Data Flow Computer Performance for the GISS Weather Model

Computation Structures Group Memo 159
March 1978

David R. Nadler

Submitted to the Department of Electrical Engineering and Computer Science at MIT in partial fulfillment of the requirements for the degree of Bachelor of Science.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under contract N00014-75-C-0661.

DATA FLOW COMPUTER PERFORMANCE FOR THE GISS WEATHER MODEL

by
David Randolph Nadler

Submitted to the department of electrical engineering and computer science on January 20, 1978 in partial fulfillment of the requirements for the degree of Bachelor of Science at the Massachusetts Institute of Technology

ABSTRACT

The concept of data flow program representation and a data flow computer architecture are described. A summary of the Goddard Institute for Space Studies Global Circulation Model for numerical weather forecasting is presented. An algorithm for the implementation of the weather model on a data flow computer is described, and the problems of pipelining and secondary memory handling encountered are analyzed. Finally, specifications for a data flow computer capable of a hundred-fold speed improvement over the existing weather program are derived.

THESIS SUPERVISOR: Jack B. Dennis
TITLE: Professor of Computer Science and Engineering

Table of Contents

List of Figures

0 Introduction

A data flow computer is a major departure from existing computer architectures that directly realizes the parallelism inherent in most computations. It can achieve enormous increases in speed over conventional machines by performing many operations concurrently, offering a possible solution to computations (such as the GISS weather model) which are severely handicapped by the speed of contemporary computers.

This thesis is an attempt to look at the data flow computer in perspective; to look at the entire machine as applied to a real-life problem. Prior to this, no detailed implementation analysis of a large scale program on a data flow computer has been performed. By explicitly taking into account the proposed architecture it is hoped that this thesis will illuminate any hereto undiscovered problems with the architecture as well as provide concrete performance results and demonstrate the feasibility of constructing such a machine.

The concept of data flow programming and the architecture proposed to execute a program expressed in data flow representation are developed in the first chapter. The second chapter describes the Goddard Institute for Space Studies Global Weather Model for numerical weather simulation and forecasting. The third chapter explores problems encountered in trying to implement the weather model, and derives specifications for a data flow computer capable of achieving a hundred-fold speed increase over the existing model implementation. The implementation analysis includes discussions of the pipelining and structure manipulation problems as well as presentation of an algorithm for simulation of the GISS model.

1.1 The Data Flow Concept

Until recently attempts at high speed computing have stayed within the realm of simple sequential programming, with emphasis on increased speed via pipeline or vector processing techniques. In a pipelined processor the execution of sequential instructions may be overlapped, but what speed may be gained is had at the expense of very complex logic necessary to insure that instructions executed concurrently are independent. Vector or array processors get very high execution rates by applying the same instruction to a vector of data items concurrently, but few problems are easily expressable in terms of such vector or array operations. Attempts at loosely coupled multi-processor computer systems have tended to show that the cost of mechanisms needed to coordinate the separate processors outweighs the advantages to be gained for any more than a few processors.

The central concept of data flow is the representation of a program as a data-driven rather than instruction-driven. Rather than a sequential list of instructions applied to a set of data, the program is represented as a set of instructions, each of which may be performed when all the necessary input data is available. Because any instructions whose input data are available may be concurrently executed, extremely high processing rates are theoretically possible. A program may be shown as a directed graph, the nodes representing instructions and the links or arcs paths along which data flows. Data flow instructions are referred to as actors, data items as tokens. A link may contain no more than one token at any time. When all necessary inputs to a data-flow actor are available and no tokens are present on any of its output arcs, it is enabled and may be performed or fire.

Typical actors might perform an arithmetic function, or perform a test and yield a boolean result (a true or false token). To handle decisions, special actors are required in addition to simple arithmetic and comparison types. An example of such operations are the merge and true or false gating actors.

A merge actor has two data inputs 'T' and 'F', and a control input. The result is taken from the input specified by the boolean control token, the token on the third input remains available, and in fact need not be present for the merge actor to fire. Thus unlike other actors, a merge may fire when a token is not present on one of the input arcs.

A true or false (T or F) actor has a data input and a control input. If the value of the boolean control token matches the gate type the result is taken from the data input, otherwise the data token is discarded and no output is produced.

A constant actor has no inputs, and generates a stream of constant tokens as fast as they are absorbed by succeeding instructions, subject to the usual restriction that no more than one token may ever be present on its output arc.

An example data flow program to compute the nth power of a number is shown in figure 1.1, along with its high level representation:

```
/* initialize the arc variables i,j to 1 */
for i:=1, j:=1
        /* if finished, yield j as the result */
        if i=n then j
        /* else iterate, replacing i with i+1, j with j × input */
        else iter i+1, j × input;
```

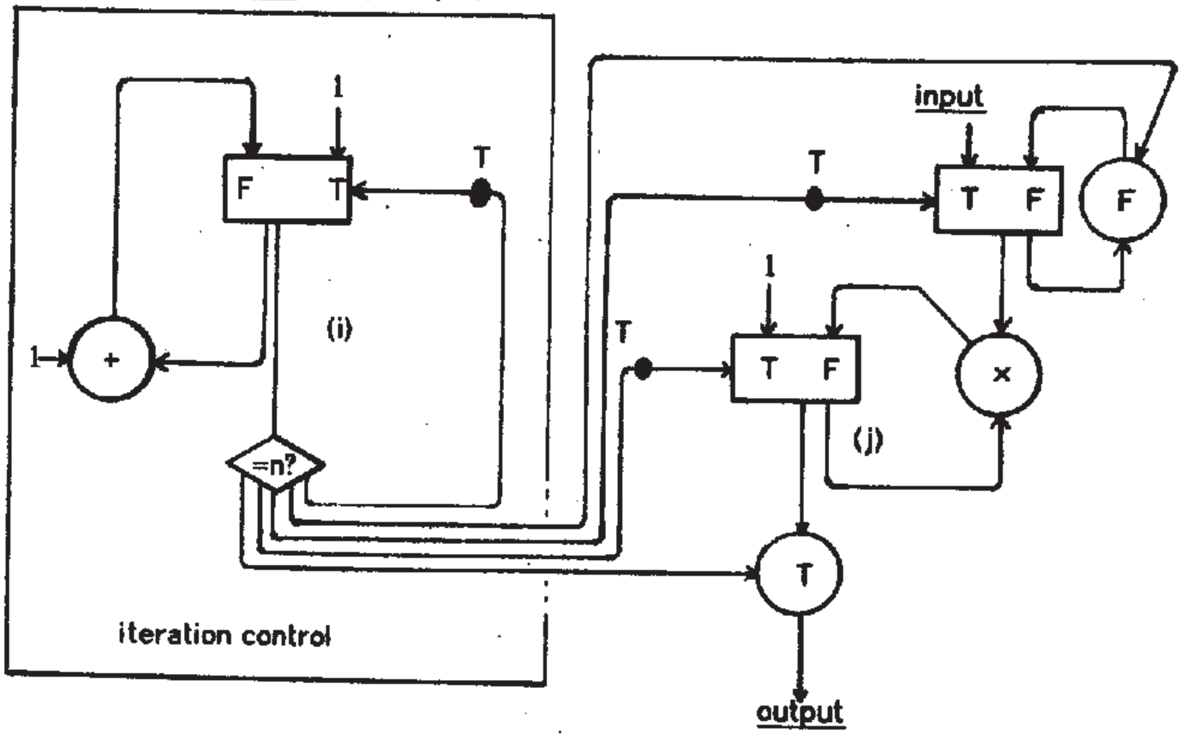


Figure 1.1 - A Sample Data Flow Program

Note that the tokens shown represent initial values. The merge and gating operators are used to control the iteration. The variables i and j exist only as tokens on links, the iteration being performed by replacing i with i+1 and j with j × input. After performing the computation the program will once again be in the state illustrated, ready for use. As shown two merge actors are free to absorb the initial value 'True' control tokens and yield constant one as a result. Because only one token may be present on a link at any one time and an actor may not fire until no tokens are present on any of its output arcs, the comparison operator will not fire until the input has been received and the merge actor fires, as this will remove the last token from the output arcs of the comparison actor.

1.2 A Data Flow Computer Architecture

A data flow computer is a computer suitable for executing programs expressed in a data flow representation. The design of such a computer has been studied by Dennis et al [6,12]. This architecture is an example of a packet communication architecture, in which each module is a separate unit communicating by transmission of asynchronous, fixed length data streams referred to as packets [5]. Such a machine is functionally independent of the speed of any of its component units, and any unit which causes a bottleneck may be upgraded.

The computer consists of a set of instruction cells, an arbitration network, a set of operation units, and a distribution network (figure 1.2). Each instruction cell contains one operation to be performed and an address (or list of addresses) where the result should be sent, plus a storage register for each of the operands. When all of the required operands are present and no tokens are present on any of the output links, the instruction cell is enabled and may fire, and an operation packet consisting of the operands, the operation to be performed, and the destination address(es) for the result is presented to the arbitration network. The arbitration network routes the operation packet to an appropriate operation unit (for example, a floating-point multiplication unit) which performs the operation and presents the result packet(s) to the distribution network. The distribution network forwards the results of the operation to the instruction cell(s) specified in the result packet, to be used as operands of subsequent instructions.

