Sample Programs for a Hypothetical Computer

Gary J. Nutt*

CU-CS-058-74 October 1974

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

*This work was supported by the National Science Foundation under grant no. GJ-42251

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

ABSTRACT

The multi associative processor (MAP) computer system is a hypothetical machine employing multiple control units and pools of identical processing elements. Each control unit, along with a subset of the processing elements, operates on a single instruction stream and multiple data streams. Processing elements are activated and deactivated by conditions preserved in an internal register, hence the term associative processor. In this paper, a brief overview of the MAP system is given, and a battery of programs for the MAP system is discussed. The programs are executed via an interpreter to investigate possible application areas for this architecture as well as to test various designs that have been incorporated into the model.

Introduction

The multi associative processor (MAP) computer system is a hypothetical machine employing multiple control units and pools of identical processing elements. Each control unit, in combination with a subset of the processing elements, operates on a single instruction stream through the control unit and multiple data streams, one through each processing element, (i.e., SIMD operation). Each processing element contains a *select register* for which the result of up to eight conditional tests can be stored (corresponding to the activity bit in an array processor). Processing element participation in instruction execution is determined by the current contents of the select register, and a corresponding *key* broadcast to each PE by the control unit. It is this capability that leads to the name "associative processor," and the existence of multiple control units that leads to the name "multi associative processor."

In this paper a brief overview of the system is given along with a description of some programs for the MAP system. The programs may be considered to be benchmark programs although they are not used so much for comparative purposes [9,14], but rather as a medium to investigate possible application areas for the architecture. These programs are executed on an interpreter in order that their MAP resource utilization can be monitored and their ability to perform calculations can be tested. Other "synthetic programs" are also discussed, the purpose of these programs being to exercise certain portions of the MAP architecture.

The MAP Architecture

MAP might just as well be an acronym for "multi array processor" as "multi associative processor." The MAP system is composed of eight control units (CUs) coupled to an arbitrary number of processing elements (PEs), each with its own private memory for storing data (see Figure 1). Each control unit decodes an instruction stream from the main memory system, and broadcasts appropriate microinstructions to a subset of the PEs which have been previously allocated to that CU. In Figure 1, the dashed lines indicate a crossbar switch between each control unit and each processing element. The connections at the crossbar are determined by the PE allocation state of the machine, i.e., if PE_{10} is allocated to CU_3, then the instruction bus leading to PE_{10} would be connected only to the instruction bus leading from CU_3. This connection is static as long as the PE is allocated to the CU. The crossbar switch for instruction broadcast lines is conceivable because each instruction bus is only 6 bits wide; conflicts for connections should disappear with a proper PE allocation scheme.

The data bus in Figure 1 is 32-bits in width and thus begins to increase the cost of implementing a straight-forward crossbar. MAP employs <u>bus sectors</u> to relieve this situation; a bus sector is a mechanism by which a pool of k PEs can share a single bus connected to a CU via a crossbar switch. Thus, if there are j data bus sectors, each serving k PEs, the crossbar connection points are reduced from 8jk to 8j when compared to a conventional crossbar switch. It is possible for two or more CUs to have PEs in a common bus sector; in this case, the bus sector must be multiplexed among the involved control units. We note that this design is possible since a data bus need not be used by a control unit during the entire decode and execution cycle, in fact, it may not be used at all for a given machine instruction that would be used in an associative processor instruction set. The design of the bus system is crucial to the success of the entire MAP system, and it is this portion of the architecture toward which most of the initial work has gravitated,[12]. A more complete discussion of the bus system architecture appears in the paper by Arnold [1].

In the MAP system, PEs are a dynamically allocatable system resource, each PE being identical to all of its counterparts, (i.e., there is no "PE array address" as determined by physical bus connections, at the programming level). A CU is allocated as many PEs as are necessary for a given task, and whenever the CU has completed its task, the excess PEs are returned to the pool to be allocated to other CUs. The anonymity of PEs precludes the possibility of data passing from one PE to another on the basis of some array address. Data may be passed from PE-to-PE via the data bus system, but on programmed considerations based on current PE states rather than array address. It is, of course, possible to superimpose the rigid array address on a pool of PEs by explicitly storing an array address in each PE memory, and using this array address to simulate conventional array processor activity.

Each PE memory contains a data stream for SIMD computations. A PE memory must be appropriately loaded by the owning CU when the PE is allocated; this loading takes place by broadcasting data from the main memory through the CU to the individual PEs.

In computational ability, each PE can be thought of as a 16-bit minicomputer with 1024 words of memory. The instruction decode hardware is replaced by an associative unit to allow selective enabling and disabling of the PE based on a number of conditions. The reader is again referred to the architecture paper for details of the PE.

