GRAPH PROGRAM SIMULATION

BY

EDWARD NELSON

STAN-CS-70-185
OCTOBER 1970

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

GRAPH PROGRAM SIMULATION

Edward Nelson

Computer Science Department
Stanford University
Stanford, California

This research was supported by the U. S.
Atomic Energy Commission under Contract
number AT(04-3) 326, PA 23

ABSTRACT

This reportsthe simulation of a parallel processing system based on a directed graph representation of parallel computations. The graph representation is based on the model developed by Duane Adams in which programs are written as directed graphs whose nodes represent operations and whose edges represent data flow. The first part of the report describes a simulator which interprets these graph programs. The second part describes the use of the simulator in a hypothetical environment which has an unlimited number of processors and an unlimited amount of memory. Three programs, a trapezoidal quadrature, a sort and a matrix multiplication, were used to study the effect of varying the relative speed of primitive operations on computation time with problem size. The system was able to achieve a high degree of parallelism. For example, the simulator multiplied two n by n matrices in a simulated time proportional to n.

TABLE OF CONTENTS

FIGURES

TABLES

INTRODUCTION

Many approaches have been taken to the problem of parallel computation. One set of approaches, characterized by ILLIAC IV, allows only one instruction stream, but allows each instruction to be carried out on many data items simultaneously. This approach does not lead to serious problems of sequencing, but it is suitable principally for problems using large arrays. To take advantage of the fact that most problems require many operations which are independent and can, therefore, be carried out simultaneously requires one to use several independent instruction streams. This leads to sequencing problems, however, since concurrently executing sections of code may refer to the same piece of data in an indeterminate order. One approach to these problems has been to require the programmer to specify where parallel execution may occur and to leave to him the problem of making sure that no conflict may occur between concurrently executing sections of code. This approach is typified by the FORK and JOIN statements proposed for ALGOL. A similar approach is to attempt to isolate the data items which are referred to by more than one piece of concurrently executing code and then to provide semi-automatic protection for these. This is the approach taken by Dijkstra's semaphore system.

These approaches suffer from the fact that the burden of providing parallel execution is on the programmer. The sequencing problem arising in multiple instruction stream parallelism will thus become a source of programming bugs since the programmer will not always use the interlocks correctly. Furthermore, because of the additional programming required to use interlocks etc., the programmer will not take full advantage of the opportunities for parallelism inherent in an algorithm, particularly at a very local (i.e., intrastatement) level.

An approach less prone to error is one which provides for multiple instruction streams where the sequencing, and thus the degree of parallelism, is specified implicitly rather than explicitly. This requires that the program be written in a different representation than that provided by conventional programming languages, since the sequencing implicit in these does not distinguish between those cases in which one operation must logically follow another and those in which there is no such logical necessity. In other words, it is desirable to have a representation in which operations are implicitly simultaneous unless they are logically dependent on one another. Directed graphs provide one such representation. In this representation, the nodes of the graph represent operations performed on data stored on edges directed into the node. A data item has no permanent location in this representation, but rather "travels" along the edges of the graph to the operations which are performed on it. An example of this approach is the computation graph model of Duane Adams. Adams' model allows one to program sophisticated algorithms, such as matrix inversion, in a way which allows both the single instruction stream type of parallelism and multiple instruction stream parallelism down to a very low level.

A program in Adams' model consists of a set of directed graphs called graph procedures. Graph procedures consist of two types of nodes, primitive nodes and procedure nodes. Primitive nodes represent the basic operations performed by the system (addition, multiplication, etc.). Procedure nodes cause invocation of another graph procedure, i.e., they specify that the computation to be performed by that node is the one represented by the named graph procedure. Edges specify the sequencing of the operations performed by the nodes; if there is an edge directed from node i to node j, then the result of the operation specified by node i is an input to the operation specified by node j. The edges act as first-in first-out queues, i.e, the data items are operated on by node j in the order in which they were output by node i. There are two types of primitive nodes, p-nodes and s-nodes. P-nodes can execute when there is at least one data item on each edge directed into the node. If there is more than one data item on each input edge, the operation may be performed simultaneously on each set of input items. This allows the single instruction stream type of parallelism to be performed within the model. In order to insure that multiple instances of an operation terminate in the same order in which they initiated, the model specifies that there be an initiation queue associated with each node. An identifier is placed on the initiation queue for each instance of the operation which is initiated, and that instance does not terminate until its identifier is at the head of the initiation queue.

The other type of primitive node is the s-node. Associated with each edge directed into an s-node is a status bit whih specifies that the edge is either locked or unlocked. An s-node can initiate when there is at least one data item on each unlocked input edge, regardless of whether or not there is data on any of the locked edges. The values of the edge status bits are reset at the end of the operation specified by the node. The new values are a function of the old status values and of the data input to the node from the unlocked edges. Since the conditions for the initiation of an s-node depend on the results of the last operation performed by that node, only one instance of the operation specified by an s-node can be carried out at a time.

Procedure nodes specify that the named directed graph is to be executed using the values on the input edges to the procedure node. They are initiated as p-nodes, so that more than one instance of a given graph procedure may be executed concurrently. Also, the graph procedure named by a procedure node may be the one in which the node is contained so that recursive execution of graph procedures is possible.

This report describes a simulator which interprets Adams's graph programs, carrying out the computations specified by a set of graph procedures and keeping statistics on the timing and resource usage, and it describes experiments performed with the simulator. Simulations were run on a number of small programs, including a matrix multiply program, a quadrature program, and a sort program. The programs were run using varying amounts of data, various speeds for the primitive operations, also with and without allowing multiple instances of a p-node to execute simultaneously. All of the simulations described here were run using the assumption that the machine specified by the simulator had an unlimited number of processors to carry out the operations specified by the primitive nodes and an unlimited amount of memory. Of course, this is an unrealistic assumption. These simulations were run in an attempt to discover the "inherent" resource usage characteristic of the programs and to discover the effect of varying the relative speed of primitive operations independently of effects due to different algorithms for allocating processors in an environment with a finite number of processors. These effects can then be controlled during simulations run in the more realistic environment of a finite machine.

Experiments conducted to discover efficient algorithms for allocating processors in a finite environment will be described in a subsequent report.

THE GRAPH PROGRAM SIMULATOR

The simulator described here may be thought of as a parallel computer, although it was not my intention to simulate any particular machine architecture. As such, it has the following components:

- Storage for graph procedures
- Storage for data (edges), initiation queues, and the status of nodes and edges in an executing graph procedure
- A pool of processors with input and output registers
- Logic for performing the operations specified by the primitive nodes
- Control logic for determining which nodes are ready to execute, assigning processors to those nodes, recognizing that a processor is done, and putting the results on the output edges in the order dictated by the initiation queue.

The first type of storage is static during the execution of a graph program, while the second is dynamic. Besides the above components, the simulator also has the code necessary to gather statistics on the simulation, provide a trace, etc.

Two distinct machine models are possible for the simulator, one in which each processor is a specialized functional unit, able to execute only a single type of primitive node and one in which the processors are all general processors so that each can execute all of the primitive nodes. I will call the first the functional unit model and the second the multiprocessor model. In terms of an actual implementation, the functional unit model has the advantage that it is not necessary to duplicate the decoding and control circuitry required to decode operations in each processor. It has the disadvantage of limiting the flexibility of processor allocation algorithms. In addition, if the mix of functional units available on the machine does not closely match that required by a given program, many of the functional units will be idle much of the time. The savings gained by not duplicating control circuitry may thus be lost to increased inefficiency. The distinction between the two models is not too important in the unlimited resource environment, since it makes no sense to ask what the optimum ratio of adders to multipliers is, for example, if one has an infinite supply of both. In the finite environment, however, the simulator can be used to determine the cost in functional unit idleness of the functional unit model, and these costs could then be weighed against the costs of duplicating control circuitry.

Each processor in the simulator contains three input registers and three output registers by means of which data is gated from and to the edges. A gating bit is associated with each of the registers. For the input registers, these indicate whether the corresponding edge was locked or unlocked and, thus, whether there is data in the register. For the output edges, the gate bits indicate whether or not the processor produced output in that register so that the control circuitry will know whether to gate the contents of the register onto the corresponding output edge. This allows a processor to produce output conditionally. Each processor also has a completion bit and a timer associated with it. The timer is a simulator expedient which allows the processor to execute for a particular number of cycles. A block diagram of a processor is shown in Figure 1a. Figure 1b shows how the processors would be arranged in the functional unit model. The availability queues indicate whether a processor is free or assigned to some node. If it is assigned, they indicate to which node.

Program storage can be divided into two parts: that which is static during execution of the program and that which is dynamic. The static storage contains the graph procedure definition and the dynamic storage contains the edges, initiation queues, and node and edge status flags. For each graph procedure, three arrays are needed. Two are one dimensional arrays with one entry for each node in the procedure. One gives the type of each node (i.e. the operation code), and the other identifies the graph procedure named by the node if it is a procedure node.

The graph itself is represented by a connection matrix whose i, jth entry is non-zero only if there is an edge directed from node i to node j in the graph program. If the entry in the connection matrix is non-zero it is an integer which identifies the edge connecting the two nodes. The static storage is shown in Figure 3.

The dynamic storage consists of node and edge status flags, pointers to edges and initiation queues, the edges and initiation queues, and storage for structured operands. These are shown in Figures 3, 4, and 5. Only the status bits and edge initiation queue pointers (Figure 3) are copied when a new procedure is initiated.

The status bits for a node indicate whether it is idle, ready to initiate, or executing. P-nodes may be both executing and ready to initiate at the beginning of the same simulator cycle, since more than one copy of the node may execute on that cycle. The status bit associated with an edge indicates whether it is locked or unlocked. If the edge is directed into a p-node, its status is always unlocked.

The basic data structure of the graph model is the first-in first-out queue. Queues are used as a basic ordering device to maintain the

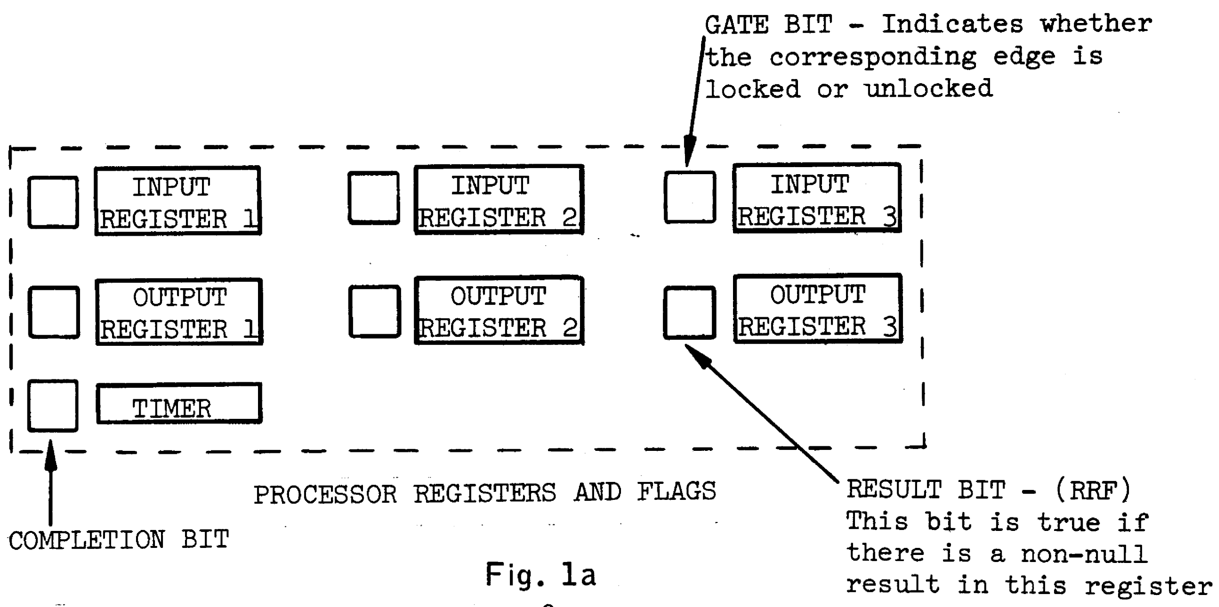

GATE BIT - Indicates whether the corresponding edge is locked or unlocked
INPUT REGISTER 1
INPUT REGISTER 2
INPUT REGISTER 3
OUTPUT REGISTER 1
OUTPUT REGISTER 2
OUTPUT REGISTER 3
TIMER
PROCESSOR REGISTERS AND FLAGS
COMPLETION BIT
RESULT BIT - (RRF) This bit is true if there is a non-null result in this register

Fig. 1a

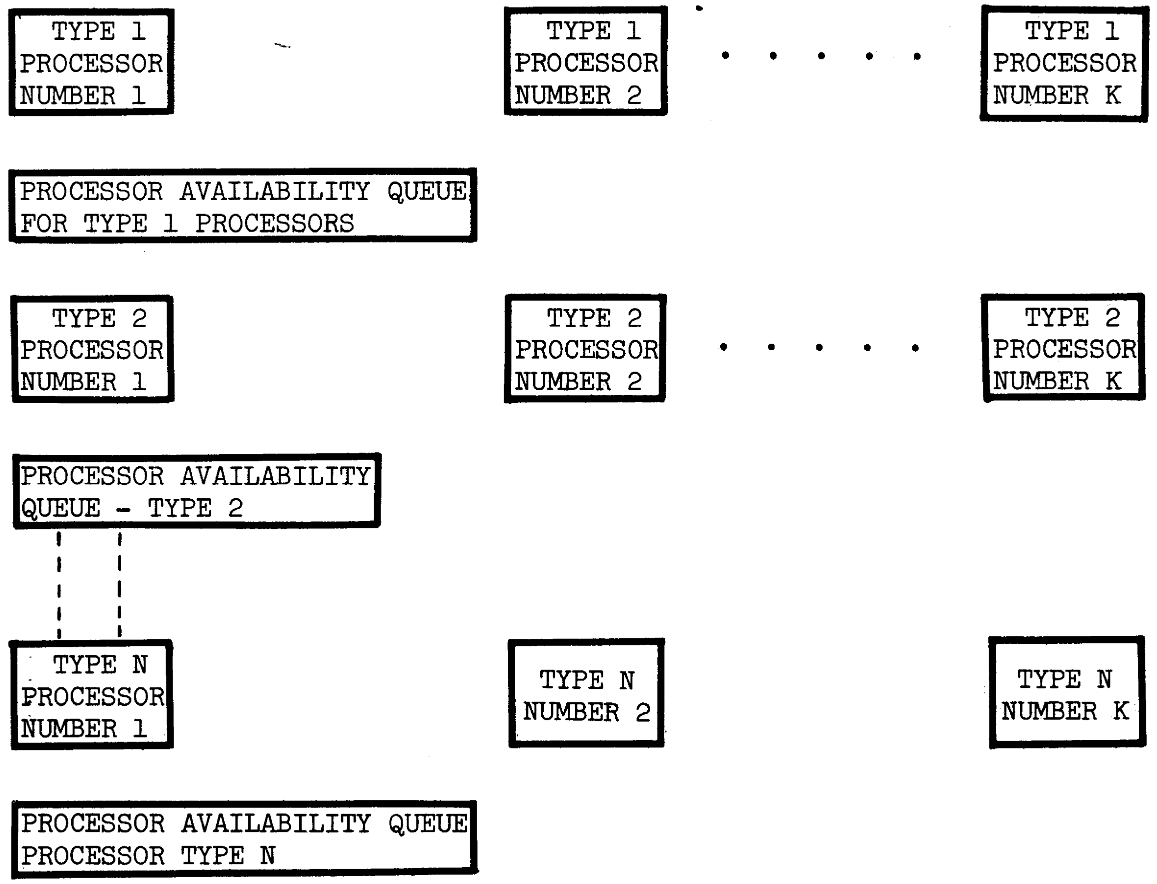
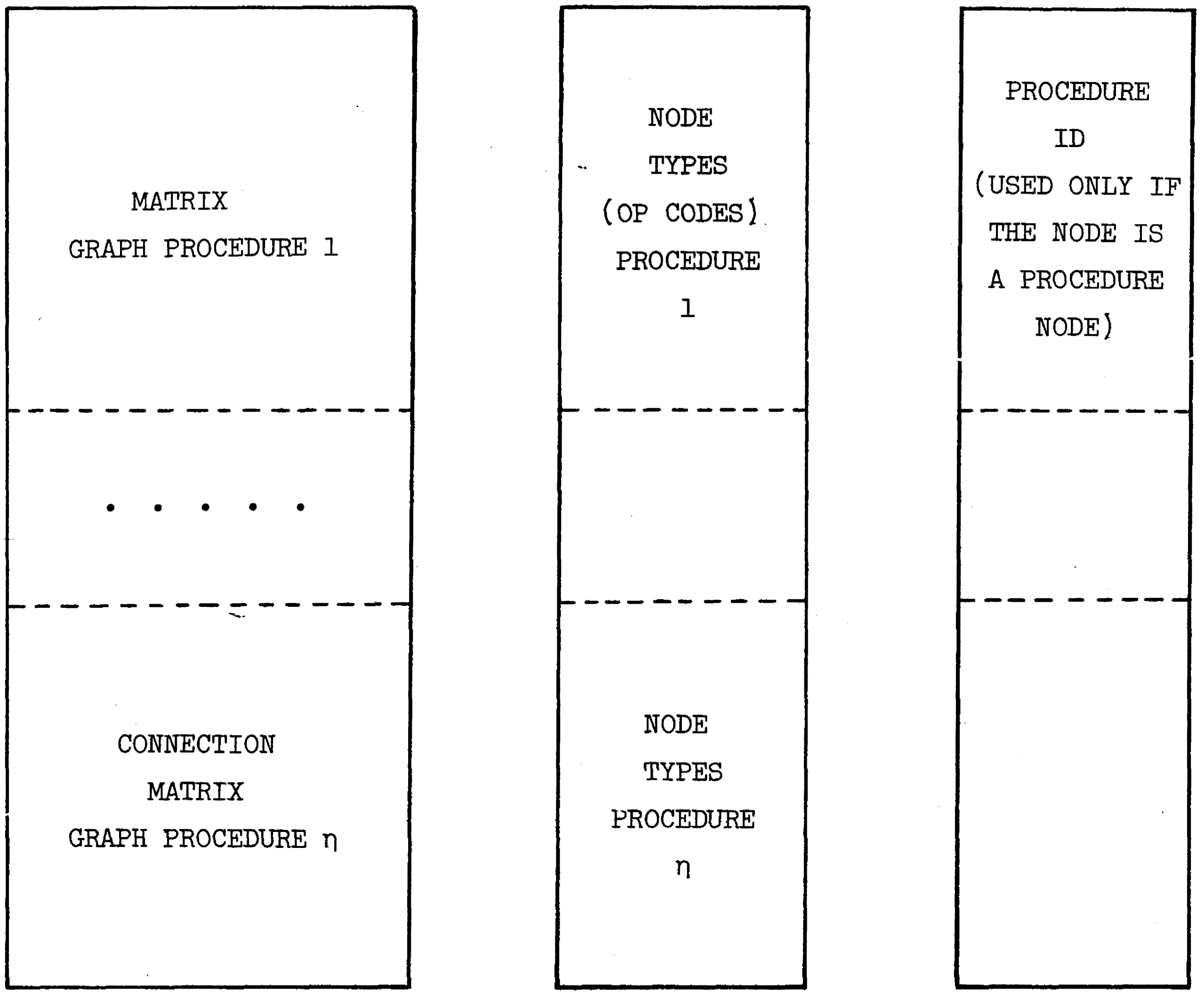

TYPE 1 PROCESSOR NUMBER 1
TYPE 1 PROCESSOR NUMBER 2
TYPE 1 PROCESSOR NUMBER K
PROCESSOR AVAILABILITY QUEUE FOR TYPE 1 PROCESSORS
TYPE 2 PROCESSOR NUMBER 1
TYPE 2 PROCESSOR NUMBER 2
TYPE 2 PROCESSOR NUMBER K
PROCESSOR AVAILABILITY QUEUE - TYPE 2
TYPE N PROCESSOR NUMBER 1
TYPE N NUMBER 2
TYPE N NUMBER K
PROCESSOR AVAILABILITY QUEUE PROCESSOR TYPE N

Fig. 1b

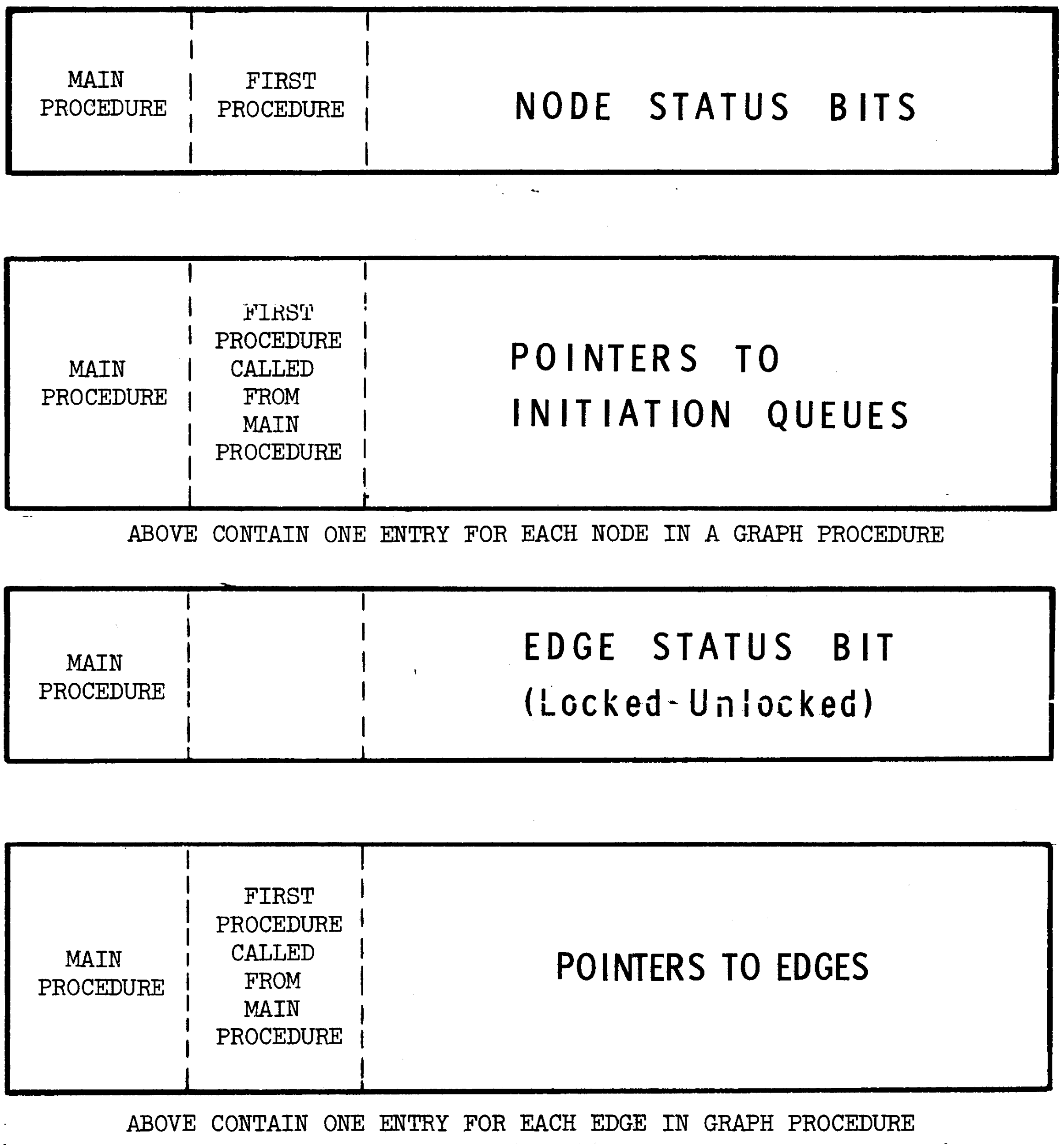
Fig. 1 PROCESSORS



STATIC PROGRAM STORAGE

One block of this storage is allocated for each procedure definition in the Graph Program.

Fig. 2


MAIN PROCEDURE
FIRST PROCEDURE
NODE STATUS BITS
MAIN PROCEDURE
FIRST PROCEDURE CALLED FROM MAIN PROCEDURE
POINTERS TO INITIATION QUEUES
ABOVE CONTAIN ONE ENTRY FOR EACH NODE IN A GRAPH PROCEDURE
MAIN PROCEDURE
EDGE STATUS BIT (Locked-Unlocked)
MAIN PROCEDURE
FIRST PROCEDURE CALLED FROM MAIN PROCEDURE
POINTERS TO EDGES
ABOVE CONTAIN ONE ENTRY FOR EACH EDGE IN GRAPH PROCEDURE

DYNAMIC PROGRAM STORAGE

One block of this storage is allocated for each copy of a procedure which is executing.

Fig. 3

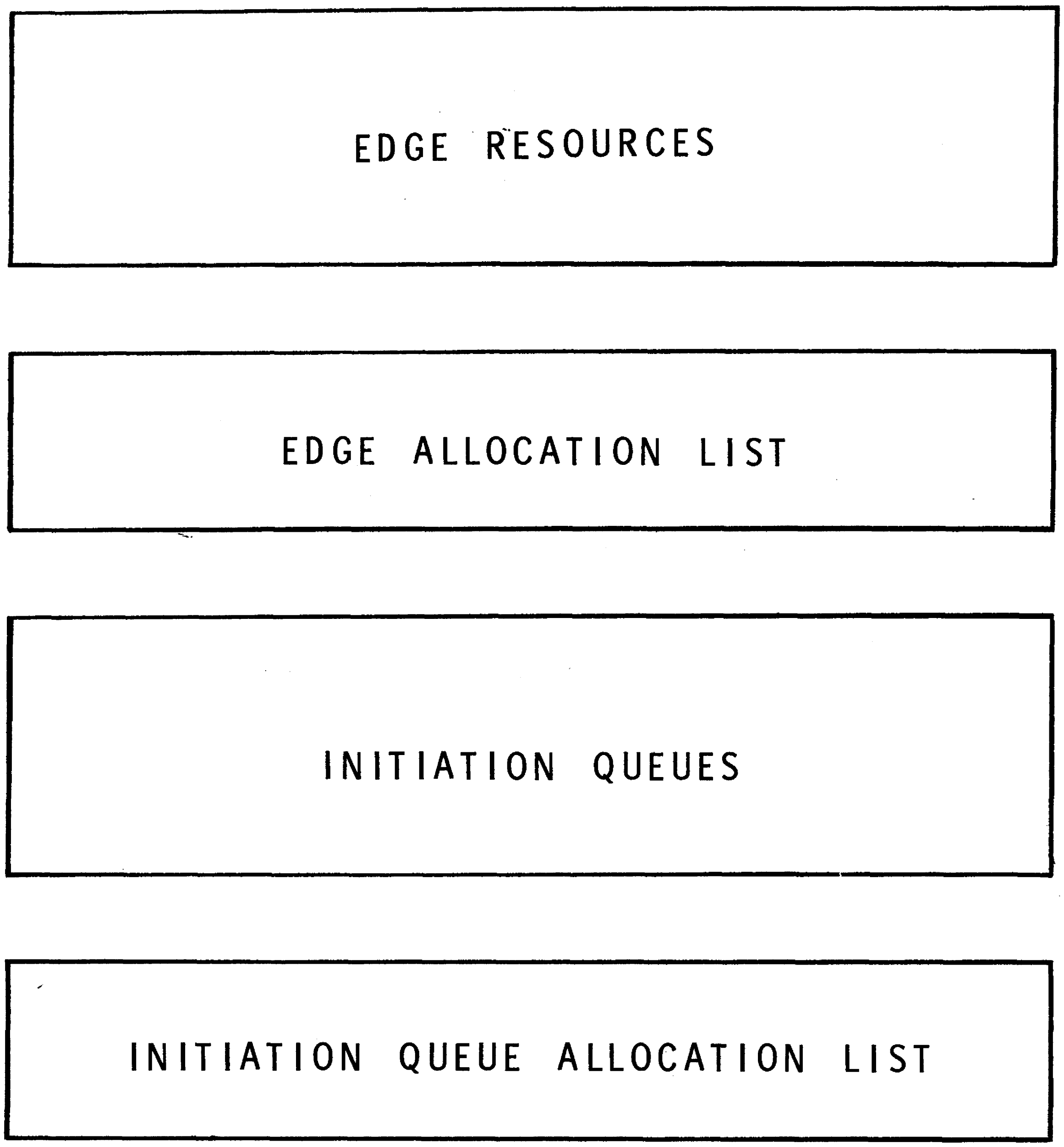
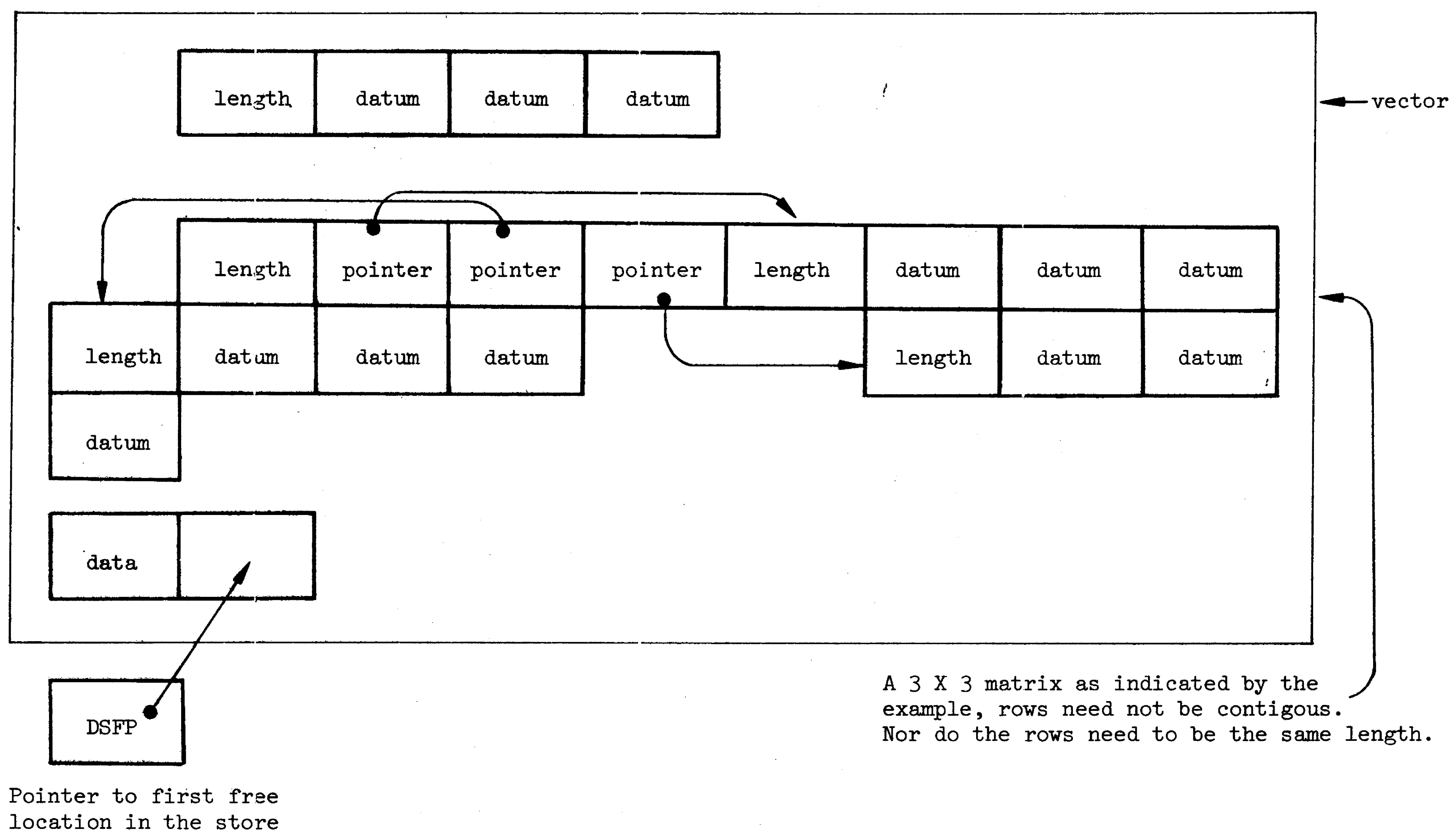

EDGE RESOURCES
EDGE ALLOCATION LIST
INITIATION QUEUES
INITIATION QUEUE ALLOCATION LIST

Fig. 3b


length
datum
datum
datum
vector
length
pointer
pointer
pointer
length
datum
datum
datum
length
datum
datum
datum
length
datum
datum
datum
data
DSFP
Pointer to first free location in the store
A 3 X 3 matrix as indicated by the example, rows need not be contigous. Nor do the rows need to be the same length.

