J. LEVY

DEC. 11, 1967
CETM 40

Implementing a Graph Model of Parallel Computation

Abstract

A graph model of computation has been developed by Karp and Miller in which data flows in FIFO queues along directed edges between nodes representing computation steps. Extensions and particularizations of this model are being developed by Adams. This paper is an initial attempt to describe a direct implementation of the model in hardware and software which is currently feasible and to discuss possible variations in the model and the implementation.

Introduction

There are two basic ways of enhancing the computational "power" of a computer. One is to increase the speed of its components. The other is to elaborate its structure in an attempt to perform more operations concurrently. Physical limitations in speed are already foreseeable. Thus, concentration on methods of invoking multiple "processors" in performing a computation is both reasonable and urgent.

One of the basic difficulties in studying multiple-processor systems is that it is difficult to visualize the execution of a computation. In machines where one processor is dominant, or central, a single "instruction stream" is usually an adequate description of the computation. This convenience is lost when multiple processors of equal importance are invoked. Graph models of computation therefore have a dual usefulness for multiple

processor systems: providing a theoretical basis for studying parallelism, and yielding immediate, intuitive visual models.

The Theoretical Model

The model used here is based on those of R. M. Karp and R. E. Miller,[1] and J. Rodriguez,[2] as modified and particularized by D. Adams.[3] It must be emphasized that the following is preliminary and possibly incomplete. A computation (or program) consists of a graph, of nodes and edges. Data flows along edges of the graph between nodes which represent steps of the computation. The edges are directed and consist of first-in-first-out queues of variable length. The elements of the queues may consist of simple scalar data or more highly structured items. Tags or typing information may be associated with each queue element. The queue as a whole is characterized by its length, the type of data it may contain, and its input and output connections. It may also have a status condition associated with its output connection. A node represents an operation which accepts data from its input edges and produces results on its output edges.

Activation of a node consists of three stages: Initiation, in which data is removed from the input queues; Execution, in which the operation is performed; and Termination, in which the results are attached to the output queues and the node activity ceases. Conditions are imposed on the activation of nodes in order to preserve determinacy -- that is, to assure that a unique result is produced by each unique graph with given initial conditions. First, a node is not a candidate for activation unless data is present on each of its input queues. Initiation consists of removing exactly one element from each queue. (Exceptions to this occur in some variants of Adams' model.)

A very useful implicit parallelism is immediately possible: whenever more than one element is present on each input queue, multiple "instances" of the node may be activated, operating concurrently. One important restriction is that the results must be attached to the output queues in the order corresponding to the order of initiation of instances.

We now have the basis for performing a computation. We are given a graph, and an initial state of its queues. Nodes are activated in any sequence consistent with the restrictions above. If the computation will ever terminate, then at some time later all activity will have ceased. (Another possibility -- that some nodes are still executing but cannot terminate -- is a case which Rodriguez, Van Horn,[4] and Adams have considered.)

In Karp's model, instances of a node are initiated one at a time as elements are found on the input queues, and the sequence of initiations is maintained in the "μ" attribute of the node. This attribute is a list of the instances which have been initiated and which have not yet terminated (although the operation involved in the execution may be complete, the results may not be output out of sequence). This list, called the "initiation queue" by Adams, has a direct counterpart in the simulation model below.

Adams is considering an extension of the basic model by allowing recursive definition of "procedure nodes", defined in terms of an inner structure which is a graph. Thus a distinction is made between "primitive nodes", which perform pre-determined operations, and "procedure nodes" which may invoke both primitives and procedures. Only primitive nodes are considered in the simulation model.

A variant considered by Adams and suggested by Rodriguez' model allows a "locked/unlocked" status on an input queue. When locked, the queue is

not examined for data, and the node may be a candidate for activation even when there are no elements on that queue. Change of locked/unlocked status is controlled by the data in the queue and occurs between the initiation and execution stages.

The Simulation Model

The following model was originally formulated using the framework of a (simulated) micro-programmable computer proposed by A. J. Nichols.[5] This computer has available a read-only store, micro-instruction execution logic, 14 hardware register, 6 arithmetic units, and many memory banks including associative (content-addressable) storage. Although only one processor was proposed in the framework above, we assume in the simulation model that at least two identical processors are operating concurrently.

Fig. 1 shows a block diagram of the hardware configuration of the simulation model.