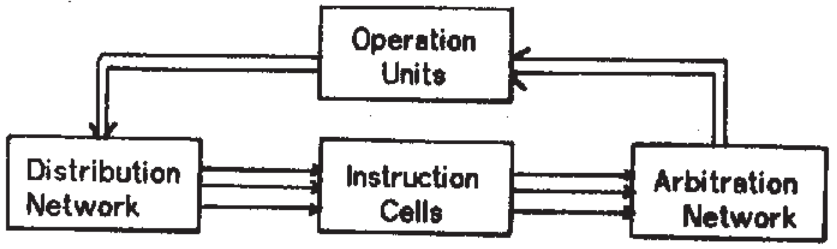


Figure 1.2 - A Data Flow Architecture

The machine described provides no mechanism for ensuring that no tokens are present on any output arc of an instruction cell prior to its firing. This problem is easily solved [11] by requiring acknowledge packets to be sent 'backwards' along each arc to signal when the destination actors are ready to receive new data tokens. An actor may fire only when it has received an acknowledge along each of its output arcs. Each time an instruction cell fires, it sends an operation packet (as before) and additionally acknowledge packets to cells at the other ends of its input arcs. The acknowledge packets are routed similarly to operation packets, through the arbitration network, identity operators, and finally the distribution network to the destination cells.

The cycle time of a data flow machine is the average time delay from the firing of an instruction cell to the time a subsequent cell receives the result and may fire. In a machine with the above mechanism, the cycle time remains constant but the minimum time between succesive firings of an instruction cell doubles, as it may not refire until its operation packet has been processed, received, and acknowledged (requiring two complete cycles).

1.3 Structure Processing in Data Flow

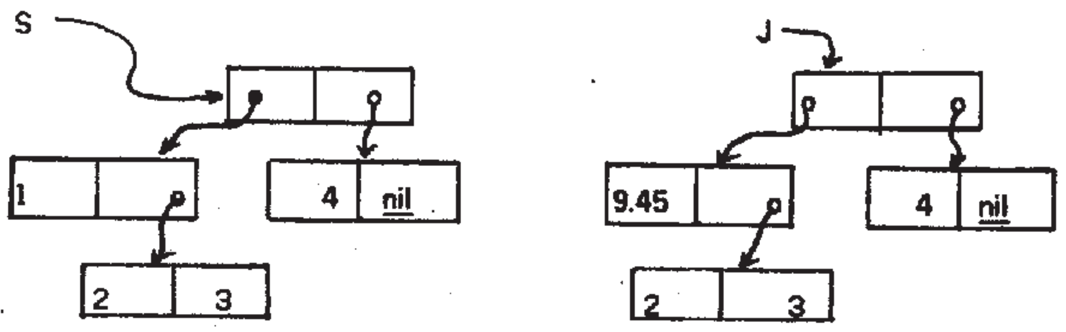
So far, data items have existed only as tokens along data arcs, or correspondingly in the proposed architecture, as operands in storage in instruction cells or packets in transit. In processing large volumes of data, as is the case in the weather problem, this would require huge numbers of instruction cells just to buffer all the data. What is needed is a secondary memory or structure memory system which can be accessed via special actors, and an additional data type 'structure pointer'. In the GISS GCM, all global state variables will be stored in such a secondary memory.

The simplest fully general structure representation allowing for dynamic size change is the binary tree, well known for its use in the programming language LISP [10]. A structure is a pair of entries, each either a simple value, a pointer to another structure, or the null structure pointer nil. Operations that may be performed on a structure are selection of one half of the structure, referred to as a select, and modifying one half of a structure, referred to as an append. The inputs to these instructions are a structure and a selector which indicates which half of the structure is to be selected or replaced. Append also requires the replacement value, which in turn may be a simple value, a structure pointer, or nil. The result of a select operation may be a simple value or a structure pointer, the result of an append is always a pointer to a new structure. For the purpose of illustration define a zero selector to mean the left half of a structure and a one to indicate the right half.

Extending the definition of a selector from a single bit to an arbitrary length bit string gives compound selectors, which will simply be defined as the concatenation of simple select operations:

select[<structure>, ab] ≡ select[select[<structure>,a], b]

Figure 1.3 illustrates the operation of select and append:



select[S, 010] = 2
select[S, 11] = nil
append[S, 00, 9.45] = J

Figure 1.3 - Operation of select and append Actors

Structures can be created by appending to the null structure pointer nil. Deletion is accomplished by destroying the last pointer to a structure or substructure, either by destroying a token of type 'structure pointer' in the data flow program or by replacing a pointer to a substructure with nil in a structure.

A simple linear data array or vector of size 2^d can be represented as a simple tree of depth d-1, requiring d simple select operations to retrieve an arbitrary simple value, or as a linked list of 2^{d+1} structure elements, requiring n simple select operations to retrieve the nth value. The simple tree method is suitable for random access structures, though the cost of waiting for several selections may be quite high as will be discussed later. The linked list representation is well suited for sequential access structures, as the next value is always accessible in a single select. Figure 1.4 illustrates the two representations:

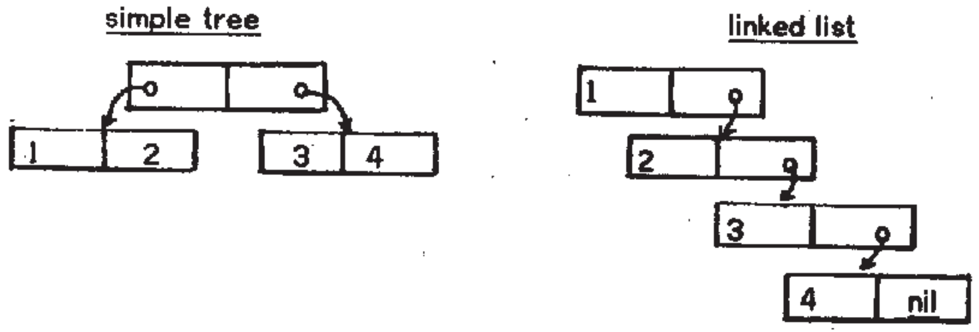
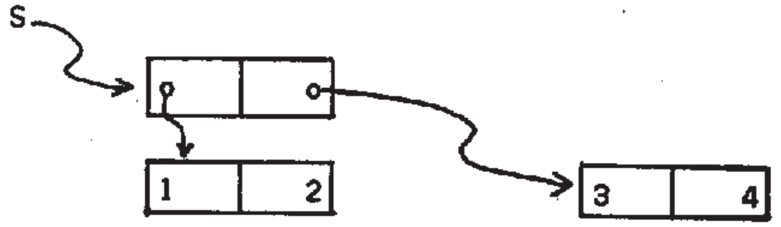


Figure 1.4 - Simple Tree and Linked List Data Vector Representations.

The above description of the secondary memory has assumed that the number of pointers to a structure and its substructure to be indefinite, implying that the structure and its substructures may be shared as in LISP. Unfortunately this can lead to rather severe problems due to the inherently parallel nature of data flow programs.

Suppose two pointers to a structure exist, and in separate and indefinitely ordered portions of a data flow program a select and an append operation are attempted. The results are indeterminate, as the order of the two operations is uncertain, and the select operation may be performed before or after modification of the structure. Requiring that a data flow program be determinate puts a severe constraint on the ability to modify data structures: No structure or substructure to which more than one pointer exists may be modified. It has been proposed by Ackerman and Misunas [1,13] that in a case where more than one pointer exists an attempt to modify a structure should return a modified copy of the structure and leave the original unchanged, this is referred to as the copy rule. Figure 1.5 gives an example of an append operation under the copy rule:

Given that more than one pointer exists to structure S,



the operation append[S, 0, nil] yields the structure S':

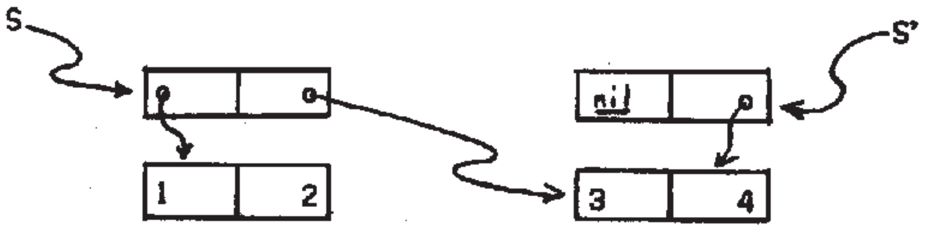


Figure 1.5 - Illustration of the Copy Rule.

In the implementation, the secondary memory must have a mechanism for recording the number of pointers that exist to (or reference count of) each structure element. Any actor that might make copies of or delete a structure pointer (such as a merge actor with multiple output arcs, or a gating actor that absorbs tokens) must inform the structure memory so that the reference count may be adjusted accordingly. Note that in the case of creating copies of the only copy of a structure, the structure memory must acknowledge that the reference count has been updated before the copies are distributed, lest the structure be modified while the reference count is still one. An attractive feature of the reference count scheme is the ease of implementation of garbage collection, any structure element whose reference count drops to zero may be made available for reuse.