STRUCTURED DATA STORAGE

Fig. 4

sequence of operations during a computation. Their use in the edges provides implicitly the array structures which are specified explicitly by indexing in conventional programs. The programming of the simulator is thus facilitated by a programming language which allows queues as a data structure. The resulting simulator is also a better description of the graph model since the ordering provided by queues is implicit as in the model. Queues can be programmed in PL/I by using structures and compile time macros.

Edge and initiation queues are represented by PL/I structures having four parameters which determine the access to the queue and an array which holds the values in the queue. The four parameters are: (1) the index of the array element which holds the head of the queue; (2) the index of the element holding the tail; (3) the number of elements currently stored in the queue; and (4) the maximum number of elements which the queue can hold. A PL/I compile time procedure is used to define QUEUE as a data type in the simulator, i.e., to produce the proper structure declaration when a simulator variable is declared to be of type QUEUE. Special access procedures are used for entering and deleting values which treat the array associated with the queue as a circular buffer. These procedures, together with the compile time macros have the effect of making QUEUE a basic data type within the simulator.

Edges are then represented as an array of queues as are initiation queues. Both arrays have an associated allocation list whose entries indicate whether the corresponding queue is allocated and if so, to which node or edge. When the simulator wishes to allocate an edge, it searches the allocation list until it finds an entry which is zero. The edge number is then put in this entry, and the edge pointer is set to the corresponding queue. The allocation list entry is reset to zero when the edge is released. When no edge resources have been allocated to an edge, the pointer is zero. Initiation queues are allocated for all nodes in a graph procedure when the procedure is called.

Representation of structured data in the simulator differs from that in the Adams model in two respects. First, structured elements are not stored directly on the edges in the simulator. Instead, they are stored in a separate array and pointers to the location of the structure within the array are kept on the edges in their place. At most, one instance of a given pointer may be on the edges at one time so that the pointer "represents" the bracketed data structure on the edge. Second, rather than use a special bracket symbol at the beginning and end of the structure, the starting location (denoted by the pointer value) and a count of the number of items is used. Items may themselves be pointers, so the structure is recursive just as Adams bracket notation is. The format of structures is: ⟨length⟩ ⟨item⟩* where ⟨length⟩ is an integer and the number of ⟨item⟩s must be equal to the value of ⟨length⟩. It is easy to show that the pointer-count representation allows exactly the same structures as does the bracket notation (*). However, having the length explicitly available simplifies storage allocation for the simulator and also avoids the problems of setting aside a special value for the bracket character and of examining each element in the structure to find the closing bracket. Pointers are not explicitly distinguished from data in the simulator. Rather it is assumed that each type of primitive node knows what type of data to expect and that graph programs will use the correct primitive nodes. This requires different primitive node types for the same operation on scalar and structured data, but it has the advantage that the edge access procedures do not have to examine each item so that the same queue access procedures can be used for all queues in the simulator. In a hardware implementation this advantage would be outweighed by the flexibility gained by using a single bit to distinguish between pointers and data.

Simulations take place in three stages. First, machine characteristics (number and speeds of processors, amount of storage, etc.) are read in followed by the graph program definition and the simulator is initialized. Second, successive machine cycles are simulated until a cycle occurs during which no node executes. This indicates the program has terminated. Finally, the memory and processor use is printed in bar graph form together with some statistics on the simulation. Figures 6-10 show the simulator flowchart.

The simulation of a single machine cycle is done in three stages. In the first stage all those nodes which are ready to initiate are marked. This is done by examining all the non-zero entries in the row of the connection matrix which corresponds to the node in question, i.e., all the input edges for that node. If any edge is both unlocked and empty then the node is not ready to initiate. Otherwise, it is ready to initiate. A p-node may be marked ready to initiate even though it is already executing if data has arrived which permits a second copy of the node to initiate.

Allocation of processors among those nodes which are ready to initiate is done by a self-contained procedure so that the allocation algorithm can be readily changed. This procedure puts the processor identifier in the nodes initiation queue and changes the node status from ready to executing. It is also responsible for determining whether multiple copies

ADAMS GRAPH SIMULATOR - FLOWCHART

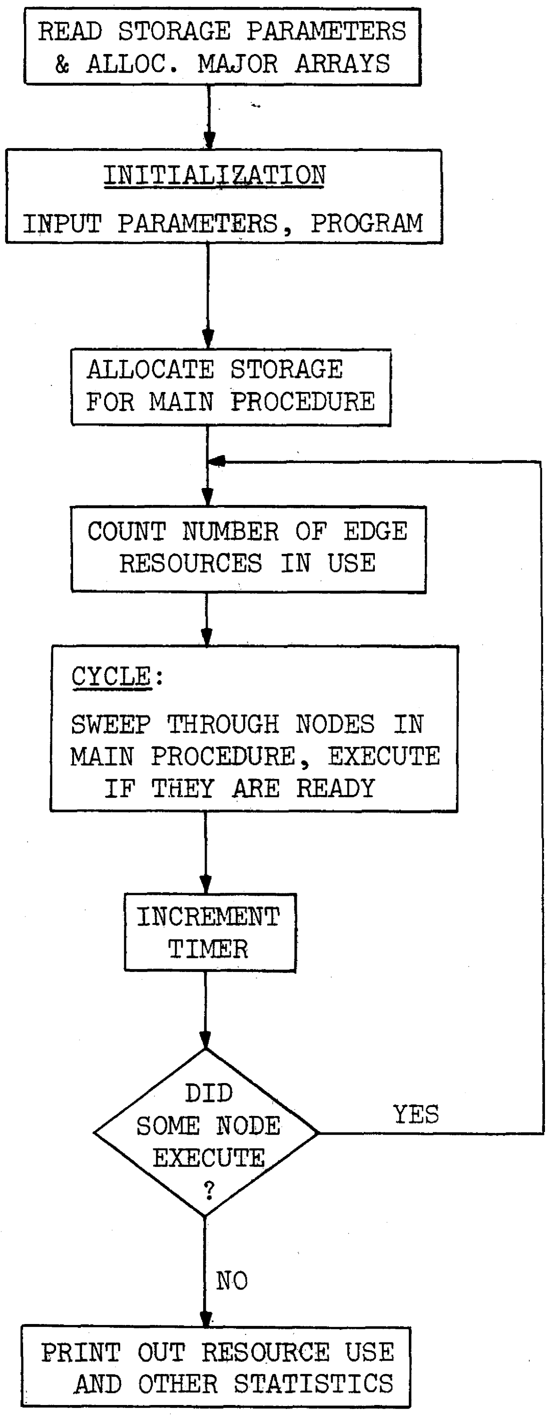
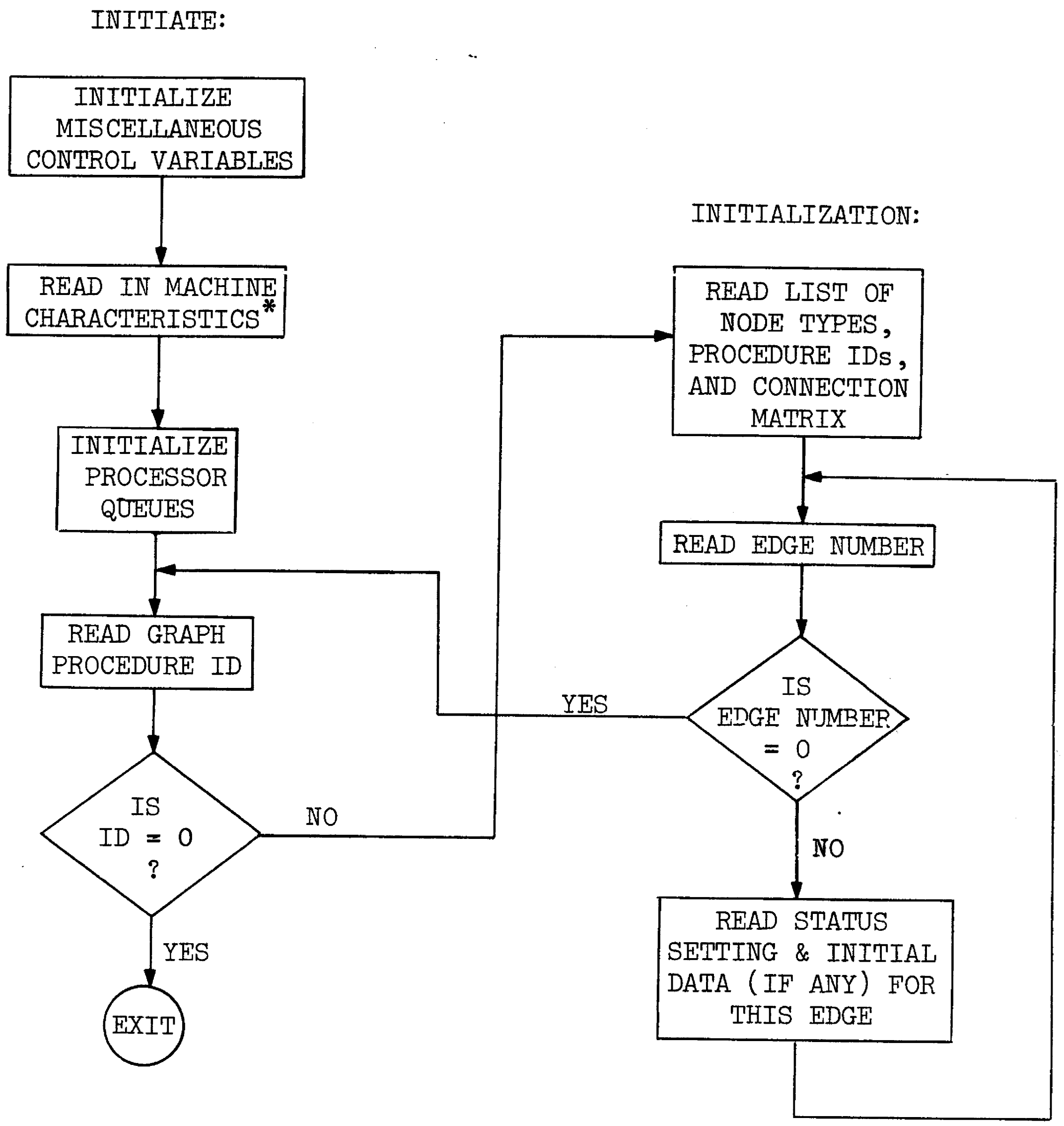

READ STORAGE PARAMETERS & ALLOC. MAJOR ARRAYS
INITIALIZATION
INPUT PARAMETERS, PROGRAM
ALLOCATE STORAGE FOR MAIN PROCEDURE
COUNT NUMBER OF EDGE RESOURCES IN USE
CYCLE:
SWEEP THROUGH NODES IN MAIN PROCEDURE, EXECUTE IF THEY ARE READY
INCREMENT TIMER
DID SOME NODE EXECUTE ?
YES
NO
PRINT OUT RESOURCE USE AND OTHER STATISTICS

Fig. 5 BASIC SIMULATOR LOOP

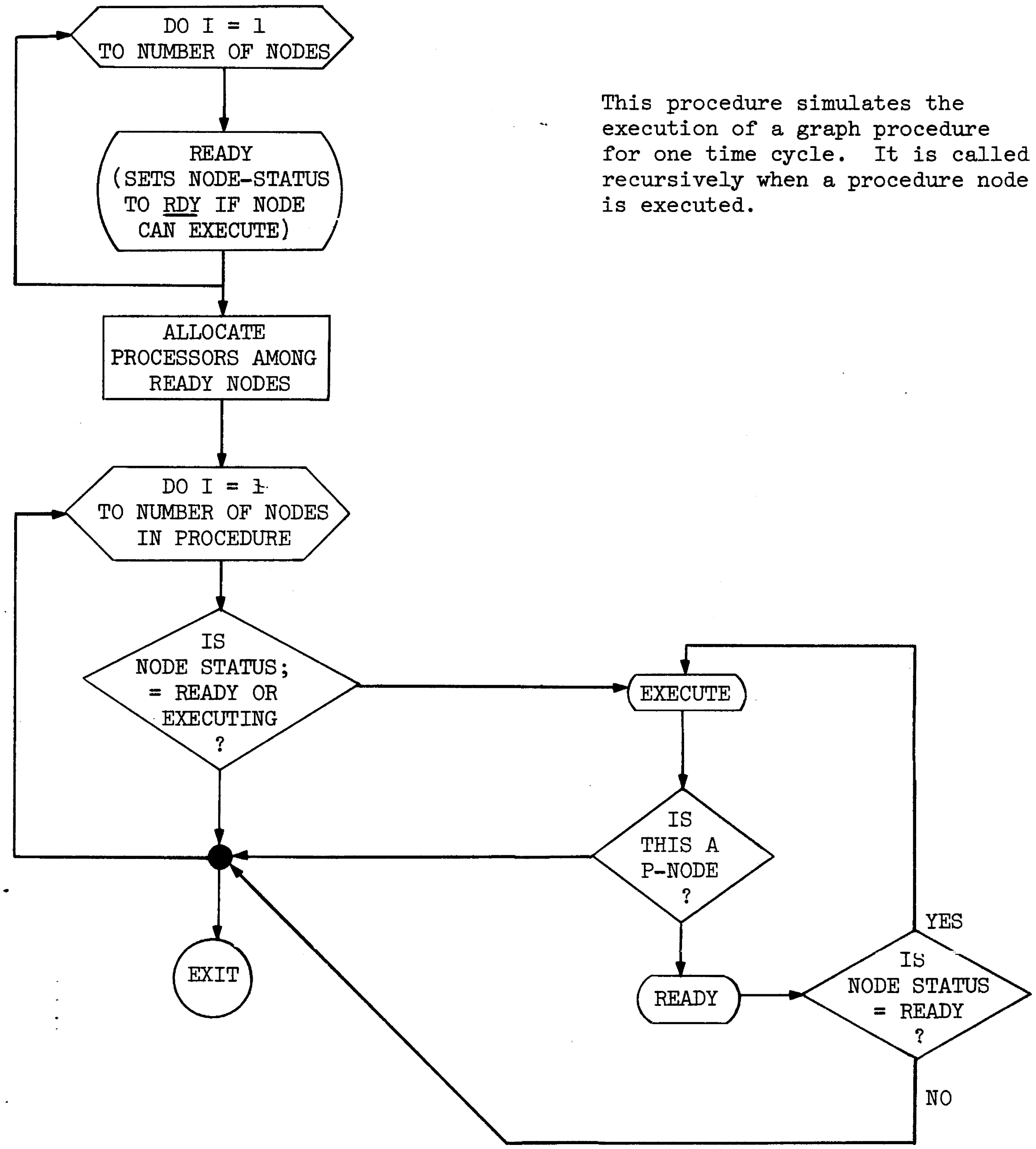
SIMULATOR FLOWCHART


INITIATE:
INITIALIZE MISCELLANEOUS CONTROL VARIABLES
READ IN MACHINE CHARACTERISTICS*
INITIALIZE PROCESSOR QUEUES
READ GRAPH PROCEDURE ID
IS ID = 0 ?
NO
YES
EXIT
INITIALIZATION:
READ LIST OF NODE TYPES, PROCEDURE IDs, AND CONNECTION MATRIX
READ EDGE NUMBER
IS EDGE NUMBER = 0 ?
YES
NO
READ STATUS SETTING & INITIAL DATA (IF ANY) FOR THIS EDGE

*Execution time and number of processors for each processor type.

Fig. 6

SIMULATOR FLOWCHART

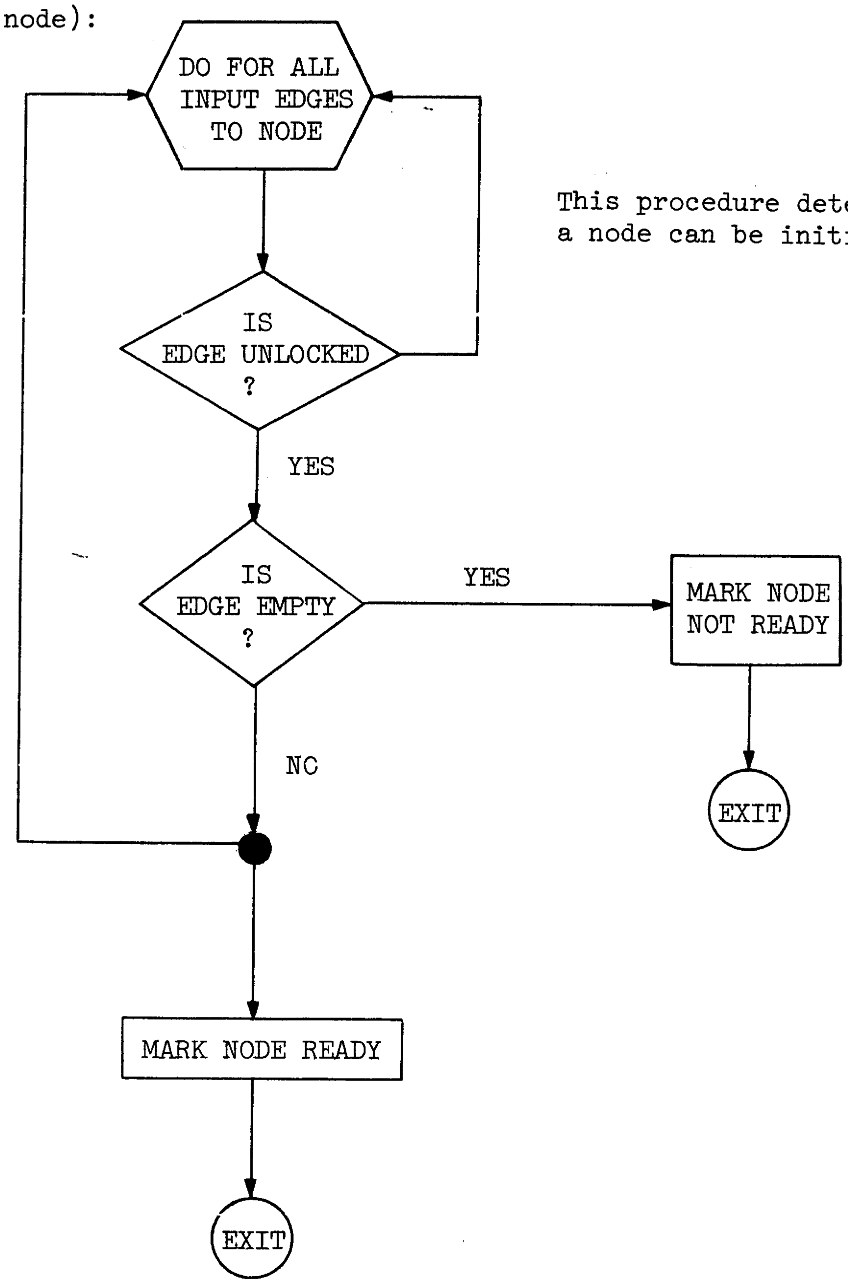


This procedure simulates the execution of a graph procedure for one time cycle. It is called recursively when a procedure node is executed.

Fig. 7

SIMULATOR FLOWCHART

READY (node):



This procedure determines whether a node can be initiated.

Fig. 8

SIMULATOR FLOWCHART

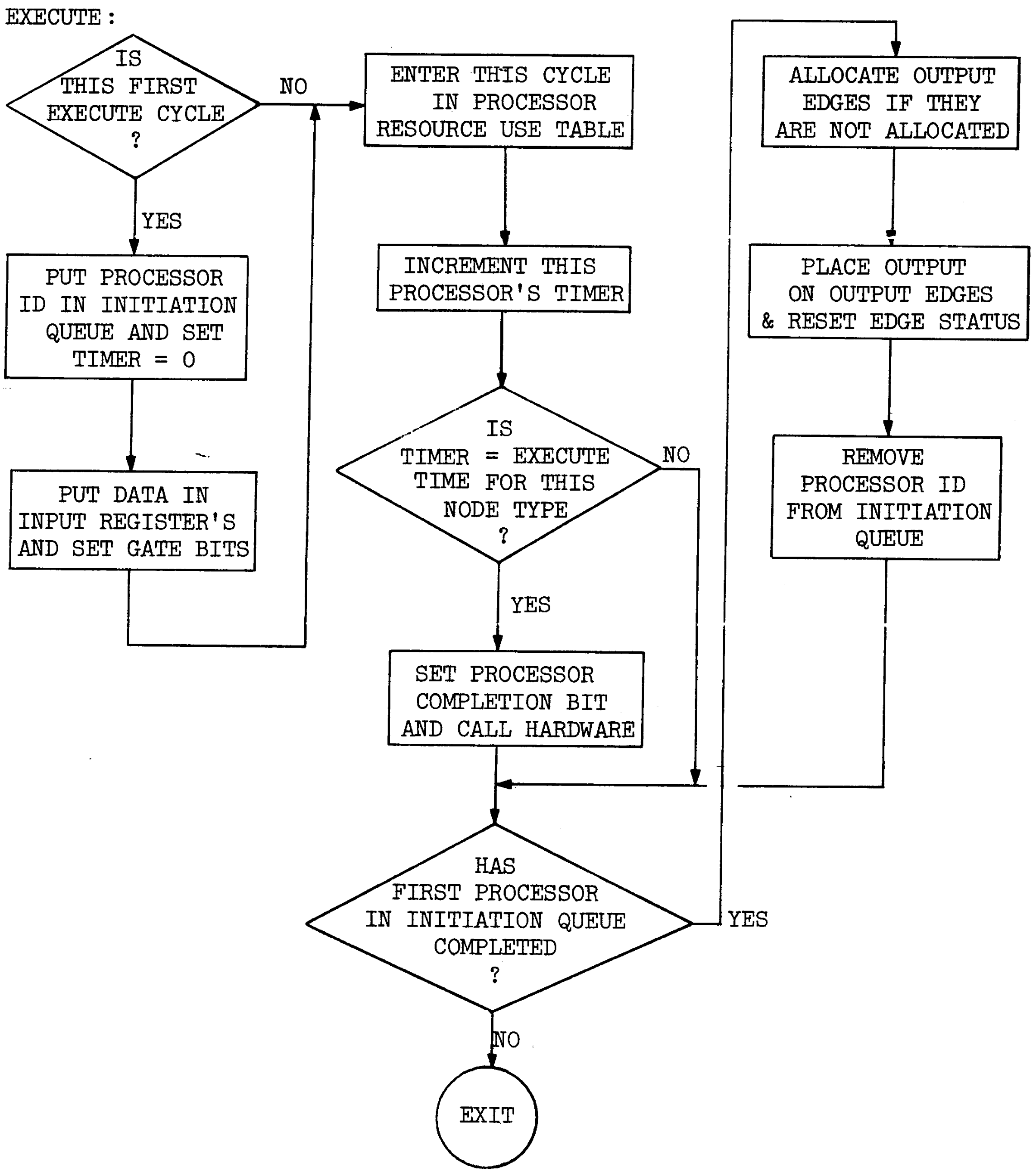

EXECUTE:
IS THIS FIRST EXECUTE CYCLE ?
NO
YES
PUT PROCESSOR ID IN INITIATION QUEUE AND SET TIMER = 0
PUT DATA IN INPUT REGISTER'S AND SET GATE BITS
ENTER THIS CYCLE IN PROCESSOR RESOURCE USE TABLE
INCREMENT THIS PROCESSOR'S TIMER
IS TIMER = EXECUTE TIME FOR THIS NODE TYPE ?
NO
YES
SET PROCESSOR COMPLETION BIT AND CALL HARDWARE
HAS FIRST PROCESSOR IN INITIATION QUEUE COMPLETED ?
YES
NO
EXIT
ALLOCATE OUTPUT EDGES IF THEY ARE NOT ALLOCATED
PLACE OUTPUT ON OUTPUT EDGES & RESET EDGE STATUS
REMOVE PROCESSOR ID FROM INITIATION QUEUE

Fig. 9

executing. It is also responsible for determining whether multiple copies of the node should be initiated. Each processor has a unique number assigned to it and entered into the availability queue when the simulator is initialized. The size of each queue determines the number of processors which are available for the corresponding node type. When a unit is assigned to a node the processor number is removed from the proper processor available queue and put onto the initiation queue for that node. One data item is then removed from each input edge and put into "input registers" associated with the processor. If the node is a p-node and there is still data on each input edge, another processor is taken from the available queue and put onto the initiation queue of the node. This process is repeated until some input edge has no data. The process provides the vector parallelism required by the graph program model.

Associated with each unit is a timer. When the unit is taken from the unit pool this variable is set to zero. After the ready nodes have been initiated, the timer of each executing processor is incremented and compared against the time required for that type of node. When the two are equal, i.e., when the node has executed the required number of time steps, the simulator transfers to code which carries out the actual operation. The transfer is by means of a switch on the node type. If the processor identifier is now first on the initiation queue of the node, the results are put on the output edges and the processor identifier is removed from the initation queue and placed on the proper unit pool queue. If another processor is first on the initiation queue, this processor is not terminated, but if that processor subsequently terminates in the same time step, the simulator looks again at the initiation queue and terminates this one without waiting for the next time step. Thus, the order imposed by the initation queue is maintained, but the simulator carries out as many terminations at a time as it can.

When a procedure node is encountered, a copy must be made of the defining graph. The nodes and edges in this copy must be renamed so as to be distinguishable from other copies executing concurrently. In addition, the initial data on the edges must be present each time the graph procedure is called. The creation of a copy is accomplished by adding a new level of naming to the PL/I structures containing the edges and the node ~~data~~. Thus, the array of queue EDGES is actually the fully qualified name COPY (I,J) · EDGES. This is the J-th call of graph procedure I. COPY (I,∅) is the definiton of the graph procedure I, while for J>0 COPY (I,J) is the copy which is actually executed. When procedure I is called its edges can then be initial executing the structure assignment statement.

