each grid point for time $t + \Delta t$ is computed using only values from the data base for time t. For the most part, this information is associated with the same vertical cell in the atmosphere, or with adjacent grid points in the horizontal plane. This implies a high degree of concurrency is possible in performing the computation since the new state of each grid point (all $9 \times 87 \times 144 = 112{,}752$ of them!) may be computed independently without indulging in any redundant computation.

Thus the weather model is a good problem with which to evaluate the potential of a data flow supercomputer. In fact, the computation offers far more parallelism than is reasonable to exploit.

As a basis for performance comparison with conventional supercomputers, we have chosen the global weather model developed at the Goddard Institute of Space Studies (GISS) [16]. This is a large Fortran program that extends earlier work by Arakawa and Mintz [3]. The GISS model runs on conventional computers such as the IBM 360/95 and the CDC 7600. The Fortran program computes one time step, representing 20 minutes of real weather, in about two minutes of computation.

9. Mapping the Weather Model onto the Static Architecture

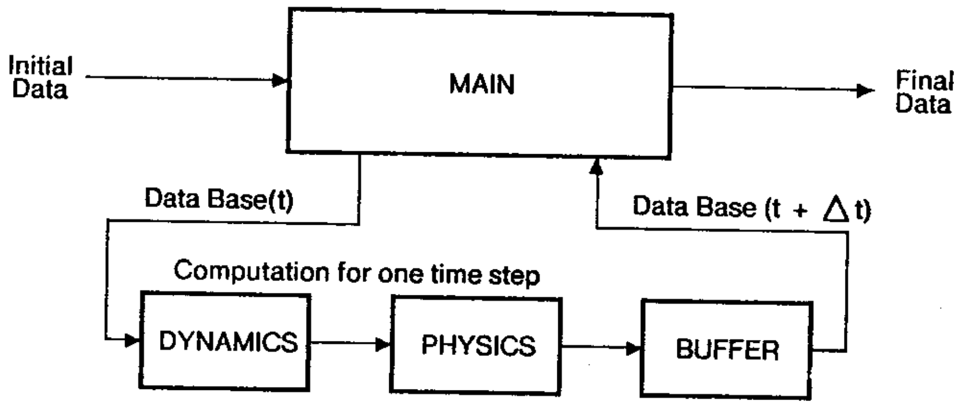


Figure 14. Overall structure of computation for the global weather model.

We have reprogrammed the GISS weather model in VAL, and discuss here the structure of corresponding machine code for execution by a static data flow supercomputer. The computation performed for each time step consists of two major parts: the first part models the *dynamics* of the atmosphere — the internal hydrodynamics and heat flow processes; the second part models the *physics* — the absorption and release of thermal energy by the atmosphere through processes including absorption of solar radiation, radiation into space, condensation and precipitation. The overall form of the VAL program is shown in Figure 14. The main function passes the data base through the dynamics and physics computations once for each time step until the desired prediction is obtained. The buffer holds the data base between computations for successive time steps.

In view of the high potential parallelism in the weather model, it is very attractive to process the data base by pipelined operation of the data flow machine code for the dynamics and physics computation. Specifically we choose to use spatial parallelism over the five physical quantities and over the nine layers of the grid, and to process the data for each quantity in each layer using pipelining. Following this scheme the input to the time step computation is 37 arrays of data type *Plane*: one array for the surface pressure and nine each for the longitudinal wind, latitudinal wind, temperature, and water content. For pipelined processing, each array is represented in the data flow machine as a single stream of $87 \times 144 = 12{,}528$ values.

Our analysis has focused on the dynamics computation where most interdependence among grid points is found. This portion of the program consists of about 1200 blocks of code, each of which constructs a stream of 12,528 values from streams generated by other code blocks or supplied as input to the dynamics computation.

Each code block computes an array of intermediate results associated with one layer of the global grid. Pairs of code blocks communicate in producer/consumer fashion — the producer generates a stream of values which are processed concurrently by the consumer as soon as they become available. Thus each code block receives several streams of input representing intermediate arrays of data, and generates an output stream. Each block has the same form as the body of the LaPlace solver (Figure 2). In fact, the machine code has essentially the same structure as that in Figures 9 through 11.