We presume that the memory busses have a built-in priority in case of simultaneous requests for data transfer. Two busses are shown to allow multiple concurrent access to interleaved addresses.

Figure 2 shows the representations of nodes and edges in the model. Each processor performs both "supervisory" and "execution" functions largely independent of the other processors. All (macro-) programs for the nodes are stored in "read-only" areas of memory. That is, the source of processor instructions is not to be modified during the running of a "program".

Edge queues are maintained as linked lists -- two-word blocks of conventional memory in which the second word "points" to its successor. This form of storage permits efficient allocation of storage, but it is somewhat

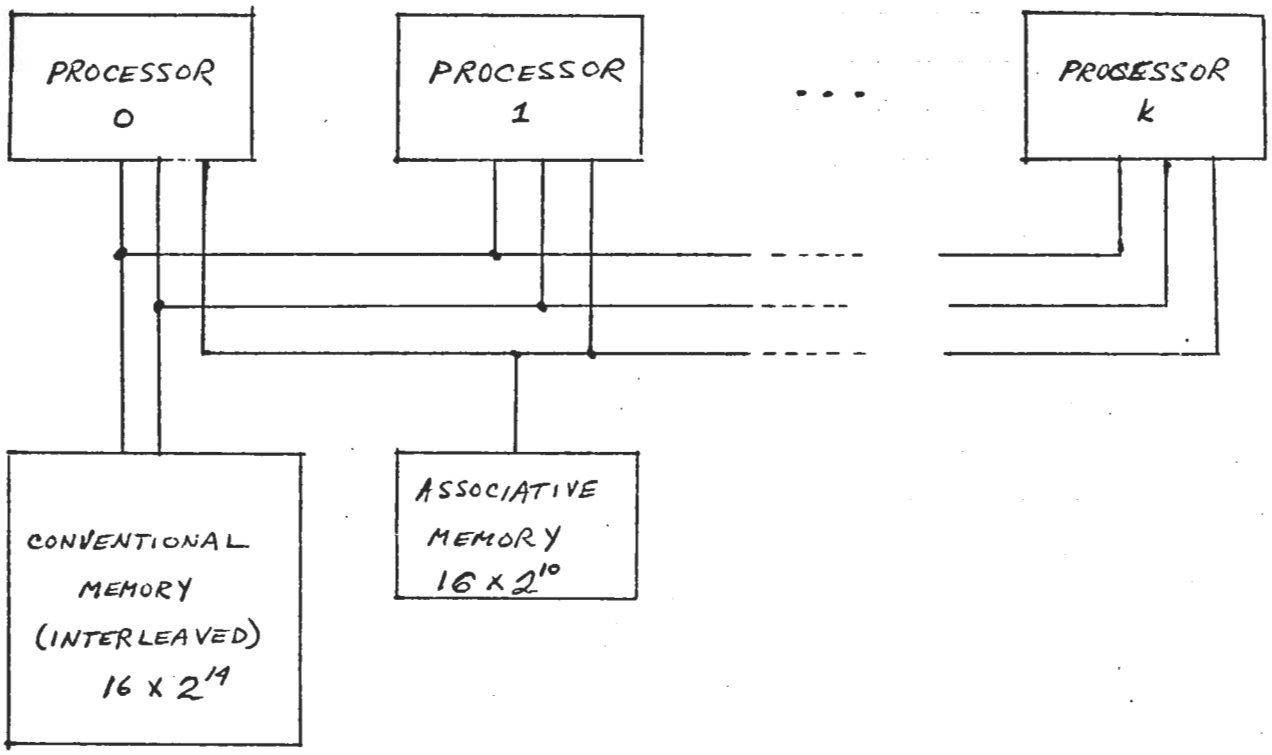
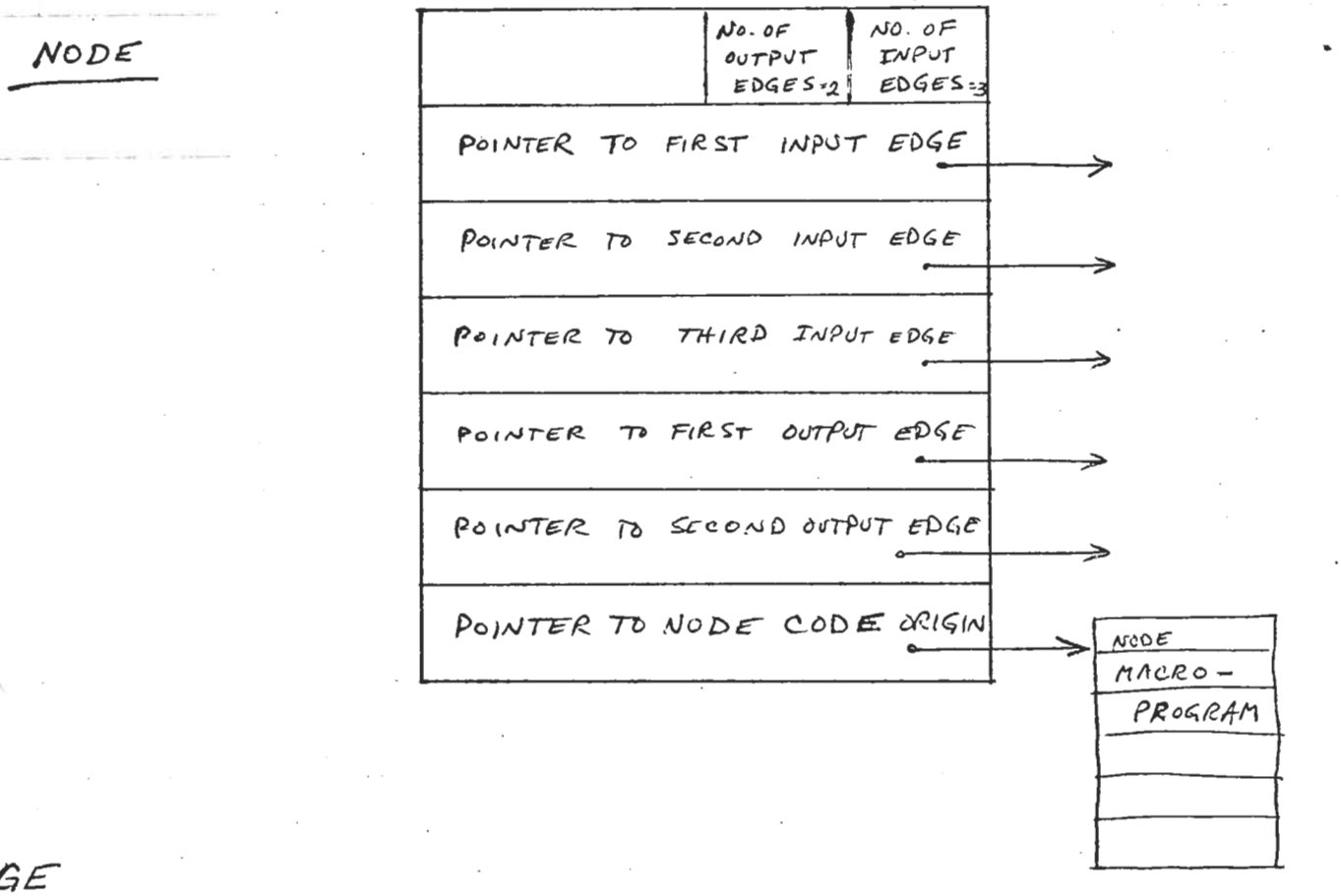