COPY (I,J) · EDGES = COPY (I,∅) · EDGES

Initially, the simulator assigns COPY (1,0) and executes the graph procedure consisting of COPY (1,1). When a procedure node is encountered, I is reset to the name of the procedure and a copy of the node is executed for one time step (i.e. each node in the procedure is executed one time step). If the procedure has not terminated at the end of the time step, control returns to the calling procedure but the node remains in execute status. When the node terminates, it is taken out of execute status and this indicates to the simulator that control is not to be passed to the node on subsequent time steps. The edge initialization only takes place when the node is in the ready-to-initiate state.

In simulating a given type of node the actual execution takes place on the last of the n cycles specified for the execution time of that node. The first n-1 cycles are simply delay cycles and no action takes place during them. Procedure nodes must be specified as a procedure call operator whose argument is the name of the procedure to be invoked. The procedure operator itself has an execution time of n cycles, which represents the setup time (resetting pointers, allocating storage, etc.) necessary for that invocation of the graph procedure, and the invoked procedure does not begin to execute until the last of these cycles, so that the total time required for a procedure node is the time required for the procedure call operator plus the time required to execute the constituent nodes.

Although execution takes place only on the last cycle of the node's execution, data is taken off the input edges prior to initiation and the processor is allocated to the node throughout the execution period. Thus, the simulator acts externally as if the processor were executing for n cycles. When the node is initiated, a processor is assigned to it by removing the processor number from the appropriate availability queue and placing it in the node's initiation queue. The data from each unlocked input edge is transferred to the corresponding input register in the assigned processor and the gate bits of all input registers are set to reflect the edge-status bits. If the node is an s node, the processor resets the gate bit at the end of execution. The gate bit is then used to reset the edge-status bit.

The execution of a node is carried out by two procedures, EXECUTE and HARDWARE. EXECUTE determines which nodes are ready to initiate, calls the processor allocation algorithm, transfers the data from edges to registers and sets the gate bit. After a delay which represents the execution time of the node, it calls HARDWARE to apply the functions associated with the node. HARDWARE operates only on the registers of the assigned processor; it does not know the edge connections of the node to which the processor is assigned. When control is returned to it, EXECUTE resets the edge status bits according to the processor gate bits, and transfers data from the output registers to the output edges. In some cases, the processor may return a null result in one or more output registers so that the value in the register is undefined. The processor flag RRF indicates to EXECUTE whether or not the corresponding output register value is to be put onto an output edge.

EXECUTE also has the task of assuring that results are put onto the output edges in the order dictated by the initiation queue. This is accomplished by checking whether the first processor in the initiation queue has completed. If not, no other processors in the queue are checked on that cycle. Otherwise, the data from that processor is put onto the output edges and the process is repeated for the next item in the initiation queue. Completion is indicated by the processor flag DONE. In this version of the simulator all nodes of a given type are constrained to have the same execution time. The order of initiation and termination would thus remain constant even without the initiation queue mechanism.

Allocation of edge resources is done by the procedure M-ALLOCATE. This procedure is called by EXECUTE before it transfers output from processor registers to an edge. It is also called by the procedure call operator in order to allocate storage for initial values to be placed on the procedure's edges before initiation. The current version of M-ALLOCATE allocates edge resources in fixed size blocks. In the unlimited resources model 15 edge-resources are allocated for each edge when M-ALLOCATE is called. This has proven ample for all of the programs which have been simulated.

SIMULATOR INPUT AND OUTPUT

The simulator first reads in a set of graph procedures defining the program to be simulated. It then simulates each time step of the program's execution until no nodes are able to execute. Simulation of a time step consist in first marking all the nodes in the graph which are ready to initiate, then allocating processors to these nodes, and finally, executing all the nodes which are able to execute on that time step. The number of processors used during the time step is recorded for each node type, as well as the number of edge resources in use at the beginning of the cycle. This information is printed at the end of the simulation.

The input to a simulation consists of two parts, machine characteristics and the graph program. The first part specifies three types of parameters: (1) whether the execution is to have vector parallelism; (2) the execution time for each primitive node type; and (3) the number of processors for each primitive node type. Parallelism is specified by a bit constant - '1'B for vector parallel mode, '0'B for concurrency only mode. In the latter mode only one copy of a p-node can execute at a time. This bit is followed by a list of pairs of integers giving the time in cycles that each processor type requires to execute and the number of processors of that type.

The graph program is read in as a set of graph procedures. The format for the input of the graph program is best described by a bnf syntax.

```
⟨graph program⟩     ::=  0
                        |⟨graph procedure⟩ ⟨graph program⟩

⟨graph procedure⟩   ::=  ⟨name⟩ ⟨procedure definition⟩ ⟨ini-
                         tial data⟩
```

```
⟨name⟩                   ::=  ⟨positive integer⟩

⟨procedure definition⟩   ::=  ⟨node count⟩ ⟨node list⟩ ⟨con-
                                        nection matrix⟩

⟨node count⟩             ::=  ⟨integer⟩

⟨node list⟩              ::=  ⟨op list⟩ ⟨name list⟩

⟨op list⟩                ::=  {list of integers}

⟨name list⟩              ::=  {list of integers}

⟨connection matrix⟩      ::=  {list of integers}

⟨initial data⟩           ::=  0
                              | ⟨edge information⟩ ⟨initial
                                                  data⟩

⟨edge information⟩       ::=  ⟨edge number⟩ ⟨status bit⟩
                                          ⟨data list⟩

⟨edge number⟩            ::=  ⟨positive integer⟩

⟨status bit⟩             ::=  '1'B  /*locked*/
                             |'0'B  /*unlocked*/

⟨data list⟩              ::=  ⟨count⟩ ⟨data⟩

⟨count⟩                  ::=  ⟨non-negative integer⟩

⟨data⟩                   ::=  {list of floating point numbers}
                              |(empty)
```

Zeros terminate both the data list and the set of graph procedures.

The integer ⟨name⟩ identifies the graph procedure being defined while those in the ⟨name list⟩ identify those procedure nodes which are constituents of that procedure. Procedures can be read in any order and may contain nodes naming procedures not yet read in. The main procedure must have the name 1, and execution begins with this procedure.

The number of entries in ⟨op list⟩ and ⟨name list⟩ must be equal to ⟨count⟩, while the ⟨connection matrix⟩ must have ⟨count⟩^{2} entries.

Only those edges specified by an edge number are initialized. If an edge is initialized its initial status setting must be given. Edges leading into p-nodes are set to unlocked. The status of all edges which are not explicitly initialized are set to unlocked before the simulation begins.

Simulator Storage Parameters

The following parameters can be varied to adjust the storage used by the simulator in order to fit the requirements of the graph being interpreted. M#T is the maximum number of time steps the computation will run. Simulation results are stored in a M#T by NT#+1 array, where NT# is the number of primitive node types. ERM is the maximum number of edges and IQM the maximum number of initiation Queues which can be allocated. IQM must be >= the number of procedures executing at one time times the number of nodes in each. The arrays used are of size: ERM by EGLNMX+4; IQM by EGLNMX+4; ERM; and IQM. EGLNMX is the maximum number of data items which can be held on an edge at one time. EGMX and NDMX refer to storage of graph procedure definitions. NDMX is the maximum number of nodes in any one procedure (excluding copies), and EGMX is the maximum number of edges in any one procedure. The major arrays used are: 2 of size PROCM by NDMX; 1 of length PROCM by NDMX+1 by NDMX+1; 1 of length PROCM by EGMX by EGLNMX+2; 2 of length GMAX by EGMX, where PROCM is the number of graph procedures in the graph program being simulated and GMAX is the maximum number of procedures which can be active at one time, including multiple calls to the same procedure. (Hence this parameter limits the depth of recursion).

Certain of these parameters (M#T, PROCM, ERM, IQM, EGMX, GMAX, and NDMX) are read in by the simulator at the start of each run. They are read in DATA format, and so may be entered in any order. They are the first data read in by the simulator.

There are two types of output from the simulator, trace output and resource use summary output. Trace output is printed during the simulation and consists of identification of nodes in execution, procedures which have been invoked, input and output register contents, etc. It is primarily useful in debugging graph programs. The resource use summary is printed at the end of the simulation. For each type of resource, including edge resources, the following information in printed: (1) A bar graph showing the number of resources of that type used at each time step of the simulated computation; (2) The total number of resource cycles used for that type of resource; (3) the percent utilization of that type of resource; (4) the average number of resources used per time step; and (5) the maximum number of resources used at any time step. The same information is also summarized for all processor resources. The total number of resource cycles provides a measure of the "cost" of the computation, the percent utilization measures the efficiency with which resources are being used, and the average resources used per time step gives an estimate of the degree of parallelism attained.

Representation of structured data in the simulator differs from that in the Adams model in two respects. First, structured elements are not stored directly on the edges in the simulator. Instead, they are stored in a separate array, and pointers to the location of the structure within the array are kept on the edges in their place. At most, one instance of a given pointer may be on the edges at one time so that the pointer "represents" the bracketed data structure on the edge. Second, rather than use a special bracket symbol at the beginning and end of the structure the starting location (denoted by the pointer value) and a count of the number of items is used. Items may themselves be pointers so the structure is recursive just as Adams' bracket notation is. The format of structures is: ⟨length⟩ ⟨item⟩* where ⟨length⟩ is an integer and the number of ⟨item⟩s must be equal to the value of ⟨length⟩. It is easy to show that the pointer-count representation allows exactly the same structures as does the bracket notation. However, having the length explicitly available simplifies storage allocation for the simulator and also avoids the problems of setting aside a special value for the bracket character and of examining each element in the structure to find the closing bracket. Pointers are not explicitly distinguished from data in the simulator. Rather it is assumed that each type of primitive node knows what type of data to expect and that graph programs will use the correct primitive nodes. This requires different primitive node types for the same operation on scalar and structured data, but it has the advantage that the edge access procedures do not have to examine each item so that the same queue access procedures can be used for all queues in the simulator. In a hardware implementation this advantage world be outweighted by the flexibility gained by using a single bit to distinguish between pointers and data.

PRIMITIVE NODES

The choice of which operations were to be implemented in the simulator was somewhat arbitrary. Since no hardware constraints or cost considerations were available as a guide, primitive nodes were chosen primarily because they were convenient for writing the programs to be simulated. Any hardware implementation of this model would include primitive nodes similar to those implemented here, although they would undoubtedly differ in some details.

The following table lists the twenty-eight primitive node types in the simulator. The first column gives the operation code used by the simulator, the second the name of the node type together with the symbol used in drawing the graph procedures, the third and fourth the data types of inputs and outputs, and fifth gives the functions which determine edge status settings for s-nodes. Only two s-nodes were needed, but these were used frequently. Loop control, type 11, selects its first input on the first execution and the second on all subsequent executions of the same node. Select route, type 12, selects either its second or its third input depending on the value of its first input, which is boolean. If the first input is true, the second input is selected, otherwise the third.

The arithmetic and boolean operations (zero test, negation, plus, increment, decrement, multiply, subtract, divide, less than, GTEQ, AND, OR) work in the obvious way. The equivalent of branchlng in a conventional computer is provided by the conditional route and branch route nodes. The conditional route node has two inputs, the first of which is a boolean value. If the value of the boolean is true, the second

TABLE 1 - PRIMITIVE NODES

CODE	NAME		INPUTS	OUTPUTS	EDGE STATUS
1	Procedure Call		Any	Any	P-node
2	Zero Test	(=0)	Float	Boolean	P-node
3	Negation	($\neg$)	Boolean	Boolean	P-node
4	Plus	(+)	Float, Float	Float	P-node
5	Increment	(+1)	Float	Float	P-node
6	Decrement	(-1)	Float	Float	P-node
7	Multiply	(*)	Float, Float	Float	P-node
8	Two Copies	(2)	Scalar	Scalar, Scalar	P-node
9	Conditional Route	(Cond)	Bool, Float	Float	P-node
10	Branch Route	(BR)	Bool, Float	Float, Float	P-node
11	Loop Control	(LC)	Float, Float	Float	U,L$\rightarrow$L,U; L,U$\rightarrow$L,U
12	Select Route	(SR)	Bool, Float, Float	Float	True, U,L,L$\rightarrow$ LUL, F,ULL $\rightarrow$LU LUL $\rightarrow$ ULL, LLU$\rightarrow$ULL
13	Subtract	(-)	Float, Float	Float	P-node
14	Divide	(÷)	Float, Float	Float	P-node
15	Less Than	(<)	Float, Float	Boolean	P-node
16	First		Vector	Float	P-node
17	Rest		Vector	Vector	P-node
18	First - Rest		Vector	Float, Vector	P-node
19	Null Test		Vector	Vector, Boolean	P-node
20	Length		Vector	Vector, Float	P-node
21	Unbracket		Vector	Float	P-node
22	Split		Vector	Vector, Vector	P-node
23	GTEQ	(>=)	Float, Float	Boolean	P-node

24	And	$(\wedge)$	Boolean,Boolean	Boolean	P-node
25	Or	$(\vee)$	Boolean,Boolean	Boolean	P-node
26	Insert		Vector, Float	Vector	P-node
27	Two Copies-Vector	$(\vec{2})$	Vector	Vector, Vector	P-node
28	Identity	ID	Any	Any	P-node

input is placed on the output edge. Otherwise, there is not output. Branch route has two inputs and two outputs. The first input is a boolean. If it is true, the second input is placed on the first output edge and nothing is placed on the second output edge. Otherwise there is no output on the first edge and the second input is placed on the second input edge.

The TWO COPIES node takes one input and puts it onto the two output edges. This is by far the most commonly occurring node in the graph programs which I have written. Because the implementation of structured operands requires that there be one and only one copy of a pointer to a vector, a special node type is needed to copy vectors. The vector itself is copied to a new location in structured operand storage, and a pointer to the new location is output together with the pointer to the original location.

The UNBRACKET node causes a vector of length n to be split into its components. The n components are put onto the output edge. This is the only primitive node which puts more than one item on a single output edge so that it must be treated as a special case by the execution logic. Rather than putting the contents of the processor output register onto the output edges, the register is used as a pointer and the contents of the structured operand storage pointed at are put on the output edge.

FIRST, REST, FIRST-REST, and SPLIT all operate on vectors. FIRST puts out the first component of the vector. REST decrements the length field of the vector, moves the length field to the position occupied by the first component of the vector and outputs a pointer to the new vector thus created. FIRST-REST combines these operations, outputting the first component and a pointer to a vector containing the remaining components. SPLIT outputs pointers to two vectors containing the first half and second half of the components of the input vector. If length of the input is odd, the first half is one longer.

Length inputs a vector and outputs the original vector and its length. NULL TEST inputs a vector and outputs the vector and a boolean whose value is true if the vector is NULL (has a length field equal to zero) and false otherwise. INSERT inputs a vector and a scalar and outputs a new vector of length n+1 which has the scalar as its last component.

The PROCEDURE CALL node requires the most complex logic of the primitive nodes. It must allocate space for the named graph procedure, transfer the contents of the processor input registers to the input edges of the procedure, detect termination of the procedure and transfer the contents of the output edges to its output registers, bracketing if necessary. Bracketing is done by creating a new vector in structured operand storage and putting a pointer to this vector in the output register. Finally, the space allocated to the graph procedure must be freed.

USE OF THE SIMULATOR

This section describes three graph programs which were written for the simulator and the results of simulations run using them. The programs are a trapezoidal rule quadrature, a sort, and a matrix multiplication. The simulations show how the computation time, processor use and degree of parallelism vary with the amount of data, the effect of changing the relative speed of primitive node types, and, in one case, the dependence of computation time on data values. Each program, and the simulations run with it, is described separately and the results are summarized in the conclusion.

TRAPEZOIDAL RULE QUADRATURE

In order to determine what processor speeds should be used for simulation, the time required for various operations on several existing computers were compared. The results are shown in the following table. In the second half of the table the times are normalized so that integer addition equals one. The time for floating point addition then ranges from 1.33 to slightly over 2 and the time for floating point division from 5.0 to 17.1.

From the studies of varying processor speeds done on the sort and trapezoidal quadrature program, it appears that the main effect of changing processor speeds from a uniform execution time of one cycle to a varied set of times falling within the range of existing computers is to scale the time required for the computation by an amount equal to the mean execution time of the nodes in the program. Second order effects, caused by delays in the execution of nodes which depend on the output of slower nodes, are not significant unless the variance in processor speeds is higher than that in existing computers, e.g. unless one node is much slower than the others.

The trapezoidal quadrature program calculates the polynomial $h* \sum_{i=0}^{(b-a)/h} f(a+ih) - (f(a) + f(b))/2$. The values of h, a, and b are inputs to the procedure, and the function to be integrated is specified by supplying a graph procedure which computes the value of that function. Successive values of a+ih are generated by adding h to the previous value. This loop is terminated when the value of a+ih equal to b has been generated. These values are fed into the procedure node for f(x), and the output of that node is fed into a summation loop. Generation of the last value of x causes the value in the summuation loop to be fed into a subtract node which subtracts the value (f(a) + f(b))/2, calculated from the initial values, from the sum. The resulting difference is multiplied by h to give the value of the integral.

TABLE 2

REPRESENTATIVE EXECUTION TIMES FOR SOME EXISTING COMPUTERS

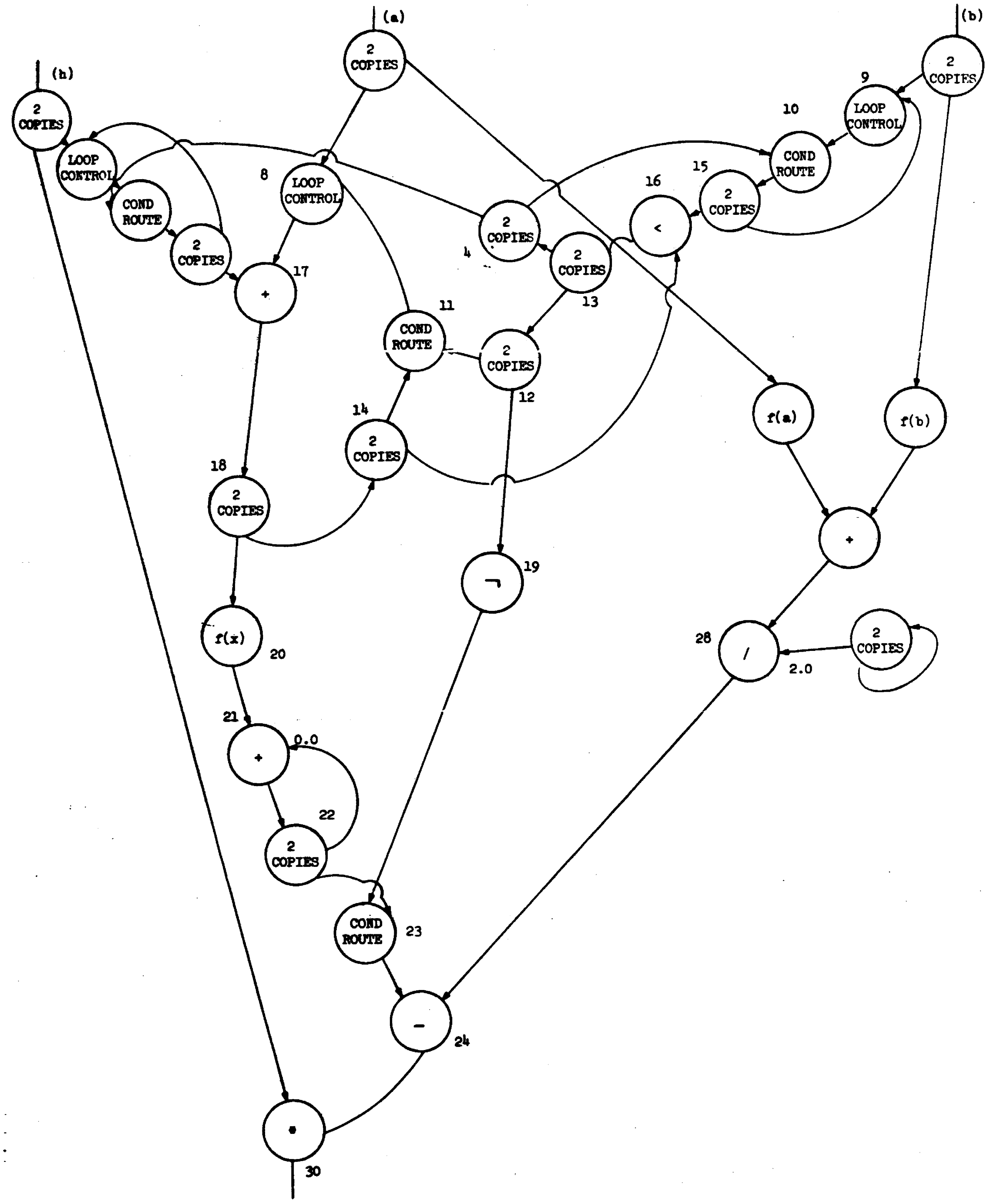
EXECUTION TIMES

	6600	PDP10	360/91	360/75	360/40	7600
FP +	400ns	4.46u	2cy	.83	14.3	4cy
FP -	400ns	4.64u	2cy	.83	14.3	4cyc
I +	300ns	2.53u	1cy =60ns	.39	7.5	2cy
FP x	1000ns	10.29u	3cy	2.05	76.3	5cy
FP ÷	2900ns	14.1u	9cy	3.80	128.1	20cy
∧	300ns	2.35u	1	.59	7.5	2cyc
¬	300ns	1.5u	1	.39	7.5	2cyc
Br	1500ns	1.36	6+	1.10	5.02	11
BC	1500ns	1.68u	7+	.39+1.10	7.5c	11
Subrout. Branch		2.21		.99	6.88	13 1cy=27.5ns

Ratios add (integer)=1

	6600	PDP10	360/91	360/75	360/40	7600
F+	1.33	1.76	2.00	2.13	1.91	2.00
F-	1.33	1.76	2.00	2.13	1.91	2.00
I+	1.00	1.00	1.00	1.00	1.00	1.00
FPX	3.33	4.07	3.00	5.26	10.2	2.50
FP÷	9.67	5.58	9.00	9.75	17.1	5.00
∧	1.00	0.93	1.00	1.51	1.00	1.00
¬	1.00	0.595	1.00	1.00	1.00	1.00
BR	5.00	0.54	6.00	2.82		5.50
BC	5.00	0.664	7.00	3.82	1.00	5.50
Sub-routine Branch				2.52		6.50

Since the values of a+ih are generated by a sequential loop, the time to perform the quadrature is at best proportional to the number of points used. For functions which require little calculation, this loop will dominate the quadrature time. However, if f(x) is sufficiently complex, the time required to compute it will be much larger than the time required to compute all the a+ih. The computation of f(a+ih) will then proceed approximately in parallel for all values of i. In this case, the computation time still has the form $k_1 n + k_0$, where n is the number of points, but k_0 will be much larger than k_1 so that the $k_1 n$ term will not be significant except for very large n.



TRAPEZOIDAL RULE QUADRATURE

$$\int_a^b f(x)dx = h * \left(\sum_{i=0}^{b-a} f(a+ih) - \frac{f(a) + f(b)}{2}\right)$$

Fig. 10

$\int_1^2 |x|dx$ - varied processor times

```
TOTAL PROCESSOR RESOURCE USAGE
   1:###
   2:#######
   3:########
   4:########
   5:#########
   6:#######
   7:#####
   8:###
   9:####
  10:#####
  11:#####
  12:#####
  13:######
  14:#######
  15:######
  16:#####
  17:###
  18:###
  19:##
  20:####
  21:####
  22:####
  23:####
  24:#####
  25:######
  26:#####
  27:####
  28:##
  29:##
  30:##
  31:####
  32:####
  33:####
  34:####
  35:#####
  36:######
  37:#####
  38:####
  39:##
  40:##
  41:##
  42:####
  43:####
  44:####
  45:####
  46:#####
  47:######
  48:#####
  49:####
  50:##
  51:##
  52:##
  53:####
  54:####
  55:####
  56:####
  57:#####
  58:######
```

```
 59:#####
 60:####
 61:##
 62:##
 63:##
 64:####
 65:####
 66:####
 67:####
 68:#####
 69:######
 70:#####
 71:####
 72:##
 73:##
 74:##
 75:####
 76:####
 77:####
 78:####
 79:#####
 80:######
 81:#####
 82:####
 83:##
 84:##
 85:##
 86:####
 87:####
 88:####
 89:####
 90:#####
 91:######
 92:#####
 93:####
 94:##
 95:##
 96:##
 97:####
 98:####
 99:####
100:####
101:#####
102:######
103:#####
104:####
105:##
106:##
107:##
108:####
109:####
110:####
111:####
112:#####
113:######
114:#####
115:####
116:##
117:##
118:##
```

```
  119:####
  120:####
  121:####
  122:####
  123:#####
  124:######
  125:##
  126:#
  127:#
  128:#
  129:#
  130:#
  131:#
  132:#
  133:#
  134:#
  135:#
  136:#
  137:#
  138:
  139:
TOTAL RESOURCE CYCLES USED =      520   % UTILIZATION =        2
AVERAGE RESOURCES USED PER TIME STEP =       4    MAXIMUM =    9
```

ABS

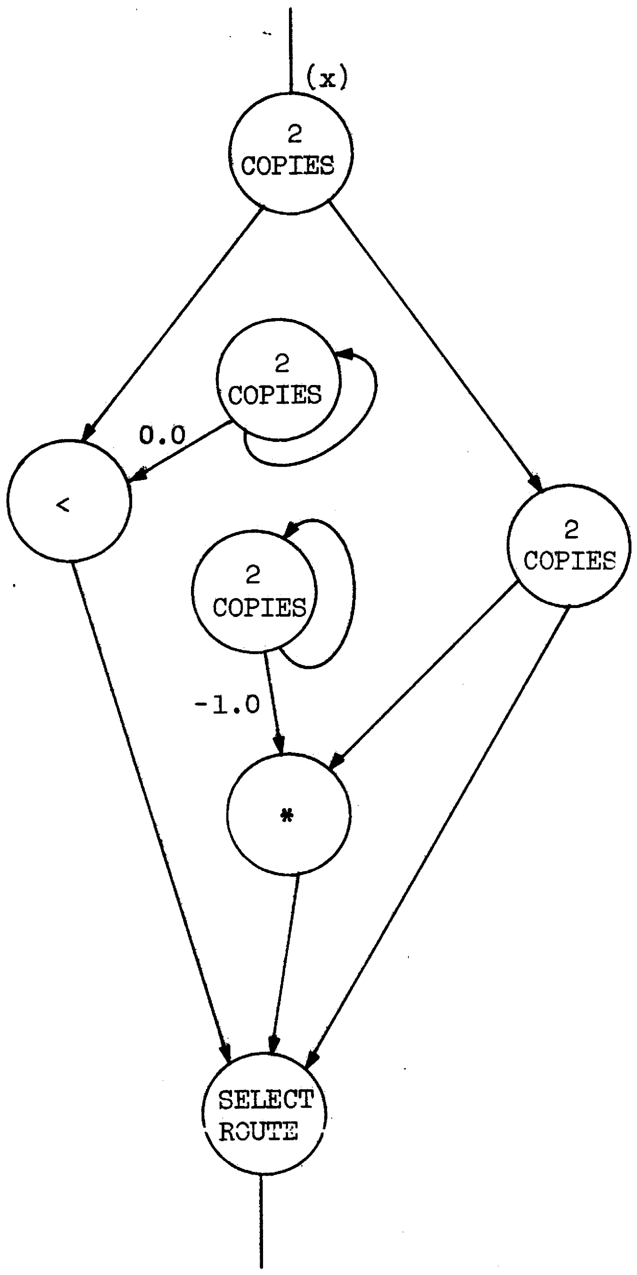
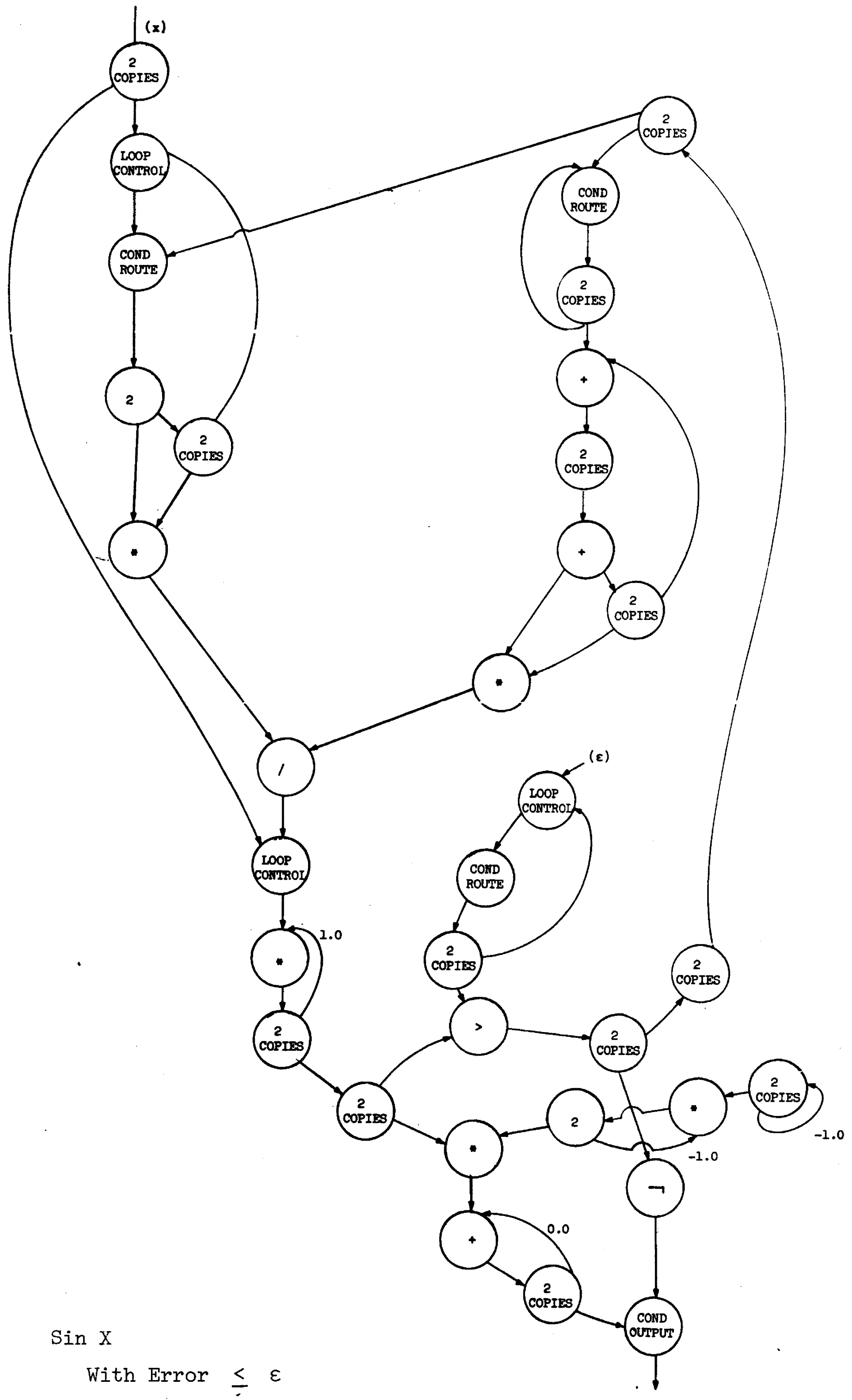