In the LaPlace solver example, we saw two uses of FIFO buffers:

1. To align array elements for adjacent columns of the grid so they arrive together at the body of the computation. These buffers were implemented as one or more ID instructions.

2. To align adjacent rows of the data array for processing together. This required the ability to buffer full rows of $n + 2$ elements apiece. These buffers could be implemented using groups of instruction cells in processing elements, or using blocks of storage allocated in array memory modules, as was discussed earlier in Section 7.

A third requirement for buffering is illustrated in Figure 15, which shows a situation that appears many places in the weather code. Here the state variable *U* is used by block 1 to compute *UT* which is used by block 2 to compute *UT1*. In block 3, *U* and *UT1* are consumed to create *UU*. Since the two paths in the figure have different lengths, a FIFO

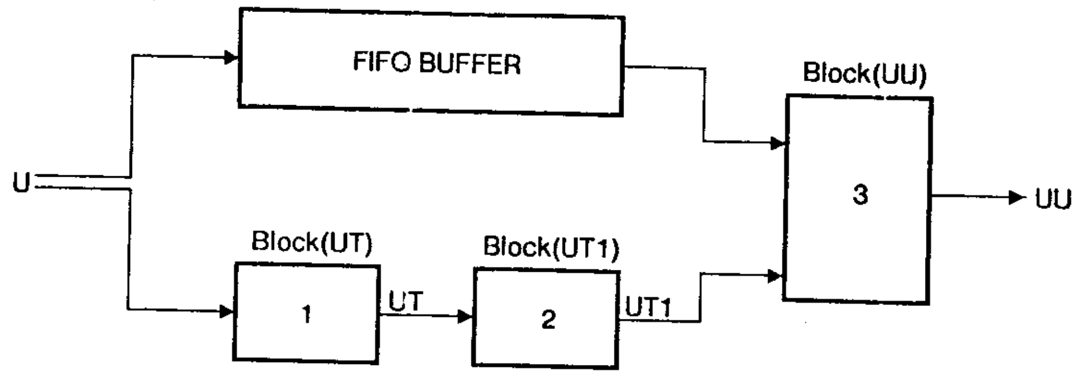


Figure 15.

buffer of appropriate size must be introduced between input *U* and block 3. For the weather code, the number of pipeline stages in most code blocks is between three and ten, so these buffers can be implemented by just a few ID cells.

An example of one code block from the dynamics computation is shown as a VAL program in Figure 16. In this form the block takes as inputs the arrays

U[*k*] *PU*[*k*] *PV*[*k*] *UT*[*k*]

associated with layer *k* of the three-dimensional grid, and computes the array *UT1*[*k*]. The data flow machine code for this code block has the same structure as that for the body of the LaPlace solver. The differences are that block *UT1* has many inputs instead of just one, there is much more computation in the kernel, and the boundary conditions are somewhat more complicated. In *UT1* the computation of each result value in the principal case requires 48 additions and 16 multiplications and has depth seven.

A complication absent in this example is interaction among layers. However, since the computation for all layers of the grid is proceeding simultaneously and closely in step, corresponding code blocks for different layers may readily exchange data pertaining to the same or adjacent cells of the horizontal grid.

10. Analysis of Computation Rate

In this section we determine the size of data flow machine required to achieve some specific performance level for the general circulation model using the machine level program structures described above. The general form of the proposed data flow machine

```
UT1: [array[array[real]] :=
  forall j in [1, n], i in [1, m]
    im: integer := if i = 1 then m else i - 1 endif;
    ip: integer := if i = m then 1 else i + 1 endif;
    jm: integer := if j = 1 then 1 else j - 1 endif;
    jp: integer := if j = n then n else j + 1 endif;

    FluxA: real := C1
       * (PU[jm,i] + PU[jm,im] + PV[jm,i] + PV[j,i])
       * (U[j,i] + U[jm,im]);
    FluxB: real := C0
       * (PU[j,i] + PU[jm,i] + PU[j,im] + PU[jm,im])
       * (U[j,i] + U[j,im]);
    FluxC: real := C1
       * (- PU[j,i] - PU[j,im] + PV[j,i] + PV[jp,i])
       * (U[jp,im] + U[j,i]);
    FluxD: real := C0
       * (PV[jm,i] + PV[jm,ip] + PV[j,i] + PV[j,ip])
       * (U[jm,i] + U[j,i]);
    FluxE: real := C1
       * (- PU[jm,ip] - PU[jm,i] + PV[jm,ip] + PV[j,ip])
       * (U[j,i] + U[jm,ip]);
    FluxF: real := C0
       * (PU[j,ip] + PU[jm,ip] + PU[j,i] + PU[jm,i])
       * (U[j,ip] + U[j,i]);
    FluxG: real := C0
       * (PV[j,i] + PV[j,ip] + PV[jp,i] + PV[jp,ip])
       * (U[j,i] + U[jp,i]);
    FluxH: real := C1
       * (PU[j,ip] + PU[j,i] + PV[j,ip] + PV[jp,ip])
       * (U[jp,ip] + U[j,i])
  construct
    UT[j,i] +
      if j = 1 then 0.0
      elseif j = n then FluxA + FluxB + FluxD + FluxE - FluxF
      else FluxA + FluxB - FluxC + FluxD
          + FluxE - FluxF - FluxG - FluxH
      endif
endall
```