FIGURE 1 : HARDWARE CONFIGURATION OF THE SIMULATION MODEL.

FIGURE 2: NODE AND EDGE BLOCK LAYOUTS



wasteful of memory. Each edge is characterized by a 6-word block of storage (each word here is 16 bits wide). Locked/unlocked status, queue type, length of queue (number of elements), and pointers to the queue elements are contained in this block. It also contains pointers for the initiation queue (corresponding to Karp's "μ" property) and local and global indices ("names") of the successor node.

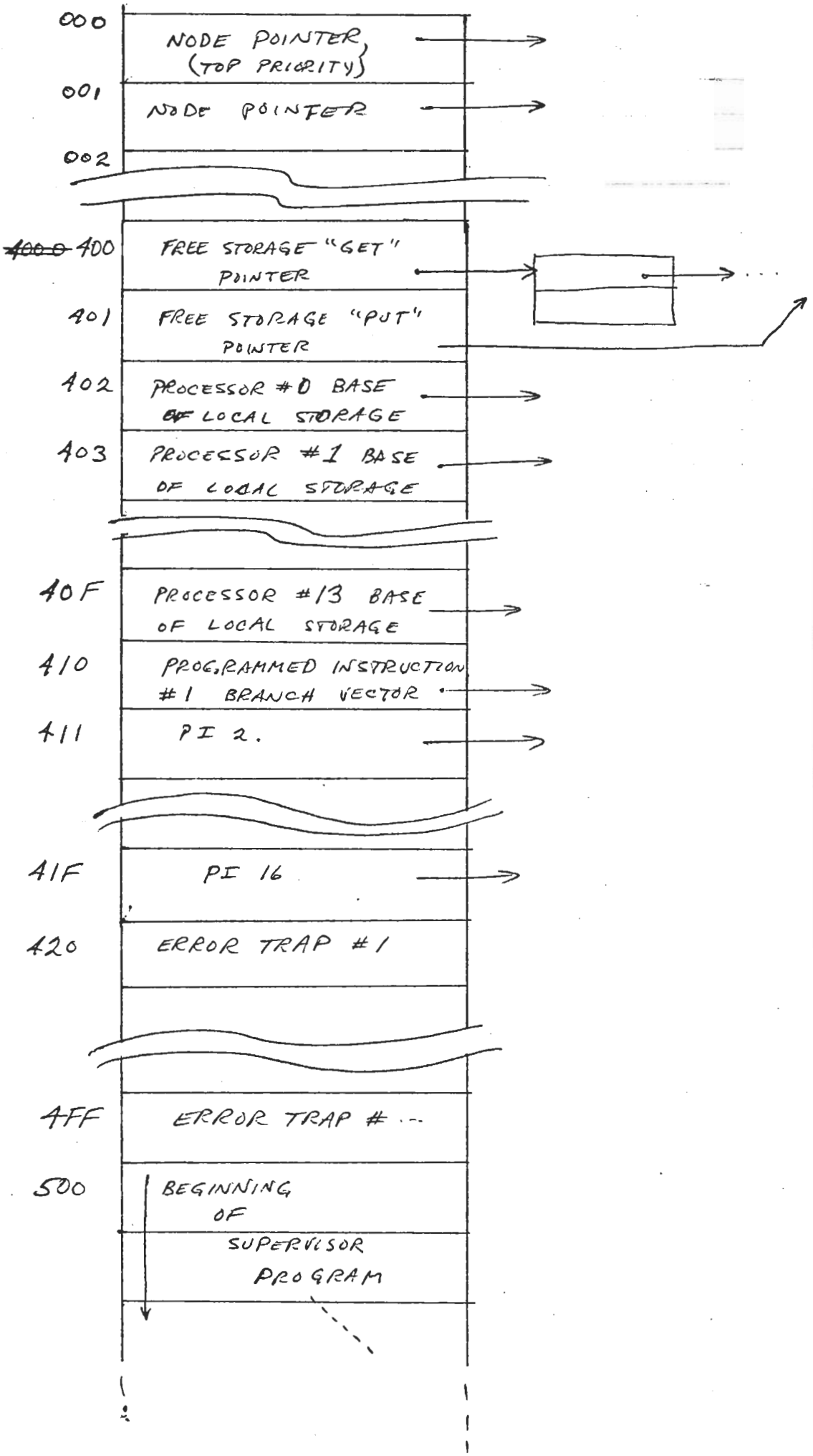
A node is characterized by a block of length 2 + k, where k is the number of edges attached to it. The first word gives the indices of input and output edges (these are "locally" indexed by their position in this block). The last word in the block points to the origin of the read-only macro code which is to be executed during the execution stage.

"Free-storage" pointers, traps, certain base pointers and branch vectors are maintained in fixed locations in the conventional memory as shown in Fig. 3. Address 0 - 3FF are used in conjunction with the association table.

The associative memory is the key to allocation of processors to tasks (nodes) and to maintenance of node activation candidacy. We allow up to 15 edges to be attached to each node, locally indexed 0-E. In the associative memory, each of up to 2^{10} nodes is allotted one word for its edge status. Each of 15 bits in the word corresponds to one of the possible edges attached to the node. When at least one element is present on a queue, the corresponding bit is set (=1). When there are no elements in the queue, the bit is reset (=0). Exception to this is taken when the edge is in the "locked" status. In this case, the bit remains set at least until the status changes to "unlocked". When there are fewer than 15 edges attached to a node, the "empty" bits are defaulted to 1 (set).