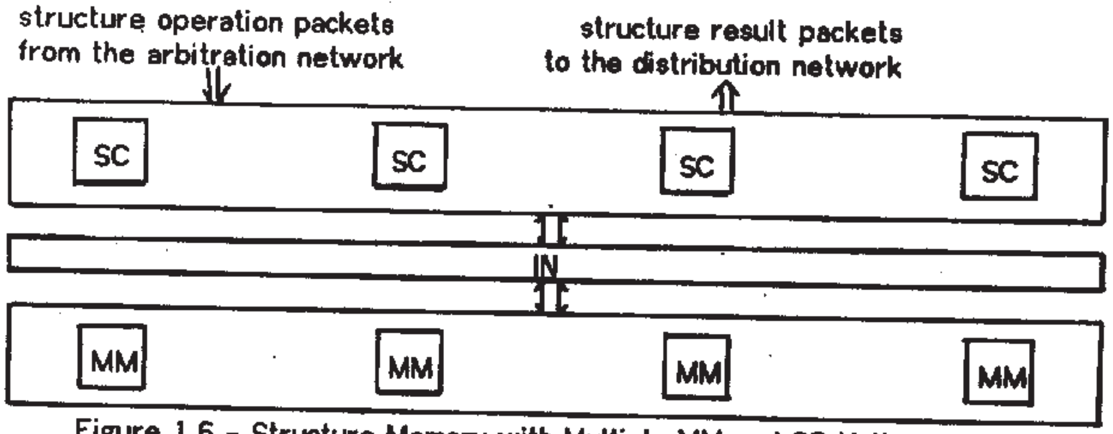
If a series of modifications are to be applied to some substructure, a logical course of optimization would seem to be to attempt to save selection operations by obtaining a pointer to this substructure and applying the modifications to this substructure pointer, thereby eliminating the redundant selections necessary to specify the substructure in each modification operation. Unfortunately, if the principal structure is left intact and a pointer is obtained to the substructure, the reference count of the substructure becomes greater than one and any attempt to modify the substructure using this pointer will cause the copy rule to be invoked and the principal structure will remain unmodified. Two courses of action are possible to achieve the desired modification: use the redundant long selects for each modification operation; or dismantle the structure, modify the desired component substructure, and reassemble the structure. In the implementation, dismantling, modifying, and reassembling the structure requires many instruction cycles, while a compound select can be performed within the secondary memory system in the course of one instruction cycle. For these efficiency reasons the secondary memory should support compound selectors.

1.4 Implementation of a Structure Memory

In this discussion all secondary memory will be assumed to be of a simple random access type. Implementation of a structure memory with rotating access devices has been studied by Ackerman [1], but is not appropriate for the amount of secondary storage required in the weather computation. The simple random access memory unit will be referred to as MM.

The simplest form of structure memory would consist of a single operation unit called a structure controller (SC) and a single MM unit, to which all operation packets dealing with the structure memory would be sent. It is required to process append and select operations with compound selectors, implement the copying rule, and correctly handle reference counts and garbage collection. Such a system would be relatively simple to implement and would function correctly, but would not yield the bandwidth required for effective realization of parallel processing as required in the weather program.

A more general solution is the use of multiple SC operation units, multiple memory units MM, and an interconnection network IN to route requests and replies between the two sets of modules, as illustrated in figure 1.6. The large address space of the single MM unit is now partitioned into many pages, each contained in an MM unit and all accessible concurrently to yield the required bandwidth. Elements of a structure might typically be scattered amongst several MM units. IN would resemble an MM unit with the full address space to each structure controller. The structure controller presents a read request to the IN, the request is passed to the appropriate MM unit, and after some retrieval delay the MM unit returns the result to the SC unit through IN.


structure operation packets
from the arbitration network
structure result packets
to the distribution network
SC
SC
SC
SC
IN
MM
MM
MM
MM
Figure 1.6 - Structure Memory with Multiple MM and SC Units.

As described each structure controller will remain idle while waiting for a response to a request passed to IN. A more efficient design for the SC would allow multiple transactions to be processed concurrently, such a unit has been described in detail by Ackerman [1,2]. The bandwidth of the structure memory can be increased by using more complicated SC units (or simply increasing the number of units), but the bandwidth of the system is also limited by the number and speed of MM units used.

The discussion of the GISS GCM will assume that the implementation of the structure controller operation unit supports compound selectors for both select and append operations. When a structure node is accessed, both halves are available for use in the SC unit. It will be assumed that a single select operation can return both halves along different result arcs, as this simplifies processing of linked lists (this makes it possible to get the next value and a pointer to the remainder of the linked list in one operation).

1.5 Performance Analysis of Data Flow Computers

In order to assess the performance of a data flow computer, one must determine the effective bandwidth of each subsystem of the machine and pinpoint the most limiting areas or bottlenecks. In a well designed data flow computer all subsystems should be matched, so that in all but pathological cases all components are operating at or near their maximum bandwidth. The fundamental reason that a data flow computer can achieve high speed lies in its inherently parallel nature and not in extremely low cycle time, in fact the cycle time of a data flow computer might be quite long when compared to many of today's conventional machines.

In analyzing the weather problem, the program size (and hence number of instruction cells required) will be determined. From analysis of the program structure, the required bandwidth of the structure memory and arithmetic operation units necessary to achieve the desired level of performance will be determined. By making reasonable assumptions about the speed of each component the number of each type of operation unit and structure memory component will be determined.

2.1 Introduction to the GISS GCM

The Goddard Institute for Space Studies fourth order general circulation model is a finite difference model of the earth's atmosphere for numerical weather forecasting, similar to the earlier GISS second order model and the UCLA model [3,9,14]. The model partitions the earth's atmosphere into a three dimensional grid, of which each point has state values for the wind vector *W*, temperature T, and moisture *Q*. An additional two dimensional grid contains state values for the normalized surface pressure Π. The GISS GCM grid is partitioned by latitude (φ), longitude (λ), and sigma (σ) coordinates. The current implementation uses a uniform grid (constant $\Delta\varphi$, $\Delta\lambda$, $\Delta\sigma$), with N intervals of latitude, M intervals of longitude, and K vertical levels. The northern- and southern- most grid lines represent the poles.

The forecast computation is performed using "leapfrog" integration, in which the next state $\underline{N}$ is computed from the previous state $\underline{P}$ and the current state $\underline{C}$ for each state value S. The state change $dS/_{dt}$ is computed using only the current state, with the next state value computed by:

$$S^N(i,j,k) = S^P(i,j,k) + 2\Delta t \cdot d/_{dt} S^C(i,j,k).$$

for each point i,j,k, where the indices specify longitude, latitude, and sigma coordinates respectively.

In addition to state variable forecasts, the model requires periodic application of damping computation to minimize instability. Fourier filtering is used to dampen high frequency oscillations which tend to occur as a result of convergence of longitude lines near the poles, and an averaging computation is applied to compensate for the inherent instability of the leapfrog integration scheme. Also, calculation of evaporation, condensation, radiation, and solar heating terms (E, C, R, Q) are performed infrequently to minimize computation time.

There are four principal forecast equations that must be evaluated for each point i,j,k on the grid, these are the computations of the change rate ($^d/_{dt}$) for ΠT, ΠQ, ΠU, and ΠV, where U and V are the two components of the wind vector W and the state variables are expresed as products with the surface pressure Π. The last principal forecast equation is for computing $^d/_{dt}\Pi$ at each surface point i,j.

Each forecast equation is a finite difference approximation to the continuous field laws which describe the atmosphere. Each of these field laws is expressed in differential form, and the operations $^d/_{d\varphi}$ and $^d/_{d\lambda}$ are approximated by $^\Delta/_{\Delta\varphi}$ and $^\Delta/_{\Delta\lambda}$. Second order differencing uses the immediate or second order neighbors on the φ and λ axis (i,j). Fourth order differencing uses fourth order (two grid units away on the φ or λ axis) neighbors as well as second order neighbors to get a better approximation to the infinitesimal derivative. In the GISS GCM the second order terms are weighted 8 times as heavily as the fourth order terms, and in no case are the diagonal neighbors used.

In the horizontal plane the forecast equations for ΠT, ΠQ, ΠU, and ΠV use fourth order differencing, using differences in state values one and two grid points away along a latitude or longitude line to approximate the infinitesimal derivative. In the vertical direction each equation depends on the second order neighbors, that is the grid points one unit above or below the point i,j,k. The forecast equation for Π depends on the fourth order horizontal neighbors of each point in the column i,j.

The forecast equations for ΠT, ΠQ, ΠU, and ΠV depend not only on the state values at the points illustrated, but also on certain derived quantities computed by the diagnostic equations. These quantities, the geopotential Φ and the vertical velocity $\dot{\sigma}$, are computed using vertical stepwise integration. This involves taking a boundary value (at the earth's surface, vertical level K) and using a relation between the values at the points i,j,k and i,j,k+1 repeatedly to compute the values for all K vertical levels. Furthermore, each of these require a summation of quantities computed for each of the points in the column i,j to determine the boundary value.

The forecast computations performed at the poles are treated as a special case, utilizing a different coordinate scheme to overcome difficulties caused by convergence of the longitude lines. This is a relatively insignificant part of the total computation and will be ignored in this thesis.

2.2 The GISS GCM Equations

The σ vertical coordinate of some point i,j,k is defined as the dimensionless fraction of the time and location dependent surface pressure minus the constant pressure at the top of the model P_{top}, as illustrated in figure 2.1:

$$\sigma \equiv \frac{P_{ijk} - P_{top}}{P_{surface} - P_{top}}$$

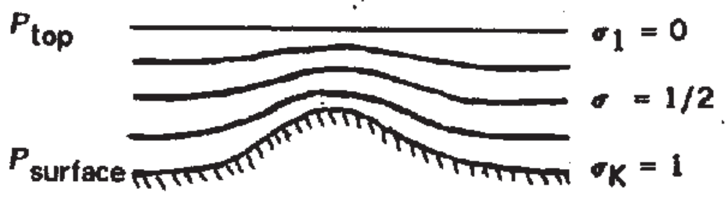


Figure 2.1 - The Sigma Coordinate Scheme.