Figure 16. Example of a code block from the weather model.

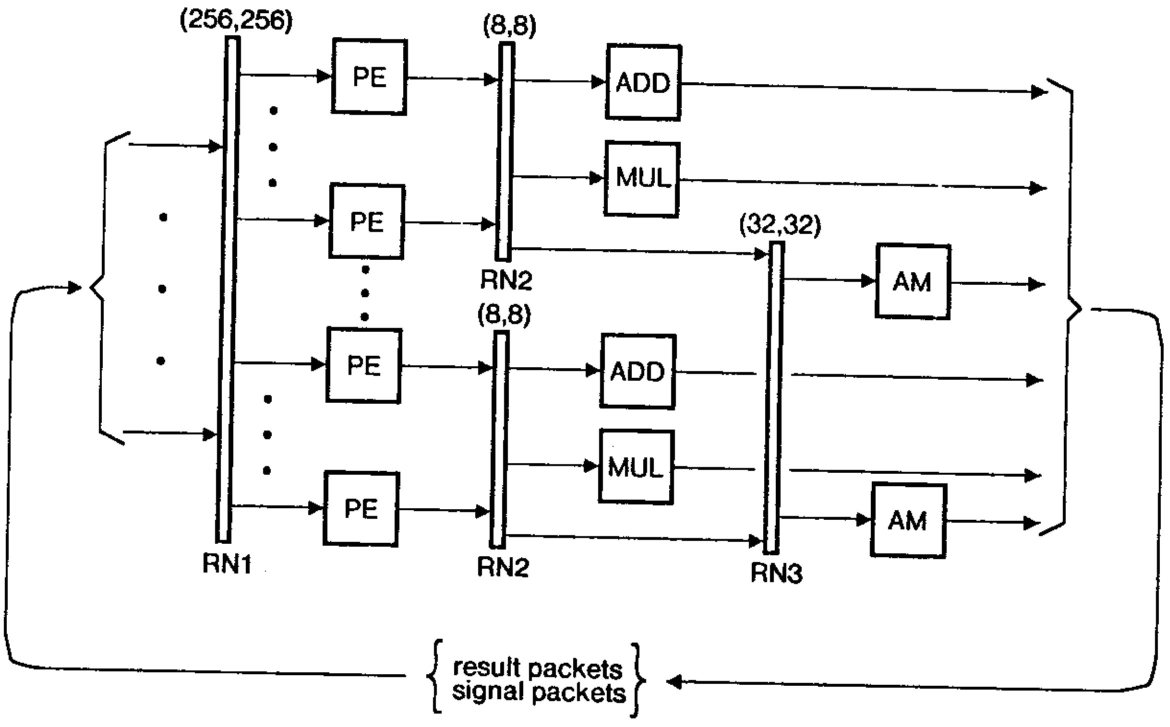


Figure 17. Data Flow Supercomputer.

is shown in Figure 17. The main units of the machine are:

- PE: Processing Elements. These units hold instructions of the machine level program. They receive result packets from other instructions and determine which instructions are ready for execution.

- ADD, MUL: Functional Units. These units perform the basic scalar operations for floating point numbers.

- AM: Array Memories. These memory units hold the array values that form the structured data of the computation to be carried out.