The other major component shown in Figure 1 is the main memory system. The interconnection problems encountered here are quite similar to those encountered in connecting any set of eight Von Neumann-type processors to a common main memory, and this area is the subject of much current work (see e.g. [R,15]). The primary differences that appear in a MAP system would tend to loosen performance constraints, since the number of memory references during the execution phase of the instruction cycle would be drastically reduced. Secondly, associative processor programs (and array processor programs) tend to be more straight-line than the corresponding conventional processor programs [8]. This allows for more effective instruction look ahead in the control unit, decreasing the possibility of memory conflicts.

Analyzing the System

The MAP system is a hypothetical computer system that has been proposed as a medium of investigation into SIMD architecture, measurement and evaluation techniques for array processors, and operating systems design employing parallel techniques [11]. Although we agree that the ultimate proof of the desirability of having such machines is dependent upon actual hardware construction and use, there were at least three major reasons why we did not pursue this goal: First, many portions of the architecture are subject to redesign and refinements, while other portions of the design have not yet been considered (e.g., the I/O subsystem). Second, the range of user applications for such a machine is unknown, although others are currently involved in related studies to sample this range (see e.g. [4]). It appears that there exist several problem areas that could profit from implementations on either array processors or associative array processors, and we shall discuss a few of those areas below. Finally, the possibility of funding the construction of a MAP system, in its current design state, does not exist.

As an alternative to actual machine hardware construction, we are basing our analysis of the MAP system on a series of simulation models ranging in complexity from an interpreter to simple conflict models. The foundation for most of this work is the associative processor program interpreter, called MAPSIM. This program is a simulation of a single control unit and an arbitrary number of processing elements each with an arbitrary memory size, as determined by input parameters. The purpose of MAPSIM is to interpretively execute a MAP program written in a MAP assembly language. The interpreter is written in the assembly language of a Control Data 6400 (called COMPASS), and hence the interpreter must represent the PE parallelism by employing quasi-parallel programming techniques. At the present time, two versions of the interpreter exist, reflecting two different sets of machine instructions. (The interpreter does not really execute microinstructions, hence some portion of the interpreter must be rewritten for each new instruction set that is tested.) Each MAP instruction set is defined as a symbolic assembly language, employing many of the pseudo operations available in the CDC 6400 assembler. An assembler for the experimental instruction set is then implemented by purging all of the mnemonic instruction definitions normally in COMPASS and defining the MAP mnemonics using macros and data definition statements. This approach allows assemblers for the various languages to be rapidly written and modified, while still providing an assembler with relatively good capabilities, (namely, the COMPASS assembler).

Employing the symbolic assembler and interpreter, associative processor programs can be written and executed, allowing the current model to be tested for attractiveness of the instruction set or to analyze bus and memory reference streams, etc. When choosing a program for implementation on the interpreter, it is possible that the program may exhibit one or the other or both of the following two properties:

- The program illustrates an application area for MAP,
- The program intensively tests some portion of the design.

The ideal situation, from the analyst's viewpoint, is that all programs exhibit both features; it serves as a prototype program to explore a new application area and also provides a driving force for the model components. In the next section, two programs are described that fit this mold, a program to solve a linear system of equations and another program to find the minimum weighted path length through a graph.

It is also possible that a program can be written that illustrates a useful application of the machine, but is not particularly useful to be run on the interpreter for some reason. The scheduling program presented in the next section is an example of this type of program. Since the interpreter does not handle multiple control units nor does it represent any job queue for the simulated system, there is no realistic environment in which to execute the scheduling program.

The primary constraint on the programs mentioned above is that they illustrate some application for which the associative array processor can be particularly useful. It is not possible to ensure that these programs will provide a maximum load on the various components of the machine (e.g., the memory-CU interface or the CU-PE interface). Thus, we have also contrived a set of programs that will exhibit controlled behavior with respect to certain portions of the machine. These programs allow the designer to place a controlled load on any particular portion of the system that he desires. For example, the designer could use a program of this type to place a maximum load on the memory system by frequently executing instructions that make data references to the main memory. The description of such a contrived program is included in a later section.