Now the process of finding a candidate node for activation is straightforward. The associative memory is "associated" on a mask of all ones.

FIGURE 3: LAYOUT OF LOWER AREA OF CONVENTIONAL MEMORY



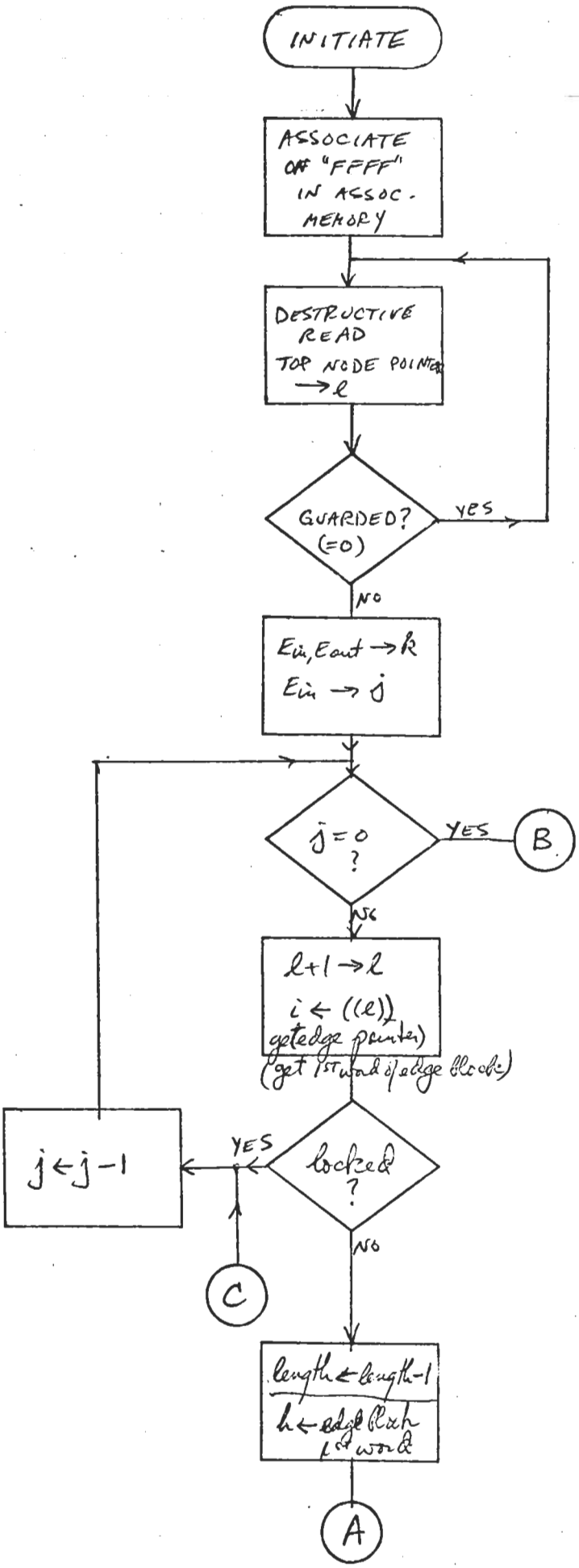
Flags are set by each word which contained all ones, corresponding to those nodes which have elements present on each input queue. This is exactly the condition required by the theoretical model for activation of a node. The processor which is looking for a task will then retrieve the index of the "top" node which is ready for activation, and then find a pointer to that node in the corresponding position in the table in the lower part of the conventional memory. It can then proceed to activate that node. Note that an inherent priority exists in the ordering of the table, since the "top" flag is accessed after the association.

