MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Laboratory for Computer Science

Computation Structures Group Memo 147

Application of Data Flow Computation to the Weather Problem

by

Jack B. Dennis
Ken K.-S. Weng

(A paper to be published in the Proceedings of the Symposium on High Speed Computer and Algorithm Organization.)

This research was supported in part by the National Science Foundation under grant DCR75-04060, and in part by the Advanced Research Projects Agency of the Department of Defense under contract N00014-75-C-0661.

May 1977

APPLICATION OF DATA FLOW COMPUTATION TO THE WEATHER PROBLEM[1]

Jack B. Dennis
Kan K.-S. Weng
MIT Laboratory for Computer Science

ABSTRACT

Computer processors and memory systems organized to execute programs in data flow form show promise of overcoming the barrier to highly parallel computation without concomitant loss of programmability. The principles and advantages of data flow programming and computer architecture are illustrated in this paper by their application to a general atmosphere circulation model for numerical weather forecasting. The paper develops the structure of a data flow program for a basic global circulation model and discusses the performance achievable for this computation by a data flow computer.

I. INTRODUCTION

The past decade has witnessed the evolution of data flow languages from primitive concepts of data driven instruction execution [1, 2, 3] to fully developed schemes for representing algorithms in a form that exposes the natural concurrency of their parts [4, 5, 6, 7]. Recently, several interesting proposals have been advanced for organizing computer hardware to interpret data flow programs in a data driven mode [8, 9, 10, 11]. Although each of these authors makes a good case that his proposed system implements a well defined level of data flow language, their effectiveness as instruments for performing practical computations has not been demonstrated.

---

[1]This research was supported in part by the National Science Foundation under grant DCR75-04060, and in part by the Advanced Research Projects Agency of the Department of Defense under contract N00014-75-C-0661.

The aim of the present paper is to help fill the gap by studying the performance potential of a data flow computer. In an earlier study [12] submitted for publication, we evaluated the performance of a limited data flow machine for the fast Fourier transform computation. Here we use a more general data flow language that includes data structure operations on arrays, and we describe the structure of a corresponding extended data flow computer. Since data flow computers differ radically in structure from conventional machines, meaningful comparisons are only possible through the study of specific applications. For the present study, we have chosen a global general circulation model (GCM) for numerical weather forecasting.

In Section II of the paper we describe the general circulation model; the structure of a corresponding data flow program is developed in Section III. Section IV presents the overall structure of a data flow computer appropriate for the language used to express the GCM computation. In Section V the manner in which array operations are handled by the machine is studied in detail because efficient processing of arrays is crucial to realizing high performance in the GCM computation. Our performance study is presented in Section VI; this analysis shows how a one hundred-fold improvement in computation rate for the GCM computation may be achieved by a data flow computer.

II. THE GENERAL CIRCULATION MODEL

The General Circulation Model used in this study is the GISS fourth order model developed by Kalnay-Rivas, Bayliss and Storch [13] in which the atmospheric state is represented by the surface pressure, the wind field, temperature, and the water vapor mixing ratio. These state variables are governed by a set of partial differential equations in the spherical coordinate system formed by latitude (ϕ), longitude (λ) and normalized atmospheric pressure (σ). In this fourth order model, the computation is carried out on a three-dimensional grid that partitions the atmosphere vertically into K levels and horizontally into M intervals of longitude and N intervals of latitude of size $\Delta\lambda$ and $\Delta\phi$, respectively.

We denote a value of a state variable (the temperature for example) by T(i, j, k) where i, j and k index over the ϕ, λ and σ coordinates, respectively. The model computes each state variable for the next time instant using "leap frog" integration: Thus the temperature $T^N(i, j, k)$ for the next time instant is

computed from the temperature $T^P(i, j, k)$ for the preceding time instant and the time derivative $\partial/\partial t\ T^C(i, j, k)$ evaluated for the current time instant

$$T^N(i, j, k) = T^P(i, j, k) + 2\Delta t \frac{\partial}{\partial t} T^C(i, j, k)$$

The main computation is the evaluation of the time derivatives of the state variables from the current atmospheric state using the physical laws that govern the atmosphere.