MAPSIM has been designed so that simulated hardware monitors can easily be inserted into the interpreter. This allows the model to be intensively monitored during each execution of a program. These monitors can either collect selected trace data, or compute distributions which describe the performance of the individual programs. It is then possible to use this data to drive lower detail models which incorporate multiple control units, and to conduct the performance analysis of shared resources in these models, [12]. [illegible] on the MAP study [illegible] with such use of the [illegible] the [illegible]

Some Sample Programs

Much of the initial motivation for the design of a multi associative processor was provided by the suspicion that portions of an operating system could be carried out as parallel tasks. It had been observed that the associative and array systems that we knew about used some sort of sequential processor to actually implement the operating system, for example, the STARAN employs a PDP-11 for this purpose [6] and the Illiac IV uses a Burroughs B6700 [2]. In order for the operating system to take advantage of the SIMD operation to execute certain parallel tasks, it is necessary to either multiplex a single control unit, or provide more than one control unit. Although the former alternative would result in a simpler hardware design, it would tend to degrade performance from a resource utilization point of view since any operation by a process requiring k data streams would cause all n-k PEs to be idle. In an operating system, we can expect the amount of sequential processing to be significant, and our experience with other application programs indicates that there may also be considerable portions of time in which the corresponding process is sequential. Therefore, the multi control unit approach was chosen.

Scheduling processes to resources can be viewed as a parallel task if there exists more than one process competing for some set of resources. The first program, shown in Figure 2 as a flowchart, is a scheduling program for PE allocations. Using a preemptive policy, this program might be executed by the supervisor each time a process makes a PE request. We are assuming that each process has previously been assigned a priority, and that each PE assigned to a process has a word in its PE memory representing the priority of the process to which it is currently allocated as well as a process identifier. (In the sample program, it is assumed that PE symbolic address PRIORITY is the process priority, and PROCESSID is the process identity.) The "preempt" instruction in box 2 allows the operating system control unit to temporarily obtain all PEs. In box 3, the PEs are activated and in box 4, each PE loads the PRIORITY of the owning process into its accumulator. The lowest priority processes are temporarily preempted in box 6 to see if they are running at a priority less than the requested process; if not, boxes 10 and 11 place all PEs in their state when the program was in box 1 (after queueing the PE request) and resumes processing. If the priority of the NewProcess is higher than the temporarily preempted process(es), the program checks to see if a sufficient number of PEs are available. If not, the cycle through boxes 5,6,7,8, and 9 are repeated until the program exits to box 10 or box 12, where the PE allocation to the NewProcess is carried out.

In this program, the parallel activity takes place in nearly all positions of the algorithm, as indicated by the boxes with four vertical lines rather than two. The only looping required to carry out the algorithm is in the systematic discovery of lower priority processes, one level at a time (i.e., boxes 5-9). The sequential portions of the algorithm occur when the requesting process is queued due to lack of PEs (Box 10).

Two other system functions that benefit from SIMD operation are high level language compilation and deadlock detection and avoidance. The PEPE system has been used to compile parallel FORTRAN programs, using both a "horizontal" and a "vertical" approach [7]. The horizontal approach involves multiple data stream operation on a single program statement at a time, while the vertical approach compiles many statements in parallel.

Deadlock detection algorithms generally involve some systematic search over a set of states for all processes and all resources to determine if there exists some sequence of resource allocations and deallocations that will allow the system to continue operation. Avoidance algorithms must perform a similar search to determine a resource allocation policy. In both cases, it would appear that the associative processor approach could substantially reduce the involved search time.

A second application for which the MAP system can be efficiently programmed comes from numerical mathematics. The example program presented here is rather simple compared to some programs that could be written to solve problems in this area, but the program does illustrate the methodology [10]. The flowchart given in Figure 3 solves a linear system of equations, $Ax = b$, using the Gauss-Jordan method with maximum pivoting. This method is chosen over the usual Gauss elimination with back substitution, since the approach given here actually results in fewer operations than the back substitution method. Each column of the coefficient matrix is stored in a single PE memory, along with an initial column number designation, ICOLNO, and a current column number designation, CCOLNO; (see boxes 1-3). Box 4 finishes the sequential loading process by storing the b-vector in the last PE memory, and initializes a row/column counter, CURRENTELT. In Box 5, all PEs containing column vectors to the right of the "current column" are activated (on the first loop through boxes 5-9, all PEs would be activated). To choose the maximum pivotal element, the program will load each accumulator with the corresponding scalar row-by-row. After each row is loaded, the maximum absolute value is chosen from the accumulators, and this maximum can be compared with any maximum obtained from a previous row. Whenever a potentially maximum element is chosen, the value, row designation, and current column numbers are stored in the main memory. Once the maximum element has been determined, a row exchange can be carried out by activating all PEs and then doing a simple load-store exchange sequence. Column exchanges take place by exchanging CCOLNO contents. In box 8, with all PEs having CCOLNO greater than or equal the CURRENT ELT, coefficients off the diagonal can be eliminated. Once the coefficient matrix has been reduced to an identity matrix, the resulting solution to the system can be obtained by using ICOLNO and CCOLNO to determine the column transformations that were used.