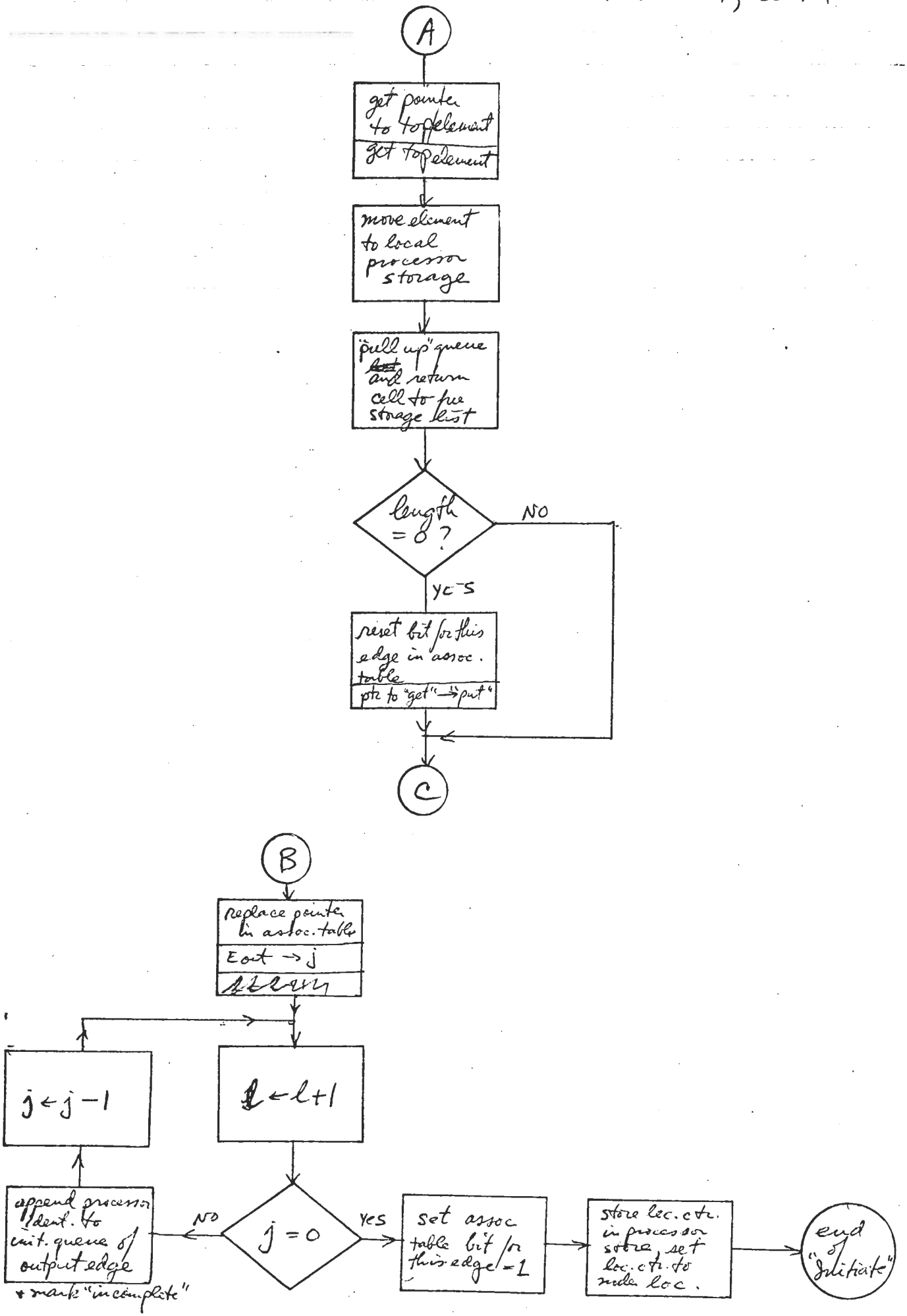
The association table, the nodes and edges, and some of the supervisor area can be modified by any one of the processors. How does one avoid hopeless entanglements? First, let us describe a typical cycle of operation for one processor. (see flow charts, Fig. 4)