The K^th level denotes the surface layer ($\sigma_K = 1$) and the 1st level ($\sigma_1 = 0$) denotes the top of the model. The normalized surface pressure is defined as:

$$\Pi \equiv P_{surface} - P_{top}.$$

The fourth order horizontal divergence of a state variable g at the point i,j,k is:

$$
\begin{aligned}
D_{ijk}(g) \equiv \frac{1}{\alpha\cos\varphi_j} \Big\{ & + (1/3\Delta\lambda)(\Pi U_{ijk}+\Pi U_{i+1jk})(g_{ijk}+g_{i+1jk}) \\
& - (1/3\Delta\lambda)(\Pi U_{ijk}+\Pi U_{i-1jk})(g_{ijk}+g_{i-1jk}) \\
& - (1/24\Delta\lambda)(\Pi U_{ijk}+\Pi U_{i+2jk})(g_{ijk}+g_{i+2jk}) \\
& + (1/24\Delta\lambda)(\Pi U_{ijk}+\Pi U_{i-2jk})(g_{ijk}+g_{i-2jk}) \\
& + (1/3\Delta\varphi)(\Pi V_{ijk}\cos\varphi_j+\Pi V_{ij+1k}\cos\varphi_{j+1})(g_{ijk}+g_{ij+1k}) \\
& - (1/3\Delta\varphi)(\Pi V_{ijk}\cos\varphi_j+\Pi V_{ij-1k}\cos\varphi_{j-1})(g_{ijk}+g_{ij-1k}) \\
& - (1/24\Delta\varphi)(\Pi V_{ijk}\cos\varphi_j+\Pi V_{ij+2k}\cos\varphi_{j+2})(g_{ijk}+g_{ij+2k}) \\
& + (1/24\Delta\varphi)(\Pi V_{ijk}\cos\varphi_j+\Pi V_{ij-2k}\cos\varphi_{j-2})(g_{ijk}+g_{ij-2k}) \Big\}
\end{aligned}
$$

The eight coefficients of the sums of g will be refered to as the <u>divergence coefficients</u>. The term $D_{ijk}(1)$ will refer to the value obtained by replacing each sum of g by 1 in the above equation. α is defined as the radius of the earth, and φ_j is the degree of latitude specified by index j.

The forecast equation for surface pressure is:

$$
\frac{d\Pi_{ij}}{dt} \equiv - \Delta\sigma \sum_{k=0}^{K} D_{ijk}(1) \quad .
$$

The stepwise vertical diagnostic equation for the vertical velocity is:

$$\dot{\sigma}_{ijk} \equiv \dot{\sigma}_{ijk+1} - (1/\Pi_{ij})\left(d\Pi_{ij}/dt - D_{ijk}(1)\right) ,$$

subject to the surface level boundary condition:

$$\dot{\sigma}_{ijK} = 0.$$

The pressure at point i,j,k is defined as:

$$P_{ijk} \equiv \sigma_k \cdot \Pi_{ij} + P_{top}.$$

The term PK_{ijk} is defined as:

$$PK_{ijk} \equiv \frac{(P_{ijk}{}^{\kappa+1} - P_{ijk-1}{}^{\kappa+1})}{(\kappa+1)(P_{ijk} - P_{ijk-1})}$$

where $P_{ijk}{}^{\kappa+1}$ is simply P_{ijk} raised to the constant power $(\kappa+1)$.

The stepwise vertical diagnostic equation for the geopotential is:

$$\Phi_{ijk} \equiv \Phi_{ijk+1} + (C_p/2)(PK_{ijk+1} - PK_{ijk})\left[\frac{T_{ijk}}{PK_{ijk}} + \frac{T_{ijk+1}}{PK_{ijk+1}}\right]$$

subject to the surface level boundary condition:

$$\Phi_{ijK} \equiv \Phi S_{ij} + \sum_{k=2}^{K} T_{ijk}\left\{ \frac{\Pi_{ij}\sigma_k \Delta\sigma R}{P_{ijk}} - (C_p/2)\left[\sigma_k\left(\frac{PK_{ijk+1}}{PK_{ijk}} - 1\right) - \sigma_{k-1}\left(\frac{PK_{ijk-1}}{PK_{ijk}} - 1\right) \right] \right\}$$

where C_p and R are constants and ΦS_{ij} is the surface geopotential from a table of surface characteristics. Before use the geopotential Φ undergoes an additional normalization step:

$$\Phi'_{ijk} \equiv \Phi_{ijk} + C_p \theta PK_{ijk} \quad ,$$

where θ is a constant.

The forecast equation for temperature is:

$$\frac{d\Pi T}{dt}_{ijk} \equiv -D_{ijk}(T) - \frac{PK_{ijk}\Pi_{ij}}{2\Delta\sigma}\left[\dot{\sigma}_{ijk}\left(\frac{T_{ijk}}{PK_{ijk}}+\frac{T_{ijk+1}}{PK_{ijk+1}}\right)-\dot{\sigma}_{ijk-1}\left(\frac{T_{ijk}}{PK_{ijk}}+\frac{T_{ijk-1}}{PK_{ijk-1}}\right)\right]$$

$$+ \frac{\Pi_{ij}\sigma_k T_{ijk}}{P_{ijk}}\left[\frac{d\Pi_{ij}}{dt}\right.$$

$$+ \frac{U_{ijk}}{\alpha\cos\varphi_j}\left\{\frac{2(\Pi_{i+1j}-\Pi_{i-1j})}{3\Delta\lambda}-\frac{(\Pi_{i+2j}-\Pi_{i-2j})}{12\Delta\lambda}\right\}$$

$$\left. + \frac{V_{ijk}}{\alpha}\left\{\frac{2(\Pi_{ij+1}-\Pi_{ij-1})}{3\Delta\varphi}-\frac{(\Pi_{ij+2}-\Pi_{ij-2})}{12\Delta\varphi}\right\}\right]$$

$$+ \Pi_{ij}((Q-R)/C_p)_{ijk} \quad ,$$

where $((Q-R)/C_p)_{ijk}$ is the net heating term (solar heating minus radiation loss) from a table computed only periodically.

The forecast equation for moisture is:

$$\frac{d\Pi Q}{dt}_{ijk} \equiv - D_{ijk}(Q) - (1/2)\Pi_{ij}\left\{\dot{\sigma}_{ijk}(Q_{ijk}+Q_{ijk+1}) - \dot{\sigma}_{ijk-1}(Q_{ijk}+Q_{ijk-1})\right\}$$

$$+ \Pi_{ij}(E-C)_{ijk} \quad ,$$

where the term $(E-C)_{ijk}$, the difference between the evaporation and condensation rates, is from a table computed only periodically.

The forecast equations for the two components of the wind vector are:

$$\frac{d\Pi U}{dt}_{ijk} \equiv - D_{ijk}(U) - \frac{\Pi_{ij}}{2\Delta\sigma}\left\{\dot{\sigma}_{ijk}(U_{ijk}+U_{ijk+1})+\dot{\sigma}_{ijk-1}(U_{ijk}+U_{ijk-1})\right\}$$

$$+ \frac{\Pi_{ij}}{\alpha\cos\varphi_j}\left\{\frac{2}{3\Delta\lambda}\left[\Phi'_{i+1jk}-\Phi'_{i-1jk}+\sigma_k RT'_{ijk}\left(\frac{\Pi_{i+1j}-\Pi_{i-1j}}{\Pi_{ij}}\right)\right]\right.$$

$$\left. - \frac{1}{12\Delta\lambda}\left[\Phi'_{i+2jk}-\Phi'_{i-2jk}+\sigma_k RT'_{ijk}\left(\frac{\Pi_{i+2j}-\Pi_{i-2j}}{\Pi_{ij}}\right)\right] \right\}$$

$$+ (F_{ij}+U_{ijk}\tan\varphi_j/\alpha)\Pi V_{ijk} \quad , \quad \text{and:}$$

$$\frac{d\Pi V}{dt}_{ijk} \equiv - D_{ijk}(V) - \frac{\Pi_{ij}}{2\Delta\sigma}\left\{\dot{\sigma}_{ijk}(V_{ijk}+V_{ijk+1})+\dot{\sigma}_{ijk-1}(V_{ijk}+V_{ijk-1})\right\}$$

$$+ \frac{\Pi_{ij}}{\alpha\cos\varphi_j}\left\{\frac{2}{3\Delta\lambda}\left[\Phi'_{ij+1k}-\Phi'_{ij-1k}+\sigma_k RT'_{ijk}\left(\frac{\Pi_{ij+1}-\Pi_{ij-1}}{\Pi_{ij}}\right)\right]\right.$$

$$\left. - \frac{1}{12\Delta\lambda}\left[\Phi'_{ij+2k}-\Phi'_{ij-2k}+\sigma_k RT'_{ijk}\left(\frac{\Pi_{ij+2}-\Pi_{ij-2}}{\Pi_{ij}}\right)\right] \right\}$$

$$+ (F_{ij}+V_{ijk}\tan\varphi_j/\alpha)\Pi U_{ijk} \quad ,$$

where F_{ij} is a friction term used only at the surface level, retrieved from a table of surface characteristics, and T' is a normalization of temperature determined as follows:

$$T'_{ijk} \equiv T_{ijk} - \theta PK_{ijk} .$$

2.3 Performance of the Existing Model

The current GISS fourth-order GCM is implemented in Fortran on an IBM 360/95 with 4 megabytes of memory. The program is configured with $\Delta\lambda=5°$, $\Delta\varphi=4°$, and $\Delta\sigma={}^1/_9$, corresponding to a grid size of N=45, M=72, and K=9. The time step Δt used is 5 minutes. Computation of the solar heating, radiation, condensation, and evaporation terms is performed every half hour, and the application of the filtering and averaging computations are performed every two hours. The program simulates 24 hours (480 time steps) in about 1 hour of computer time. Computations are performed principally with 32-bit floating point quantities.

A conservative estimate of time spent in the principal forecasting (as opposed to the periodic calculations mentioned above) is about 80%, implying that a time step is performed in about:

$$(.80)(3600\text{sec}/24\text{hr simulation})(24\text{hr simulation}/480\text{ time step})$$
$$= 6\text{ sec/time step.}$$

The remainder of this thesis will ignore the periodic calculations (except in determination of the number of instruction cells required), as analysis of these calculations shows that they are similar in form to the principal forecast equations [3,9,14], and the degree of parallelism and increased speed obtainable is about the same.