The third program illustrates an application from the field of combinatorial mathematics. This program reads a graph composed of nodes and arcs with associated weights (see Figure 4). A path from the initial node to the final node is determined such that the sum of the weights on the arcs of the path is minimal. Again, the program is sequential while loading the graph description into the PEs. Each node in the graph is allocated a PE and the following information is stored in the PE memory: the node identification, a list of all nodes to which this node is connected, a list with elements corresponding to the previous list which denotes the weight associated with the directed arc. The algorithm initially chooses the terminal node as a distinguishing element in the graph (box 4). It then determines which nodes have directed arcs leading into the distinguished node; (this causes a sequential loop in the program that is bounded by the maximum number of connections leading from any node). The set of all nodes leading into the distinguished elements can then be used to compute new estimates to a minimum path (box 6). Choosing the resulting PE with the current minimum path length (weight) as the distinguished element, the above operations are continued until the initial node is chosen as the distinguished element.

Other application areas for which the MAP system might be useful include linear programming, radar tracking (as discussed in several other array processor descriptions [3,8,12]), artificial intelligence applications that involve large searches through trees, etc. We have yet to explore these areas, although subsequent work includes writing sample MAP programs for some of these areas.

Some Contrived Programs

As was mentioned previously, there are two reasons for writing and testing programs for the MAP system: To explore application areas for the machine and to provide actual programs to exercise the various components of the machine. In the preceding section, a few examples were provided to illustrate the utility of the architecture. In this section some programs are discussed that have been contrived in order to exercise individual components of the machine. It is necessary to manufacture such programs, since the sample application programs do not necessarily represent all characteristics of all classes of programs that could be executed on a MAP system. This approach of using "synthetic programs" is motivated by Buchholz, although his synthetic programs were produced in order to gain an approximation of the performance of two different machines on a set of jobs that required a large amount of file maintenance [5].

The critical portions of the MAP architecture are the main memory design and the CU-PE bus system design. The main memory system has crucial interfaces between itself and the I/O system as well as between itself and the control units. Although we do not view the MAP system as being useful for programs requiring a high frequency of input or output operations (relative to computational effort), it is desirable to exercise the I/O system interface with the main memory.

There are requirements for at least four distinct synthetic programs to test the MAP design: The first synthetic program is designed to test the control unit/main memory interface, and we shall describe it in some detail to explain the approach used in all of these programs. The parameters to this program must describe the frequency of execution of instructions that make data references to the main memory, the main memory addresses that are referenced, and the length of time for which the program should run. The frequency and address can be specified using a pair of distribution functions, (that may be sampled as the program executes), to compute the amount of time between memory references to data and the address involved in the data reference. In Figure 5 a flowchart of the memory reference program is given, where the boxes made up of dashed lines constitute artifact to the program compared to its specification. The synthetic program is to be executed on the interpreter, hence the artifact that appears in the flowchart can be removed by executing the corresponding code outside the realm of the interpreter. Provisions have been made in the interpreter to allow for this, by including a call to the monitoring routine at the completion of each instruction cycle simulated by the interpreter. Hence, the program is written to do normal initialization (such as request PEs, etc.) after which the monitor will perform the processing described in boxes 2,3,4, and either 5 or 7. At that point, the monitor returns control to the interpreter for the execution of MAP code corresponding to box 6 or 8, etc. The monitoring routine also gathers the data describing machine performance.

The second program should allow the designer to specify a distribution of input/output requests to be made by the program, as well as a service distribution to describe the number and length of records to be transmitted between the I/O system and the main memory. The contrived program will then simulate an I/O load, inserting appropriate computation time between I/O requests based on samples of the two distributions.

The third required program exercises the multiplexed data bus that connects the control units to the bus sectors (see Figure 1). This program will explicitly deal with PE activation and deactivation as well as the provision of an appropriate number of instructions that require data transmission over the bus system, (note that not all instructions use the data bus, and that those instructions that do transmit data do not require the bus for the duration of the instruction execution phase). In order to specify a load for this program, the user must provide distributions describing the frequency of activation/deactivation instructions, the number and identity of the PEs involved, and the frequency of instructions requiring that data bus. It is then possible to control the load on the data bus system in order to test the bus sector design and its parameters.

The last contrived program needed to test the architecture is similar to the previous program, except the PE allocation and deallocation operations must be taken into account.

The four synthetic programs have been designed to exercise various components of the architecture, but it is apparent that each program constitutes only a single instruction stream. Hence, the programs do not really provide an adequate test of the architecture, although they are helpful. The solution to this problem can be obtained in one of two ways: the interpreter can be rewritten to simulate multiple control units; or simpler models that incorporate multiple control units can be written, and these models can be tested with reference streams obtained from the interpreter. Our present approach is the latter one, since the interpreter program is already very large and time-consuming to execute. A discussion of the result of experiments involving the CU-memory and the CU-PE interfaces appears in another paper, [12].