In addition to the main computation, three additional computations must be performed to make the scheme workable: (1) Polar computation -- computation of state variables at the poles is treated as a special case; (2) Filtering -- to ensure stability in spite of the convergence of longitude lines toward the poles, spatial filtering is used to suppress high frequency waves at high latitudes; (3) Sum-of-Neighbors -- since the leap frog integration rule is inherently unstable, an averaging computation is performed once every so many time steps.

The GISS model has been implemented in Fortran and runs on an IBM 360/95 machine equipped with 4 megabytes of addressable core memory. Using a grid having nine vertical levels, 72 intervals of longitude, 45 intervals of latitude, and a time step of five minutes, this implementation can simulate one day of atmospheric activity in about one hour of computer time. Reliable long-range forecasts require that the simulation be carried out on a finer grid for more time steps, and thus demand a much faster processing rate.

III. THE DATA FLOW PROGRAM

To present the structure of the data flow program for the General Circulation Model, we shall use the language of data flow schemas [5] in terms of which the concurrency of execution of the computation on a data flow processor can be easily seen. In practice we envision that programs prepared for execution on a data flow computer will be written in a high-level textual language. The design of a high level language that permits straightforward translation into data flow schemas has been studied by Weng [14].

To construct a data flow program for the GCM computation, we represent the atmospheric state by a nested array data structure; for example, the temperature component T of the state

is of type A1 where

```
type A1 = array 0..M+3 of A2
type A2 = array 0..N+3 of A3
type A3 = array 0..K+1 of real
```

The South and North poles correspond to latitude indices i = 1 and i = M+2, and the values for longitude indices j ∈ [N-1..N+3] are copies of the values for j ∈ [0..4].

The new values of each state variable are computed for i ∈ [2..M+1], j ∈ [2..N+1], and k ∈ [1..K]; the remaining components of the data structure provide neighboring values for the fourth order spacial difference formulas along the boundaries of the horizontal grid.

The overall structure of the GCM computation, represented as a data flow schema, is shown in Fig. 1. In this figure, the notation T(i, j, *) denotes the K+2-element array containing the temperature values for horizontal grid point (i, j); T(i, *, *) is the array containing all temperature values on the i^{th} line of latitude; and T(*, *, *) is the complete data structure of temperature values. The figure shows the blocks making up the data flow computation of the next temperature state T^N(*, *, *) from the preceding state T^P(*, *, *) and current values of all state variables including T^C(*, *, *). The next state data structure and the current state data structure become the current state and preceding state for the next cycle of computation (the data paths and control for this are omitted in Fig. 1 for simplicity).

The data flow schema in Fig. 1 is organized so the parallelism of the GCM computation is exposed in two major ways: First, in the main computation, evaluation of the time derivative is carried out concurrently for all K atmospheric levels. This is accomplished by using K copies of the data flow program appropriate for a single grid point. Second, the main computation program block is coded so sets of data values for successive cells of the horizontal grid are processed concurrently by the several stages of the program block. Thus the streams of K+2-element arrays entering the main computation block are processed in pipeline fashion.

The function of each get operator (defined in Fig. 2a) is to convert each array arriving at its a-input into a stream of component values selected from the array by successive elements of the sequence of integers presented at the s-input. Each get operator in the first rank of Fig. 1 converts the temperature data structure T(*, *, *) into a stream of M arrays where each