- RN1: Routing Network. This network delivers result packets to the cell block units that contain their target instructions.

- RN2: Routing Network. This network transmits ready instructions (with their operands) to an appropriate functional unit, or to an array memory unit if an operation on structured data is called for.

- RN3: Routing Network. This network routes instructions that operate on structured data to the particular memory module that holds the relevant data.

Each of the routing networks is built of individual two-by-two router units. In general, assuming n is an integral power of two, an (n, n) routing network will consist of $log_2(n)$ stages, each stage having $n / 2$ two-by-two router units.

In a static data flow computer instructions of the machine-level program are assigned to processing elements by the compiler and program loader, and this allocation is fixed for the duration of the computation. The operands of instructions are stored as part of the instructions, so an instruction cannot be reused until the result of is previous execution is consumed.

Our method of determining performance is to find the rate of transmission of operation, result and signal packets that must be supported by hardware units to achieve a target speed of computation. Then, using assumed speeds for the various types of hardware units, we can find how many units of each type are required, and the needed capacity of the routing networks. Finally, we must be convinced that sequencing constraints in the machine code do not prevent achievement of this computation rate.

The parallelism supported by our machine code for the weather model lies in the concurrency of computing the five physical quantities for all layers of the atmosphere concurrently and in the pipelining of computation over the sequence of i, j pairs that define the horizontal grid. It is this high degree of concurrency that makes fast computation possible even though complete execution of an individual instruction may require several microseconds.

We set as our speed objective completion of one time step in five seconds of computing. This is roughly twenty times faster than the GISS weather model currently runs on a CDC 7600 machine. We start by finding how many and what types of operation packets must be processed for one time step of the computation.

Analysis of the weather code in VAL shows that for each time step the dynamics computation requires 216×10^6 addition/subtraction operations and 144×10^6 multiplication operations. We know that arithmetic operations in the dynamics computation account for about 40 percent of the total arithmetic operations per time step for the dynamics and physics computations. Thus the total numbers of additions and multiplications are about 540×10^6 and 360×10^6, respectively.

The number of operation packets sent to the array memories is more difficult to determine. A lower bound may be found by assuming one read operation and one write operation for each element of the data base per time step; this yields

$$2 \times 37 \times 144 \times 87 = 927{,}072$$

The actual number will be increased from several sources: FIFO buffers implemented in the array memory; retrieval of auxiliary data stored in array memory. From our understanding of the GISS code, we estimate that the number of array memory packets generated will be of the order of 50×10^6. Even this is far less than the main memory accesses required in running the GISS code on a conventional computer because almost all storage of intermediate results in the data flow machine is in the local memories of the processing elements.

In addition to operation packets for arithmetic operations and memory accesses, we estimate that about 175×10^6 miscellaneous operation packets will be generated. Note, however, that execution of many ID and MERGE instructions, Boolean operations, and other control instructions will take place entirely within processing elements without generating packet traffic through the routing networks, functional units and array memories.

Thus our estimate of the operation packet traffic required to complete one time step every five seconds is:

Packet Type	*Packet Rate*
Addition Packets	108 MHz.
Multiplication Packets	72 MHz.
Array Memory Packets	10 MHz.
Miscellaneous Packets	35 MHz.
Total	225MHz.

These numbers suggest using the following quantities of machine parts:

Type of Unit	*Number of Units*
Processing Element	256
Array Memory Module	32
Add Unit	128
Multiply Unit	96

In this configuration, each processing element of the machine must transmit operation packets at the rate of 0.88 MHz. The addition units must operate at 0.84 MHz and the multiply units at 0.75 MHz. The 32 array memory units can run at an average speed of 0.31 MHz.

The three types of routing network are configured as follows:

Network	*Configuration*	*Number of two-by-two Routers*
RN1	(256,256)	1024
RN2	(8,8)	12
RN3	(32,32)	80

The machine shown in Figure 17 includes two forms of memory: instruction memory in the processing elements and array memory units to hold the data base for the application. To estimate the amount of instruction memory required, let us suppose that one instruction cell is needed for each add, multiply, miscellaneous, or memory operation performed in processing one vertical cell of the grid. This yields

$$(540 + 360 + 175 + 50)\times 10^6)/(144 \times 87) = 89{,}800$$

instruction cells. If each instruction uses four 32-bit words then an instruction memory of 4K words in each processing element, or one million words total, will fill the bill with lots of room to spare for implementing FIFO buffers and other functions. If the array memory units each contain 64K words, then the total array memory is two million words compared to the primary weather application data base of

$$37 \times 144 \times 87 = 463{,}536$$

words.