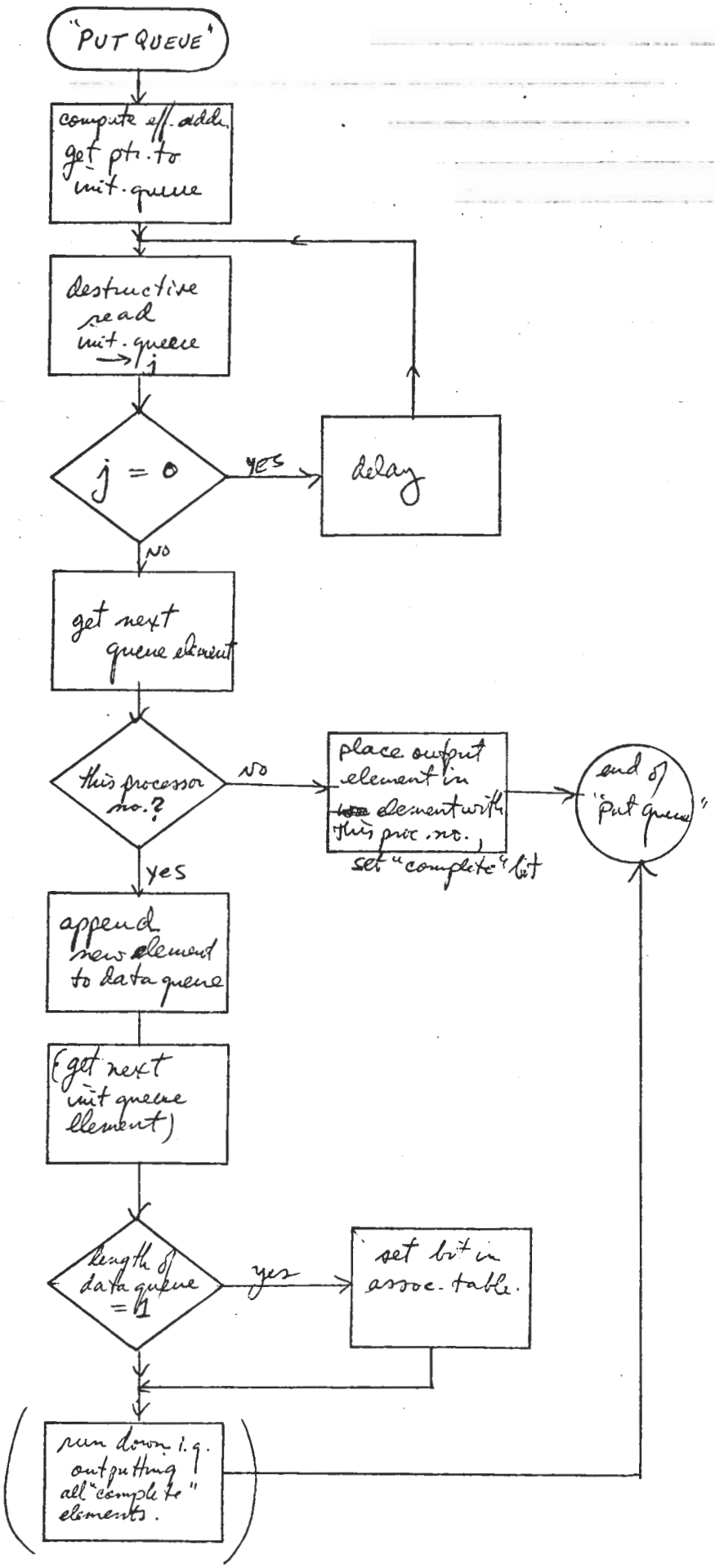
The processor presumably starts in the "supervisor" program. At some point, the program (i.e., the code in lower ~~case~~ core) decides to dedicate the processor to node execution. It does this by performing (executing) the "Initiate" macro-instruction.

The following is a description of the macro-instruction "Initiate", as implemented in micro-instructions in the processor. First, the processor "associates" in the node status table and selects a node which is ready for activation. It retrieves a pointer to that node and begins the actual "initiation" stage as previously defined in the theoretical model. It removes one element from the queue of each input edge, updating the length and queue pointers, and places the elements in storage local to the processor. If the length of any queue becomes zero at this time, the corresponding bit is reset in the association table. It then attaches an element to the end ("newest") of the initiation

FIGURE 4 FLOWCHARTS




A
get pointer to top element
get top element
move element to local processor storage
"pull up" queue and return cell to free storage list
length = 0 ?
NO
YES
reset bit for this edge in assoc. table
ptr to "get" → "put"
C
B
replace pointer in assoc. table
Eout → j
j ← j-1
l ← l+1
append processor ident. to init. queue of output edge
+ mark "incomplete"
NO
j = 0
YES
set assoc table bit for this edge = 1
store loc. ctr. in processor store, set loc. ctr. to node loc.
end "Initiate"


"PUT QUEUE"
compute eff. addr. get ptr. to init. queue
destructive read init. queue → j
j = 0
yes
delay
no
get next queue element
this processor no.?
no
place output element in element with this proc. no., set "complete" bit
end of "put queue"
yes
append new element to data queue
(get next init queue element)
length of data queue = 1
yes
set bit in assoc. table.
run down l.q. outputting all "complete" elements.