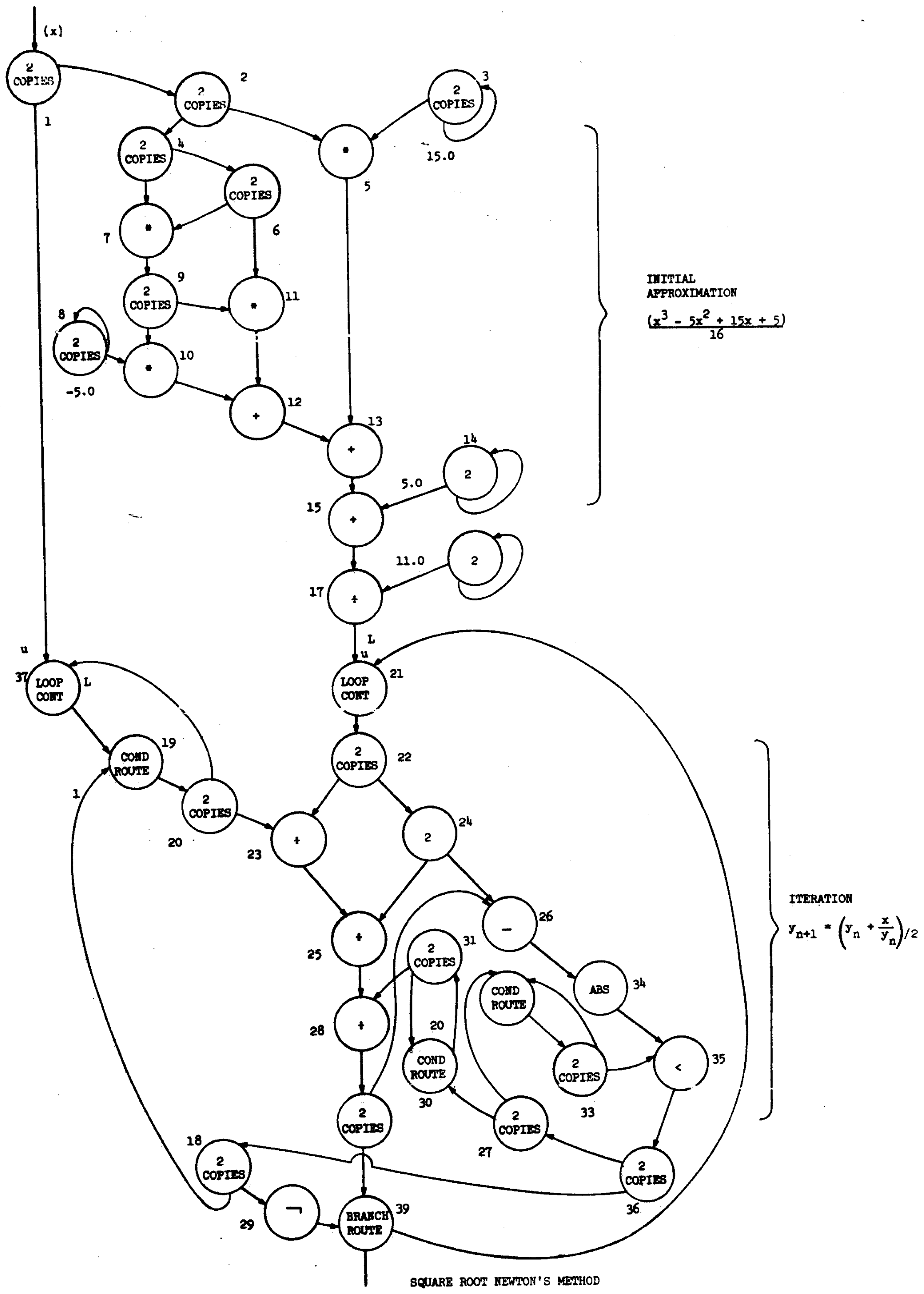
Fig. 12



Sin X

With Error $\leq$ ε

Fig. 13



SQUARE ROOT NEWTON'S METHOD

$y_0 = (x^3 - 5x^2 + 15x + 5)/16$

$y_{n+1} = \left(y_n \div \frac{x}{y_n} /2\right)$ $\qquad y_n \to \sqrt{x}$

Fig. 14

Square Root Procedure

This graph procedure calculates the square root of a positive floating point number by Newton's method. The initial approximation is provided from the polynomial $(x^3-5x^2+15x+5)/16$. This is derived from the 4 term Taylor series expansion for $(1+y)1/2 = 1 + y/2 - y^2/8 + y^3/16$ by setting $y = x - 1$. This polynomial is computed by nodes 1 through 17. The remaining nodes compute the approximation $Y_n+1 = (Y_n + x/Y_n)/2$ to $y = x$ and test for an error below a specified limit. The iteration stops when $|y_n - y_{n+1}|<\varepsilon$ where ε is the constant placed on the edge between nodes 32 and 33, in this case 10^{-5}. The test is computed by nodes 26, to 32-35, and the resultant boolean is distributed by nodes 36, 27, and 18 to the gating nodes which either enable another iteration or halt the computation and gate the result to the output edge of the procedure through node 39.

Fig. (15a) shows the processor resource usage for SQRT (2.0) under the assumption that all processors executed in equal times. Fig. (15b) shows the same computation with processor times which assume gating and similar operations take 1 cycle, additon, subtraction, logical operations and compares 2 cycles, multiplication 4 cycles, and division 6 cycles.

Newton's method is inherently sequential, so there is little overlap in the execution of this graph procedure. The maximum number of processors executing in any cycle was four. The time for execution was 133 cycles, but the total of processor cycles used was 200 so that 67 cycles were overlapped or 1/3 of the total. To put it another way, with strictly sequential execution the computation would have taken 1/3 longer.

```
  SQRT(2.0)   -  UNIFORM PROCESSOR SPEEDS
TOTAL PROCESSOR RESOURCE USAGE
    1:###
    2:##
    3:###
    4:##
    5:##
    6:#
    7:##
    8:#
    9:#
   10:#
   11:#
   12:#
   13:#
   14:##
   15:#
   16:#
   17:#
   18:#
   19:##
   20:###
   21:###
   22:##
   23:#
   24:#
   25:#
   26:##
   27:####
   28:####
   29:##
   30:#
   31:##
   32:#
   33:#
   34:#
   35:#
   36:##
   37:###
   38:###
   39:##
   40:#
   41:#
   42:#
   43:##
   44:####
   45:####
   46:##
   47:#
   48:##
   49:#
   50:#
   51:#
   52:#
   53:##
   54:###
   55:###
   56:##
   57:#
```

```
      58:#
      59:#
      60:##
      61:####
      62:#
      63:
      64:
  TOTAL RESOURCE CYCLES USED =  111.00   % UTILIZATION =    0.39
  AVERAGE RESOURCES USED PER TIME STEP =   1.73    MAXIMUM =    4
```

SQRT (2.0) - Varied Processor Speeds

```
TOTAL PROCESSOR RESOURCE USAGE
    1:###
    2:##
    3:###
    4:###
    5:###
    6:##
    7:#
    8:#
    9:#
   10:##
   11:##
   12:##
   13:##
   14:#
   15:#
   16:#
   17:#
   18:#
   19:#
   20:#
   21:#
   22:#
   23:#
   24:#
   25:#
   26:#
   27:#
   28:#
   29:#
   30:#
   31:##
   32:#
   33:#
   34:#
   35:#
   36:#
   37:#
   38:#
   39:#
   40:#
   41:#
   42:#
   43:#
   44:#
   45:#
   46:#
   47:#
   48:#
   49:#
   50:##
   51:###
   52:###
   53:###
   54:###
   55:##
   56:#
   57:#
   58:#
```

```
 59:#
 60:##
 61:####
 62:####
 63:##
 64:#
 65:#
 66:##
 67:#
 68:#
 69:#
 70:#
 71:#
 72:#
 73:#
 74:#
 75:#
 76:#
 77:#
 78:#
 79:#
 80:#
 81:#
 82:#
 83:#
 84:#
 85:##
 86:###
 87:###
 88:###
 89:###
 90:##
 91:#
 92:#
 93:#
 94:#
 95:##
 96:####
 97:####
 98:##
 99:#
100:#
101:##
102:#
103:#
104:#
105:#
106:#
107:#
108:#
109:#
110:#
111:#
112:#
113:#
114:#
115:#
116:#
117:#
118:#
```

```
  119:#
  120:##
  121:###
  122:###
  123:###
  124:###
  125:##
  126:#
  127:#
  128:#
  129:#
  130:##
  131:####
  132:#
  133:#
  134:
  135:
TOTAL RESOURCE CYCLES USED =      200    % UTILIZATION =        1
AVERAGE RESOURCES USED PER TIME STEP =       1     MAXIMUM =     4
```

$\int_{1}^{1.5} \sqrt{x}\, dx \qquad h = .1$

```
TOTAL PROCESSOR RESOURCE USAGE
    1:###
    2:###########
    3:########
    4:##########
    5:###########
    6:#########
    7:#######
    8:#####
    9:#########
   10:########
   11:###########
   12:###########
   13:############
   14:#############
   15:##########
   16:#########
   17:#######
   18:########
   19:############
   20:###########
   21:############
   22:###########
   23:###########
   24:###########
   25:############
   26:############
   27:###########
   28:##########
   29:##########
   30:##############
   31:#############
   32:###############
   33:#############
   34:#############
   35:#############
   36:##############
   37:##############
   38:#############
   39:#############
   40:############
   41:################
   42:###############
   43:###############
   44:###############
   45:###############
   46:###############
   47:################
   48:################
   49:################
   50:##############
   51:################
   52:######################
   53:#####################
   54:#####################
   55:#####################
   56:###################
   57:#################
   58:###################
```

```
 59:####################
 60:####################
 61:####################
 62:########################
 63:########################
 64:####################
 65:##################
 66:#################
 67:##################
 68:###################
 69:#######################
 70:#################
 71:#################
 72:################
 73:###############
 74:##############
 75:##############
 76:#############
 77:#############
 78:#############
 79:################
 80:#################
 81:###############
 82:###############
 83:##############
 84:##############
 85:############
 86:#############
 87:##############
 88:##############
 89:###############
 90:##################
 91:##################
 92:###############
 93:###############
 94:##############
 95:##############
 96:#############
 97:###############
 98:############
 99:############
100:############
101:###############
102:################
103:###############
104:###############
105:###############
106:###############
107:#############
108:############
109:###########
110:###########
111:###########
112:##############
113:################
114:#################
115:##############
116:##############
117:#############
118:###########
```

```
119:##########
120:#########
121:#########
122:##########
123:###########
124:############
125:#############
126:##########
127:##########
128:##########
129:#########
130:########
131:#######
132:#######
133:#######
134:######
135:#######
136:##########
137:########
138:########
139:#######
140:#######
141:######
142:#####
143:#####
144:#####
145:####
146:#####
147:########
148:######
149:######
150:#####
151:#####
152:####
153:###
154:###
155:###
156:##
157:###
158:#####
159:##
160:##
161:#
162:#
163:#
164:#
165:#
166:#
167:#
168:#
169:#
170:#
171:#
172:#
173:#
174:
175:

IHE500I SUBSCRIPTRANGE IN STATEMENT 00225 AT OFFSET +00812 FROM
```

$\int_{1}^{2.0} \sqrt{x}\,dx \qquad h = .1$

```
TOTAL PROCESSOR RESOURCE USAGE
    1:###
    2:###########
    3:########
    4:##########
    5:###########
    6:#########
    7:#######
    8:#####
    9:#########
   10:########
   11:###########
   12:###########
   13:############
   14:#############
   15:##########
   16:#########
   17:#######
   18:########
   19:############
   20:###########
   21:############
   22:###########
   23:###########
   24:###########
   25:############
   26:############
   27:###########
   28:##########
   29:##########
   30:##############
   31:#############
   32:###############
   33:#############
   34:#############
   35:#############
   36:##############
   37:##############
   38:#############
   39:#############
   40:############
   41:################
   42:###############
   43:###############
   44:###############
   45:###############
   46:###############
   47:################
   48:################
   49:################
   50:##############
   51:################
   52:######################
   53:#####################
   54:#####################
   55:#####################
   56:###################
   57:#################
   58:###################
```

```
 59:####################
 60:####################
 61:####################
 62:########################
 63:########################
 64:####################
 65:##################
 66:#################
 67:##################
 68:###################
 69:#######################
 70:####################
 71:####################
 72:#################
 73:################
 74:###################
 75:##################
 76:##################
 77:##################
 78:##################
 79:#####################
 80:#######################
 81:#####################
 82:####################
 83:##################
 84:##################
 85:####################
 86:####################
 87:#####################
 88:#####################
 89:######################
 90:#########################
 91:##########################
 92:#######################
 93:######################
 94:####################
 95:####################
 96:#######################
 97:########################
 98:########################
 99:######################
100:######################
101:#########################
102:############################
103:##########################
104:##########################
105:########################
106:########################
107:##########################
108:########################
109:#######################
110:#######################
111:########################
112:###########################
113:##############################
114:###############################
115:############################
116:##########################
117:#########################
118:###########################
```

```
119:#########################
120:########################
121:#########################
122:###########################
123:#############################
124:################################
125:##############################
126:##########################
127:#########################
128:########################
129:######################
130:#####################
131:####################
132:######################
133:####################
134:######################
135:#######################
136:########################
137:######################
138:#####################
139:####################
140:##################
141:#################
142:################
143:################
144:#################
145:###################
146:####################
147:#####################
148:###################
149:##################
150:#################
151:###############
152:##############
153:#############
154:#############
155:##############
156:################
157:##################
158:###################
159:################
160:################
161:###############
162:#############
163:############
164:###########
165:###########
166:############
167:##############
168:################
169:#################
170:#################
171:###############
172:##############
173:############
174:############
175:##########
176:##########
177:###########
178:#############
```

```
179:##############
180:###############
181:###############
182:#############
183:#############
184:############
185:############
186:##########
187:##########
188:##########
189:##########
190:###########
191:##############
192:###############
193:##############
194:##############
195:##############
196:##############
197:############
198:###########
199:##########
200:##########
201:###########
202:##############
203:################
204:#################
205:##############
206:##############
207:#############
208:###########
209:##########
210:#########
211:#########
212:##########
213:###########
214:############
215:#############
216:##########
217:##########
218:##########
219:#########
220:########
221:#######
222:#######
223:#######
224:######
225:#######
226:##########
227:########
228:########
229:#######
230:#######
231:######
232:#####
233:#####
234:#####
235:####
236:#####
237:########
238:######
```

```
 239:######
 240:#####
 241:#####
 242:####
 243:###
 244:###
 245:###
 246:##
 247:###
 248:#####
 249:##
 250:##
 251:#
 252:#
 253:#
 254:#
 255:#
 256:#
 257:#
 258:#
 259:#
 26C:#
 261:#
 262:#
 263:#
 264:
 265:
TOTAL RESOURCE CYCLES USED =     3840   % UTILIZATION =        7
AVERAGE RESOURCES USED PER TIME STEP =     14    MAXIMUM =    32
```

Variation of Relative Processor Speeds

The trapezoidal quadrature program was run with all processors executing in one cycle (Fig. 18) and with the times for different processors varied from one cycle for simple gating nodes to 6 cycles for divide (Fig. 19) 19); with uniform times the computation took 70 cycles and with varied execution times it took 122 cycles. In order to compare the two cases meaningfully, the uniform execution run must be scaled so that the common processor speed is equal to the mean of the speeds of the nodes in the graph program under the varied execution case. Otherwise, the first case just represents a run with faster hardware than the second. Not counting dummy nodes which do do not execute, the 69 nodes in the graph program for $\int\sqrt{x}dx$ represent a total of 121 cycles using the timings of the varied processor speed simulation. This gives a mean execution time of 121/69 = 1.75 cycles. Hence we have the following:

Tvaried = 122 cycles

Tuniform = 70*1.75 cycles = 122.5 cycles

In order to test the effect of slowing down a single processor type to the point where it could cause significant delays, we re-ran the simulation of varied processor times with the divide slowed down to 16 cycles and other times the same (Fig. 20). This is a slower divide, relative to other operations, than is found in current large scale computers. Normalized to fixed point add, one finds divide times ranging from 5.00 (CDC 7600) to 9.75 (IBM 360*/75).* The computation took 172 cycles with the slow divide. The total number of cycles represented by the nodes in the graph program is 161 when divide = 16, so that the mean time for a node to execute is 2.33 cycles giving

Tuniform = 2.33 x 70 cycles = 163.1 cycles

Tslow divide = 172 cycles

Putting in a very slow divide unit thus results in a slower computation than increasing the mean processor execution time by a corresponding amount. Thus there are probably significant delays when other nodes are idle waiting for the result of a divide. However, the increase in execution time is only 5.6 percent even in this case where one node type is four times slower than the next slowest node, the multiply. In the case where the divide time is more nearly comparable to other processor speeds, the difference between varied processor speeds and a uniform execution time, which keeps the mean processor execution time constant, is negligible.

Trapezoidal Runs

The trapezoidal quadrature program was run using SQRT (x) and SIN (x) as the functions to be integrated. These functions are complex enough so that they will execute concurrently for several values of x. Further more, the execution of SQRT(x) is data dependent since it is an iterative approximation program whose initial approximation becomes worse as x moves away from 7.0. This dependence is illustrated in figs. 21a-21e, which show the processor usage for SQRT(x), x=2,3,4,5, and 10. The computation took from 133 to 308 cycles and from 200 to 470 processor cycles were used. None of the other programs simulated is data dependent in the sense that the amount of computation depends on the value of the data used.

As a result, the time required to compute $\sqrt{x}ds$ is not simply a function of the number of points used in the quadrature. Rather, it has the form t = f(n) + g(a,b), where n is the number of points used and (a, b) is the interval over which $\sqrt{x}$ is integrated.

*The IBM 360/40, however, has a relative divide speed of 17.1.

```
TOTAL PRCCESSCR RESOURCE LSAGE        ∫√x dx      all processors execute in
                                                  one cycle
    1:###
    2:###########
    3:########
    4:##########
    5:#########
    6:#######
    7:#########
    8:##########
    9:#########
   10 :#########
   11:###########
   12:#########
   13:##########
   14:###########
   15:############
   16:###########
   17:##########
   18:###########
   19:############
   20:###############
   21:###############
   22:#################
   23:##############
   24:#############
   25:#############
   26 :###############
   27:###################
   28:##################
   29:#############
   30:##########
   31:########
   32:###########
   33:#############
   34:##########
   35:#########
   36:##########
   37:##########
   38:############
   39:############
   40:##############
   41:#############
   42 :############
   43:#########
   44:##########
   45:###########
   46:########
   47:########
   48:##########
   49:############
   50:#########
   51:########
   52:########
   53:######
   54:#######
   55:########
   56:#########
   57:###########
   58:########
```

Fig. 18

```
   59: #######
   60:#####
   61:#####
   62: #######
   63 :####
   64: ####
   65:#####
   66:##
   67:#
   68 :#
   69:#
   70:#
   71:
   72:
TOTAL RESOURCE CYCLES USED =      663   % UTILIZATION =      920
AVERAGE RESOURCES USED PER TIME STEP =        9    MAXIMUM =   19
```

```
TOTAL PROCESSOR RESOURCE USAGE
    1:###
    2:###########
    3:########
    4:##########
    5:###########
    6:#########
    7:#######
    8:#########
    9:########
   10:#########
   11:###########
   12:############
   13:#############
   14:############
   15:#########
   16:#######
   17:############
   18:###########
   19:############
   20:############
   21:###########
   22:###########
   23:############
   24:############
   25:###########
   26:##########
   27:##############
   28:#############
   29:################
   30:#############
   31:#############
   32:#############
   33:##############
   34:###########
   35:###########
   36:###########
   37:###########
   38:###########
   39:###########
   40:##########
   41:##########
   42:##########
   43:##########
   44:###########
   45:##########
   46:############
   47:##############
   48:##############
   49:##############
   50:##############
   51:############
   52:###########
   53:############
   54:#############
   55:############
   56:##############
   57:#################
   58:#############
```

Fig. 19

```
 59:###########
 60:#########
 61:#########
 62:############
 63:#############
 64:##########
 65:##########
 66:########
 67:######
 68:######
 69:######
 70:######
 71:########
 72:###########
 73:###########
 74:#########
 75:########
 76:#######
 77:#######
 78:######
 79:#######
 80:########
 81:#########
 82:###########
 83:###########
 84:########
 85:######
 86:######
 87:#######
 88:######
 89:#######
 90:#########
 91:######
 92:######
 93:#####
 94:######
 95:#######
 96:#######
 97:#######
 98:#######
 99:######
100:#####
101:#####
102:#####
103:####
104:######
105:#########
106:######
107:######
108:#####
109:###
110:##
111:##
112:##
113:##
114:###
115:#####
116:##
117:##
118:#
```

```
 119:#
 120:#
 121:#
 122:#
 123:
 124:
TOTAL RESOURCE CYCLES USED =     1062   % UTILIZATION =       61
AVERAGE RESOURCES USED PER TIME STEP =       8    MAXIMUM =   17
```

```
TOTAL PROCESSOR RESOURCE USAGE                 ∫√x dx    divide = 16 cycles
   1:###
   2:###########
   3:########
   4:##########
   5 :###########
   6:#########
   7 :#######
   8:#########
   9:########
  10:#########
  11:###########
  12:############
  13:#############
  14:############
  15:#########
  16 :#######
  17:############
  18:###########
  19:############
  20:############
  21 :###########
  22:###########
  23:############
  24:############
  25:###########
  26:##########
  27:##############
  28:#############
  29:##############
  30:#############
  31:#############
  32:#############
  33:##############
  34:###########
  35:##########
  36:###########
  37 :###########
  38:###########
  39:#############
  40:##########
  41:##########
  42:##########
  43:##########
  44:##########
  45:###########
  46:##########
  47:##########
  48:##########
  49:##########
  50:##########
  51:##########
  52:##########
  53 :##########
  54:###########
  55:##########
  56:##########
  57:##########
  58:##########
```