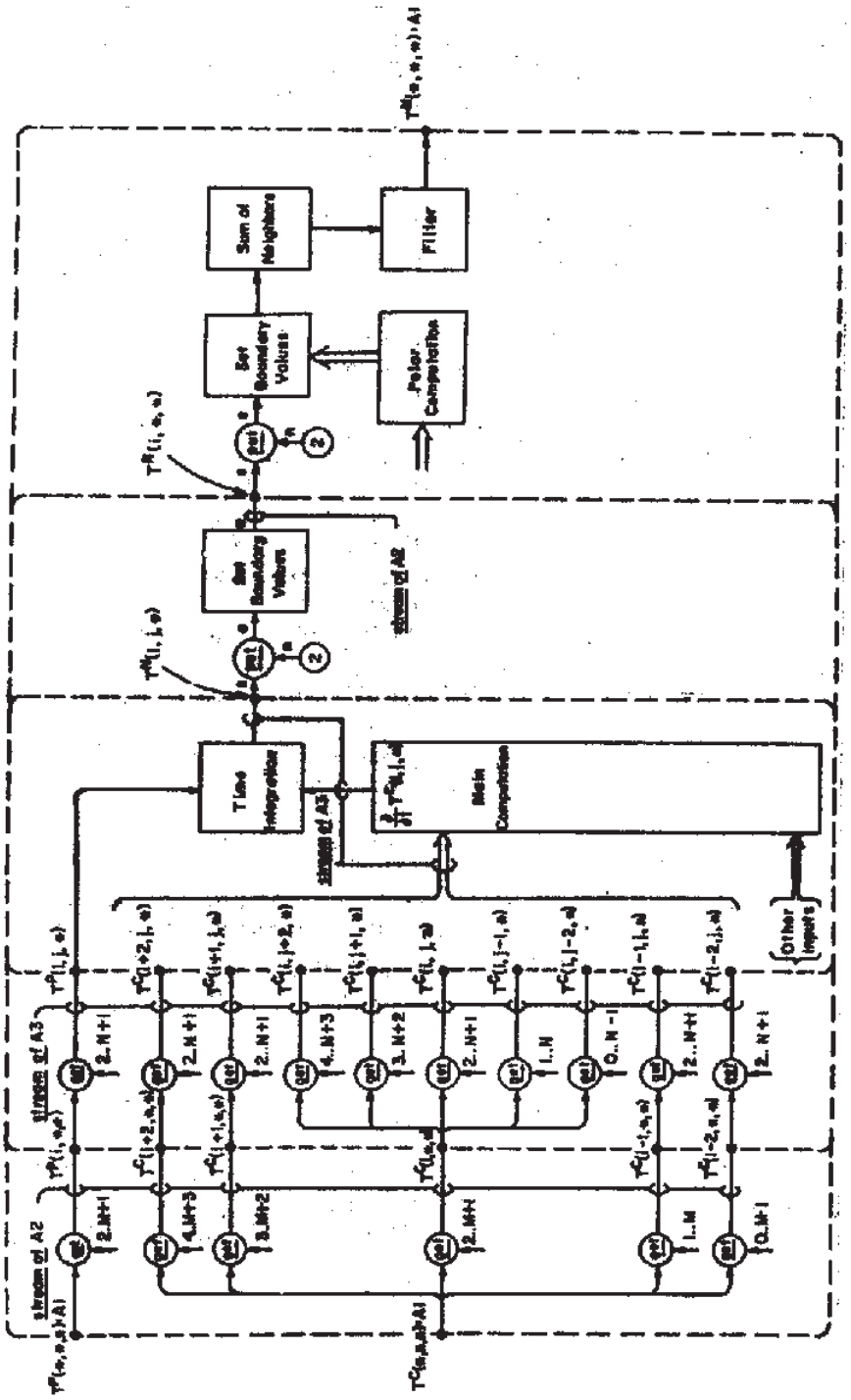

Fig. 1. Structure of a data flow program for the General Circulation Model.

array holds the temperature values for one latitude line. The second rank of get operators further converts the state data structure so each element of the resulting streams is a K+2-element array of values for all grid points having the same horizontal coordinates.

The output of the main computation and time integration is M streams of N arrays apiece, each containing the K+2 temperature values $T^N(i, j, *)$ for one horizontal grid point. The put operator converts each stream into an array of type array 2..N+1 of A3 with components representing $T^N(i, j, *)$, for $j \in [2..N+1]$, the set of new temperature values for the i^{th} latitude line. The block labelled Set Boundary Values adds to the resulting array the temperature values for the boundary indices $j \in [0, 1, N+1, N+3]$, yielding an array of type A2 representing $T^N(i, *, *)$. A final put operator generates the array containing as components the temperature values $T^N(i, *, *)$ for $i \in [2, \ldots, M+1]$. The full array of temperature values $T^N(*, *, *)$ is obtained by adding boundary values for $i \in [0, 1, M+2, M+3]$ using the results of the polar computation. This array is further averaged and filtered to ensure computational stability before becoming the next value of the current atmospheric state.