For the routing networks to support this computation rate they must be able to transmit one operation packet each microsecond. It is attractive to suppose that each operation packet is transmitted as a sequence of eight 16-bit bytes. In this case the links connecting the various hardware units must transmit bytes at eight MHz, which is reasonable.

To be sure that the processing rates discussed above are achievable with the machine program structure we have outlined, we must check that data dependency constraints in the code do not limit the speed of computation. By the *instruction processing time* we mean the time interval from the instant an instruction cell becomes enabled to the instant all result and acknowledge packets have been received by their target instruction cells. The instruction processing time must be small enough that the required rate of pipelining can be maintained. Since an instruction cell can be fired again only after its target instruction cells have fired, it is sufficient that the instruction processing time is less than half the inter-packet interval of pipeline operation.

For the program structure we have explained, the inter-packet time interval is the five seconds allowed per time step divided by the number of points in the horizontal grid:

$$5.0/(144 \times 87) = 400 \text{ microseconds.}$$

There should be no difficulty in achieving an instruction processing time far below half of this interval.

11. Conclusion

The analysis presented shows that, at least for one numerical computation of practical interest, a data flow supercomputer should achieve performance beyond that of conventional supercomputers. It seems likely that similar results will be obtained for other important large scale computations such as seismic modeling of the earth's crust, numerical wind tunnel experiments for the study of turbulence, finite element methods applied to building structures, and calculations for nuclear reactor and weapons design.

Our analysis shows only that data flow computers of a given configuration can achieve a specified level of performance. The cost of designing and fabricating the parts for constructing these machines has not been determined. The construction of practical data flow supercomputers depends on successful production of custom design LSI parts meeting the postulated level of throughput.

Further work is needed in other areas as well. We believe it is possible to construct an optimizing compiler that could generate the kind of machine level program structures discussed in this paper from a VAL program for the global weather model with minimal advise from the programmer. Yet the validity of this claim has not been demonstrated and we do not know to how broad a class of programs our techniques will apply.

12. Acknowledgments

The work leading to the supercomputer design proposed in this paper was mostly done by the Computation Structures Group of the MIT Laboratory for Computer Science. The graduate students who have participated in the development of concepts and methodology have been a continuing source of inspiration. They are:

William B. Ackerman	Sheldon Borkin	G. Andy Boughton
J. Dean Brock	Randal E. Bryant	Peter J. Denning
David Ellis	John Fosseen	Gao Guang-Rong
D. Austin Henderson	Earl C. Van Horn	Paul Kosinsky
Clement K. C. Leung	John Linderman	Fred Luconi
Suhas S. Patil	Chander Ramchandani	George Rodriguez
Joseph Qualitz	Kenneth W. Todd	Kung-Song Weng

We wish to acknowledge the efforts of Bill Ackerman, Dean Brock, Jim McGraw, and Charles Wetherell for their collaboration in the development of the VAL language, and in the design and programming of a translator and interpreter to run on the Decsystem 20 at the MIT Laboratory for Computer Science. We also acknowledge with appreciation the Data Flow Engineering Model, a test bed for evaluating proposed architectures, which has been designed and constructed by Andy Boughton, Clement Leung, Willie Lim, and Ed Shaw.

Professor Arvind, our colleague on the MIT faculty, is actively engaged in a significantly distinct approach to the realization of practical computers using data flow concepts. Yet our two projects have much in common and the free exchange of knowledge and experience has reinforced our confidence in the directions of our work.

It has also been satisfying to see data flow ideas taken up by workers at other institutions and to have the opportunity to meet occasionally for an exchange of notes on progress, concepts and methodology. These include:

- Jean Syre and his colleagues at CERT, Toulouse, France, who have built an operating machine based on the concepts of single assignment language.

- Al Davis, whose group at Burroughs built the first (to the best of our knowledge) operating hardware using data driven instruction execution, and is continuing his work at the University of Utah [6].