Fig. 20

```
 59: ##########
 60 :##########
 61:##########
 62: ##########
 63:##########
 64: ###########
 65: ##########
 66:##########
 67:##########
 68: ##########
 69:##########
 70: ##########
 71:##########
 72: ##########
 73: ##########
 74 :##########
 75: ##########
 76:############
 77:##############
 78: ##############
 79:##############
 80: ##############
 81:############
 82:###########
 83: ############
 84: ############
 85:############
 86: ##############
 87:#################
 88: #############
 89: ###########
 90 :#########
 91: #########
 92:############
 93:#############
 94: ##########
 95:##########
 96:########
 97:######
 98:######
 99: ######
100: ######
101:########
102: ###########
103:###########
104:#########
105: ########
106 :#######
107: #######
108:######
109:######
110: ######
111:#######
112:#########
113:#########
114:#######
115:######
116: ######
117:#######
118: ######
```

```
 119:######
 120:######
 121:######
 122:######
 123:######
 124:######
 125:######
 126:######
 127:######
 128:######
 129:#######
 130:########
 131:########
 132:########
 133:########
 134:#######
 135:######
 136:######
 137:######
 138:######
 139:#######
 140:#########
 141:######
 142:######
 143:#####
 144:######
 145:#######
 146:#######
 147:#######
 148:#######
 149:######
 150:#####
 151:#####
 152:#####
 153:#####
 154:#######
 155:##########
 156:#######
 157:#######
 158:######
 159:####
 160:###
 161:###
 162:###
 163:##
 164:###
 165:#####
 166:##
 167:##
 168:#
 169:#
 170:#
 171:#
 172:#
 173:
 174:
TOTAL RESOURCE CYCLES USED =     1492   % UTILIZATION =      61
AVERAGE RESOURCES USED PER TIME STEP =       8    MAXIMUM =    17
```

```
SQRT(2)      VARIED PROCESSOR TIMES
TOTAL PROCESSOR RESOURCE USAGE
    1:###
    2:##
    3:###
    4:###
    5:###
    6:##
    7:#
    8:#
    9:#
   10:##
   11:##
   12:##
   13:##
   14:#
   15:#
   16:#
   17:#
   18:#
   19:#
   20:#
   21:#
   22:#
   23:#
   24:#
   25:#
   26:#
   27:#
   28:#
   29:#
   30:#
   31:##
   32:#
   33:#
   34:#
   35:#
   36:#
   37:#
   38:#
   39:#
   40:#
   41:#
   42:#
   43:#
   44:#
   45:#
   46:#
   47:#
   48:#
   49:#
   50:##
   51:###
   52:###
   53:###
   54:###
   55:##
   56:#
   57:#
```

```
 58:#
 59:#
 60:##
 61:####
 62:####
 63:##
 64:#
 65:#
 66:##
 67:#
 68:#
 69:#
 70:#
 71:#
 72:#
 73:#
 74:#
 75:#
 76:#
 77:#
 78:#
 79:#
 80:#
 81:#
 82:#
 83:#
 84:#
 85:##
 86:###
 87:###
 88:###
 89:###
 90:##
 91:#
 92:#
 93:#
 94:#
 95:##
 96:####
 97:####
 98:##
 99:#
100:#
101:##
102:#
103:#
104:#
105:#
106:#
107:#
108:#
109:#
110:#
111:#
112:#
113:#
114:#
115:#
116:#
117:#
```

```
 118:#
 119:#
 120:##
 121:###
 122:###
 123:###
 124:###
 125:##
 126:#
 127:#
 128:#
 129:#
 130:##
 131:####
 132:#
 133:#
 134:
 135:
TOTAL RESOURCE CYCLES USED =   200.00    % UTILIZATION =     0.63
AVERAGE RESOURCES USED PER TIME STEP =    1.48    MAXIMUM =     4
```

```
SQRT(3)      VARIED PROCESSOR TIMES
TOTAL PROCESSOR RESOURCE USAGE
    1:###
    2:##
    3:###
    4:###
    5:###
    6:##
    7:#
    8:#
    9:#
   10:##
   11:##
   12:##
   13:##
   14:#
   15:#
   16:#
   17:#
   18:#
   19:#
   20:#
   21:#
   22:#
   23:#
   24:#
   25:#
   26:#
   27:#
   28:#
   29:#
   30:#
   31:##
   32:#
   33:#
   34:#
   35:#
   36:#
   37:#
   38:#
   39:#
   40:#
   41:#
   42:#
   43:#
   44:#
   45:#
   46:#
   47:#
   48:#
   49:#
   50:##
   51:###
   52:###
   53:###
   54:###
   55:##
   56:#
   57:#
```

```
 58:#
 59:#
 60:##
 61:####
 62:####
 63:##
 64:#
 65:#
 66:##
 67:#
 68:#
 69:#
 70:#
 71:#
 72:#
 73:#
 74:#
 75:#
 76:#
 77:#
 78:#
 79:#
 80:#
 81:#
 82:#
 83:#
 84:#
 85:##
 86:###
 87:###
 88:###
 89:###
 90:##
 91:#
 92:#
 93:#
 94:#
 95:##
 96:####
 97:####
 98:##
 99:#
100:#
101:##
102:#
103:#
104:#
105:#
106:#
107:#
108:#
109:#
110:#
111:#
112:#
113:#
114:#
115:#
116:#
117:#
```

```
  118:#
  119:#
  120:##
  121:###
  122:###
  123:###
  124:###
  125:##
  126:#
  127:#
  128:#
  129:#
  130:##
  131:####
  132:####
  133:##
  134:#
  135:#
  136:##
  137:#
  138:#
  139:#
  140:#
  141:#
  142:#
  143:#
  144:#
  145:#
  146:#
  147:#
  148:#
  149:#
  150:#
  151:#
  152:#
  153:#
  154:#
  155:##
  156:###
  157:###
  158:###
  159:###
  160:##
  161:#
  162:#
  163:#
  164:#
  165:##
  166:####
  167:#
  168:#
  169:
  170:
TOTAL RESOURCE CYCLES USED =  254.00   % UTILIZATION =    0.64
AVERAGE RESOURCES USED PER TIME STEP =   1.49    MAXIMUM =    4
```

```
SQRT(4)      VARIED PROCESSCR TIMES
TCTAL PROCESSOR RESOURCE USAGE
   1:###
   2:##
   3:###
   4:###
   5:###
   6:##
   7:#
   8:#
   9:#
  10:##
  11:##
  12:##
  13:##
  14:#
  15:#
  16:#
  17:#
  18:#
  19:#
  20:#
  21:#
  22:#
  23:#
  24:#
  25:#
  26:#
  27:#
  28:#
  29:#
  30:#
  31:##
  32:#
  33:#
  34:#
  35:#
  36:#
  37:#
  38:#
  39:#
  40:#
  41:#
  42:#
  43:#
  44:#
  45:#
  46:#
  47:#
  48:#
  49:#
  50:##
  51:###
  52:###
  53:###
  54:###
  55:##
  56:#
  57:#
```

```
 58:#
 59:#
 60:##
 61:####
 62:####
 63:##
 64:#
 65:#
 66:##
 67:#
 68:#
 69:#
 70:#
 71:#
 72:#
 73:#
 74:#
 75:#
 76:#
 77:#
 78:#
 79:#
 80:#
 81:#
 82:#
 83:#
 84:#
 85:##
 86:###
 87:###
 88:###
 89:###
 90:##
 91:#
 92:#
 93:#
 94:#
 95:##
 96:####
 97:####
 98:##
 99:#
100:#
101:##
102:#
103:#
104:#
105:#
106:#
107:#
108:#
109:#
110:#
111:#
112:#
113:#
114:#
115:#
116:#
117:#
```

```
118:#
119:#
120:##
121:###
122:###
123:###
124:###
125:##
126:#
127:#
128:#
129:#
130:##
131:####
132:####
133:##
134:#
135:#
136:##
137:#
138:#
139:#
140:#
141:#
142:#
143:#
144:#
145:#
146:#
147:#
148:#
149:#
150:#
151:#
152:#
153:#
154:#
155:##
156:###
157:###
158:###
159:###
160:##
161:#
162:#
163:#
164:#
165:##
166:####
167:####
168:##
169:#
170:#
171:##
172:#
173:#
174:#
175:#
176:#
177:#
```

```
 178:#
 179:#
 180:#
 181:#
 182:#
 183:#
 184:#
 185:#
 186:#
 187:#
 188:#
 189:#
 190:##
 191:###
 192:###
 193:###
 194:###
 195:##
 196:#
 197:#
 198:#
 199:#
 200:##
 201:####
 202:#
 203:#
 204:
 205:
TOTAL RESOURCE CYCLES USED =  308.00   % UTILIZATION =    0.64
AVERAGE RESOURCES USED PER TIME STEP =   1.50    MAXIMUM =    4
```

```
SQRT(5)      VARIED PROCESSOR TIMES
TOTAL PROCESSOR RESOURCE USAGE
    1:###
    2:##
    3:###
    4:###
    5:###
    6:##
    7:#
    8:#
    9:#
   10:##
   11:##
   12:##
   13:##
   14:#
   15:#
   16:#
   17:#
   18:#
   19:#
   20:#
   21:#
   22:#
   23:#
   24:#
   25:#
   26:#
   27:#
   28:#
   29:#
   30:#
   31:##
   32:#
   33:#
   34:#
   35:#
   36:#
   37:#
   38:#
   39:#
   40:#
   41:#
   42:#
   43:#
   44:#
   45:#
   46:#
   47:#
   48:#
   49:#
   50:##
   51:###
   52:###
   53:###
   54:###
   55:##
   56:#
   57:#
```

```
 58:#
 59:#
 60:##
 61:####
 62:####
 63:##
 64:#
 65:#
 66:##
 67:#
 68:#
 69:#
 70:#
 71:#
 72:#
 73:#
 74:#
 75:#
 76:#
 77:#
 78:#
 79:#
 80:#
 81:#
 82:#
 83:#
 84:#
 85:##
 86:###
 87:###
 88:###
 89:###
 90:##
 91:#
 92:#
 93:#
 94:#
 95:##
 96:####
 97:####
 98:##
 99:#
100:#
101:##
102:#
103:#
104:#
105:#
106:#
107:#
108:#
109:#
110:#
111:#
112:#
113:#
114:#
115:#
116:#
117:#
```

```
118:#
119:#
120:##
121:###
122:###
123:###
124:###
125:##
126:#
127:#
128:#
129:#
130:##
131:####
132:####
133:##
134:#
135:#
136:##
137:#
138:#
139:#
140:#
141:#
142:#
143:#
144:#
145:#
146:#
147:#
148:#
149:#
150:#
151:#
152:#
153:#
154:#
155:##
156:###
157:###
158:###
159:###
160:##
161:#
162:#
163:#
164:#
165:##
166:####
167:####
168:##
169:#
170:#
171:##
172:#
173:#
174:#
175:#
176:#
177:#
```

```
  178:#
  179:#
  180:#
  181:#
  182:#
  183:#
  184:#
  185:#
  186:#
  187:#
  188:#
  189:#
  190:##
  191:###
  192:###
  193:###
  194:###
  195:##
  196:#
  197:#
  198:#
  199:#
  200:##
  201:####
  202:#
  203:#
  204:
  205:
TOTAL RESOURCE CYCLES USED =  308.00   % UTILIZATION =    0.64
AVERAGE RESOURCES USED PER TIME STEP =   1.50    MAXIMUM =    4
```

```
SQRT(10)     VARIED PROCESSOR TIMES
TOTAL PROCESSOR RESOURCE USAGE
    1:###
    2:##
    3:###
    4:###
    5:###
    6:##
    7:#
    8:#
    9:#
   10:##
   11:##
   12:##
   13:##
   14:#
   15:#
   16:#
   17:#
   18:#
   19:#
   20:#
   21:#
   22:#
   23:#
   24:#
   25:#
   26:#
   27:#
   28:#
   29:#
   30:#
   31:##
   32:#
   33:#
   34:#
   35:#
   36:#
   37:#
   38:#
   39:#
   40:#
   41:#
   42:#
   43:#
   44:#
   45:#
   46:#
   47:#
   48:#
   49:#
   50:##
   51:###
   52:###
   53:###
   54:###
   55:##
   56:#
   57:#
```

```
 58:#
 59:#
 60:##
 61:####
 62:####
 63:##
 64:#
 65:#
 66:##
 67:#
 68:#
 69:#
 70:#
 71:#
 72:#
 73:#
 74:#
 75:#
 76:#
 77:#
 78:#
 79:#
 80:#
 81:#
 82:#
 83:#
 84:#
 85:##
 86:###
 87:###
 88:###
 89:###
 90:##
 91:#
 92:#
 93:#
 94:#
 95:##
 96:####
 97:####
 98:##
 99:#
100:#
101:##
102:#
103:#
104:#
105:#
106:#
107:#
108:#
109:#
110:#
111:#
112:#
113:#
114:#
115:#
116:#
117:#
```

```
118:#
119:#
120:##
121:###
122:###
123:###
124:###
125:##
126:#
127:#
128:#
129:#
130:##
131:####
132:####
133:##
134:#
135:#
136:##
137:#
138:#
139:#
140:#
141:#
142:#
143:#
144:#
145:#
146:#
147:#
148:#
149:#
150:#
151:#
152:#
153:#
154:#
155:##
156:###
157:###
158:###
159:###
160:##
161:#
162:#
163:#
164:#
165:##
166:####
167:####
168:##
169:#
170:#
171:##
172:#
173:#
174:#
175:#
176:#
177:#
```

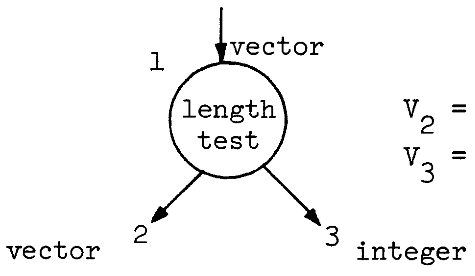
```
178:#
179:#
180:#
181:#
182:#
183:#
184:#
185:#
186:#
187:#
188:#
189:#
190:##
191:###
192:###
193:###
194:###
195:##
196:#
197:#
198:#
199:#
200:##
201:####
202:####
203:##
204:#
205:#
206:##
207:#
208:#
209:#
210:#
211:#
212:#
213:#
214:#
215:#
216:#
217:#
218:#
219:#
220:#
221:#
222:#
223:#
224:#
225:##
226:###
227:###
228:###
229:###
230:##
231:#
232:#
233:#
234:#
235:##
236:####
237:####
```

```
238:##
239:#
240:#
241:##
242:#
243:#
244:#
245:#
246:#
247:#
248:#
249:#
250:#
251:#
252:#
253:#
254:#
255:#
256:#
257:#
258:#
259:#
260:##
261:###
262:###
263:###
264:###
265:##
266:#
267:#
268:#
269:#
270:##
271:####
272:####
273:##
274:#
275:#
276:##
277:#
278:#
279:#
280:#
281:#
282:#
283:#
284:#
285:#
286:#
287:#
288:#
289:#
290:#
291:#
292:#
293:#
294:#
295:##
296:###
297:###
```

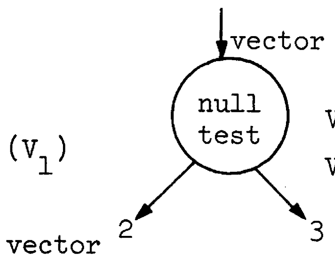
```
  298:###
  299:###
  300:##
  301:#
  302:#
  303:#
  304:#
  305:##
  306:####
  307:#
  308:#
  309:
  310:
TOTAL RESOURCE CYCLES USED =   470.00    % UTILIZATION =      0.65
AVERAGE RESOURCES USED PER TIME STEP =    1.52     MAXIMUM =      4
```

SORT PROGRAM

The SORT program was written by Duane Adams. The version used for simulation differs from his in two respects. First, the primitive node set is different for certain vector (record) operations. For certain operations, such as length or null, one almost always wants to use the operand later, as well as the result of the operation. Thus, in Adams' program, length is proceeded by a two copies node, one output of which is fed into the length node. Since making a copy of a vector or record is bound to be a time consuming operation, in this version the primitive node length outputs both the length of the vector and the vector itself. Thus, there is no need to make a second copy of the vector. The relevant primitive nodes are shown below:



$V_2 = V_1$
$V_3 = \text{length } (V_1)$



$V_2 = V_1$
$V_3 = V_1 = \emptyset$ then true else false

The second respect in which my program differs from Adams' is that the procedure ROUTE SELECT was rewritten to allow for more parallelism. In Adams' version, shown in Figure 3a, comparison of the first element of the two two records must wait until the determination of whether either record is null. In mine, it proceeds simultaneously with the null check, and the conditional output nodes (4 and 5 in figure 3a) is moved to the bottom of the graph. This has the disadvantage that the procedure may take the first of a null record, which works in my implementation but gives a meaningless result. The result of a meaningless comparison is never output, however, and the procedure works much faster for the common case where neither record is null (6 cycles vs. 11 cycles).

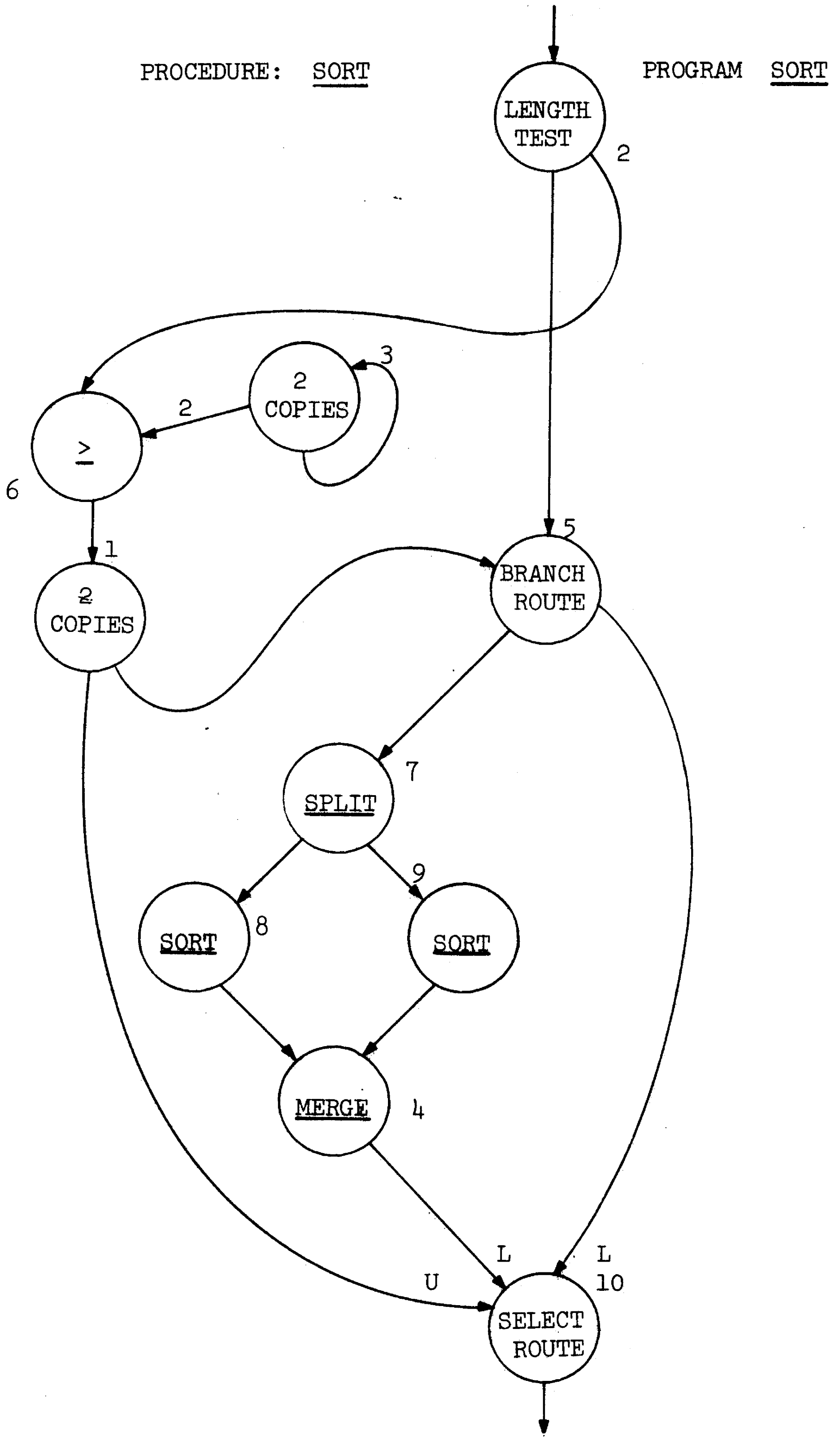

PROCEDURE: SORT
PROGRAM SORT
LENGTH TEST
2
2 COPIES
3
2
≥
6
1
2 COPIES
5
BRANCH ROUTE
SPLIT
7
9
SORT
8
SORT
MERGE
4
L
L
U
10
SELECT ROUTE

Fig. 22

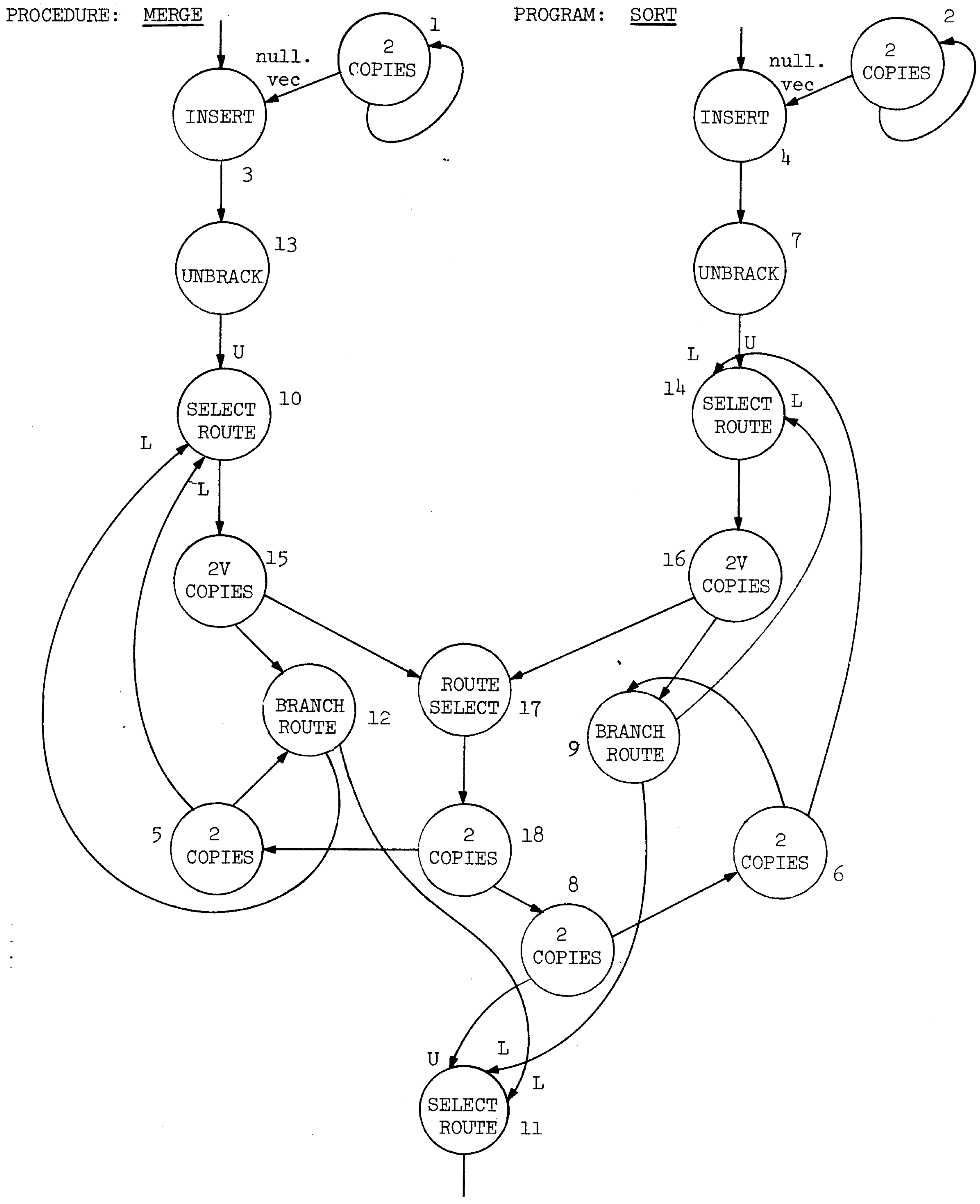

PROCEDURE: MERGE
PROGRAM: SORT
2 COPIES 1
null. vec
INSERT 3
UNBRACK 13
U
SELECT ROUTE 10
L
L
2V COPIES 15
BRANCH ROUTE 12
ROUTE SELECT 17
2 COPIES 5
2 COPIES 18
2 COPIES 8
2 COPIES 2
null. vec
INSERT 4
UNBRACK 7
U
L
SELECT ROUTE 14
L
2V COPIES 16
BRANCH ROUTE 9
2 COPIES 6
U
L
L
SELECT ROUTE 11

Fig. 23

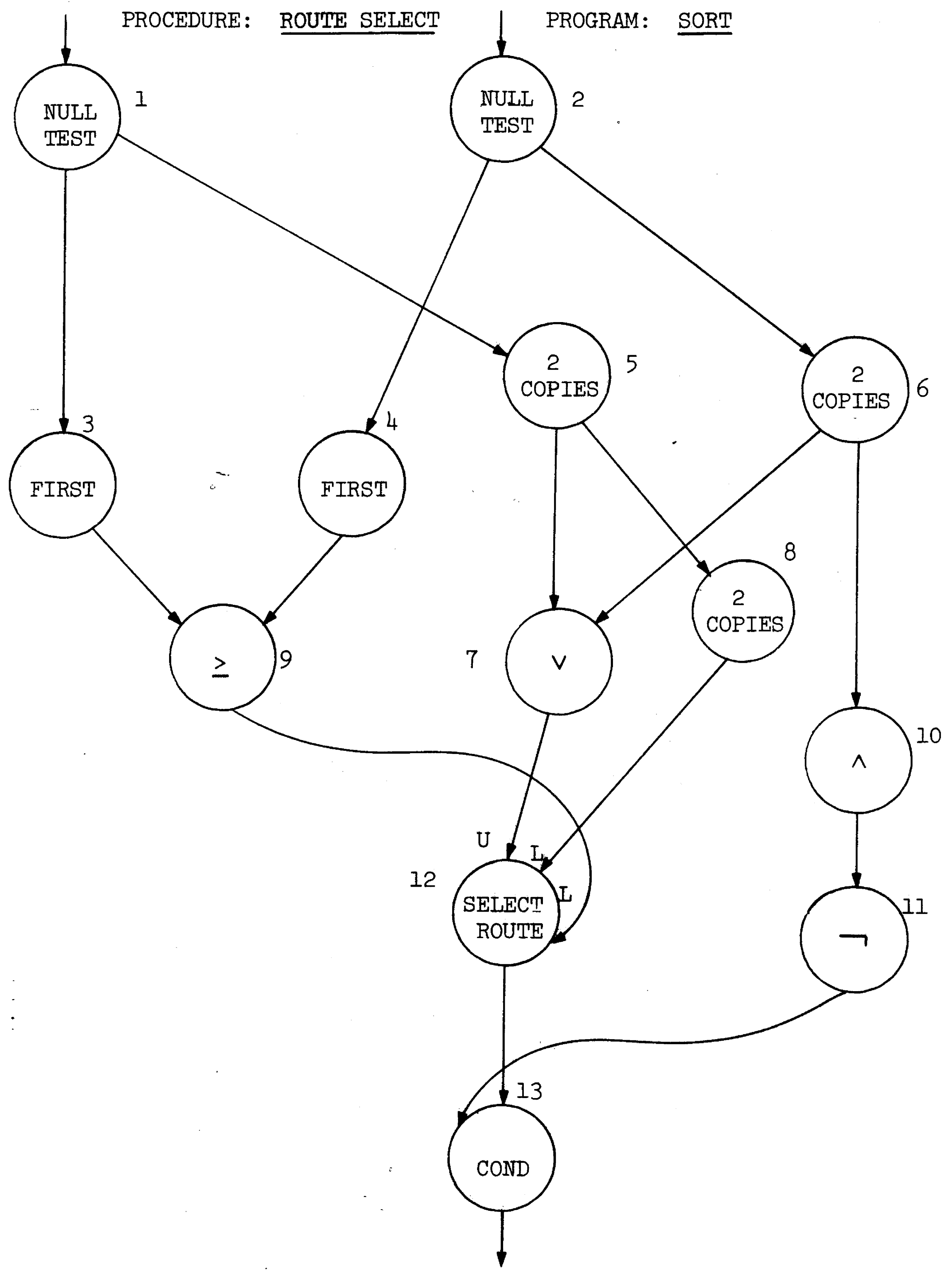

PROCEDURE: ROUTE SELECT
PROGRAM: SORT
NULL TEST
1
NULL TEST
2
2 COPIES
5
2 COPIES
6
3
FIRST
4
FIRST
8
2 COPIES
≥
9
7
∨
10
∧
U
L
12
SELECT ROUTE
L
11
¬
13
COND

Fig. 24

PROCEDURE: ROUTE SELECT PROGRAM: SORT

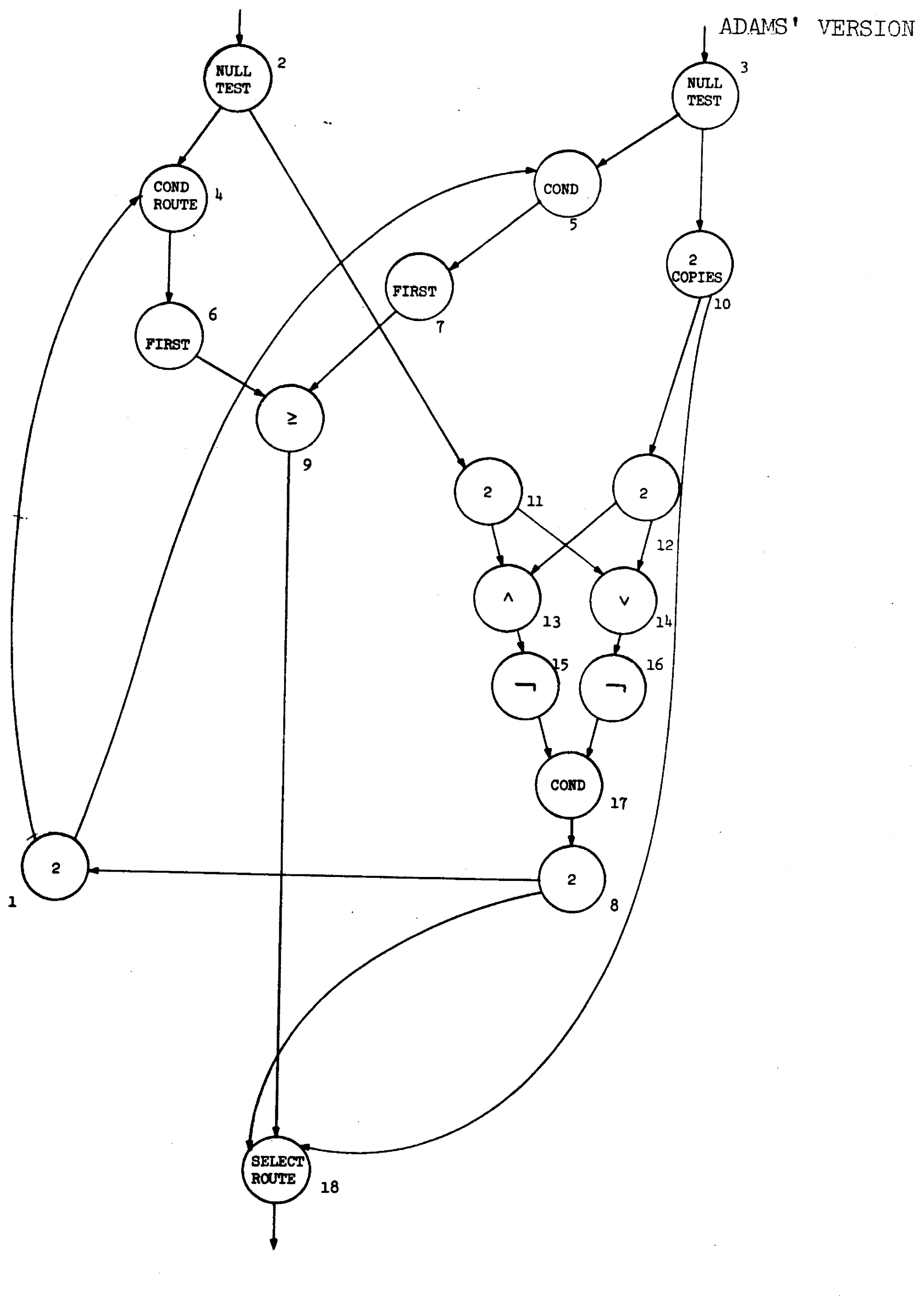


Fig. 25

Since the only data type implemented in the simulator is floating point, a record is identical to a vector. A file is just a vector each of whose elements is a record (vector). If a file contains m records each of length n, its representation is identical to that of an m by n matrix.

Variation of Execution Time with File Size

The processor usage for sorts of various length files are shown.