Data flow schemas for the get and put operators are shown in Fig. 2. These schemas are composed of data flow *actors* [5] interconnected by links that convey data and truth values from one actor to another. In addition to the basic actor types introduced in [5], these schemas use some special actors to implement operations on streams of values [14]: A stream is represented in a data flow schema by a sequence of value-bearing tokens followed by a special token called an *end-of-stream token*. The actor est generates an end-of-stream token; the predicate eos yields true if an end-of-stream token is received and false otherwise.

IV. THE DATA FLOW COMPUTER

Now we are ready to explain how our data flow program for the GCM computation will run on a data flow computer. The organization of a computer that implements the appropriate level of data flow language is shown in Fig. 3. This machine is similar in structure to the data flow processor described in [12], but the machine provides, in addition to the basic scalar operations and control mechanisms, support for data structure operations in its Structure Processor.

Before discussing operation of the Structure Processor, let us review the basic scheme of operation of the data flow

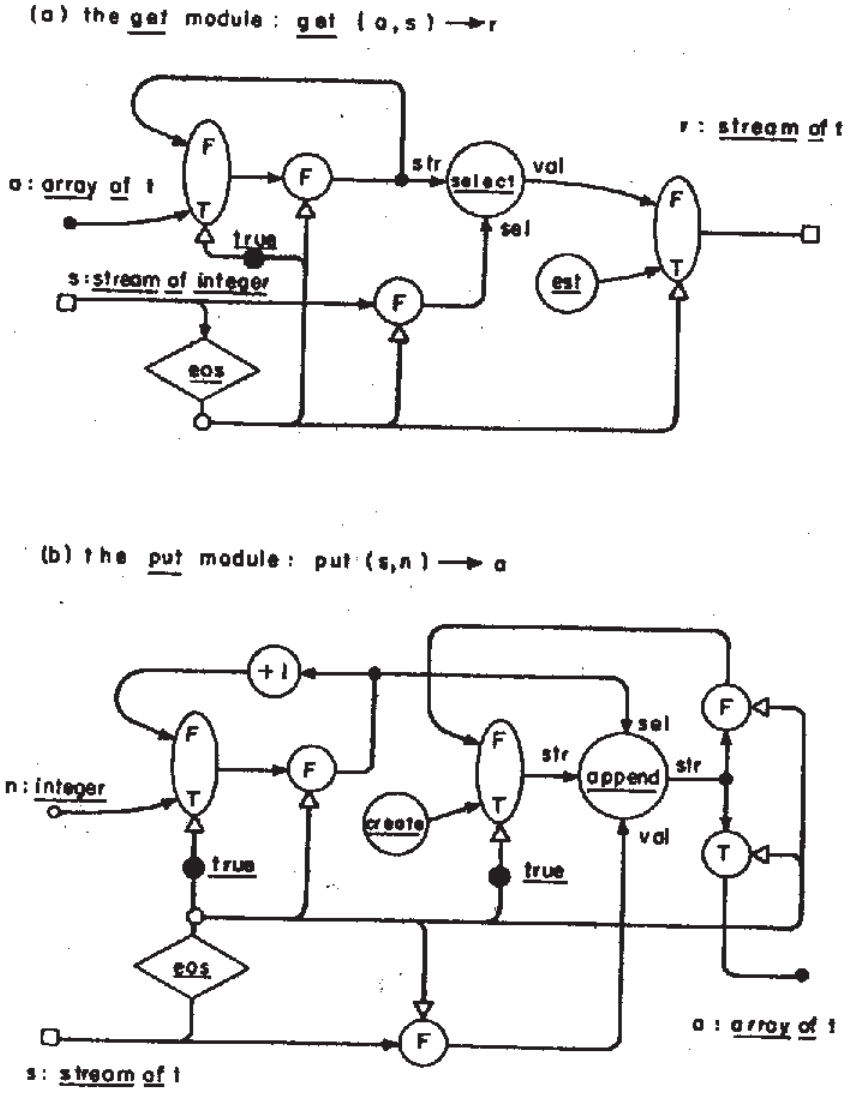

Fig. 2. The get and put operators as data flow schemas.