- The group led by Don Oxley and Merrill Cornish at the Texas Instruments Company, Austin, Texas, which constructed the first operating data flow multiprocessor system and is continuing pursuit of feasible practical forms of data flow computers.

- John Gurd, Ian Watson and their colleagues at the University of Manchester, England, who have built an experimental multiprocessor data flow computer using the tagged token principle [17].

- Robert Keller and his colleagues at the University of Utah, who are engaged in basic research on concepts of data flow program execution using the Lisp language as the cornerstone of their work [12].

- And others who have made worthwhile contributions: Arthur Oldehoeft and Roy Zingg, Phillip Treleaven, and Bruce Shriver.

Our work on data flow concepts began with the graph model of Jorge Rodriguez published in 1967 [15]. Since then we have enjoyed support from the Advanced Research Projects Agency, the National Science Foundation, the Lawrence Livermore Laboratory, the Basic Sciences Program of the Department of Energy, and the NASA Ames Research Center. Through this period the interest and support of the Director of the Laboratory for Computer Science, especially Robert Fano and Michael Dertouzos, has been most helpful and encouraging. We are also deeply indebted to George Michael of the Livermore Laboratory for his recognition of the possible importance of data flow concepts and his continuing interest and support of our work.

References

1. Ackerman, W. B. and Dennis, J. B. VAL -- A Value - Oriented Algorithmic Language: Preliminary Reference Manual. Technical Report TR-218, Laboratory for Computer Science, MIT, Cambridge, MA 02139, June, 1979.

2. Ackerman, W. B. Data Flow Languages. In *AFIPS Conference Proceedings, Volume 48: Proceedings of the 1979 National Computer Conference*, AFIPS, 1979, pp. 1087-1095.

3. Arakawa, A. Design of the UCLA GCM. Department of Meteorology, UCLA, 1972.

4. Arvind, Kathail, V., and Pingali, K. A Dataflow Architecture with Tagged Tokens. Technical Memo TM-174, Laboratory for Computer Science, MIT, Cambridge, MA 02139, September, 1980.

5. Boughton, G. A. Routing Networks in Packet Communication Architectures. Master Th., Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, MA 02139, June, 1978.

6. Davis, A. L. The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine. The Fifth Annual Symposium on Computer Architecture, April, 1978, pp. 210-215.

7. Dennis, J. B. Packet Communication Architecture. Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, August, 1975, pp. 224-229.

8. Dennis, J. B., and K.-S. Weng. An Abstract Implementation for Concurrent Computation with Streams. Proceedings of the 1979 International Conference on Parallel Processing, August, 1979, pp. 35-45.

9. Dennis, J. B. Data Flow Supercomputers. *Computer 13*, 11 (November 1980), 48-56.

10. Dennis, J. B., Boughton, G. A., and Leung, C. K. C. Building Blocks for Data Flow Prototypes. Proceedings of the 7th Annual Symposium on Computer Architecture, May, 1980, pp. 1 - 8.

11. Gurd, J., and Watson, I. Data Driven System for High Speed Parallel Computing - Part 1: Structuring Software for Parallel Execution. *Computer Design 19*, 6 (June 1980), 91-100.

12. Keller, R. M., G. Lindstrom, and S. S. Patil. A Loosely-Coupled Applicative Multiprocessing System. In *AFIPS Conference Proceedings, Volume 48: Proceedings of the 1979 National Computer Conference*, AFIPS, 1979, pp. 613-622.

13. Keller, R. M., Lindstrom, G., and Patil, S. S. Data-Flow Concepts for Hardware Design. COMPCON Spring 80, February, 1980, pp. 105-111.

14. Mesinger, F. and Arakawa, A. Numerical Methods used in Atmospheric Models. Tech. Rep. 17, GARP Publication Series, 1976.

15. Rodriguez, J. R. A Graph Model for Parallel Computation. Ph.D. Th., MIT Department of Electrical Engineering, Cambridge, MA, 1967.

16. Somerville, R. *et. al.*. The GISS Model of the Global Atmosphere. *Journal of Atmospheric Science*, 31 (1974), 84-117.

17. Watson, I. and J. Gurd. A Practical Data Flow Computer. *Computer 15*, 2 (February 1982), 51-57.