Results and Conclusions

The utility of the associative array processor system in executing algorithms has been explored by writing some sample algorithms for various application areas. The scheduling algorithm indicates that there are good applications for the architecture in operating systems. The most pleasing observation to be made about this application area is that the multiple control units make it conceivable to have the operating system executed on the same hardware as the user programs. This provides a capability for graceful degradation in the event of certain hardware failures, and also encourages the design of hierarchical software systems. However, the current work on the operating system for MAP has not been very successful due in part to the absence of multiple control units in MAPSIM.

There is very little that needs to be said about the applicability of array-type processors to numerical problems, since there is a large body of information on this application produced as a by-product of the Illiac IV studies. The primary improvements to be made in algorithm execution by a MAP system are concerned with the use of the select register to save conditions encountered during a computation, hence allowing more convenient activation and deactivation of various subsets of the processing elements. Another consideration that may either hamper or help the MAP programmer when solving matrix problems in the absence of any PE address. This may be beneficial if PE-PE data exchanges are not of the "north-south-east-west" variety, but may be a hindrance if the exchanges are of the prescribed form.

The examples presented here point out the problem with implementing algorithms that require a significant amount of PE memory loading. The current architecture requires that this be a serial task, and the anonymity of PEs makes STARAN-type or Illiac IV-type I/O impossible. This restriction excludes many problems which might otherwise be solved using a vertical approach, e.g., a payroll program that simultaneously processed n employees.

A nice programming effect induced by the architecture is that parallel programs tend to have fewer branches than the corresponding sequential program. A branch point normally occurs because a data dependent condition forces one section of code to be executed for some executions and another section of code to be executed for others. In an array processor both sections will, in general, be executed for different subsets of PEs.

The synthetic programs are generally not very useful to test the single control unit execution. The primary purpose of such programs is to observe the machine reaction at shared resource areas, and no shared resources are included in MAPSIM. On the other hand, the monitored data from the synthetic programs has been quite valuable when used to drive the lower detail simulation programs that do model shared resources.

The most significant result to come from algorithm encoding and execution was in the area of instruction set analysis. The design premise behind instruction set selection was that the control unit should be a microprocessor so that a variety of machine instruction sets could be tried, without drastically affecting the basic hardware design. All instruction sets must then be implementable from the microinstructions. The first instruction set was built from our intuition of how an associative processor set should look, (we had no previous experience). This set included the usual arithmetic/logic instructions, some main memory instruction, some branch instructions, and a set of associative instructions. After we wrote a few programs in the first version of the instruction set, several weaknesses became apparent. For example, although we had provided an instruction to set a switch (in the PE) dependent upon accumulator value, two more associative instructions were required to find out how the switch had been set. The application programmer experience was fed back to the system architects, (in this case, the same people constituted both groups), and a new version of the instruction set resulted. This implied that parts of the interpreter had to be rewritten, although incorporating a new instruction set is not nearly as difficult as writing the first interpreter.

The current s[illegible] [illegible] that the [illegible] version of the interpreter is [illegible] [illegible]ned for production, while we are continuing to write a few programs [illegible] [illegible]. This is desirable in order to generate data for [illegible] simulation [illegible] [illegible] will be to [illegible] the programs to the new instruction set, and again monitor their execution to derive [illegible] data.

Aknowledgement

The work described in this paper would not have been possible without the financial support of the National Science Foundation under grant number GJ-42251. The author also acknowledges the contributions of Clarence Ellis to the early ideas in MAP and to Roger Arnold and Karl Williamson for their participation in this work.

References

1. Arnold, R.D., "Multi Associative Processor Systems Architecture," University of Colorado Department of Computer Science Technical Report No. CU-CS-051-74 (August 1974).

2. Barnes, G.H.; Brown, R.M.; Kato, M.; Kuck, D.J.; Slotnick, D.L.; and Stokes, R.A., "The Illiac IV Computer," IEEE Transactions on Computers, Vol. C-17, No. 8 (August 1968), pp. 746-757.

3. Batcher, K.E., "STARAN Parallel Processor System Hardware," AFIPS Proceedings of the NCC, Vol. 43 (1974), pp. 405-410.

4. Berra, P.B., "Some Problems in Associative Processor Applications to Data Base Management," AFIPS Proceedings of the NCC, Vol. 43 (1974), pp. 1-5.

5. Buchholz, W., "A Synthetic Job for Measuring System Performance," IBM Systems Journal, Vol. 8, No. 4 (1969), pp. 309-318.