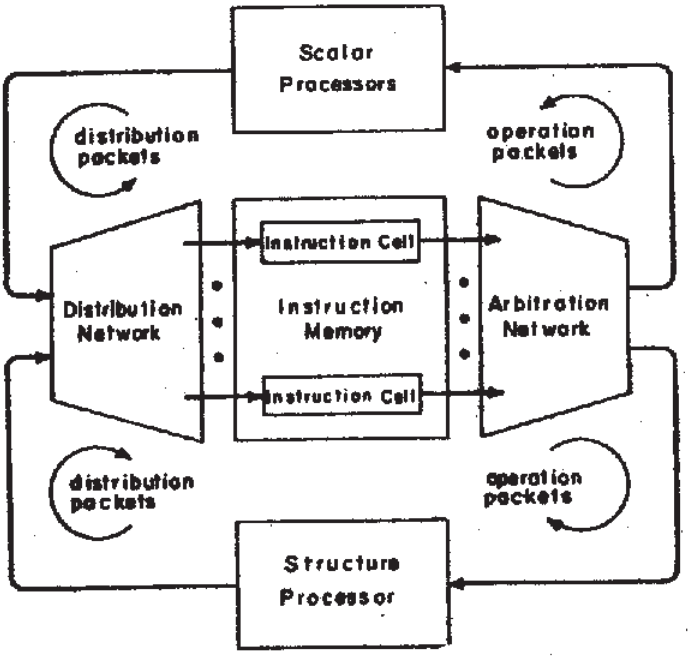

Fig. 3. Structure of the data flow computer.

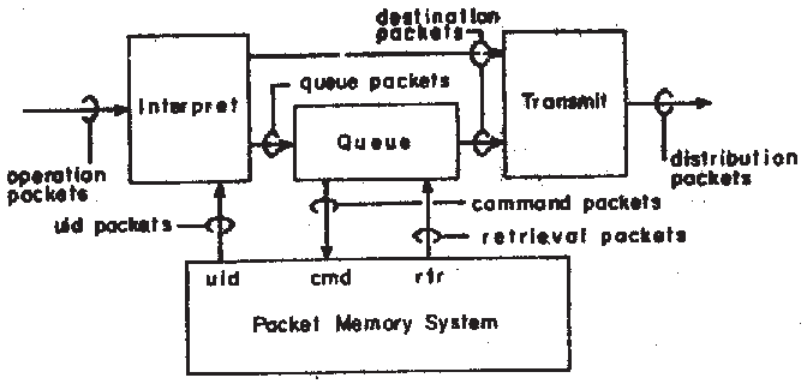

Fig. 4. The Structure Processor.

machine. Each Instruction Cell in the Instruction Memory holds one instruction which corresponds to an actor of a data flow program. Once an Instruction Cell has received (via the Distribution Network) all required operand values and the necessary number of acknowledge signals, the Cell is *enabled* and delivers its contents to the Arbitration Network for transmission to the appropriate Processor. The result value produced by the Processor is transmitted through the Distribution Network to the Instruction Cells which require it as an operand, and *acknowledge* signals are sent to control the enabling of Cells. Even though roughly 20 microseconds may be required for an instruction to be enabled, sent to the Processing Section, executed, and the results transmitted back to other Instruction Cells, the computer is capable of high performance because a large number of instructions may be in various stages of execution simultaneously.

In this form of data flow processor, congestion of the Distribution Network is possible if Instruction Cells are reenabled repeatedly without waiting for previously generated results to be consumed by other Instruction Cells; this congestion can even lead to deadlock -- the complete cessation of computation (see [12]). We avoid these problems of congestion and deadlock by requiring that an Instruction Cell not be reenabled until the data and control packets generated by the previous execution of the instruction have been absorbed by their destination cells. Machine language programs which satisfy this condition are said to be *safe*. Safety is achieved through the use of acknowledge signals generated by an instruction to control the enabling of instructions that produce the data required by the instruction.