The sort program can be considered as having two parts, the first of which recursively splits the file into subfiles, and the second of which merges the subflies together again. The merge is not initiated until the split has reached the lowest level. The number of stages required to split the original file into subfiles of length 1 is equal to $\lceil \log_2 n \rceil$, where n is the number of records in the file. There will then be a similar number of merge stages, at each of which the subfiles are merged pairwise. The time taken by the merge procedure to merge two files of length n_1 and n_2 respectively will be proportional to $n_1 + n_2$ since each comparison results in one record being put on the output edge and there are $n_1 + n_2$ records in the output file. Since the merging of a subfile pair at any stage proceeds in parallel with that of all other pairs at the same stage, the time for each merge stage is determined by the length of the longest subfile produced by this stage. The total merge time is the sum of the times taken by each stage, and will thus be proportional to the sum of the lengths of the longest file produced at each stage. The last stage produces one file of length n, the next-to-last stage produces the longest of the two inputs to the the last stage, i.e. a file of length $\lceil n/2 \rceil$, the stage before that a file of length $\lceil n/4 \rceil$ etc. The time to merge is thus proportional to

$$n + \lceil n/2 + n/4 \rceil + \ldots + 2 = \sum_{i=0}^{\lceil \log_2 n \rceil - 1} \lceil n/2^i \rceil$$

If we then write time taken by the sort as T= const. + t_s + t_m where t_s is the time to split the file into subfiles and t_m the time to merge the subfiles, then we have

$$(1) \qquad T = k_0 + k_1\lceil \log_2 n\rceil + k_2 \sum_{i=0}^{\lceil \log_2 n\rceil -1} \lceil n/2^i\rceil$$

When n is a power of 2, $n = 2^m$, the series in the last term is equal to 2n-2, i.e.

$$\sum_{i=0}^{[\log_2 2^m] -1} 2^m/2^i = \sum_{i=0}^{m-1} 2m/2^i = 2^{m+1} - 2 = 2n - 2$$

Giving $T = k_o + k_1 \ \log_2 n \ + k_2(2n-2)$

Otherwise

$$2n-2 < \sum_{i=0}^{\lceil \log_2 n\rceil-1} \lceil n/2^i\rceil \leq 2n + \lceil \log_2 n\rceil - 2$$

The right side follows from

$$\sum_{i=0}^{\lceil \log_2 n\rceil-1} [n/2^i] \leq \sum_{i=0}^{\lceil \log_2 n\rceil-1} (n/2^i+1) = \sum_{i=0}^{\lceil \log_2 n\rceil-1} (n/2^i) + \sum_{i=0}^{\lceil \log_2 n\rceil-1} 1$$

$$= \sum_{i=0}^{\lceil \log_2 n\rceil-1} (1/2^i) + \lceil \log_2 n\rceil$$

$$= n \left(\frac{1-(1/2)^{\lceil \log_2 n\rceil}}{1-(1/2)}\right) + \lceil \log_2 n\rceil$$

$$= n \left(\frac{2\cdot 2^{\lceil \log_2 n\rceil}-2}{2^{\lceil \log_2 n\rceil}}\right) + \lceil \log_2 n\rceil$$

$$= (2^{\log_2 n - \lceil \log_2 n \rceil})(2 \cdot 2^{\lceil \log_2 n \rceil} - 2) + \lceil \log_2 n \rceil$$

$$= 2 \cdot 2^{\log_2 n} - 2 \cdot 2^{\log_2 n - \lceil \log_2 n \rceil} + \lceil \log_2 n \rceil$$

$$\leq 2 \cdot 2^{\log_2 n} - 2 + \lceil \log_2 n \rceil \text{ since } 2^{\log_2 n - \lceil \log_2 n \rceil} \leq 1$$

$$= 2n - 2 + \lceil \log_2 n \rceil$$

The sort program was run for files of length 3,4,5,6,7 with all processor types executing in one cycle. The resulting elapsed times fit EQN 1 exactly with $k_0=5$, $k_1=19$, and $k_2=13$.

TABLE 3

N	$\lceil \log_2 n \rceil$	$\sum_{i=0}^{\lceil \log_2 n \rceil - 1} \lceil n/2^i \rceil$	T CALCULATED	T OBSERVED
3	2	5	5+38+65=108	108
4	2	6	5+38+78=121	121
5	3	10	5+57+130=192	192
6	3	11	5+57+143=205	205
7	3	13	5+57+169=231	231
9	4	19	5+75+247=328	328

The time taken by this sort is independent of the original order of the records in the file since neither the number of subfiles produced nor the number of comparisons required to merge two files depends on the contents of the records.

SORT - 3 Record File

```
TOTAL PROCESSOR RESOURCE USAGE
    1:#
    2:#
    3:#
    4:##
    5:#
    6:####
    7:####
    8:####
    9:######
   10:####
   11:######
   12:#####
   13:#####
   14:#######
   15:#####
   16:###
   17:####
   18:####
   19:####
   20:####
   21:#####
   22:#######
   23:######
   24:#####
   25:#####
   26:####
   27:###
   28:###
   29:####
   30:######
   31:######
   32:####
   33:###
   34:#####
   35:#######
   36:######
   37:#####
   38:#####
   39:####
   40:###
   41:###
   42:####
   43:######
   44:#####
   45:#####
   46:###
   47:#####
   48:#######
   49:######
   50:#####
   51:#####
   52:####
   53:###
   54:##
   55:##
   56:#
   57:###
   58:###
```

Fig. 26a

```
   59:###
   60:###
   61:####
   62:######
   63:#####
   64:####
   65:####
   66:###
   67:##
   68:##
   69:###
   70:#####
   71:#####
   72:###
   73:##
   74:####
   75:######
   76:#####
   77:####
   78:####
   79:###
   80:##
   81:##
   82:###
   83:#####
   84:#####
   85:###
   86:##
   87:####
   88:######
   89:#####
   90:####
   91:####
   92:###
   93:##
   94:##
   95:###
   96:#####
   97:####
   98:####
   99:##
  100:####
  101:######
  102:#####
  103:####
  104:####
  105:###
  106:##
  107:#
  108:#
  109:
  110:
TOTAL RESOURCE CYCLES USED =      423   % UTILIZATION =       1
AVERAGE RESOURCES USED PER TIME STEP =       4    MAXIMUM =     7
```

SORT - 4 Record File

```
TOTAL PROCESSOR RESOURCE USAGE
    1:#
    2:#
    3:#
    4:##
    5:#
    6:####
    7:####
    8:####
    9:######
   10:####
   11:##########
   12:##########
   13:##########
   14:##############
   15:##########
   16:######
   17:########
   18:########
   19:########
   20:########
   21:##########
   22:##############
   23:############
   24:##########
   25:##########
   26:########
   27:######
   28:######
   29:########
   30:############
   31:###########
   32:#########
   33:######
   34:##########
   35:##############
   36:############
   37:##########
   38:##########
   39:########
   40:######
   41:######
   42:########
   43:############
   44:###########
   45:#########
   46:######
   47:##########
   48:##############
   49:############
   50:##########
   51:##########
   52:########
   53:######
   54:####
   55:####
   56:##
   57:###
   58:###
```

Fig. 26b

```
 59:###
 60:###
 61:####
 62:######
 63:#####
 64:####
 65:####
 66:###
 67:##
 68:##
 69:###
 70:#####
 71:#####
 72:###
 73:##
 74:####
 75:######
 76:#####
 77:####
 78:####
 79:###
 80:##
 81:##
 82:###
 83:#####
 84:####
 85:####
 86:##
 87:####
 88:######
 89:#####
 90:####
 91:####
 92:###
 93:##
 94:##
 95:###
 96:#####
 97:#####
 98:###
 99:##
100:####
101:######
102:#####
103:####
104:####
105:###
106:##
107:##
108:###
109:#####
110:####
111:####
112:##
113:####
114:######
115:#####
116:####
117:####
118:###
```

```
 119:##
 120:#
 121:#
 122:
 123:
TOTAL RESOURCE CYCLES USED =      678   % UTILIZATION =        2
AVERAGE RESOURCES USED PER TIME STEP =      6    MAXIMUM =    14
```

SORT - 6 Record File

```
TOTAL PROCESSOR RESOURCE USAGE
   1:#
   2:#
   3:#
   4:##
   5:#
   6:####
   7:####
   8:####
   9:######
  10:####
  11:##########
  12:##########
  13:##########
  14:##############
  15:##########
  16:##############
  17:############
  18:############
  19:################
  20:############
  21:########
  22:##########
  23:##########
  24:##########
  25:##########
  26:############
  27:################
  28:##############
  29:############
  30:############
  31:##########
  32:########
  33:########
  34:##########
  35:##############
  36:#############
  37:###########
  38:########
  39:############
  40:################
  41:##############
  42:############
  43:############
  44:##########
  45:########
  46:########
  47:##########
  48:##############
  49:#############
  50:###########
  51:########
  52:############
  53:################
  54:##############
  55:############
  56:############
  57:##########
  58:########
```

Fig. 26c

```
 59:######
 60:######
 61:####
 62:########
 63:########
 64:########
 65:########
 66:##########
 67:##############
 68:############
 69:##########
 70:##########
 71:########
 72:######
 73:######
 74:########
 75:############
 76:##########
 77:##########
 78:######
 79:##########
 80:##############
 81:############
 82:##########
 83:##########
 84:########
 85:######
 86:######
 87:########
 88:############
 89:############
 90:########
 91:######
 92:##########
 93:##############
 94:############
 95:##########
 96:##########
 97:########
 98:######
 99:######
100:########
101:############
102:############
103:########
104:######
105:##########
106:##############
107:############
108:##########
109:##########
110:########
111:######
112:####
113:####
114:##
115:###
116:###
117:###
118:###
```

```
119:####
120:######
121:#####
122:####
123:####
124:###
125:##
126:##
127:###
128:#####
129:#####
130:###
131:##
132:####
133:######
134:#####
135:####
136:####
137:###
138:##
139:##
140:###
141:#####
142:####
143:####
144:##
145:####
146:######
147:#####
148:####
149:####
150:###
151:##
152:##
153:###
154:#####
155:####
156:####
157:##
158:####
159:######
160:#####
161:####
162:####
163:###
164:##
165:##
166:###
167:#####
168:#####
169:###
170:##
171:####
172:######
173:#####
174:####
175:####
176:###
177:##
178:##
```

```
 179:###
 180:#####
 181:#####
 182:###
 183:##
 184:####
 185:######
 186:#####
 187:####
 188:####
 189:###
 190:##
 191:##
 192:###
 193:#####
 194:####
 195:####
 196:##
 197:####
 198:######
 199:#####
 200:####
 201:####
 202:###
 203:##
 204:#
 205:#
 206:
 207:
TOTAL RESOURCE CYCLES USED =     1400   % UTILIZATION =        2
AVERAGE RESOURCES USED PER TIME STEP =       7    MAXIMUM =    16
```

SORT - 7 Record File

```
TOTAL PROCESSOR RESOURCE USAGE
    1:#
    2:#
    3:#
    4:##
    5:#
    6:####
    7:####
    8:####
    9:######
   10:####
   11:##########
   12:##########
   13:##########
   14:##############
   15:##########
   16:##################
   17:#################
   18:#################
   19:#######################
   20:#################
   21:###########
   22:##############
   23:##############
   24:##############
   25:##############
   26:#################
   27:#######################
   28:####################
   29:#################
   30:#################
   31:##############
   32:###########
   33:###########
   34:##############
   35:####################
   36:###################
   37:###############
   38:###########
   39:#################
   40:#######################
   41:####################
   42:#################
   43:#################
   44:##############
   45:###########
   46:###########
   47:##############
   48:####################
   49:##################
   50:################
   51:###########
   52:#################
   53:#######################
   54:####################
   55:#################
   56:#################
   57:##############
   58:###########
```

```
 59:########
 60:########
 61:#####
 62:########
 63:########
 64:########
 65:########
 66:##########
 67:##############
 68:############
 69:##########
 70:##########
 71:########
 72:######
 73:######
 74:########
 75:############
 76:###########
 77:#########
 78:######
 79:##########
 80:##############
 81:############
 82:##########
 83:##########
 84:########
 85:######
 86:######
 87:########
 88:############
 89:###########
 90:#########
 91:######
 92:##########
 93:##############
 94:############
 95:##########
 96:##########
 97:########
 98:######
 99:######
100:########
101:############
102:###########
103:#########
104:######
105:##########
106:##############
107:############
108:##########
109:##########
110:########
111:######
112:#####
113:######
114:#######
115:######
116:####
117:###
118:#####
```

```
119:#######
120:######
121:#####
122:#####
123:####
124:###
125:##
126:##
127:#
128:###
129:###
130:###
131:###
132:####
133:######
134:#####
135:####
136:####
137:###
138:##
139:##
140:###
141:#####
142:#####
143:###
144:##
145:####
146:######
147:#####
148:####
149:####
150:###
151:##
152:##
153:###
154:#####
155:#####
156:###
157:##
158:####
159:######
160:#####
161:####
162:####
163:###
164:##
165:##
166:###
167:#####
168:#####
169:###
170:##
171:####
172:######
173:#####
174:####
175:####
176:###
177:##
178:##
```

```
 179:###
 180:#####
 181:####
 182:####
 183:##
 184:####
 185:######
 186:#####
 187:####
 188:####
 189:###
 190:##
 191:##
 192:###
 193:#####
 194:####
 195:####
 196:##
 197:####
 198:######
 199:#####
 200:####
 201:####
 202:###
 203:##
 204:##
 205:###
 206:#####
 207:####
 208:####
 209:##
 210:####
 211:######
 212:#####
 213:####
 214:####
 215:###
 216:##
 217:##
 218:###
 219:#####
 220:#####
 221:###
 222:##
 223:####
 224:######
 225:#####
 226:####
 227:####
 228:###
 229:##
 230:#
 231:#
 232:
 233:
TOTAL RESOURCE CYCLES USED =    1716   % UTILIZATION =       3
AVERAGE RESOURCES USED PER TIME STEP =      7    MAXIMUM =   23
```

SORT - 9 Record File

```
TOTAL PROCESSOR RESOURCE USAGE
    1:#
    2:#
    3:#
    4:##
    5:#
    6:####
    7:####
    8:####
    9:######
   10:####
   11:##########
   12:##########
   13:##########
   14:##############
   15:##########
   16:######################
   17:######################
   18:######################
   19:##############################
   20:######################
   21:##################
   22:####################
   23:####################
   24:######################
   25:####################
   26:#####################
   27:############################
   28:#########################
   29:######################
   30:######################
   31:####################
   32:###################
   33:##################
   34:####################
   35:##########################
   36:#########################
   37:##################
   38:###############
   39:######################
   40:##############################
   41:###########################
   42:######################
   43:#####################
   44:####################
   45:###################
   46:##################
   47:####################
   48:##########################
   49:######################
   50:#####################
   51:###############
   52:######################
   53:##############################
   54:##########################
   55:#######################
   56:#####################
   57:####################
   58:###################
```

Fig. 26e

```
 59:###############
 60:##############
 61:###########
 62:##########
 63:#########
 64:########
 65:########
 66:########
 67:############
 68:###########
 69:##########
 70:##########
 71:##########
 72:###########
 73:##########
 74:##########
 75:############
 76:##########
 77:#########
 78:#######
 79:##########
 80:##############
 81:############
 82:###########
 83:#########
 84:##########
 85:###########
 86:##########
 87:##########
 88:############
 89:###########
 90:########
 91:#######
 92:##########
 93:##############
 94:#############
 95:##########
 96:#########
 97:##########
 98:###########
 99:##########
100:##########
101:############
102:##########
103:#########
104:#######
105:##########
106:##############
107:#############
108:##########
109:#########
110:##########
111:###########
112:##########
113:##########
114:############
115:###########
116:########
117:######
118:########
```

```
119:#########
120:##########
121:#########
122:#########
123:########
124:########
125:#########
126:########
127:######
128:#####
129:####
130:###
131:###
132:####
133:######
134:######
135:####
136:###
137:#####
138:#######
139:######
140:#####
141:#####
142:####
143:###
144:###
145:####
146:######
147:######
148:####
149:###
150:#####
151:#######
152:######
153:#####
154:#####
155:####
156:###
157:###
158:####
159:######
160:#####
161:#####
162:###
163:#####
164:#######
165:######
166:#####
167:#####
168:####
169:###
170:###
171:####
172:######
173:######
174:####
175:###
176:#####
177:#######
178:######
```

```
179:#####
180:#####
181:####
182:###
183:###
184:####
185:######
186:#####
187:#####
188:###
189:#####
190:#######
191:######
192:#####
193:#####
194:####
195:###
196:##
197:##
198:#
199:###
200:###
201:###
202:###
203:####
204:######
205:#####
206:####
207:####
208:###
209:##
210:##
211:###
212:#####
213:#####
214:###
215:##
216:####
217:######
218:#####
219:####
220:####
221:###
222:##
223:##
224:###
225:#####
226:####
227:####
228:##
229:####
230:######
231:#####
232:####
233:####
234:###
235:##
236:##
237:###
238:#####
```

```
239:#####
240:###
241:##
242:####
243:######
244:#####
245:####
246:####
247:###
248:##
249:##
250:###
251:#####
252:#####
253:###
254:##
255:####
256:######
257:#####
258:####
259:####
260:###
261:##
262:##
263:###
264:#####
265:#####
266:###
267:##
268:####
269:######
270:#####
271:####
272:####
273:###
274:##
275:##
276:###
277:#####
278:####
279:####
280:##
281:####
282:######
283:#####
284:####
285:####
286:###
287:##
288:##
289:###
290:#####
291:#####
292:###
293:##
294:####
295:######
296:#####
297:####
298:####
```

```
 299:###
 300:##
 301:##
 302:###
 303:#####
 304:####
 305:####
 306:##
 307:####
 308:######
 309:#####
 310:####
 311:####
 312:###
 313:##
 314:##
 315:###
 316:#####
 317:####
 318:####
 319:##
 320:####
 321:######
 322:#####
 323:####
 324:####
 325:###
 326:##
 327:#
 328:#
 329:
 330:
TOTAL RESOURCE CYCLES USED =     2512   % UTILIZATION =        3
AVERAGE RESOURCES USED PER TIME STEP =       8    MAXIMUM =    30
```

It is interesting to compare the time required for a parallel sort with time which world be required to run the same sort sequentially. At the i^{th} stage of the initial process of splitting the file into subfiles, there are 2^{i-1} files to be split. However, some of these are already of length 1 and thus are not split. To simplify, we consider the case where $n=2^m$. Then

$$\sum_{i=0}^{\log_2 n} 2^{i-1} = \sum_{i=0}^{\log_2 n-1} 2^i = 2^{\log_2 n-1} = 2^m-1 = \underline{n-1} \text{ stages.}$$

At any given merge stage the subfile pairs must be merged sequentially, and the time taken for all these merges is proportional to the sum of the lengths of the merged subfile pairs, i.e., to the length of the original file. Since there are $\log_2 n$ merge stages, the time taken merging is proportional to $n\log_2 n$, i.e.

$$\text{Tsequential} = k_0^1 + k_1^1 (n-1) + k_2^1 n \log_2 n$$

If we assume that the proportionality constants are the same for both sequential and parallel operation we can compare times for files of length 4, 8, and 16.

N	Tpar	Tseq
4	121	166
8	244	450
16	484	1122

It should be noted that the assumption that the proportionality constants are equal for sequential and parallel cases implies either that the sequential machine has a faster cycle time or that the sequential program is coded more effeciently, since the constants k_o, k_1, and k_2 themselves represent considerable concurrent operation.

Sort - Comparison of Relative Processor Speeds

When all processor speeds were equal the time to sort a four record file was 121 cycles. 678 processor cycles were used. When the relative processor speeds were varied in a ratio reflecting the speeds of corresponding operations on existing computers, the same computation took 159 cycles, using a total of 897 processor cycles. In the first case the execution time for all processors was one cycle. In the second, the fastest processor operated at one cycle while others were slower. To obtain a true comparison of the two cases, one ought to set the execution time in the first case to the mean of all the execution times in the second computation. An approximation to this is obtained by averaging the execution times for each node in the graph program (rather than for each node executed). The average will be off by the degree to which the mean execution time of nodes executed repeadedly weighted by number of repititions differs from the mean time for nodes in the graph.

The mean execution time of nodes in the sort program (based on processor speeds used in the second case) is 1.275 cycles. It is not necessary to rerun the program with all processor speeds equal to 1.275, since the same effect can be achieved by scaling the case of all processor times = 1 cycle. The equal processor speed case then gives a time of 121 x 1.275 = 154 cycles and a total number of processor cycles used of 678 x 1.275 = 860 cycles.

Summary

To understand the effects of relative processor speeds we must compare cases where the relative speeds of different operations vary to cases where they are all the same. For a precise comparison we should set the processor execution time in the second case to the weighted mean of the execution times in the first case

$$\tau = \left(\frac{\sum_{n_i G} \gamma_{ni} \cdot \tau_{ni}}{\sum_{n_i \varepsilon G} \gamma_{n_i}} \right)$$

where τ_{ni} = execution time of node ni

γn_i = # times node n_i is executed during the computation

To simplify we make the assumption that the above mean is well approximated by the unweighted mean

$$\tau^1 = \left(\frac{\sum_{n_i \epsilon G} \tau n_i}{N} \right)$$

where N = number of nodes in graph program

Certain nodes are "dummy" nodes (i.e. they never execute) always nodes with time = 1 we exclude them in calculating τ^1 . (in MERGE nodes 1, 2, e.g) Then for the sort program

$$\tau^1 = \frac{47}{37} = 1.297$$

Using this to scale a run where $\tau_{n_i} = 1$ all $n_i \epsilon G$ we have

	Constant Speed	Varied Speed
Time	121 x 1.297 = 157 cycles	159 cycles
Total Cycles	678 x 1.297 = 879 cycles	897 cycles

This indicates that relative processor speeds are not too important.

As a further experiment a new set of relative processor speeds (σn_i) was chosen so that the unweighted mean would be the same as for the first set, i.e. such that

$$\sum_{n_i \epsilon G} \tau_{n_i} = \sum_{n_i \epsilon G} \sigma_{n_i}$$

However, the set σ_{n_i} was such that the variance was slightly larger i.e.

11.81 for $\{\sigma_{n_i}\}$

7.41 for $\{\tau_{n_i}\}$

This was done by reducing the time for $\wedge$, length from two cycles to one and increasing $\vee$ $\neg$ to three cycles to compensate for the possibility that $\vee$ and $\neg$ (route select .7 and route select .11) were executed more often than $\wedge$ and length, the change was reversed, i.e.,

$\wedge$, length = 3

$\vee$, $\neg$ = 1

The variance is 11.81 for this case also.

For the first case the time was 156 cycles and the total processor cycles used was 893.

In the second case the computation took 162 cycles using 901 processor cycles

TABLE 4

TIME	PROCESSOR CYCLES	
157	879	Constant speed
159	897	Variable speed
156	893	Var. speed - higher var. I
162	901	Var. speed - higher var. II
162 - 156 =	6/159 = 3.77 percent	901 - 879 = 22/990 = 2.22 percent
183	992	Var = 49.60
183	992	Var = 49.60
183	992	Var = 49.60
183 - 156 = 27/169 = 17 percent		992 - 879 = 113/936 = 12.1 percent

SORT - 4 Record File   Variable Processor Speeds

```
TOTAL PROCESSOR RESOURCE USAGE
    1:#
    2:#
    3:#
    4:#
    5:#
    6:##
    7:#
    8:####
    9:####
   10:####
   11:####
   12:####
   13:######
   14:####
   15:##########
   16:##########
   17:##########
   18:##########
   19:##########
   20:##############
   21:##########
   22:######
   23:########
   24:########
   25:########
   26:########
   27:########
   28:##########
   29:##########
   30:##############
   31:############
   32:############
   33:##########
   34:##########
   35:########
   36:########
   37:######
   38:######
   39:########
   40:############
   41:###########
   42:#########
   43:########
   44:######
   45:##########
   46:##########
   47:##############
   48:############
   49:############
   50:##########
   51:##########
   52:########
   53:########
   54:######
   55:######
   56:########
   57:############
   58:###########
```

Fig. 27a

```
 59:#########
 60:########
 61:######
 62:##########
 63:##########
 64:##############
 65:############
 66:############
 67:##########
 68:##########
 69:########
 70:########
 71:######
 72:####
 73:####
 74:##
 75:###
 76:###
 77:###
 78:###
 79:###
 80:####
 81:####
 82:######
 83:#####
 84:#####
 85:####
 86:####
 87:###
 88:###
 89:##
 90:##
 91:###
 92:#####
 93:#####
 94:###
 95:###
 96:##
 97:####
 98:####
 99:######
100:#####
101:#####
102:####
103:####
104:###
105:###
106:##
107:##
108:###
109:#####
110:####
111:####
112:###
113:##
114:####
115:####
116:######
117:#####
118:#####
```

```
 119:####
 120:####
 121:###
 122:###
 123:##
 124:##
 125:###
 126:#####
 127:#####
 128:###
 129:###
 130:##
 131:####
 132:####
 133:######
 134:#####
 135:#####
 136:####
 137:####
 138:###
 139:###
 140:##
 141:##
 142:###
 143:#####
 144:####
 145:####
 146:###
 147:##
 148:####
 149:####
 150:######
 151:#####
 152:#####
 153:####
 154:####
 155:###
 156:###
 157:##
 158:#
 159:#
 160:
 161:
TOTAL RESOURCE CYCLES USED =      897   % UTILIZATION =        2
AVERAGE RESOURCES USED PER TIME STEP =       6    MAXIMUM =   14
```

SORT - 4 Record File   Variance I

```
TOTAL PROCESSOR RESOURCE USAGE
    1:#
    2:#
    3:#
    4:#
    5:##
    6:#
    7:####
    8:####
    9:####
   10:####
   11:######
   12:####
   13:##########
   14:##########
   15:##########
   16:##########
   17:##############
   18:##########
   19:######
   20:########
   21:########
   22:########
   23:########
   24:########
   25:##########
   26:##########
   27:##############
   28:############
   29:############
   30:##########
   31:##########
   32:##########
   33:########
   34:######
   35:######
   36:########
   37:############
   38:###########
   39:#########
   40:########
   41:######
   42:##########
   43:##########
   44:##############
   45:############
   46:############
   47:##########
   48:##########
   49:##########
   50:########
   51:######
   52:######
   53:########
   54:############
   55:###########
   56:#########
   57:########
   58:######
```

Fig. 27b

```
 59:##########
 60:##########
 61:##############
 62:############
 63:############
 64:##########
 65:##########
 66:##########
 67:########
 68:######
 69:####
 70:####
 71:##
 72:###
 73:###
 74:###
 75:###
 76:###
 77:####
 78:####
 79:######
 80:#####
 81:#####
 82:####
 83:####
 84:####
 85:###
 86:##
 87:##
 88:###
 89:#####
 90:#####
 91:###
 92:###
 93:##
 94:####
 95:####
 96:######
 97:#####
 98:#####
 99:####
100:####
101:####
102:###
103:##
104:##
105:###
106:#####
107:####
108:####
109:###
110:##
111:####
112:####
113:######
114:#####
115:#####
116:####
117:####
118:####
```

```
 119:###
 120:##
 121:##
 122:###
 123:#####
 124:#####
 125:###
 126:###
 127:##
 128:####
 129:####
 130:######
 131:#####
 132:#####
 133:####
 134:####
 135:####
 136:###
 137:##
 138:##
 139:###
 140:#####
 141:####
 142:####
 143:###
 144:##
 145:####
 146:####
 147:######
 148:#####
 149:#####
 150:####
 151:####
 152:####
 153:###
 154:##
 155:#
 156:#
 157:
 158:
TOTAL RESOURCE CYCLES USED =       893   % UTILIZATION =          2
AVERAGE RESOURCES USED PER TIME STEP =       6    MAXIMUM =    14
```

SORT - 4 Record File   Variance II

```
TOTAL PROCESSCR RESOURCE USAGE
    1:#
    2:#
    3:#
    4:#
    5:#
    6:#
    7:##
    8:#
    9:####
   10:####
   11:####
   12:####
   13:####
   14:####
   15:######
   16:####
   17:##########
   18:##########
   19:##########
   20:##########
   21:##########
   22:##########
   23:##############
   24:##########
   25:######
   26:########
   27:########
   28:########
   29:########
   30:########
   31:##########
   32:##########
   33:##############
   34:############
   35:############
   36:##########
   37:########
   38:########
   39:########
   40:######
   41:######
   42:########
   43:############
   44:###########
   45:#########
   46:########
   47:######
   48:##########
   49:##########
   50:##############
   51:############
   52:############
   53:##########
   54:########
   55:########
   56:########
   57:######
   58:######
```

Fig. 27c

```
 59:########
 60:############
 61:###########
 62:#########
 63:########
 64:######
 65:##########
 66:##########
 67:##############
 68:############
 69:############
 70:##########
 71:########
 72:########
 73:########
 74:######
 75:####
 76:####
 77:##
 78:###
 79:###
 80:###
 81:###
 82:###
 83:####
 84:####
 85:######
 86:#####
 87:#####
 88:####
 89:###
 90:###
 91:###
 92:##
 93:##
 94:###
 95:#####
 96:#####
 97:###
 98:###
 99:##
100:####
101:####
102:######
103:#####
104:#####
105:####
106:###
107:###
108:###
109:##
110:##
111:###
112:#####
113:####
114:####
115:###
116:##
117:####
118:####
```

```
 119:######
 120:#####
 121:#####
 122:####
 123:###
 124:###
 125:###
 126:##
 127:##
 128:###
 129:#####
 130:#####
 131:###
 132:###
 133:##
 134:####
 135:####
 136:######
 137:#####
 138:#####
 139:####
 140:###
 141:###
 142:###
 143:##
 144:##
 145:###
 146:#####
 147:####
 148:####
 149:###
 150:##
 151:####
 152:####
 153:######
 154:#####
 155:#####
 156:####
 157:###
 158:###
 159:###
 160:##
 161:#
 162:#
 163:
 164:
TOTAL RESOURCE CYCLES USED =     901   % UTILIZATION =        2
AVERAGE RESOURCES USED PER TIME STEP =        5    MAXIMUM =    14
```

SORT - 4 Record File, Processor Speed Variance = 49

```
TOTAL PROCESSOR RESOURCE USAGE
   1:#
   2:#
   3:#
   4:#
   5:#
   6:##
   7:#
   8:####
   9:####
  10:####
  11:####
  12:####
  13:######
  14:####
  15:##########
  16:##########
  17:##########
  18:##########
  19:##########
  20:##############
  21:##########
  22:######
  23:########
  24:########
  25:########
  26:########
  27:########
  28:##########
  29:##############
  30:############
  31:############
  32:##########
  33:########
  34:########
  35:########
  36:########
  37:########
  38:########
  39:########
  40:######
  41:######
  42:########
  43:############
  44:###########
  45:#########
  46:########
  47:######
  48:##########
  49:##############
  50:############
  51:############
  52:##########
  53:########
  54:########
  55:########
  56:########
  57:########
  58:########
```

Fig. 27d

```
  59:########
  60:######
  61:######
  62:########
  63:############
  64:###########
  65:#########
  66:########
  67:######
  68:##########
  69:##############
  70:############
  71:############
  72:##########
  73:########
  74:########
  75:########
  76:########
  77:########
  78:########
  79:########
  80:######
  81:####
  82:####
  83:##
  84:###
  85:###
  86:###
  87:###
  88:###
  89:####
  90:######
  91:#####
  92:#####
  93:####
  94:###
  95:###
  96:###
  97:###
  98:###
  99:###
 100:###
 101:##
 102:##
 103:###
 104:#####
 105:#####
 106:###
 107:###
 108:##
 109:####
 110:######
 111:#####
 112:#####
 113:####
 114:###
 115:###
 116:###
 117:###
 118:###
```

```
119:###
120:###
121:##
122:##
123:###
124:#####
125:####
126:####
127:###
128:##
129:####
130:######
131:#####
132:#####
133:####
134:###
135:###
136:###
137:###
138:###
139:###
140:###
141:##
142:##
143:###
144:#####
145:#####
146:###
147:###
148:##
149:####
150:######
151:#####
152:#####
153:####
154:###
155:###
156:###
157:###
158:###
159:###
160:###
161:##
162:##
163:###
164:#####
165:####
166:####
167:###
168:##
169:####
170:######
171:#####
172:#####
173:####
174:###
175:###
176:###
177:###
178:###
```

```
 179:###
 180:###
 181:##
 182:#
 183:#
 184:
 185:
TOTAL RESOURCE CYCLES USED =     992   % UTILIZATION =       2
AVERAGE RESOURCES USED PER TIME STEP =      5    MAXIMUM =   14
```

MATRIX MULTIPLICATION PROGRAM

The matrix multiply program consists of eight graph procedures. The program is written as a procedure to be called from another graph program. In the simulations which were run, a dummy procedure, whose only active node was the matrix multiply procedure, represented the other program.