3.1 A Data Flow Algorithm for the GISS GCM

Analysis of the forecast equations of the GISS GCM reveals that, except for use of the normalized geopotential Φ' of fourth order horizontal neighbors in the wind forecast, the forecast computation for each column i,j is essentially independent. In the existing model each time step is performed in two passes, first the geopotential is computed for each point and then the principal forecast computations are performed.

For the data flow implementation, to serially perform the two pass algorithm over the entire grid would require a very high bandwidth of the structure memory to achieve the desired level of performance. Forecast computations for each column require retrieval of Π and the state variables ΠU, ΠV, ΠT, ΠQ, and Φ' at each and at all the fourth order neighbors of each point in the column., plus retrieval of the previous values of ΠU, ΠV, ΠT, and ΠQ for each point in the column. This translates into about 30xK structure operations per column, just for the principal computation (i.e. not including the first pass for computation of the geopotential). Even if the program is optimized so the forecast computations for successive columns ij at a constant latitude do not perform redundant retrieval of state information of points along the latitude line (four of the five east west fourth-order neighbors of i,j,k are used in the forecast computation at i+1,j,k), the overall program still retrieves the state information at each point five times during the course of the principal computation and once during computation of the geopotential. If the program could be condensed into a single pass, requiring the state information at each point to be retrieved only once and performing the geopotential concurrently with the principal forecast, the bandwidth required of the structure memory would be reduced by almost an order of magnitude.

By concurrently performing the forecast of all the columns along a line of longitude the north south neighbors of a point are available without redundant accesses. By stepping sequentially through lines of longitude and optimizing the program so that state values of points along the same latitude as i,j,k are saved for subsequent forecast computations at i+1,j,k the number of times a point is retrieved per timestep is reduced to one. State value retrieval and computation of the geopotential of the column i+2,j,k is performed immediately prior to the forecast computation at i,j,k. By saving the geopotential similarly to the state values (along latitude j), the need to compute it in a separate pass is eliminated, as is the need to store and retrieve it with the structure memory. The quantities P and PK which are required for the geopotential computation need not be redundantly computed as in the two pass algorithm.

Performing N column forecasts concurrently not only reduces the required structure memory bandwidth, it reduces the required average completion rate for a column forecast (and hence the required cycle time) by more than a factor of N. Speed is gained via parallelism, at the expense of increasing the required number of instruction cells.

Figure 3.1 illustrates the proposed algorithm:

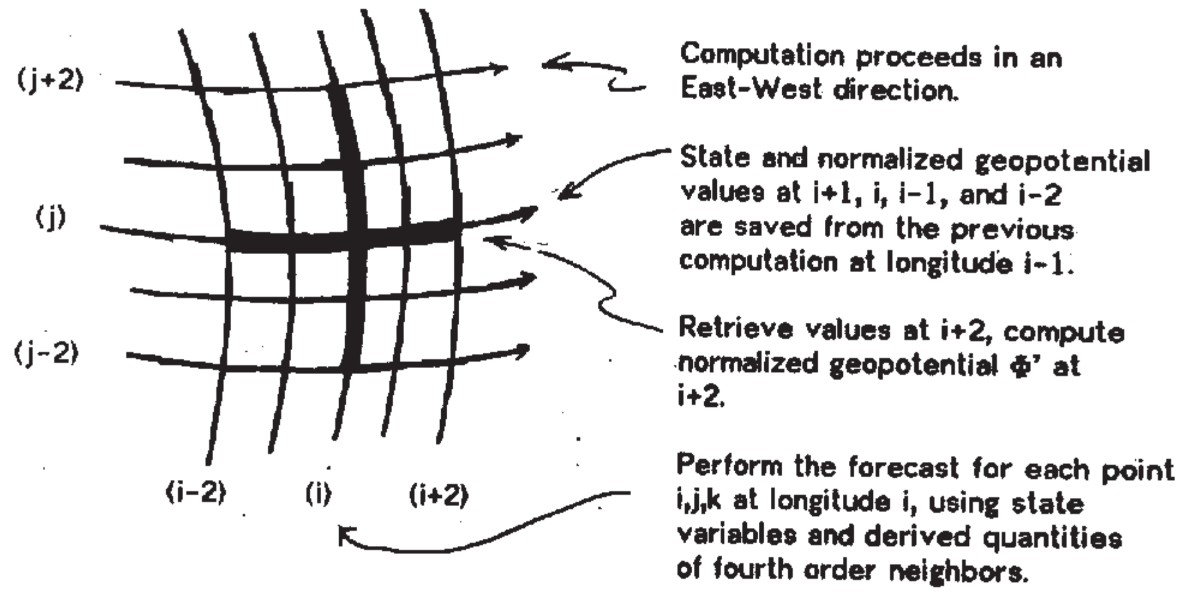

(j+2)
(j)
(j-2)
(i-2)
(i)
(i+2)
Computation proceeds in an East-West direction.
State and normalized geopotential values at i+1, i, i-1, and i-2 are saved from the previous computation at longitude i-1.
Retrieve values at i+2, compute normalized geopotential Φ' at i+2.
Perform the forecast for each point i,j,k at longitude i, using state variables and derived quantities of fourth order neighbors.

Figure 3.1 - Illustration of the Data Flow GISS GCM Algorithm

One further extension to the above algorithm is proposed: the forecast computation for all K levels of each column i,j will be performed concurrently. Because the forecast computation for a single point involves many simple functions, each with several inputs and outputs, and because of the dependence of the point forecast computations on vertical neighbors, expressing the forecast of a column as an iterative loop that performs the point forecast from the surface level to the top of the model requires a very large number of merge and gating operators to route the inputs and outputs to and from the point forecast computation. This means that, at least for relatively small values of K, the concurrent evaluation of all K levels is actually cheaper (in terms of required number of instruction cells and execution delay) than the iterative method. Note that because the stepwise vertical integration used to compute the vertical velocity $\dot{\sigma}$ and the geopotential Φ starts at the surface level (k=K), the top level forecast (k=1) will require more time to complete than the lower levels.

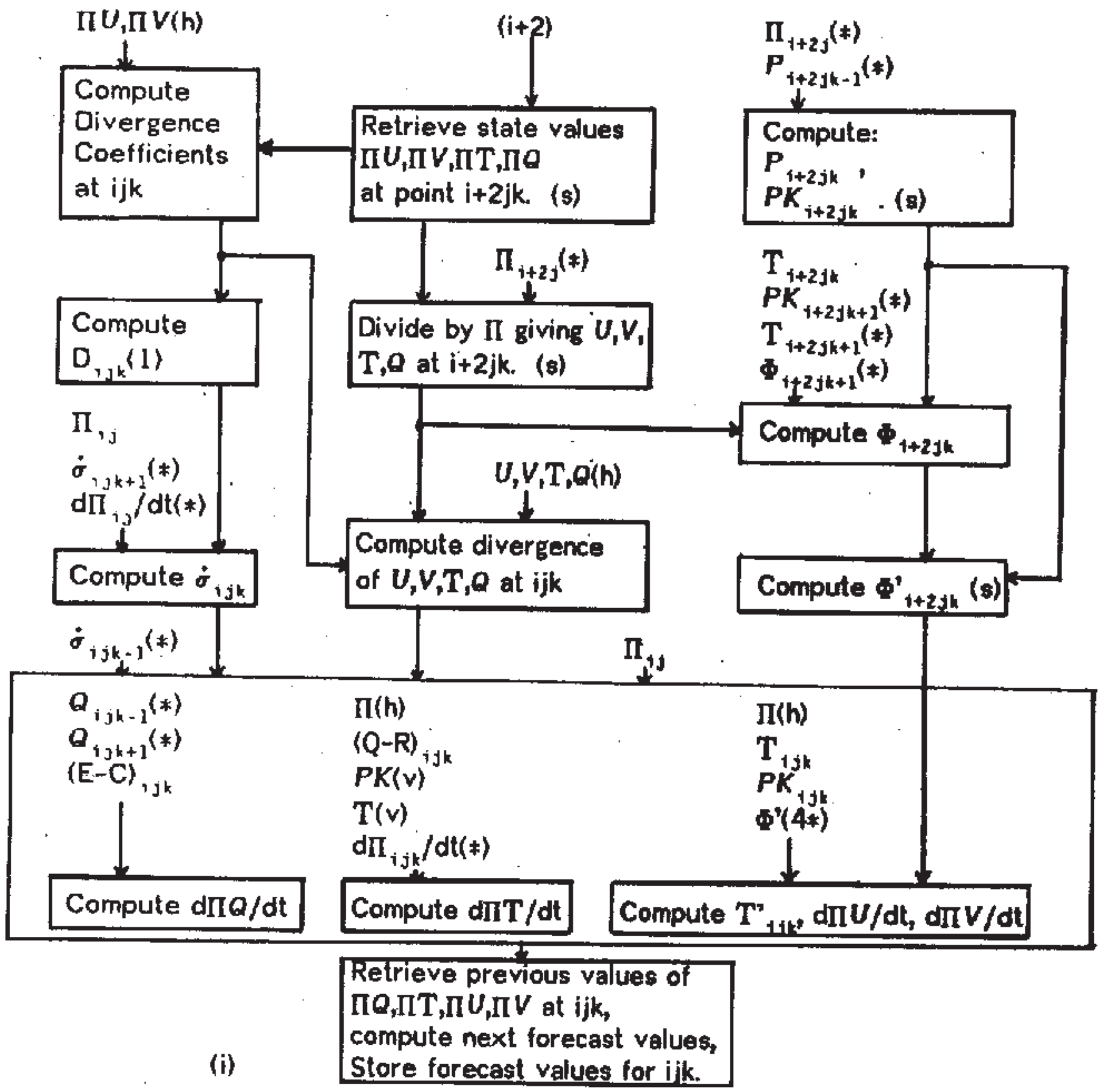
3.2 Performance Analysis of the Data Flow GCM Program

From analysis of the diagnostic and forecast equations of the GISS GCM, the computation at a point i,j,k has been mapped in detail (figures 3.2, 3.3) for an arbitrary point i,j,k and for a surface point i,j,K.