queue of each output edge. This element contains an identifying number unique to the processor, and some status bits. Throughout this process, interaction with other processors in this node must be avoided. Therefore, when the processor accesses the pointer to this node from the table, it does so by a "destructive read", leaving the former pointer word equal to all zeros. This will inhibit other processors from operating on this node while the initiation stage is under way. Clearly, this processor must also test for the condition of all zeros when it accesses an alleged node pointer. If the condition is found, then the processor will return to the association table and access the index of the next "ready" node. If none are ready, it will continue to associate and test until one is ready and unguarded. At the end of the initiation stage (still in the Initiate microprogram) the node pointer is replaced, allowing access by other processors. The processor then loads its location counter with the origin address of the node macro code, effectively branching into the node execution stage. The microprogram for "Initiate" ends at this point.

The node macro-code performs a "primitive" operation, however, ~~that~~ it has been defined by the node programmer. The node programmer has available conceptually a general-purpose computer, ~~as described elsewhere in this paper.~~ The scope of this computer is limited only by the restriction that all input data must have been given by the Initiate command, and that at most one element of output data may be loaded onto each output queue by the command "Put Queue". The structure of queue elements and allocation of storage for them is treated elsewhere. Each element may in general be a pointer to an elaborate structure, such as a vector, matrix, or tree, and may also contain bits characterizing its type.

destructive read on the edge pointer in the node, restoring it when access by other processors is "safe". When a "guarded" condition is encountered during "put" and "release", the processor must wait.

At the end of node execution, the macro-instruction "Terminate" is performed (it must be invoked by the node programmer). This instruction removes control from the node and (by modifying the instruction counter) effectively performs a return branch to the supervisory program area. The supervisor program presumably may do general bookkeeping before giving another Initiate command to send the processor into the cycle.

Discussion of Possible Extensions

The structure of the simulation model allows some features not yet described. First, the association table-node pointer table is amenable to changing priorities of nodes during the course of execution. The supervisor can interchange the respective pointers and status bits of two nodes (guarding them from access during the ~~switch~~ interchange), thus reversing the relative priorities of the two nodes. This type of (interchange) operation is probably satisfactory for implementing a dynamic priority scheme. One of the major problems in implementing graph computation models is determining or limiting the upper bound on data queue length. The following scheme would allow automatic queue length limiting.

The length of each edge queue is maintained in the edge defining block. This count increases as elements are added (by the "put" and "release" commands) and decreases as node instances are initiated. Thus, by adding a test to the "put" and "release" microprograms, we can decide to change the priorities of some nodes when the edge queue length exceeds a pre-established number. A plausible approach would be to assure that when an

edge queue exceeds a certain length, the priority of the successor node is placed above that of the predecessor node. With this approach in mind, we have included the queue length and the successor node index information in each edge block.

Graham[6] has described an ordering scheme for graph model programs which, for an elementary case, gives optimal execution time for the program. This scheme may be applicable to assigning node priorities in this model.

The lock/unlock status of an input edge is examined by the "initiate" command, but no mechanism is provided here for modifying that status. It has been purposely omitted, pending further development of Adams' model. The status will presumably be modified only by the "initiate" instruction.

"Procedure nodes", as described in the theoretical model above, are not implemented in the present model because severe complications in node and edge definition and allocation are indicated.

References

1. Karp, Richard M., and Miller, Raymond E., "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing", SIAM J. Appl. Math. 14 (Nov. 1966), 1390-1411.

   _____, "Parallel Program Schemata: A Mathematical Model for Parallel Computation", 8th Ann. Sympos. on Switching and Automata Theory, Austin, Texas, Oct. 1967.

2. Rodriguez, Jorge, "A Graph Model for Parallel Computations", Electronic Systems Laboratory, Massachusetts Institute of Technology, September 1967.

3. Adams, Duane (Computation Group Technical Memo, Stanford Linear Accelerator Center, to be published).

4. Van Horn, Earl C., "Computer Design for Asynchronously Reproducible Multiprocessing", MAC-TR-34 (M.I.T.), November 1966.

5. Nichols, A. J., "A Micro-Programmable Framework for Computer Design", Notes for course Computer Science 231, Stanford University, November 13, 1967.

6. Graham, R. L., "Bounds for Certain Multiprocessing Anomalies", BSTJ 45, (1966).