V. THE STRUCTURE PROCESSOR

The Structure Processor receives operation packets calling for the data structure operations create, select and append. Earlier concepts for the design of structure processors have been given by Rumbaugh [10] and by Misunas [15]. As shown in Fig. 4, the Structure Processor consists of a Packet Memory System [16] and three units -- the Interpret, Queue and Transmit units -- which make up the Structure Controller.

The Packet Memory: The function of the Packet Memory System is to hold representations of data structures and to provide the means for storing and accessing their components. Each data structure value held in the Packet Memory has a *unique identifier* which serves to represent the structure value in all units outside the Structure Processor. Within the Packet Memory, a data structure value (we consider here only arrays) is represented by

an *item* of the form

$$(i, (c_m, \ldots, c_n), r)$$

in which

i	is the unique identifier of the data structure value.
$c_m, \ldots, c_n$	are either all real number representations, or all unique identifiers of component structure values. Some components may be undefined, and c_k is then nil.
r	is a reference count used to detect when all references to the item have disappeared indicating that the item may be deleted from the Packet Memory.

The state of the Packet Memory is fixed by giving the collection of items held and the set of unique identifiers available for creation of new items. The behavior of the Packet Memory is conveniently specified by giving the state changes for each of the five basic transactions:

Store Transaction: In response to a *store* command packet ⟨STO, i, k, c⟩ at port cmd the item having unique identifier i is modified to have a component $c_k = c$ (the previous value of c_k is lost). If no item exists with unique identifier i, then a new item is created having $c_k = c$ as its sole component, and with its reference count set to one.

Retrieval Transaction: If a *retrieval* command packet ⟨RTR, i, k⟩ arrives at port cmd and an item $(i, (c_m, \ldots, c_n), r)$ exists where $m \leq k \leq n$, then a retrieval packet $\langle i, k, c_k \rangle$ is sent at port rtr.

Up and Down Transactions: The command ⟨UP, i⟩ adds one to the reference count of item i; the command ⟨DWN, i⟩ decrements its reference count by one. If the reference count is reduced to zero by a down command, the item is deleted from the collection of items held by PM and its unique identifier i is added to the set of available unique identifiers, and the reference count for each data structure component is decremented.

Unique Identifier Generation: A unique identifier packet ⟨i⟩ is sent at port uid, and the unique identifier i is removed from the set of available unique identifiers.

In [16] we have shown how the Packet Memory can be structured to handle many transactions concurrently at a high throughput rate.

The Structure Controller: The function of the Structure Controller is to implement the data structure operations create, append and select in terms of the memory transactions supported by the Packet Memory System. In the GCM data flow program these data structure operations occur only in the blocks labelled Set Boundary Values and the get and put routines which transform arrays into streams and vice versa; these routines have been specified as data flow schemas in Fig. 2.

To achieve the desired level of performance, it is important to exploit the capability of the Packet Memory to handle many transactions concurrently, while permitting the memory system to be slow in responding to individual retrieval requests. Thus the get routine as written in Fig. 2a is unsatisfactory because the select actor is not reenabled until after the result of its previous execution has been sent. Consequently, repeated execution of the select actor can occur only at a rate determined by the retrieval delay of the Packet Memory, and no overlap of retrieval requests is realized.

The desired overlapped execution of select operations can be achieved through the choice of an appropriate machine level instruction set and careful design of the Structure Controller. The Interpret unit of the Structure Controller interprets the data structure operations producing sequences of commands that it sends to the Packet Memory System. The Transmit unit generates result and acknowledge packets for distribution to Instruction Cells as called for by the instructions in operation packets. The Queue unit is the heart of the Structure Controller; it holds an entry for each select operation that has been initiated but not completed. Each entry includes the unique identifier and selector that specify the value to be obtained, and the destinations to which copies of the result are to be sent. Operation of the Structure Processor must be such that the results of selection are sent to the destination cell exactly in the order of select actor initiation even though variations in retrieval delay cause retrieval packets to be returned out of sequence from the Packet Memory. Otherwise, the components of the arrays constructed by the put operators of the GCM program would be incorrectly indexed. The function of the Queue module is to ensure that results of select operations are sent by the Structure Controller in the same order as the corresponding operation packets are received. When a retrieval packet is received from the Packet Memory, a matching entry in the Queue is found and the retrieval value appended

to the entry. Result packets are generated from entries containing retrieved values as they reach the end of the queue.