Note that all of the terms involving φ_j and $\dot{\sigma}_k$ are now constants, as a separate data flow subprogram will be generated for each line of latitude j and each vertical level k. Keeping this in mind, the depth and number of operations required of each computational block shown in the program map (figures 3.2, 3.3) have been tabled in figures 3.4 and 3.5. The longest serial dependency (referred to as the depth of a computation or T_{total}) in the forecast computations of a column i,j is the dependence of the wind vector $\boldsymbol{W}$ forecast equation at the top level of the model on the stepwise vertically integrated geopotential Φ'. The total depth of this serial chain is on the order of 70 cycles from the start of processing at column i,j to the forecast of a new wind vector *W*.

To obtain the desired hundredfold increase in speed, the data flow program will have to perform a timestep in 60 msec, for an average forecast computation time of 833 usec per line of longitude. This would require an average cycle time of 12 usec, including the time required to access the structure memory for retrieval of the required state values.

In order to study how the program may be further optimized to achieve the desired level of performance on a more easily realizable machine, the program will be analyzed in two parts. The forecast equations will each be modeled as a series of small sequential functions, where every step of a forecast equation is interpreted as the application of a function. The parts of the program performing the sequential access of the previous and current state structures and then creating the next state structures will be modeled as the copying of simple linear data vectors, with each vector corresponding to the state values lying along a line of latitude.



ΠU,ΠV(h)
Compute Divergence Coefficients at ijk
(i+2)
Retrieve state values ΠU,ΠV,ΠT,ΠQ at point i+2jk. (s)
Π i+2j(*)
P i+2jk-1(*)
Compute: P i+2jk, PK i+2jk. (s)
Compute D ijk(1)
Π i+2j(*)
Divide by Π giving U,V,T,Q at i+2jk. (s)
T i+2jk
PK i+2jk+1(*)
T i+2jk+1(*)
Φ i+2jk+1(*)
Π ij
σ̇ ijk+1(*)
dΠ ij/dt(*)
Compute Φ i+2jk
U,V,T,Q(h)
Compute σ̇ ijk
Compute divergence of U,V,T,Q at ijk
Compute Φ' i+2jk (s)
σ̇ ijk-1(*)
Π ij
Q ijk-1(*)
Q ijk+1(*)
(E-C) ijk
Π(h)
(Q-R) ijk
PK(v)
T(v)
dΠ ijk/dt(*)
Π(h)
T ijk
PK ijk
Φ'(4*)
Compute dΠQ/dt
Compute dΠT/dt
Compute T' ijk, dΠU/dt, dΠV/dt
Retrieve previous values of ΠQ,ΠT,ΠU,ΠV at ijk, compute next forecast values, Store forecast values for ijk.
(i)

g(v) - Value of g at ijk and vertical neighbors ijk+1, ijk-1.
g(h) - Value of g at ijk and fourth order horizontal neighbors.
g(*) - Value is from other level(s) that are being computed concurrently with the computation at ijk, and are not available at the start of the computation.

Figure 3.2 - Point Forecast Computation Map

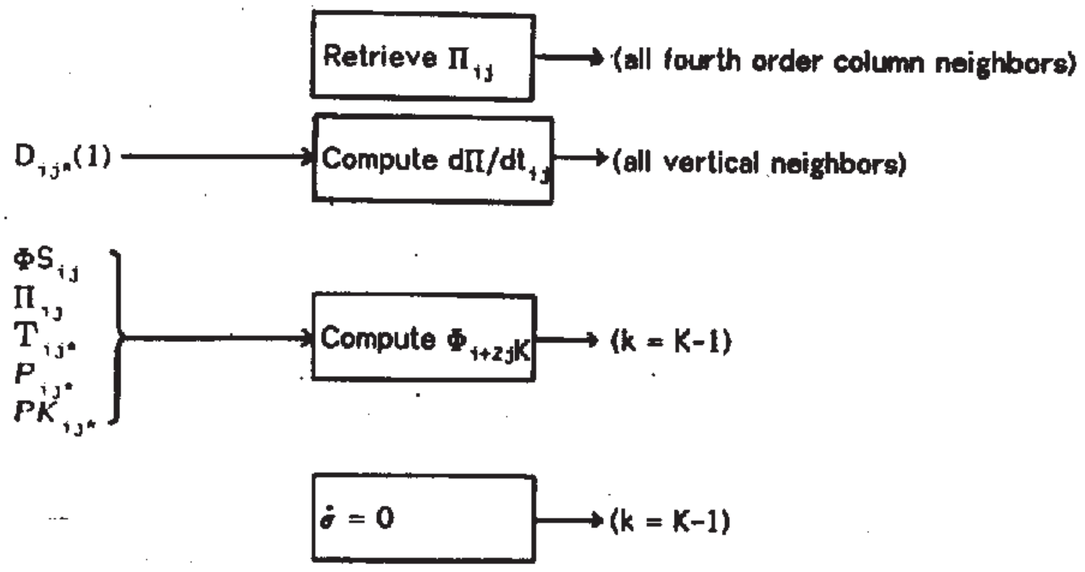


Figure 3.3 - Special Surface Computation Map

Operation	Required instructions st	×	±	/	depth
Select state values at i+2jk	10	-	-	-	1
Divide state values by Π	-	-	-	4	1
Compute divergence coefficients	-	12	8	-	3
Compute D(1) at ijk	-	1	7	-	4
Compute $\dot{\sigma}$ at ijk	-	-	2	1	3
Calculate divergence of each state variable at ijk	-	36	60	-	6
Compute P, PK at i+2jk	-	20	10	10	8
Compute Φ at ijk	-	3	3	-	4
Compute Φ' at ijk	-	1	1	-	2
Calculate $d\Pi Q/dt$	-	7	5	-	7
Calculate $d\Pi T$	-	17	12	5	8
Calculate T' of ijk, $d\Pi U/dt$, $d\Pi V/dt$	-	27	29	2	8
Retrieve previous values of $\Pi Q, \Pi T, \Pi U, \Pi V$	4	-	-	-	1
Forecast $\Pi Q, \Pi T, \Pi U, \Pi V$	-	4	4	-	2
Store forecast next values of $\Pi Q, \Pi T, \Pi U, \Pi V$	4	-	-	-	1
Total requirements at ijk:	18	128	141	22	-

Figure 3.4 - Instruction Requirements for Point Forecast.

Operation	Required instructions st	×	±	/	depth
Retrieve current Π and ΦS, F terms	3	-	-	-	1
Compute $d\Pi/dt$	-	1	8	-	5
Compute Φ_{ijK}	-	45	45	27	10
Retrieve previous Π value	1	-	-	-	1
Forecast next Π value	-	1	1	-	2
Store forecast next Π	1	-	-	-	1
Additional requirements at ijK:	5	47	54	27	-

Figure 3.5 - Additional Instruction Requirements at Surface.

3.3 Structure Manipulation in a Data Flow Computer

First, consider the problem of copying a simple data vector of size 2^d, represented as a simple tree. Each read operation on the old structure requires d simple selects (each simple select in turn requiring a single MM read operation), for a total of $d \cdot 2^d$ MM read operations. In creating the new structure, each node is written twice, the first time with one element nil and the second time replacing this element; thus requiring a requiring $4 \cdot 2^d$ MM write operations. Finally, in the course of creating the a new structure, the top node of the structure of size 2^x is read 2^x-1 times; summing this quantity recursively for each node of a simple tree of size 2^d (depth d) gives a further requirement of $(d-1) \cdot 2^d$ MM read operations. The total cost of copying a simple tree structure of size 2^d or depth d is $(2d+3) \cdot 2^d$ MM operations, or about 2,200 operations for d=7 as in the weather program.

Consider copying a linked list, again of size 2^d. Reading the list requires only 2^d MM operations, but writing the n^{th} element requires n-1 selects (each a single MM operation) followed by two write operations (addition of a pointer to the new node to the current end of the list and creation of the new node). Thus the total requirement for copying a linked list element by element is $3 \cdot 2^d + 2^{2d-1}$ MM operations, or about 8,600 operations for d=7.

A cheaper method of copying a linked list (in terms of MM operations required) involves copying the list twice, each time reversing the order of the list. This method eliminates the expensive compound selection needed to append to the end of a list, appending instead to the top. The total cost for performing two order reversing copies is two read and two write operations per node, for a total of $4 \cdot 2^d$ MM operations, or about 500 operations for $d=7$.

The speed advantage of a linked list representation lies in the ability to retrieve the next item and the structure pointer to the remainder of the list in a single MM operation. If the list is being destroyed while being read, reading a node causes its reference count to drop to zero (and it is placed on the free list), but the reference count of the subsequent node remains one, hence only one MM operation is required per node read. If the list is not to be destroyed, keeping the reference counts correct requires some additional bookkeeping. If pointers exist to the beginning of and the next node n of a list, the reference count of node n is at least two. Retrieving the contents of node n will decrease its reference count and increase the reference count of the subsequent node n+1. This requires reading of both nodes, updating their reference counts, and rewriting them; giving a requirement of 4 MM operations per node read. The cost of reading a list of size 2^d without destroying it is $4 \cdot 2^d$ MM operations. The cost of copying with list reversal and recopying to get the correct ordering is $7 \cdot 2^d$, or about 900 MM operations for $d=7$. Note that this is $(^d/_{3.5})$ times as efficient as the simple tree method for large d, and about 2.5 times more efficient for $d=7$.