6. Davis, E.W., "STARAN Parallel Processor System Software," AFIPS Proceedings of the NCC, Vol. 43 (1974), pp. 17-22.

7. Ellis, C.A., "Parallel Compiling Techniques," Proceedings of the ACM National Conference (1971).

8. Githens, J.A., "A Fully Parallel Computer for Radar Data Processing," NAECON 1970 Record (1970), pp. 290-297.

9. Lucas, H.C. Jr., "Performance Evaluation and Monitoring," ACM Computing Surveys, Vol. 3, No. 3 (September 1971), pp. 79-91.

10. Mirankar, W.L., "A Survey of Parallelism in Numerical Analysis," SIAM Review, Vol. 13, No. 4 (October 1971), pp. 524-547.

11. Nutt, G.J., "An Overview of the Multi Associative Processor Study," Proceedings of the ACM National Conference (1974).

12. Nutt, G.J., "The Analysis of Certain Critical Components of an Array Processor", submitted for publication.

13. Rudolph, J.A., "A Production Implementation of an Associative Array Processor," AFIPS Proceedings of the FJCC, Vol. 41, Pt. I (1972), pp. 229-241.

14. Strauss, J.C., "A Benchmark Study," AFIPS Proceedings of the FJCC, Vol. 41, Pt. II (1972), pp. 1225-1233.

15. Wulf, W.A. and Bell, C.G., "C.MMP -- A Multi-mini-processor," AFIPS Proceedings of the FJCC, Vol. 41, Pt. II (1972), pp. 765-777.

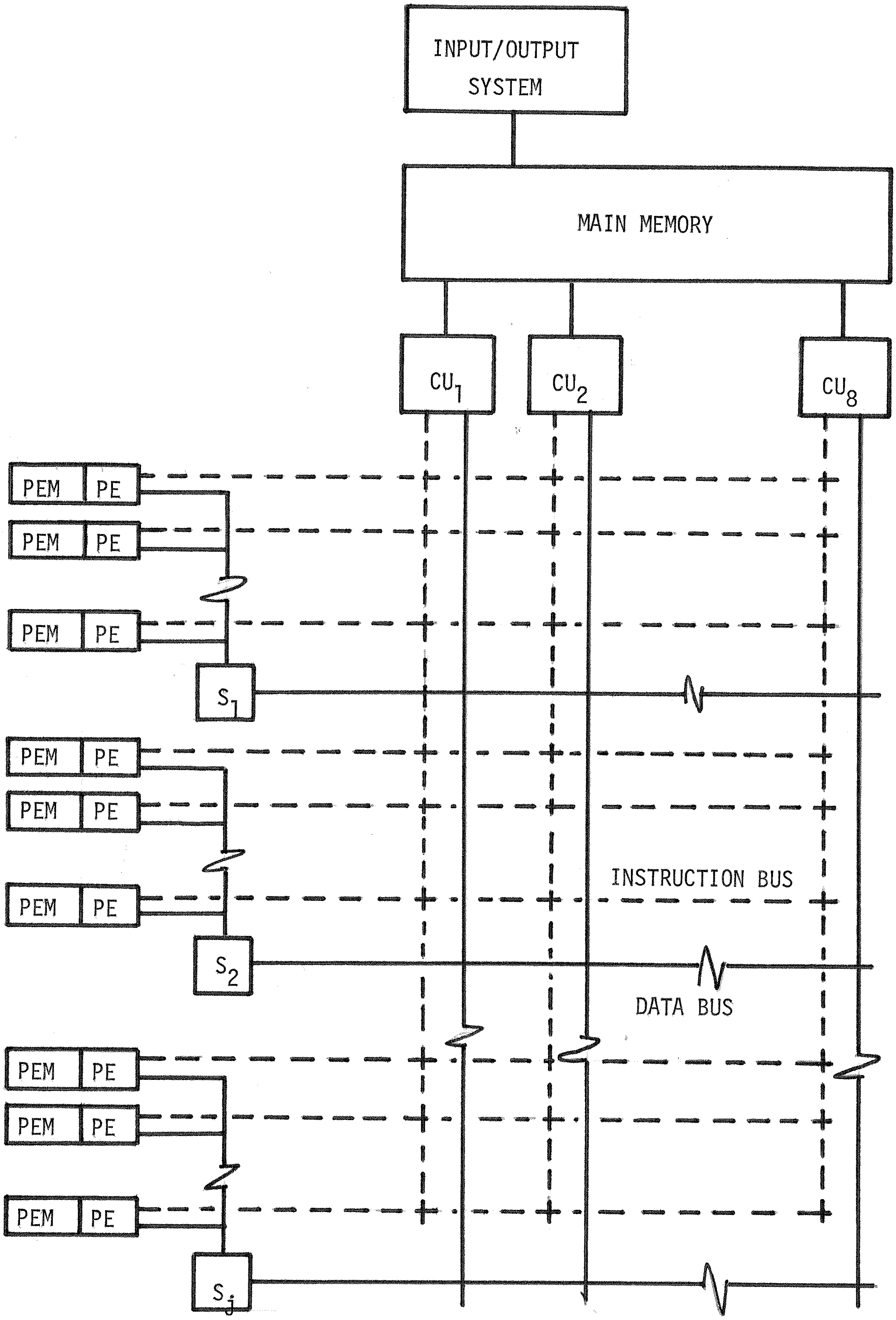
INPUT/OUTPUT SYSTEM
MAIN MEMORY
CU$_1$
CU$_2$
CU$_8$
PEM
PE
S$_1$
S$_2$
S$_j$
INSTRUCTION BUS
DATA BUS