Correct pipelined operation of select actors in the data flow program requires that after a result packet is sent to an Instruction Cell by the Structure Processor, no further result packet is sent until an acknowledge signal has been received indicating that the Instruction Cell is ready to receive it. This provision requires a machine level get routine that is more elaborate than a direct encoding of the scheme in Fig. 2a, but the details will not be covered here. Further discussion of the machine encoding of safe data flow programs may be found in [12].

The put routine in Fig. 2b generates an array by appending successive elements to the empty data structure. If each result of an append operation is viewed as a distinct value, as in the usual data flow semantics, a new copy of the partial array must be created in the Packet Memory each time an append operation is executed. In most cases, as in the put routine, each new partial array value is used only as input to the next instance of append, and it is unnecessary to retain the input array after execution of append. Therefore the append operation is implemented by the Structure Controller by adding a component to an existing item in the Packet Memory System. It is the responsibility of the programming system to ensure that no attempt is made to reference the old data structure value once the append operation is initiated.

VI. PERFORMANCE

We now turn to the analysis of the processing capacity of the data flow processor necessary to achieve the desired one hundred-fold performance over IBM 360/95 on which the GISS model is implemented. The 360 implementation simulates one day of atmospheric activity in about one hour using a 9 X 45 x 72 grid and a five-minute time step. To increase this performance by two orders of magnitude implies that our data flow computer must be able to complete all operations for computing new values of the state variables for one group of K grid points at the same latitude and longitude each 40 microseconds.

For the data flow computer shown in Fig. 3, the computation rate will be determined by which part of the machine is the bottleneck for the flow of operation and result packets. We proceed by determining the throughput required of each part of the machine if the desired performance level is to be achieved.

Analysis of the complete data flow program partially sketched in Fig. 1 reveals that the machine level program will occupy about 13,000 Instruction Cells and that computation of the new state for all grid points with the same horizontal coordinates requires processing approximately 7000 operation packets, of which 2700 are multiplications or divisions, 2700 are additions or subtractions, 900 are data structure operations, and 700 are other miscellaneous operations. If the data flow computer is to complete this processing in 40 microseconds, the Scalar Processors must be able to handle operation packets at 150 MHz and the Structure Processor must be capable of handling data structure operations at 25 MHz. The routing networks must be able to perform packet switching at 175 MHz. These rates may be achieved by using many processors and structuring the Arbitration and Distribution Networks for concurrent transmission of many packets.

In addition to these throughput requirements, we must ensure that the instruction processing time (the time interval from the instant an Instruction Cell becomes enabled to the instant all result and acknowledge packets have been received by other Instruction Cells) is small enough that instructions are enabled at the necessary rate. If a block of a data flow program is constructed to make the most effective use of the pipeline capability of the data flow computer, the period of repeated use of any actor is twice the instruction execution delay. This is because one execution cycle is needed to compute a result value and forward it to the next actor, and a second cycle is needed to return an acknowledge signal. We conclude that the two routing networks (the Arbitration and Distribution Networks) must be constructed so the instruction execution delay is no more than 20 microseconds.

Finally, the memory access time for retrieval requests handled by the Packet Memory must not be so large that values of the new atmospheric state are not available when they are needed. Since a time step is completed only once every 125 milliseconds, this requirement is easily met. However, the Queue unit of the Structure Controller must be large enough to hold all retrieval requests which have not been completed by the Packet Memory. For the arrival rate of 25 MHz even a one millisecond retrieval delay would require a capacity of 25,000 entries in the Queue, thus the Packet Memory should be implemented with storage devices having an access time well under a millisecond.