The basic algorithm used is to split off each row of the first matrix and to take the scalar product of this row with each column of the second m matrix. Thus, if we are multiplying a m by l matrix by a l by n matrix, each row of the first matrix must enter into a scalar product with n columns of the second. Furthermore, each column enters into a scalar product m times. The procedure was written to execute with the maximum amount of parallelism at the expense of storage for row and column vectors. Hence, the row vectors are each copied n times rather than being recycled after each multiplication. The same is done for column vectors, they are copied rather than looped around the graph.

The row vectors are split off the first matrix by the <u>first-rest</u> node. The <u>null test</u> and <u>not</u> nodes provide a boolean which causes a copy of the second matrix to be made for each row except the last (since the <u>rest</u> of the matrix is null for the last row). In order to provide m rather than m-1 copies of the second matrix, the edge linking node 4 to node 9 is initialized to <u>true</u>. The value <u>true</u> thus appears m times on this edge. Each copy of the second matrix is converted from row form to column form by the procedure COLS. At the same time n copies of the corresponding row vector are produced by the procedure N COPIES. Since the n rows and columns appear on the input edges to the scalar product procedure at the same time, the n x l multiplications of the scalar product can be done in parallel. For n x n matrices then, the number of operations per step is proportional

MATRIX MULTIPLY (MAIN PROCEDURE)

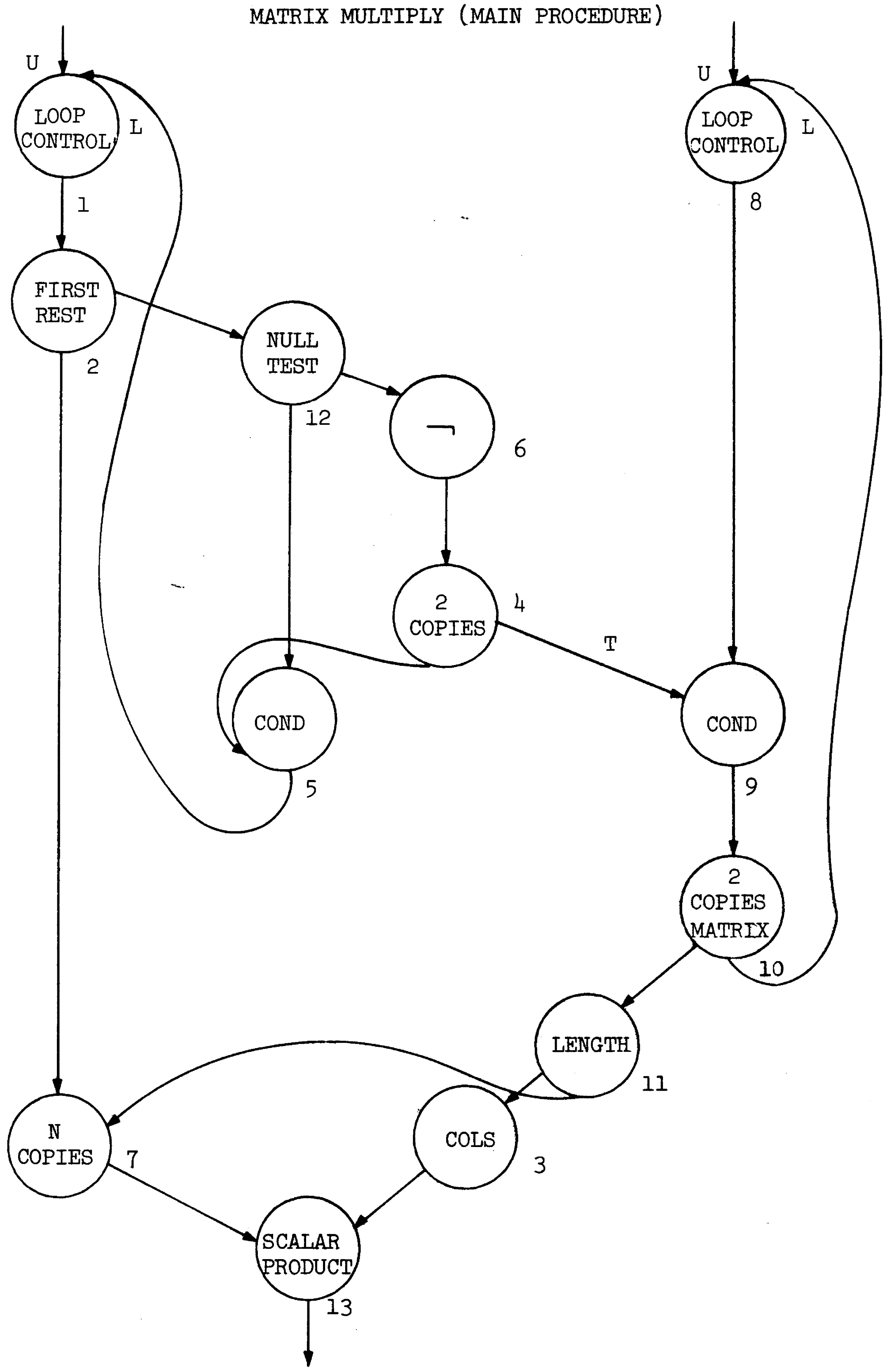


Fig. 28

to n^2, while the time to execute the procedure is proportional to n, or in general, to the number of rows in the first matrix.

Since both COLS and N COPIES bracket their outputs, the inputs to SCALAR PRODUCT are both matrices. These are unbracketed into their constituent vectors by SCALAR PRODUCT which then uses two subprocedures to compute the scalar product of each pair of vectors. SPA unbrackets each vector and multiplies the elements of each pair together. The output is bracketed to produce a vector whose elements are the products of the elements of the input vectors. The elements of this vector are summed by SPB i.e., SPA produces the vector $(a_{i1}, b_{1j}, a_{i2}, b_{2j}, \ldots, a_{i\ell} b_{\ell j})$ and SPB produces the scalar $\sum_{k=1}^{\ell} a_{ik} b_{kj}$ from this vector.

SCALAR PRODUCT invokes n copies of SPA simultaneously, once for each vector pair whose scalar product is to be computed. Each copy of SPA performs its ℓ multiplications in parallel. Thus, for an n by n matrix, n^2 multiplications are performed in parallel.

The procedure COLS turns a matrix stored in row form into a matrix of columns. The input is an m by n row matrix. The subprocedure COLS 1 unbrackets the matrix to form m row vectors. It then splits off the first element of each row vector and puts it on the first output edge. The remainder of each vector is put on the second output edge. Bracketing of the outputs produces a vector of length m on the first edge and a m by n-1 matrix on the second edge. The matrix is recycled through COLS1 by COLS until the last element is taken from each row vector. This results in no output on the second edge of COLS1, and thus terminates COLS with n column vectors of length m on its output edge. Bracketing of these vectors produces an n by m matrix of columns.

MATRIX MULTIPLY CALLING PROCEDURE

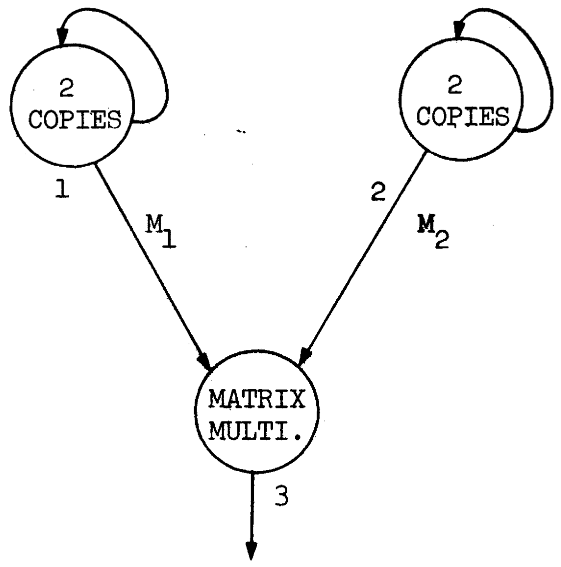


Fig. 29

SCALAR PRODUCT PROCEDURE

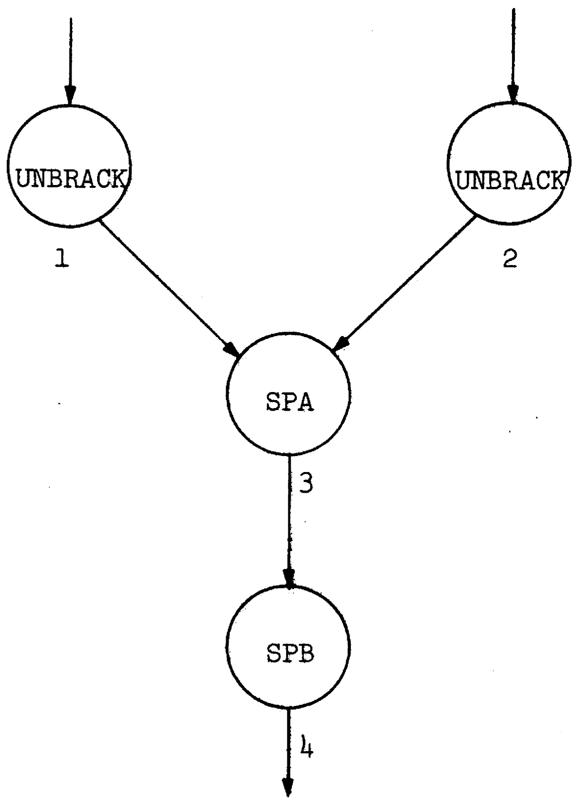


Fig. 30

PROCEDURE SPA (FIRST HALF OF SCALAR PRODUCT)

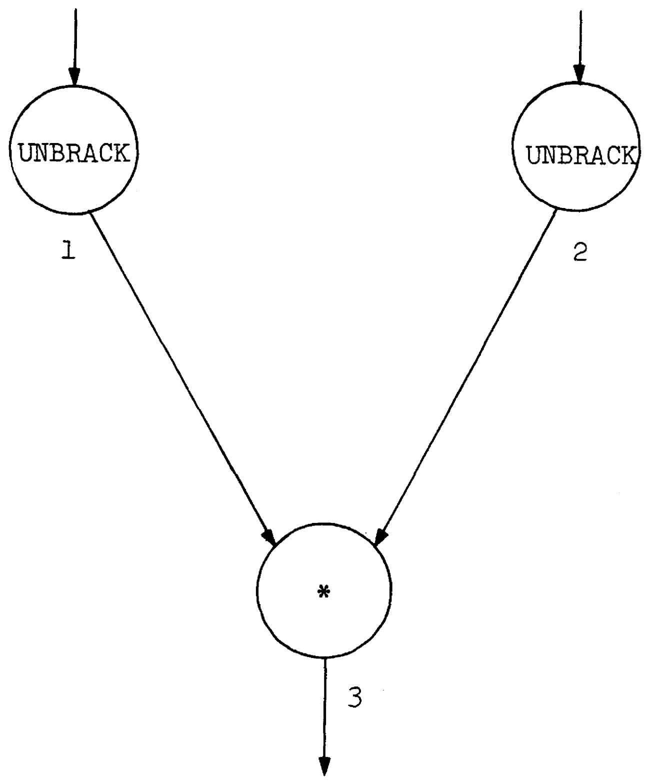


Fig. 31a

PROCEDURE SPB (SCALAR PRODUCT SUMMATION)

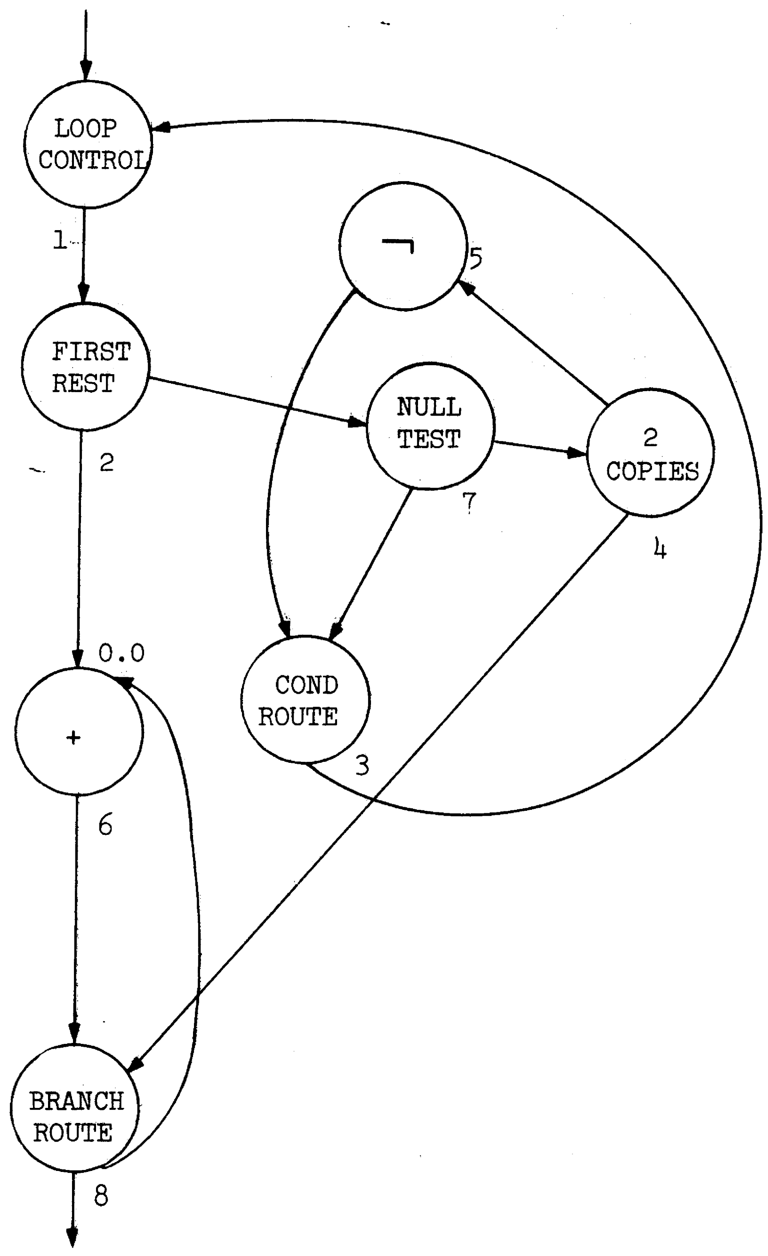


Fig. 31b

Since the first element can be split off each row vector in parallel, the execution time for COLS depends only on the number of invocations of COLS1 and thus proportional to m, the column length.

The procedure N COPIES produces n copies of a vector, where n is a parameter to the procedure. The length of time taken for its execution is directly proportional to n.

The procedure TWO COPIES MATRIX is necessary since use of the primitive node for copying a vector on a matrix would simply produce two copies of the pointer vector whose elements point to the row vector of the matrix and would not duplicate the rows themselves. Since the row vectors are duplicated in parallel, the procedure takes a fixed time independent of the size of the matrix. (Provided that the time to execute the primitive node for two copies vector is independent of vector size).

Simulation Results

The matrix multiply program was run on n by n matrices ranging in size from 2 by 2 to 6 by 6. There are n^3 multiplications required, and the program does n^2 of them at a time. This can be seen very dramatically in the figures 36-38 which show multiply processor usage for 2x2, 3x3, and 4x4 matrices. Multiplication executes in one cycle so there are exactly n cycles during which multiplication occurs.

The total processor usage for the matrices on which the program was run is shown in figures 39-43. In these runs the execution time for all processors is one cycle. As can be seen, the time required for the program is proportional to n, while the amount of computation per cycle increases approximately as n^2. The results of these runs are shown in Table 5. If t is the time required for the computation, then $t=k_1n+k$

PROCEDURE: COLS

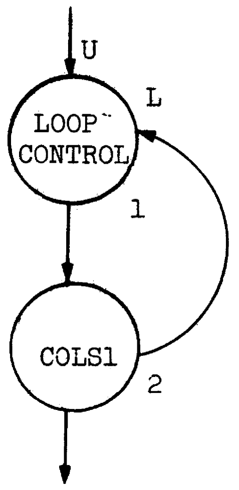


Fig. 32

PROCEDURE: COLS 1

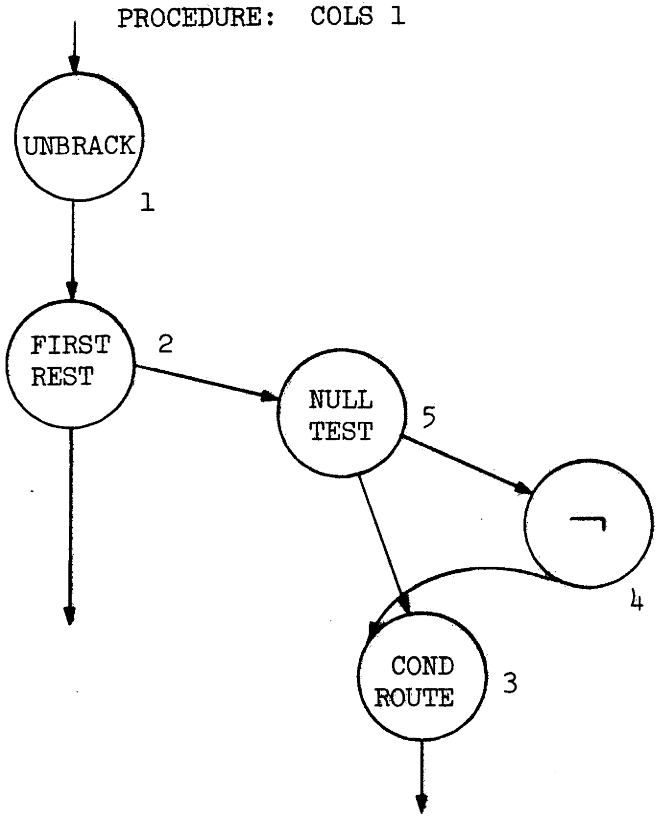


Fig. 32a

PROCEDURE: TWO COPIES OF A MATRIX

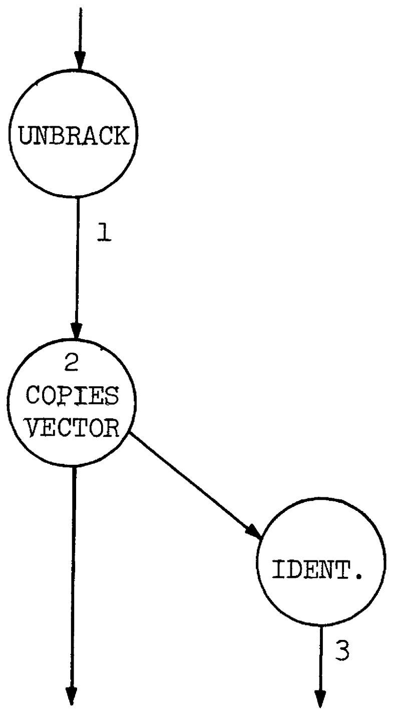


Fig. 33

PROCEDURE: N COPIES (OF A VECTOR)

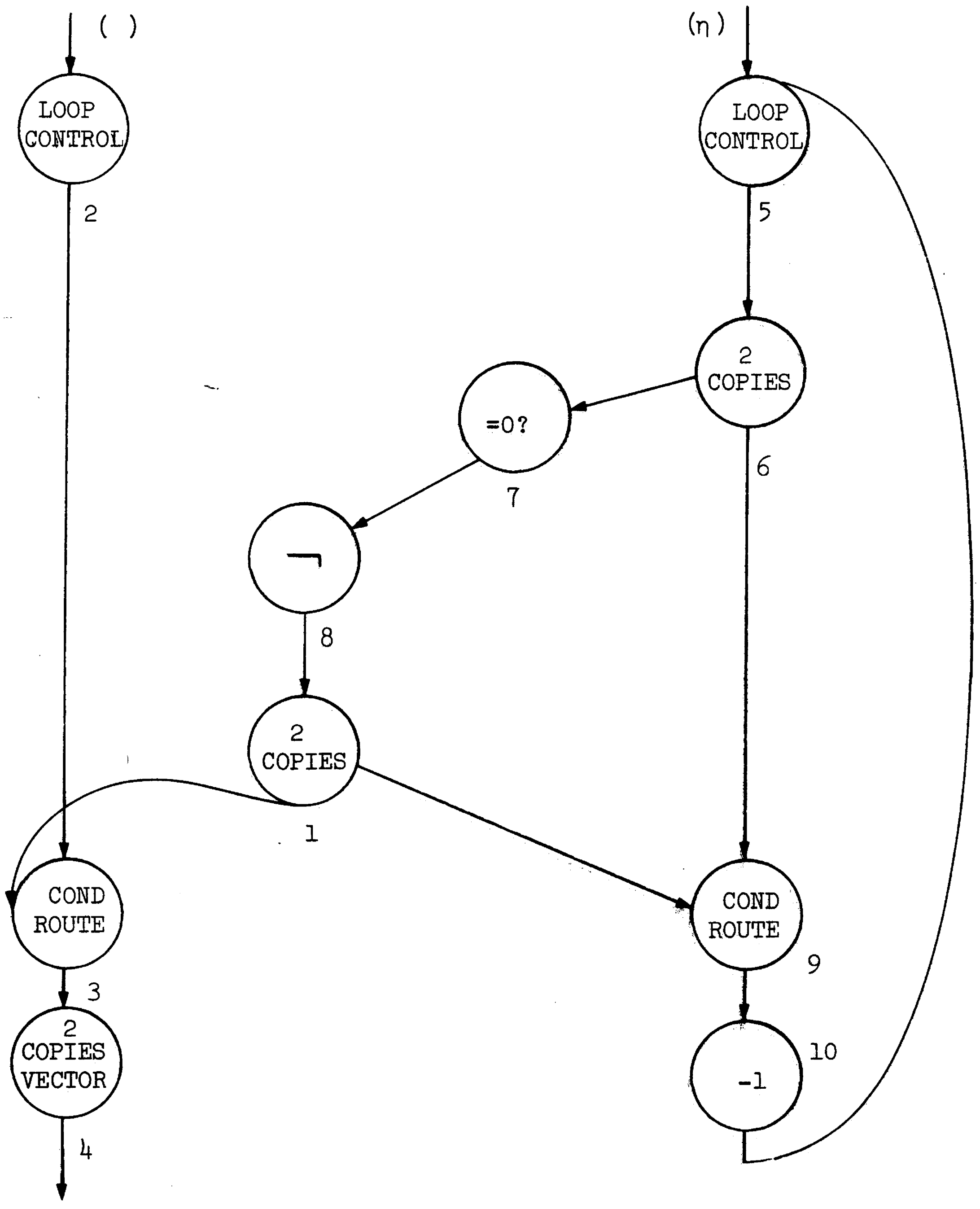


Fig. 34

From the times required we have $k_1=19, k_0=14$, so that the time required to multiply two n by n matrices is given by

(1) $t = 19n + 14$ cycles

The program was run on the same matrices with a four cycle multiplication time and all other processors executing in one cycle. The multiplier and total processor use for the 3x3 matrix is shown in figures 44 and 45. The effect of four cycle multiplication on all the matrix sizes is summarized in Table 5. In this case, we can calculate the new k_1 and k_0, and we get

$t = 19n + 17$ cycles

The value of k_1 is unchanged because the n multiplication steps are independent, i.e., the initiation of the second set of n^2 multiplications does not depend on the termination of the first set.

In an earlier version of the matrix multiplication program, TWO COPIES MATRIX used a loop control node rather than an identity node. That version of the procedure is shown in the following graph. Since loop control is an s-node, only one copy of the node can execute at a time, so that the execution time for the procedure was proportional to the number of rows in the matrix being copied. And since this procedure is in a loop whose execution time is proportional to n, the execution time for the earlier version of the program was proportional to n^2. The execution times were:

n	t	Δt	$\Delta^2 t$
2	54		
3	77	23	
4	102	25	2
5	129	27	2
6	158	29	2

which give the equation

(2) $t = n^2 + 19n + 14$

Both identity and loop control executed in one cycle; the only difference was that the first node could execute in parallel. Equations (1) and (2) illustrate the kind of major differences in program behavior which are brought about by essentially trivial programming changes.