3.4 Structure Memory Requirements

Figure 3.6 details the average number of structure manipulations required in the forecast of a longitude line and the cost (in MM operations required) of these manipulations for both linked list and simple tree representations:

	Operation	Cost List	Cost Tree
(1)	Read (destructively, if convenient) the previous state values of U, V, T and Q at each level k.	4·K	4d·K
(2)	Read (non-destructively) the current state values of U, V, T, and Q plus the tabled values if Q and (E-C) at each level k.	24·K	6d·K
(3)	Read (destructively, if convenient) the previous state value of Π, at the surface level only.	1	d
(4)	Read (non-destructively) the current state value of Π and the tabled surface characteristics ΦS and F, at the surface level only.	2	3d
(5)	Store the forecast values of U, V, T, and Q at each level k.	4·K	4(d+3)·K
(6)	Store the forecast value of Π, at the the surface level only.	1	d+3
(7)	Recopy (reading destructively) the forecast vectors for U, V, T, and Q to get the correct ordering at each level k (not applicable to the simple tree method).	8·K	-
(8)	Recopy (reading destructively) the forecast vector for Π to get the correct ordering at the surface level only.	2	-

Total MM operation cost equations and numerical results for the forecast of a longitude line:

$$\text{Linked List} = (40 \cdot K + 16) \approx 17{,}000 \quad ,$$
$$\text{Simple Tree} = ((14d+12) \cdot K + 9d + 3) \approx 47{,}500 \quad .$$

Figure 3.6 - MM operation costs for Simple Tree and Linked List methods.

Note that the linked list version of the program spends about 20% of its time in recopying the forecast lists to get the correct ordering. During this recopying time, no computation may be performed and thus to achieve the same level of performance this version requires a 25% increase in processor throughput over the simple tree version. The simple tree version requires 3 times the bandwidth of the structure processor over the linked list version to achieve the same level of performance.

To achieve the desired throughput, the structure memory must be able to perform all of the structure manipulations required for forecast of an entire longitude line (abbreviated as l.l.) in an average of 833 usec. Assuming an MM unit cycle time of 250ns, the number of MM units required to achieve the desired bandwidth may be computed. For the simple tree representation this number is:

no. MM units = (47,500 op/l.l.)(250ns/op)(1 l.l./833 usec) $\approx$ 14 units,

with an average MM operation rate of one operation every 17ns. For the linked list method;

no. MM units = (17,000 op/l.l.)(250ns/op)(1 l.l./833 usec) $\approx$ 5 units,

with an average MM operation rate of one operation every 50ns.

For the current GISS GCM size, the relative cost of using the simple tree representation is negligible, and allows much simpler encoding of the structure handling portions of the program then the linked list method. The speed requirements placed on the structure memory are such that a single SC unit could handle the required transaction rate. The difficulties encountered with the boundary conditions of the globe when using linked lists imply that the simple tree representation should be used, for example the east west neighbors of the zero longitude line are at opposite ends of the list.

The structure memory must contain three complete state variable grids, the surface characteristic tables for ΦS and F, and the values (Q-R) and (E-C) for each point of the grid, for a total of:

$$2LM \cdot (14K+5) \approx 900{,}000 \text{ structure nodes,}$$

or approximately 1 million nodes of secondary memory to be partitioned between the MM units.

3.5 Pipelining in Data Flow

A serial operation being performed upon a vector of data items and yielding another vector may be modeled by the computation shown in figure 3.7:

```
/* Initialize the iteration control variable i */
for i:=0
        /* If not yet finished, perform the ith computation */
        if i<=n
                begin
                result[i] := f2( f1( input[i] ));
                /* Iterate, replacing i with i+1 */
                iter i+1;
                end
        /* Else (finished) return the vector 'result' */
        else result;
```

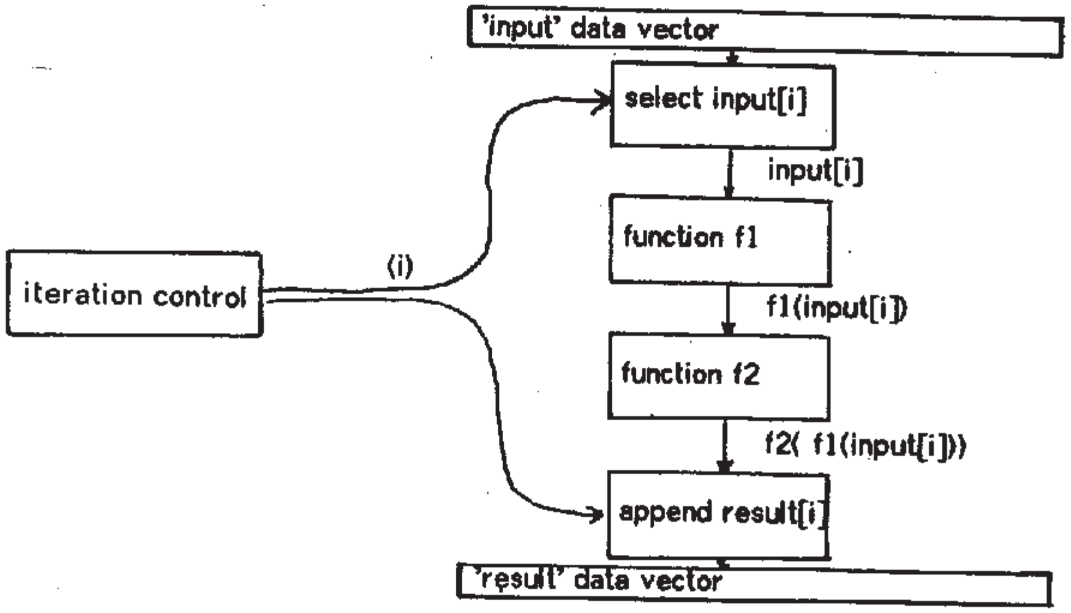


Figure 3.7 - A Pipelineable Data Flow Program.

where the iteration control produces a stream of tokens for i ranging from 0 to n. If the time T_{total} is defined as:

$$T_{total} = (T_{select} + T_{f1} + T_{f2} + T_{append}) ,$$

then time required to perform this computation is

$$T_{computation} = n \cdot T_{total} .$$

At any time during the computation only one of the four serial operations is being performed, the other three being idle.

The speed may be increased by a technique called pipelining. This involves allowing more than one of the serial parts of a computation to be performed concurrently on successive data items; for example if the f2 computation is being performed on the i^{th} item the f1 computation would be operating on the $i+1^{th}$ item and the selection in progress would be of the $i+2^{th}$ input. Each computation or serial subprogram behaves like a single data flow actor, and obeys the same firing rules: it may fire when all requisite inputs are available and no tokens are present on any of its output arcs. The time required to process a single data item input[i] remains the same, but the time for the total computation becomes

$$T_{computation} = (n-1) \cdot T_{max} + T_{total} ,$$

where T_{max} is defined as:

$$T_{max} = \text{maximum}(T_{select}, T_{f1}, T_{f2}, T_{append}) , \text{ or}$$

$$T_{computation} \approx n \cdot T_{max} \ .$$

Thus even though the delay for the processing of one data item remains constant, the average time required for this computation is approximately T_{max}. A throughput increase of a factor of about (T_{total}/T_{max}) can be achieved by increasing only the bandwidth of a data flow processor, requiring neither increased cycle time nor a substantial increase in the number of instruction cells.

As illustrated in figure 3.7, to obey the firing rule the iteration unit may not produce the i+1th token until the append actor has fired and removed the last token from its output arcs. This implies that the selection of the i+1th item will not be performed until the last of the serial processing for the ith item is completed, and the computation will not pipeline.

The situation that prevents a sequential computation from pipelining is an instance of some actor having output arcs leading to both the beginning and end of the computation (In figure 3.7, the iteration control has output links to both the select and append actors). The bottleneck is the path leading to the end of the computation. If this path could carry more than 1 token, the computation would pipeline. The solution involves partitioning the critical arc into several sequential arcs via identity operators ($\underline{I}$), allowing the arc to contain several tokens without violating the firing rule which prohibits more than one token on an arc. The number of identity operators limits the number of serial operators that may be active at one time, refered to as the degree of pipelining. Figure 3.8

illustrates a pipelined version of the computation illustrated in figure 3.7:

```
/* Initialize the iteration control variable i */
for i:=0
          /* If not yet finished, perform the ith computation */
          if i<=n
                    begin
                    /* partition the path from the iteration control
                    ** to the append operator */
                    j:=i; k:=j; l:=k;
                    /* note the use of different indices in the append and select */
                    result[l] := f2( f1( input[i] ));
                    /* Iterate, replacing i with i+1 */
                    iter i+1;
                    end
          /* Else (finished) return the vector 'result' */
          else result;
```