FIGURE 1

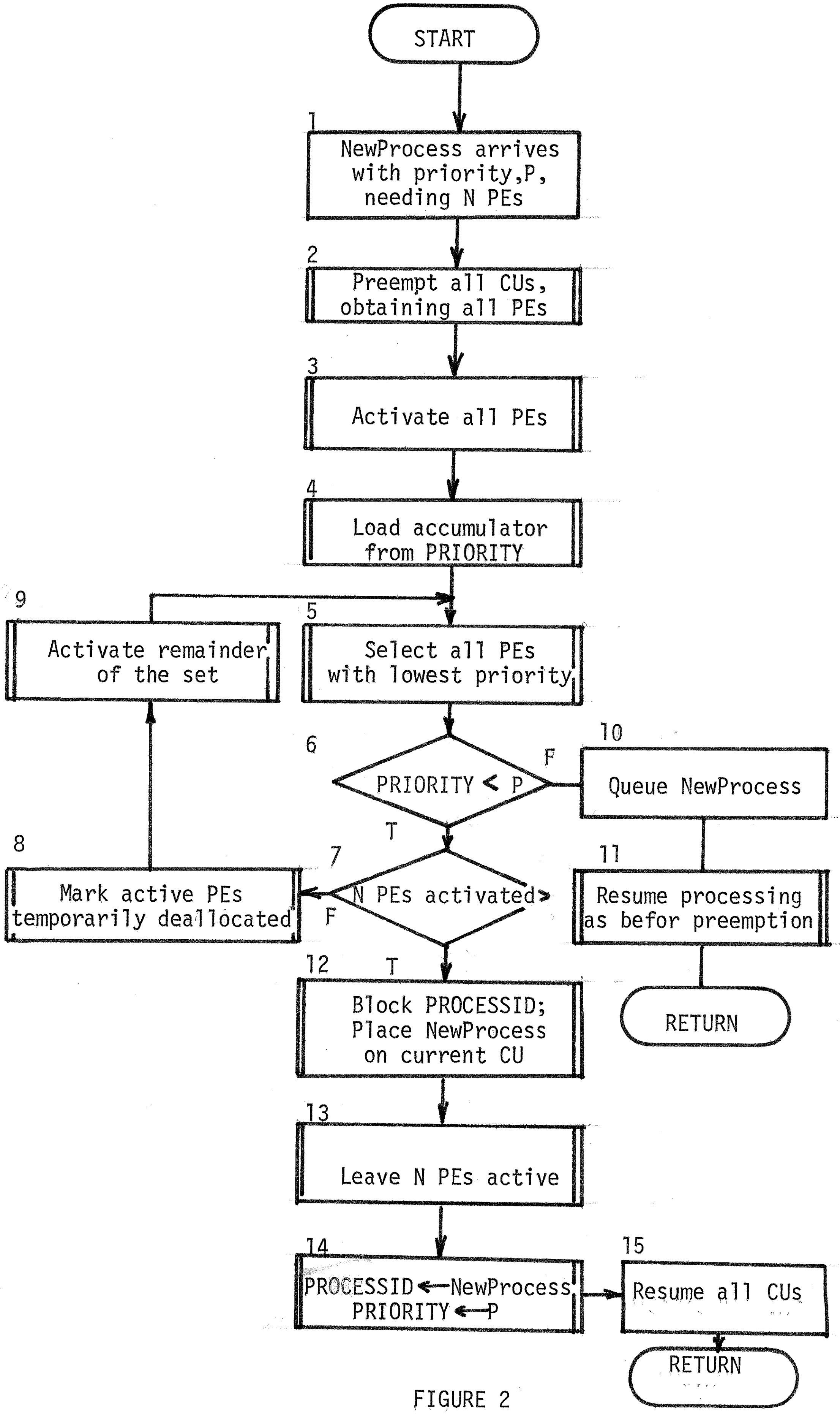
START
1
NewProcess arrives with priority,P, needing N PEs
2
Preempt all CUs, obtaining all PEs
3
Activate all PEs
4
Load accumulator from PRIORITY
9
Activate remainder of the set
5
Select all PEs with lowest priority
6
PRIORITY < P
F
10
Queue NewProcess
T
8
Mark active PEs temporarily deallocated
7
N PEs activated?
F
11
Resume processing as befor preemption
RETURN
12
T
Block PROCESSID; Place NewProcess on current CU
13
Leave N PEs active
14
PROCESSID ← NewProcess
PRIORITY ← P
15
Resume all CUs
RETURN