PROCEDURE: TWO COPIES OF A MATRIX
(SEQUENTIALIZED VERSION)

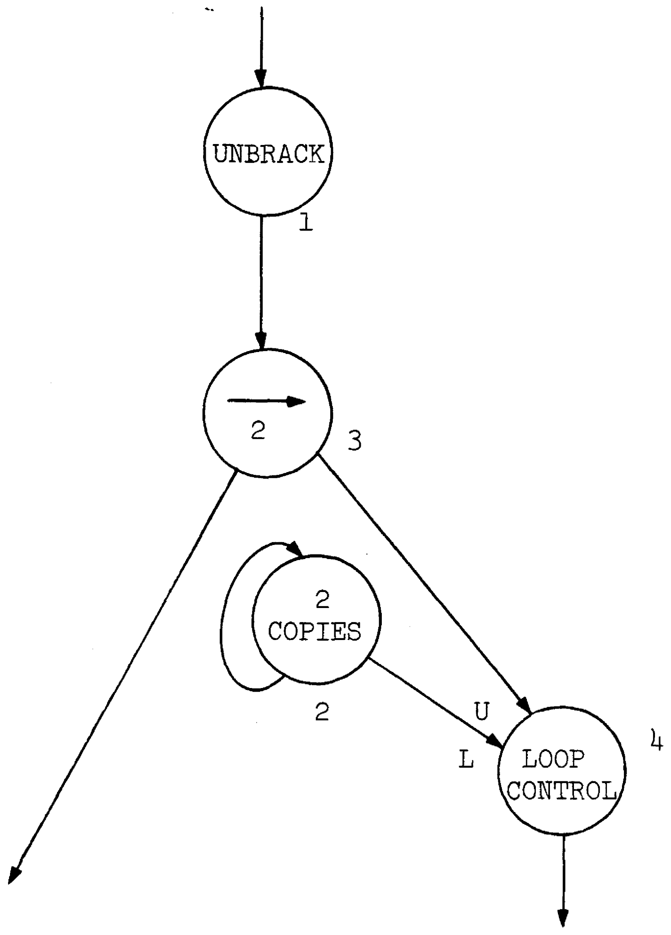


Fig. 35

```
RESOURCE USAGE OF TYPE      MATRIX MULTIPLICATION
   1:                       2 X 2 MATRICES MULTIPLIER USAGE
   2:
   3:
   4:
   5:
   6:
   7:
   8:
   9:
  10:
  11:
  12:
  13:
  14:
  15:
  16:
  17:
  18:
  19:
  20:
  21:
  22:
  23:
  24:
  25:
  26:
  27:
  28:
  29:
  30:
  31:####
  32:
  33:
  34:
  35:
  36:####
  37:
  38:
  39:
  40:
  41:
  42:
  43:
  44:
  45:
  46:
  47:
  48:
  49:
  50:
  51:
  52:
  53:
  54:
TOTAL RESOURCE CYCLES USED =        8   % UTILIZATION =        1
AVERAGE RESOURCES USED PER TIME STEP =        0    MAXIMUM =     4
```

```
RESOURCE USAGE OF TYPE          MULTIPLY PROCESSOR USE
   1:                           3 X 3 MATRIX MULTIPLY
   2:
   3:
   4:
   5:
   6:
   7:
   8:
   9:
  10:
  11:
  12:
  13:
  14:
  15:
  16:
  17:
  18:
  19:
  20:
  21:
  22:
  23:
  24:
  25:
  26:
  27:
  28:
  29:
  30:
  31:
  32:
  33:
  34:
  35:
  36:
  37:
  38:#########
  39:
  40:
  41:
  42:
  43:#########
  44:
  45:
  46:
  47:
  48:
  49:#########
  50:
  51:
  52:
  53:
  54:
  55:
  56:
  57:
  58:
     59:
     60:
     61:
     62:
     63:
     64:
     65:
     66:
     67:
     68:
     69:
     70:
     71:
     72:
     73:
  TOTAL RESOURCE CYCLES USED =       27   % UTILIZATION =        5
  AVERAGE RESOURCES USED PER TIME STEP =       0    MAXIMUM =    9
```

```
RESOURCE USAGE OF TYPE            MULTIPLY PROCESSOR USE
    1:                            4 X 4 MATRIX MULTIPLY
    2:
    3:
    4:
    5:
    6:
    7:
    8:
    9:
   10:
   11:
   12:
   13:
   14:
   15:
   16:
   17:
   18:
   19:
   20:
   21:
   22:
   23:
   24:
   25:
   26:
   27:
   28:
   29:
   30:
   31:
   32:
   33:
   34:
   35:
   36:
   37:
   38:
   39:
   40:
   41:
   42:
   43:
   44:
   45:################
   46:
   47:
   48:
   49:
   50:################
   51:
   52:
   53:
   54:
   55:
   56:################
   57:
   58:
```

```
   59:
   60:
   61:
   62:################
   63:
   64:
   65:
   66:
   67:
   68:
   69:
   70:
   71:
   72:
   73:
   74:
   75:
   76:
   77:
   78:
   79:
   80:
   81:
   82:
   83:
   84:
   85:
   86:
   87:
   88:
   89:
   90:
   91:
   92:
TOTAL RESOURCE CYCLES USED =       64   % UTILIZATION =       3
AVERAGE RESOURCES USED PER TIME STEP =       1     MAXIMUM =   16
```

```
TOTAL PROCESSOR RESOURCE USAGE      MATRIX MULTIPLICATION - 2 X 2 MATRICES
   1:###                            ALL PROCESSORS EXECUTE IN ONE CYCLE
   2:###
   3:####
   4:#####
   5:#####
   6:###
   7:####
   8:########
   9:#########
  10:###########
  11:###########
  12:#########
  13:##########
  14:############
  15:###########
  16:############
  17:#############
  18:#############
  19:##############
  20:#############
  21:###########
  22:##########
  23:#########
  24:#########
  25:#########
  26:##########
  27:#########
  28:######
  29:######
  30:##########
  31:##########
  32:######
  33:#########
  34:##########
  35:###############
  36:#############
  37:###########
  38:###########
  39:###########
  40:#############
  41:#############
  42:#############
  43:#############
  44:###########
  45:#########
  46:#########
  47:######
  48:########
  49:######
  50:####
  51:##
  52:#
  53:
  54:
TOTAL RESOURCE CYCLES USED =      466   % UTILIZATION =        5
AVERAGE RESOURCES USED PER TIME STEP =       9    MAXIMUM =    15
```

```
TOTAL PRCCESSCR RESOURCE USAGE
   1:###
   2:###
   3:####
   4:######
   5:######
   6:###
   7:####
   8:########
   9:#########
  10:#############
  11:#############
  12:##########
  13:############
  14:#############
  15:##############
  16:##################
  17:####################
  18:#################
  19:##################
  20:####################
  21:################
  22:##################
  23:####################
  24:######################
  25:#######################
  26:#####################
  27:###################
  28:################
  29:#################
  30:#################
  31:#################
  32:##################
  33:################
  34:##############
  35:###########
  36:#############
  37:####################
  38:#######################
  39:###############
  40:################
  41:################
  42:#######################
  43:#######################
  44:#################
  45:#################
  46:##################
  47:######################
  48:############################
  49:###############################
  50:######################
  51:######################
  52:######################
  53:############################
  54:#########################
  55:############################
  56:#########################
  57:######################
  58:###################
```

3 X 3 MATRIX MULTIPLICATION

ALL PROCESSORS EXECUTE IN ONE CYCLE

```
   59:######################
   60:###############
   61:#####################
   62:###############
   63:############
   64:#########
   65:###########
   66:########
   67:###########
   68:########
   69:#####
   70:##
   71:#
   72:
   73:
TOTAL RESOURCE CYCLES USED =     1114    % UTILIZATION =          7
AVERAGE RESOURCES USED PER TIME STEP =      15     MAXIMUM =    31
```

4 X 4 MATRIX MULTIPLICATION

ALL PROCESSORS EXECUTE IN ONE CYCLE

```
TOTAL PROCESSOR RESOURCE USAGE
    1:###
    2:###
    3:####
    4:#######
    5:#######
    6:###
    7:####
    8:########
    9:#########
   10:###############
   11:###############
   12:###########
   13:#############
   14:#############
   15:##############
   16:####################
   17:#######################
   18:###################
   19:#####################
   20:######################
   21:###################
   22:#########################
   23:############################
   24:###########################
   25:############################
   26:#############################
   27:#########################
   28:########################
   29:###########################
   30:##############################
   31:##################################
   32:###############################
   33:############################
   34:#########################
   35:########################
   36:##########################
   37:###########################
   38:############################
   39:#########################
   40:######################
   41:###################
   42:###################
   43:######################
   44:################################
   45:#########################################
   46:######################
   47:#######################
   48:#######################
   49:#####################################
   50:#########################################
   51:#############################
   52:##############################
   53:###########################
   54:###############################
   55:######################################
   56:##############################################
   57:##############################
   58:##############################
```

```
    59:###############################
    60:########################################
    61:#############################################
    62:#########################################################
    63:#####################################
    64:#####################################
    65:#####################################
    66:#################################################
    67:#########################################
    68:#################################################
    69:#########################################
    70:#####################################
    71:#################################
    72:#########################################
    73:############################
    74:########################################
    75:############################
    76:########################
    77:####################
    78:###########################
    79:###################
    80:###########################
    81:###################
    82:###############
    83:###########
    84:##############
    85:##########
    86:##############
    87:##########
    88:######
    89:##
    90:#
    91:
    92:
TOTAL RESOURCE CYCLES USED =     2196   % UTILIZATION =      14
AVERAGE RESOURCES USED PER TIME STEP =      24    MAXIMUM =   57
```

```
TOTAL PROCESSOR RESOURCE USAGE                5 X 5 MATRIX MULTIPLICATION
    1:###                                     ALL PROCESSORS EXECUTE IN ONE CYCLE
    2:###
    3:####
    4:########
    5:########
    6:###
    7:####
    8:########
    9:#########
   10:#################
   11:#################
   12:############
   13:##############
   14:#############
   15:##############
   16:######################
   17:##########################
   18:#####################
   19:#######################
   20:#######################
   21:###################
   22:###########################
   23:###############################
   24:##############################
   25:################################
   26:################################
   27:#############################
   28:################################
   29:####################################
   30:###################################
   31:########################################
   32:########################################
   33:###################################
   34:###################################
   35:##################################
   36:######################################
   37:###########################################
   38:###########################################
   39:#######################################
   40:###################################
   41:###################################
   42:##################################
   43:#####################################
   44:########################################
   45:####################################
   46:################################
   47:############################
   48:############################
   49:#############################
   50:#################################
   51:###############################################
   52:##########################################################
   53:#############################
   54:##############################
   55:##################################
   56:#####################################################
   57:###############################################################
   58:############################################
```

```
 59:########################################
 60:####################################
 61:#########################################
 62:#######################################################
 63:#######################################################################
 64:#############################################
 65:##############################################
 66:##########################################
 67:####################################################
 68:#########################################################
 69:#######################################################################
 70:###############################################
 71:###############################################
 72:################################################
 73:################################################################
 74:##################################################################
 75:#####################################################################
 76:########################################################
 77:########################################################
 78:########################################################
 79:#####################################################################
 80:#############################################################
 81:#####################################################################
 82:#############################################################
 83:########################################################
 84:###################################################
 85:##################################################################
 86:#############################################
 87:#################################################################
 88:#############################################
 89:########################################
 90:###################################
 91:#################################################
 92:##################################
 93:#################################################
 94:##################################
 95:#############################
 96:########################
 97:#################################
 98:#######################
 99:#################################
100:#######################
101:##################
102:#############
103:#################
104:############
105:#################
106:############
107:#######
108:##
109:#
110:
111:
TOTAL RESOURCE CYCLES USED =     3826    % UTILIZATION =       11
AVERAGE RESOURCES USED PER TIME STEP =      34    MAXIMUM =    91
```

```
TOTAL PROCESSOR RESOURCE USAGE                    6 X 6 MATRIX MULTIPLICATION
    1:###
    2:###
    3:####
    4:#########
    5:####
    6:####
    7:####
    8:####
    9:####
   10:####
   11:###
   12:####
   13:########
   14:#########
   15:###################
   16:##############
   17:##############
   18:###############
   19:#########
   20:#########
   21:#########
   22:#############
   23:##############
   24:##################
   25:####################
   26:########################
   27:###################
   28:###################
   29:#########################
   30:###################
   31:###################
   32:####################
   33:##################
   34:###################
   35:#######################
   36:##############################
   37:#################################
   38:############################
   39:#############################
   40:#############################
   41:#######################
   42:#######################
   43:#############################
   44:###########################
   45:############################
   46:#################################
   47:##################################
   48:######################################
   49:#################################
   50:#######################################
   51:#######################################
   52:#################################
   53:##################################
   54:##################################
   55:###############################
   56:##############################
   57:###################################
   58:####################################
```

```
 59:########################################
 60:####################################
 61:########################################
 62:##########################################
 63:####################################################
 64:####################################################################
 65:#########################################
 66:############################################
 67:###########################################
 68:######################################################
 69:###############################################
 70:####################################################
 71:###############################################
 72:######################################################
 73:####################################################################
 74:####################################################################
 75:####################################################
 76:################################################################
 77:########################################################
 78:############################################################
 79:#######################################################
 80:###################################################################
 81:#################################################
 82:########################################################
 83:#########################################################
 84:####################################################################
 85:####################################################################
 86:####################################################################
 87:##################################################################
 88:#################################################################
 89:###############################################################
 90:################################################################
 91:###############################################################
 92:####################################################################
 93:##########################################################
 94:####################################################################
 95:####################################################################
 96:####################################################################
 97:####################################################################
 98:####################################################################
 99:###################################################################
100:####################################################################
101:####################################################################
102:##################################################################
103:######################################################
104:######################################################
105:######################################################
106:####################################################################
107:####################################################################
108:####################################################################
109:################################################################
110:###############################################################
111:#############################################################
112:#######################################################
113:#######################################################
114:################################################
115:#################################################
116:############################################
117:#################################################################
118:####################################################################
```

```
  119:###########################################################
  120:###########################################################
  121:###########################################################
  122:###########################################################
  123:#####################################################
  124:#####################################################
  125:##############################################
  126:##############################################
  127:########################################
  128:##############################################
  129:##############################################
  130:####################################################
  131:##############################################
  132:##############################################
  133:########################################
  134:########################################
  135:##################################
  136:#################################
  137:#################################
  138:###########################
  139:###########################
  140:#################################
  141:#################################
  142:#################################
  143:#################################
  144:###########################
  145:#####################
  146:#####################
  147:##############
  148:####################
  149:##############
  150:##############
  151:##############
  152:####################
  153:##############
  154:####################
  155:##############
  156:########
  157:##
  158:#
  159:
  160:
TOTAL RESOURCE CYCLES USED =     6178    % UTILIZATION =        9
AVERAGE RESOURCES USED PER TIME STEP =      39     MAXIMUM =  104
```

```
RESOURCE USAGE CF TYPE          3 X 3 MATRIX MULTIPLY
   1:                           (MULTIPLY)
   2:
   3:
   4:
   5:
   6:
   7:
   8:
   9:
  10:
  11:
  12:
  13:
  14:
  15:
  16:
  17:
  18:
  19:
  20:
  21:
  22:
  23:
  24:
  25:
  26:
  27:
  28:
  29:
  30:
  31:
  32:
  33:
  34:
  35:
  36:
  37:
  38:#########
  39:#########
  40:#########
  41:#########
  42:
  43:#########
  44:#########
  45:#########
  46:#########
  47:
  48:
  49:#########
  50:#########
  51:#########
  52:#########
  53:
  54:
  55:
  56:
  57:
  58:
    59:
    60:
    61:
    62:
    63:
    64:
    65:
    66:
    67:
    68:
    69:
    70:
    71:
    72:
    73:
    74:
    75:
    76:
TOTAL RESOURCE CYCLES USED =      108   % UTILIZATION =        5
AVERAGE RESOURCES USED PER TIME STEP =       1    MAXIMUM =     9
```

```
TOTAL PROCESSOR RESOURCE USAGE        3 X 3 MATRIX MULTIPLICATION
   1:###
   2:###
   3:####
   4:######
   5:######
   6:###
   7:####
   8:########
   9:#########
  10:#############
  11:#############
  12:##########
  13:############
  14:#############
  15:##############
  16:##################
  17:####################
  18:#################
  19:##################
  20:####################
  21:################
  22:##################
  23:####################
  24:######################
  25:#######################
  26:#####################
  27:###################
  28:################
  29:#################
  30:#################
  31:#################
  32:##################
  33:################
  34:##############
  35:###########
  36:#############
  37:####################
  38:#######################
  39:########################
  40:######################
  41:######################
  42:#################
  43:#######################
  44:#######################
  45:##########################
  46:########################
  47:###################
  48:#########################
  49:############################
  50:###############################
  51:###############################
  52:###############################
  53:######################
  54:######################
  55:######################
  56:############################
  57:#########################
  58:############################
```

```
   59:#########################
   60:######################
   61:###################
   62:######################
   63:###############
   64:#####################
   65:###############
   66:############
   67:#########
   68:###########
   69:########
   70:###########
   71:########
   72:#####
   73:##
   74:#
   75:
   76:
TOTAL RESOURCE CYCLES USED =     1234   % UTILIZATION =        5
AVERAGE RESOURCES USED PER TIME STEP =      16    MAXIMUM =   31
```

TABLE 5

MATRIX MULTIPLY PROGRAM RUN WITH N x N MATRICES

All processors execute in 1 cycle except multiply processor

N	T mult	Time	Total Processor Cycles	Average	MAX
2	1	52 cycles	466	9 (8.97)	15
3	1	71 cycles	1114	15 (15.7)	31
4	1	90 cycles	2196	24 (24.4)	57
5	1	109	3826	34 (35.05)	91
6	1	128	6178	39	104

$t = k_1 n + k$ $k_1 = 19$ $k_0 = 14$

2	4	55 cy	511	9	15
3	4	74 cy	1234	16	31
4	4	93 cy	2451	26	57
5	4	112 cy	4294	38	91
6	4	131 cy	6895	52	133

$k_1 = 19$ $k_0 = 17$

Total Resource Usage Calculation

Since n^3 multiplications are required to do a matrix multiplication, the total number of processor resource cycles used is at least proportional to n^3. If we assume that the total processor cycles is given by

(1) $$Tot = k_3n^3 + k_2n^2 + k_1n + k_0$$

then we can use the results of simulations for four different values of n to find k_3, k_2, k_1, and k_0 by solving the linear system of equations

$$n_1{}^3k_3 + n_1{}^2k_2 + n_1k_1 + k_0 = Tot_1$$

$$n_{2_3}{}^3k_3 + n_2k_2 + n_2K_1 + k_0 = Tot_2$$

$$n_{3_3}k_3 + n_3k_2 + n_3k_1 + k_0 = Tot_3$$

$$n_4k_3 + n_4k_2 + n_4k_1 + k_0 = Tot_4$$

derived by using the values found for total processor resource cycles for the four values of n.

The values of the preceeding table for n=2,3,4,5 were used to calculate the $k_o - k_1$ for the case where multiply time = 4. The valves of the constants were

$$k_3 = 22,\ k_2 = 49,\ k_1 = 60,\ k_0 = 19$$

These values also satisfy the fifth equation for the case n=6, i.e.,

$$k_3 \cdot 6^3 + k_2 \cdot 6^2 + k_1 \cdot 6 + k_0 = 22 \cdot 216 + 49.36 + 60 \cdot 6 + 19 = 6895$$

Thus, the total processor cycles for multiplication of two n by n matrices when multiply takes four cycles is given by

(2) $$Tot = 22n^3 + 49n^2 + 60n + 19$$

Using the values of total processor cycles observed when multiplication is one cycle gives

$$k_3^1 = 19, k_2 = 46, k_1 = 57, k_0 = 16$$

So that in this case

(3) $\mathrm{Tot} = 19n^3 + 46n^2 + 60n + 16$

FREQUENCY OF EXECUTION OF PRIMITIVE NODES

So far we have only discussed combined processor useage of all node types. The simulator output also provides separate statistics for each processor type. These statistics can be used as a guide in setting up a system with a finite number of processors to determine how many processors of each type to provide. Table 6 shows the number of processor cycles used by each primitive node type for the trapezoidal quadrature program. Since each processor executed in one cycle on this run, the table also represents the number of executions of each node type except the procedure node. The number of cycles entered for the procedure processor is the number of cycles the invoked graph procedure requires to complete, so the figure given in this case is only valid for a system in which the procedure processor is reserved throughout the computation of the invoked procedure. For this reason the largest number of cycles is that used by the procedure processor. The second largest node type is the two copies node.

The breakdown into individual node types shown in table 6 is not as useful as a less detailed breakdown for three reasons: 1) Since only three graph programs were investigated the statistics gathered from them are not representative at that level of detail; 2) Since the primitive nodes implemented in the simulator were chosen arbitrarily, they are not necessarily representative of the primitive operations which might be implemented in an actual system; 3) A breakdown into individual operations is useful only for a pure "functional unit" model where separate processors are used for each type of operation. In practice it is unlikely that different processors would be used for addition and subtraction, for example. It is more likely

TABLE 6

PRIMITIVE NODE EXECUTIONS IN TRAPEZOIDAL QUADRATURE PROGRAM

NODE	NUMBER OF CYCLES	PERCENTAGE OF TOTAL
1	270	40.7%
2	0	0.0%
3	12	1.8%
4	32	4.8%
5	0	0.0%
6	0	0.0%
7	33	5.0%
8	154	23.2%
9	45	6.8%
10	9	1.4%
11	35	5.3%
12	18	2.7%
13	9	1.4%
14	25	3.8%
15	21	3.2%
16	0	0.0%
17	0	0.0%
18	0	0.0%
19	0	0.0%
20	0	0.0%
21	0	0.0%
22	0	0.0%
23	0	0.0%
24	0	0.0%
25	0	0.0%
26	0	0.0%
27	0	0.0%
28	0	0.0%
TOTAL	663	99.4%

that certain primitive operations would be grouped together to be executed by an arithmetic unit, a data routing unit, etc.

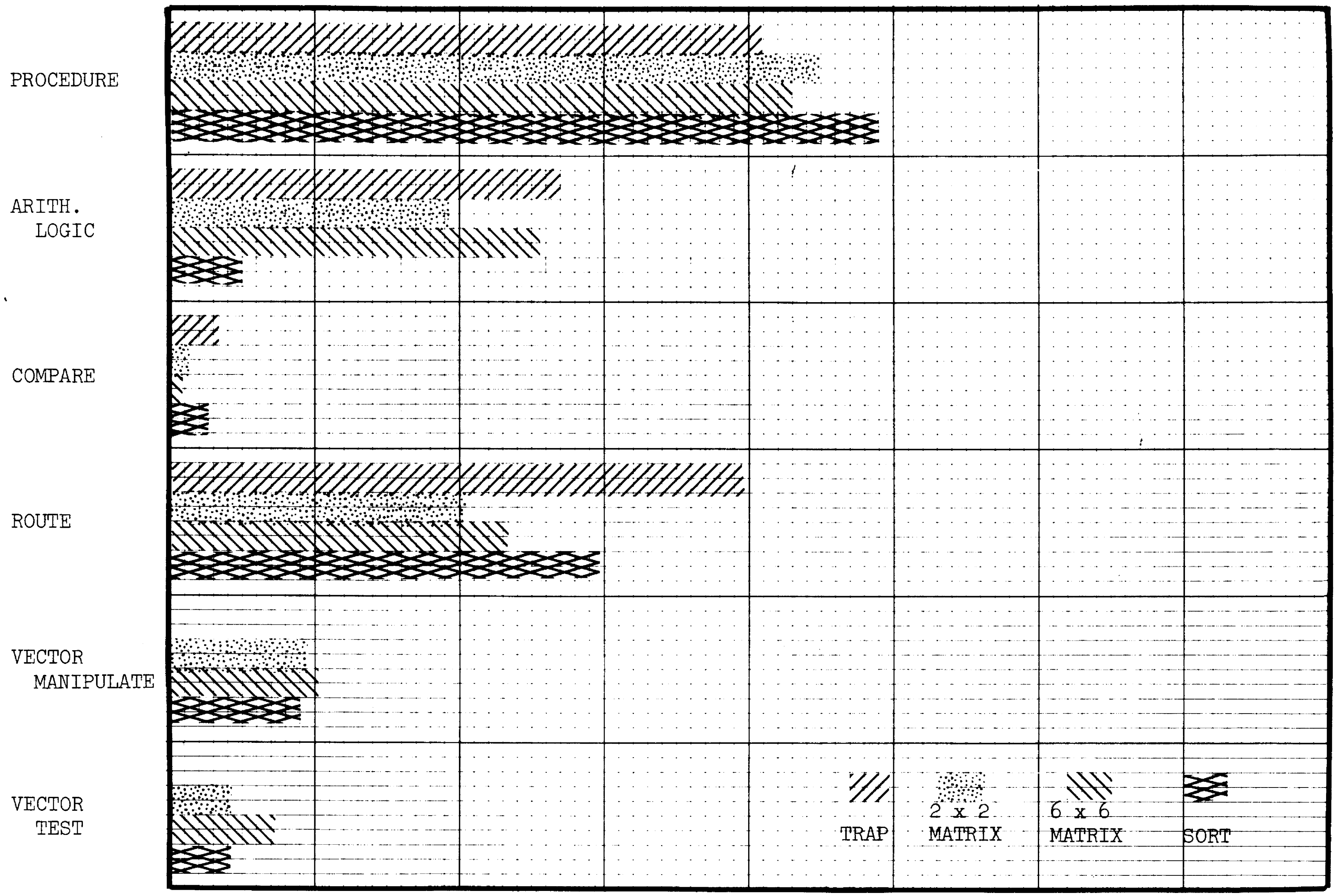
For these reasons I have grouped the primitive nodes into six classes, the procedure node, arithmetic and logical nodes, compare nodes, data routing nodes, vector manipulation nodes, and vector testing nodes. Table 7 gives the percentage of node executions falling into each class for the trapezoidal quadrature program, a 2 by 2 matrix multiplication, a 6 by 6 matrix multiplication and the sort program. It also gives the mean and standard deviation in each class for the four programs. The results are shown graphically in fig. 46. The largest number of processor cycles is used by the procedure node for the reason given above. The procedure node was put into its own class since the execution logic for a procedure call is sufficiently more complicated than that for the other nodes to justify dedicating a special processor to procedures. Procedure processors might also be used as control processors to direct the execution of nodes in the invoked graph procedure.

The second largest number of executions fall into the data routing class, which accounts for more than 1/4 of the executions on the average. The arithmetic and logical nodes and the two classes of vector operations taken together each account for about 11% of the executions, while the comparison nodes are the least used class.

TABLE 7

PERCENTAGE OF EXECUTIONS IN SIX CLASSES OF PRIMITIVE NODES

	TRAPEZOIDAL QUADRATURE	2 BY 2 MATRIX	6 BY 6 MATRIX	SORT	MEAN	σ
Procedure (1)	40.7%	54.8	43.3	49.0	46.95	5.7
Arithmetic, Logical (3,4,5,6,7,13,14,24,25)	16.8%	9.4	15.4	5.1	11.675	4.7
Compare (2,15,23)	3.2%	1.3	0.7	2.8	2.0	1.0
Route (8,9,10,11,12,28)	39.4%	20.6	23.3	29.7	28.25	7.2
Vector Manipulation (16,17,18,21,22,26,27)	0.0%	9.5	10.3	8.9	7.175	1.9
Vector Testing (19,20)	0.0%	4.3	7.2	4.5	4.0	2.6



RELATIVE USE OF PRIMITIVE NODES

Fig. 46

CONCLUSIONS

The simulator and the graph programs described here show first of all that Adams' graphs are a feasible representation in which parallel algorithms can actually be programmed and that a CPU could be constructed which uses such a representation. Writing down a graph program is roughly equivalent in difficulty to machine language programming for a conventional computer, however, and the problem of designing a suitable higher level language which can be translated into an efficient computation graph representation is still open.

The simulations also show that the graph representation is able to take advantage of opportunities for parallelism at several levels without conscious effort on the part of the programmer. The square root program and the matrix multiply are instructive extremes in this regard. Newton's method for finding the square root is inherently sequential, yet even for this algorithm a small amount of overlapped execution is possible, and the computation graph representation produces it. Matrix multiplication, on the other hand, is capable of highly parallel execution, and straightforward programming of this algorithm as a computation graph produces parallelism on the order of n^2, reducing computation time to the order of n. Besides the three programs described here, a number of other programs were written for the simulator including recursive and iterative factorial programs, SIN and COS routines and a number of polynomial evaluations. All resulted in some degree of parallel execution, although no special efforts were made to produce parallel execution.

The actual speed which could be obtained on an implementation of this model could depend very heavily on the amount of overhead or bookkeeping required for control of the system. Three sources of overhead can be distinguished: 1) The computations required to keep track of the status of nodes in the executing graph, to determine whether they are ready to execute and to initiate and terminate their execution; 2) The overhead resulting from the organization of memory into QUEUES; 3) The overhead caused by the execution of algorithms to allocate shared resources such as processors and memory. No attempt is made to refect these costs in the output statistics of the simulator because they are very dependent on specific hardware implementations. For example, the implementation of queues used in the simulator requires two memory references to fetch a data item, one to get the pointer to the head of the queue and one to get the data itself. However, if the head and tail pointers were kept in registers or in fast storage, the time could be reduced to one memory cycle.

The major portion of the execution time of the simulator itself is spent checking each node to see whether it is ready to execute. If the model were implemented with a single control processor, it would have to be much faster than the primitive node processors to provide any degree of parallelism. However, an implementation which used the procedure processor to control execution of the nodes in the graph procedure which it initiated could distribute the overhead considerably to allow a greater degree of parallelism. The overhead can also be reduced by an efficient representation of the node edge connectivity of the graph. The connection matrix representation used here is inefficient in this regard since it requires the control logic to scan the matrix to find the edges directed into a node before it can check whether those edges have data on them. An edge list representation of the graph would be more efficient in this regard.

Two main questions were studied in the three programs described in this report: first the dependence on problem size of computation time and amount of parallelism in execution, and second, the dependence of these measures on relative processor speeds.

The Trapezoidal Quadrature program, the Sort program, and the Matrix Multiplication program differ widely in the amount of parallelism which they allow. The time required to execute the trapezoidal quadrature program is proportional to the number of points used. However, the dependence lies in the generation of the n points for which f(x) is calculated, not in the calculation of f(x), so that increasing the complexity of the function being integrated does not increase the coefficient of n in the time required for the quadrature. Rather, it increases the number of values $f(x_i)$ which are being calculated concurrently. The square root procedure used in the quadrature program is inherently sequential, and its computation time depends on the value of x. The average number of nodes in execution during the square root calculation is 1.7. However, since the quadrature program calculated several values of the square root concurrently, it executed from 8 to 14 nodes on the average.

The sort program executes in a time proportional to n, the number of items in the file being sorted. Since the number of operations required is proportional to n $\log_2 n$, the average number of nodes in execution in this program is on the order of $\log_2 n$. The matrix multiplication program, on the other hand, is highly parallel. Although n^3 operations are required to multiply two n by n matrices, the program executes in a time proportional to n. Of course, the number of processors required to achieve this time is on the order of n^2, but the algorithm itself is inherently parallel, whereas the sort program increases sequentially faster than it increases its parallelism in the ratio $n/\log_2 n$, and the trapezoidal quadrature is inherently sequential, though it allows overlap in the calculation of $f(x)$.

One of the major questions which can be posed in an infinite resource environment is the degree to which variations in relative processor speeds affect the computation. In a sequential computation, the time to execute a program is just the sum of the times to perform each type of operation weighted by the number of times that operation is executed by the program. In a parallel program we might expect a secondary effect due to delays in the initiation of a node which is waiting for output from one of its predecessors. This effect did not show up in my simulations, however. The effects of different sets of varied processor speeds and of uniform processor speeds equal to the mean of the varied speeds over the nodes in the graph program are virtually identical. Moreover, this held even though the node execution times are not weighted by the number of times the node is executed in calculating the mean.

This conclusion should be taken as very tentative, since the number of programs investigated was small. In order to draw even the modest conclusions that relative processor speeds are unimportant if the mean execution time is constant for <u>many</u> (not most) programs, one should investigate a large number of programs written by different programmers under many different timings. Because of the strong dependence of program behavior on small variations of coding, even this investigation would not be completely generalizable. Several people have exhibited programs whose execution time is strongly dependent on small changes in processor speed.(1)

If the results found here hold more generally, however, they suggest a method for determining processor speeds in a hardware implementation.

First, a large sample of actual programs should be collected and the distribution of primitive node types in this sample should be determined. Then, in balancing processor speed against the per unit cost of the logic required, one should attempt to minimize the mean execution time over that distribution.

(1) E.G. Paul Richards "Parallel Programming" Report No. TO-B60-27, Technical Operations Inc., Burlington, Mass. 1960