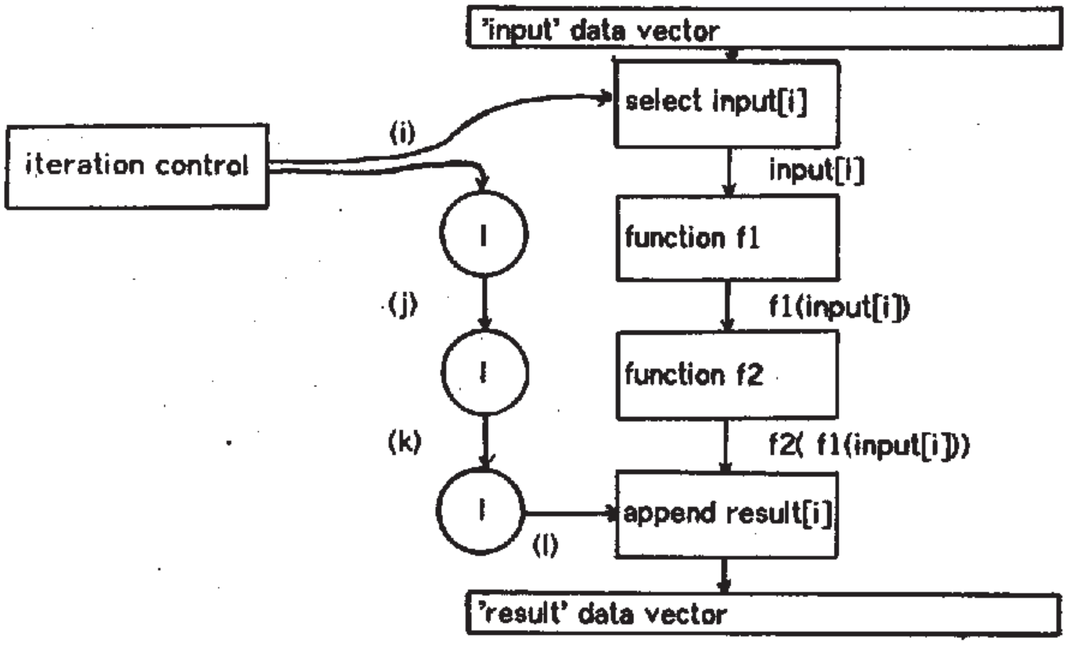


Figure 3.8 - A Pipelined Data Flow Program.

It is significant that this bottleneck condition arises often, and not just in conjunction with the for - iter construct. If two serial computations share a common input and an output of each is used in a common third computation, the degree of pipelining that will be achieved is limited by the depth of the shorter of the two computations. Again, the solution lies in partitioning the critical arc; in this case the output arc of the shorter of the two computations. In general, if a data path splits and rejoins, the computations along the two halves of the split will pipeline correctly when the depths of the two halves are equal. A totally pipelined computation is one in which all paths from the beginning to the end of the computation have the same depth.

In the data flow machine described earlier T_{max} of a totally pipelined computation is twice the average cycle time, as the firing rule requires sending an acknowledge to the last step of the computation prior to the production of each result. Because of this the maximum allowable average cycle time is half the required result production rate for a totally pipelined computation.

It is possible that a high level language compiler could look for pipelineable serial dependencies with the bottleneck condition described above and automatically insert identity actors, the number of identities needed being a function of the length (depth) of the serial operation. In any case, it is always possible to explicitly add identities (as illustrated in figure 3.8) and it is beyond the scope of this thesis to develop such an algorithm. In the remainder of this thesis the performance analysis of the GISS CGM program will assume that identities have been inserted so that the program will pipeline correctly.

Pipelined computations tend to exhibit a property called smoothing, that is the frequency of use of any particular resource (i.e. a floating point multiplier) tends to be constant, rather than in occuring in bursts during which other activity diminishes. Because of this property, the computation of required resources can be performed using average bandwidth rather than using peak bandwidth as is the case with a non-pipelined computation. This fact will be used in computation of the required number of operation units that follows.

3.6 Instruction Cell and Operation Unit Requirements

Using the operation requirements summed in figures 3.4 and 3.5, the number of operation units required to achieve the desired performance level may be computed. Assuming arithmetic operation unit delay times (typical of a modern minicomputer [8]) of 2.5 usec, 3.5 usec, and 7 usec for 32-bit floating point addition, multiplication, and division operators respectively these requirements are:

No. Add. Units = (2.5 usec/op)(60000 op/I.I.)(1 I.I./833 usec) $\approx$ 180 units.
No. Mult. Units = (3.5 usec/op)(54000 op/I.I.)(1 I.I./833 usec) $\approx$ 230 units.
No. Div. Units = (7 usec/op)(10000 op/I.I.)(1 I.I./833 usec) $\approx$ 85 units.

To compute the required number of instruction cells, the additional code for the periodic calculations, iteration control, and pipelining must be considered. Assuming use of the same basic algorithm structure for the periodic calculations the additional code required will be of approximately the same size as that of the central forecast computation. The arithmetic code for the central forecast requires about 3000 instruction cells (figures 3.4 and 3.5).

The iteration control overhead is on the order of 500 instructions per column (including the iteration startup condition handling), or about one sixth of the number of arithmetic instructions. This control mechanism is composed of identity operators for storage and shifting for sequential longitude lines of the ΠU, ΠV, ΠT, ΠQ and Φ' values at each level K and of Π at the surface level. Also included are gating operators used to discard input tokens to all but the geopotential computation for the first four lines of longitude processed, as the forecast computation does not commence until the Φ and state values for ΠU, ΠV, ΠT, and ΠQ are available from all fourth order neighbors.

Checking the data dependence map of the point forecast computation for pipeline bottlenecks shows that several instances of a common input at the beginning and end of a serial path do in fact exist (for example ΠU_{ijk} is used to compute both the divergence coefficients and the wind forecast, and the iteration control causes a bottleneck as in the example of section 3.5). Inserting identity operators to totally pipeline the point forecast computations (making all paths from the top to the bottom of the computation are of uniform depth) requires about 600 identity operators per column, or about one fifth of the number of arithmetic instructions. The required average cycle time is one half the required average forecast time per longitude line (as determined in section 3.2), so the required average cycle time is (833 usec/2), or $\approx$ 420 usec.

The total number of instruction cells required for the forecast computation is equal to N (the number of lines of latitude in the grid) times the sum of the required arithmetic, iteration control, and pipelining instructions required for a column forecast, or about 185,000 instruction cells. The total number of instruction cells required for the whole program is about twice this amount, or about 370,000 instruction cells.

4 Conclusions and Suggestions for Future Research

The achievement of a hundred-fold speed improvement in simulation of the GISS weather model has been shown to be feasible on a data flow computer with on the order of half a million instruction cells and several hundred floating point arithmetic units, requiring an average cycle time of about 400 usec. Such a computer is definitely possible to implement with contemporary technology.

The pipelining problem has been detailed, and the example given shows that at least 15% of the total required number of instructions are likely to be identities. Further research needs to be performed on detection of pipeline bottlenecks and placement of identity operators. One suggestion that has arisen out of the pipeline discussion is that identities be implemented by replacing the operand storage registers of the instruction cells with FIFO (first-in, first-out) queues, thereby decreasing the required number of instruction cells and the required bandwidth of the arbitration and distribution networks.

Though some research is in progress on the structure of routing networks in packet communication architectures [4], how arbitration and distribution networks can best be implemented to satisfy the 400 usec average cycle time needs to be carefully investigated, as it is in these networks that the bulk of an instruction cycle elapses.

5 References

1. Ackerman, W.B. A Structure Memory for Data Flow Computers, Technical Report 186, Laboratory for Computer Science, MIT, August 1977.

2. Ackerman, W.B. A Structure Controller for Data Flow Computers, Computation Structures Group Memo 156, Laboratory for Computer Science, MIT, January 1978.

3. Arakawa, A. Design of the UCLA General Circulation Model, Numerical Simulation of Weather and Climate Report 7, Department of Meteorology, University of California, Los Angeles, July 1972.

4. Boughton, G.A. Routing Networks in Packet Communication Systems, S.M. Thesis in Preparation, Department of Electrical Engineering and Computer Science, MIT.

5. Dennis, J.B. "Packet Communication Architecture", Proceedings of the 1975 Sagamore Computer Conference on Parallel Computation, also Computation Structures Group Memo 130, Laboratory for Computer Science, MIT, August 1975.

6. Dennis, J.B., Misunas, D.P., and Leung, C.K. A Highly Parallel Processor Using a Data Flow Machine Language, Computation Structures Group Memo 134, Laboratory for Computer Science, MIT, January 1977.

7. Dennis, J.B. and Weng, K.-S. "Application of a Data Flow Computer to the Weather Problem", to be published in the Proceedings of the Symposium on High Speed Computer and Algorithm Organization, also Computation Structures Group Memo 147, Laboratory for Computer Science, MIT, May 1977.

8. Digital Equipment Co., PDP 11/70 Processor Handbook, 1976.

9. Kalnay-Rivas, E., Bayliss, A., and Storch, J. "Experiments With the Fourth Order GISS Model of the Global Atmosphere", to be published in the Proceedings of the Conference on Simulation of Large-Scale Atmospheric Processes, Hamburg, Germany (1976).

10. McCarthy, J. et al LISP 1.5 Programmer's Manual, MIT Press 1966.

11. Misunas, D.P. Deadlock Avoidance in a Data Flow Architecture, Computation Structures Group Memo 116, Laboratory for Computer Science, MIT, February 1975.

12. Misunas, D.P. A Computer Architecture for Data Flow Computation, S.M. Thesis, MIT, June 1975.

13. Misunas, D.P. "Structure Processing in a Data Flow Computer", Proceedings of the 1975 Sagamore Computer Conference on Parallel Computation, also Computation Structures Group Memo 129, Laboratory for Computer Science, MIT, August 1975.

14. Somerville, R. et al "The GISS Model of the Global Atmosphere", Journal of the Atmospheric Sciences, Vol. 31 pg 84-117, 1974.

15. Weng, K.-S. Stream-Oriented Computation in Recursive Data Flow Schemas, Technical Memo 68, Laboratory for Computer Science, MIT, December 1977.