VII. CONCLUSION

Our study of the General Circulation Model as a data flow computation shows that a very high computation rate can be realized if the units of our proposed data flow computer operate at the assumed rates. This level of performance results from exposing and exploiting the inherent concurrency of the computation on a global basis. In contrast, the "lookahead" machines such as the IBM 360/195 attempt to discover parallelism through execution-time analysis of data dependencies in a small fragment of a sequential program. The vector and array machines can effectively use their highly parallel operation only to the extent that the programmer (or the compiler) can invent ways of encoding problem data into vectors or arrays that take advantage of the machine's power. Since the high performance of a data flow computer results from exposing large numbers of operations for concurrent execution, the speed with which each operation is executed is not crucial; thus a very powerful machine could be built using a large number of relatively slow logic devices. Since our data flow machines are composed of many units of similar type, these machines are ideal for effective application of LSI technology.

The open questions concerning the feasibility of practical data flow computers are: What physical structure should the Structure Controller and the Packet Memory System have? Can these units, which make up the Structure Processor, achieve the throughput assumed in our analysis? How difficult will it be to construct and debug such a large asynchronous system? How much will it cost to build data flow computers? The last question can be answered only by developing complete logic designs for the critical components of the machine. Each of these questions is under study in the Data Flow Project at the MIT Laboratory for Computer Science.

REFERENCES

1. Seeber, R. R., and Lindquist, A. B., "Associative Logic for Highly Parallel Systems," Proc. of the AFIPS Conference 24, 489-493 (1963).

2. Shapiro, R. M., Saint, H., and Presberg, D. L., "Representation of Algorithms as Cyclic Partial Orderings," Report CA-7112-2711, Applied Data Research, Wakefield, Mass., 1971.

3. Miller, R. E., and Cocke, J., "Configurable Computers: A New Class of General Purpose Machines," Report RC 3897, IBM Research Center, Yorktown Heights, N. Y., June 1972.

4. Rodriguez, J. E., "A Graph Model for Parallel Computation," Technical Report MAC TR-64, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, Mass., 1969.

5. Dennis, J. B., "First Version of a Data Flow Procedure Language," Lecture Notes in Computer Science 19, 362-376, Springer-Verlag, New York, 1974.

6. Kosinski, P. R., "A Data Flow Language for Operating Systems Programming," SIGPLAN Notices 8, 89-94 (1973).

7. Bährs, A., "Operation Patterns," Lecture Notes in Computer Science 5, 217-246, Springer-Verlag, New York, 1974.

8. Dennis, J. B., and Misunas, D. P., "A Computer Architecture for Highly Parallel Signal Processing," Proc. of the ACM 1974 National Conference, 402-409 (1974).

9. Dennis, J. B., and Misunas, D. P., "A Preliminary Architecture for a Basic Data-Flow Processor," Proc. of the Second Annual Symposium on Computer Architecture, IEEE, 126-132 (1975).

10. Rumbaugh, J. E., "A Data Flow Multiprocessor," IEEE Trans. on Computers C-26, 138-146 (February 1977).

11. Arvind, and Gostelow, K., "A New Interpreter for Data Flow Schemas and Its Implications for Computer Architecture," Technical Report 72, Department of Information and Computer Science, University of California, Irvine, 1975.

12. Dennis, J. B., Misunas D. P., and Leung, C. K., "A Highly Parallel Processor Using a Data Flow Machine Language," submitted for publication.

13. Kalnay-Rivas, E., Bayliss, A., and Storch, J., "Experiments with the 4th Order GISS Model of the Global Atmosphere," Proc. of the Conference on Simulation of Large-Scale Atmospheric Processes, Hamburg, Germany (1976), to be published.

14. Weng, K.-S., "Stream-Oriented Computation in Recursive Data Flow Schemas," Technical Memo 68, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, Mass., 1975.

15. Misunas, D. P., "Structure Processing in a Data-Flow Computer," Proc. of the 1975 Sagamore Computer Conference on Parallel Computation, IEEE, 230-234 (August 1975).

16. Dennis, J. B., "Packet Communication Architecture," Proc. of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, 224-229 (August 1975).