FIGURE 2

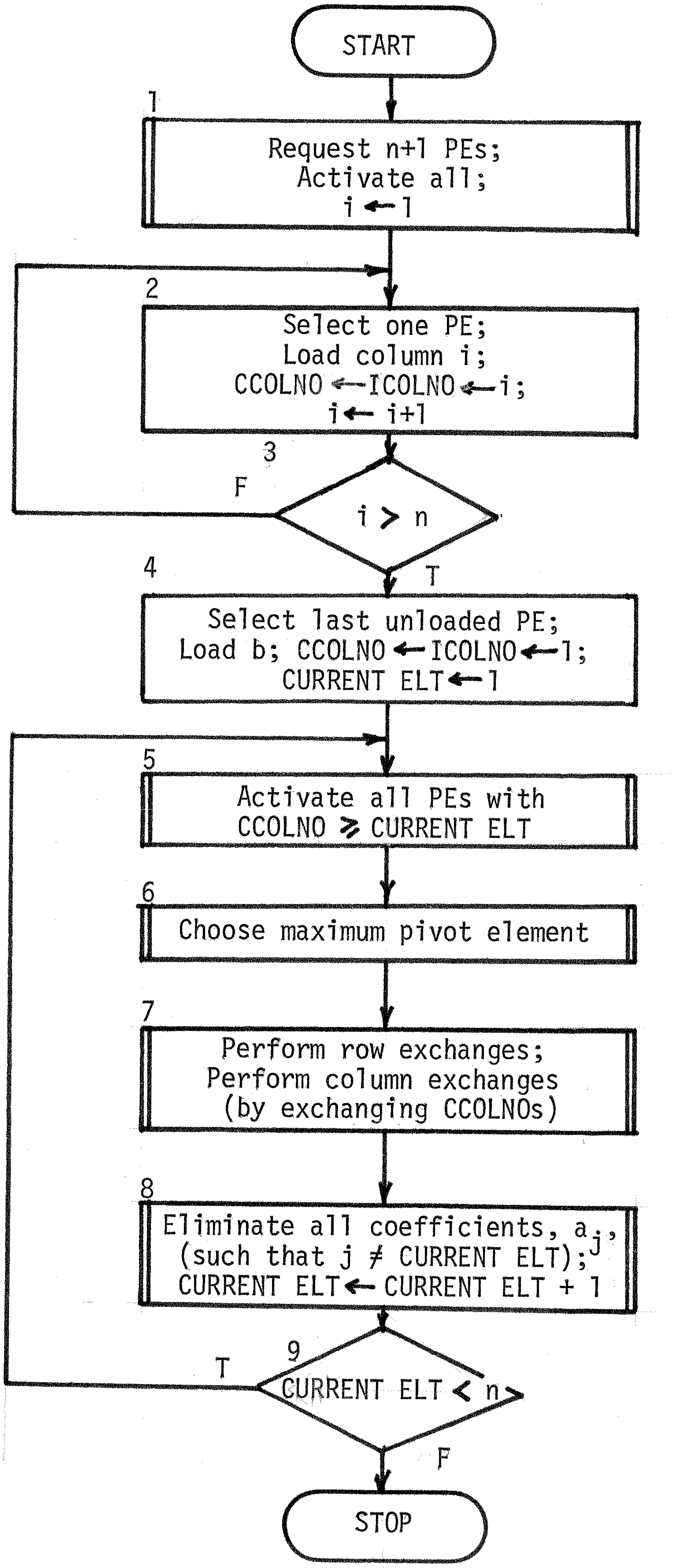
START
1
Request n+1 PEs;
Activate all;
i ← 1
2
Select one PE;
Load column i;
CCOLNO ← ICOLNO ← i;
i ← i+1
3
F
i > n
4
T
Select last unloaded PE;
Load b; CCOLNO ← ICOLNO ← 1;
CURRENT ELT ← 1
5
Activate all PEs with
CCOLNO ≥ CURRENT ELT
6
Choose maximum pivot element
7
Perform row exchanges;
Perform column exchanges
(by exchanging CCOLNOs)
8
Eliminate all coefficients, a_j,
(such that j ≠ CURRENT ELT);
CURRENT ELT ← CURRENT ELT + 1
9
T
CURRENT ELT < n
F
STOP

FIGURE 3

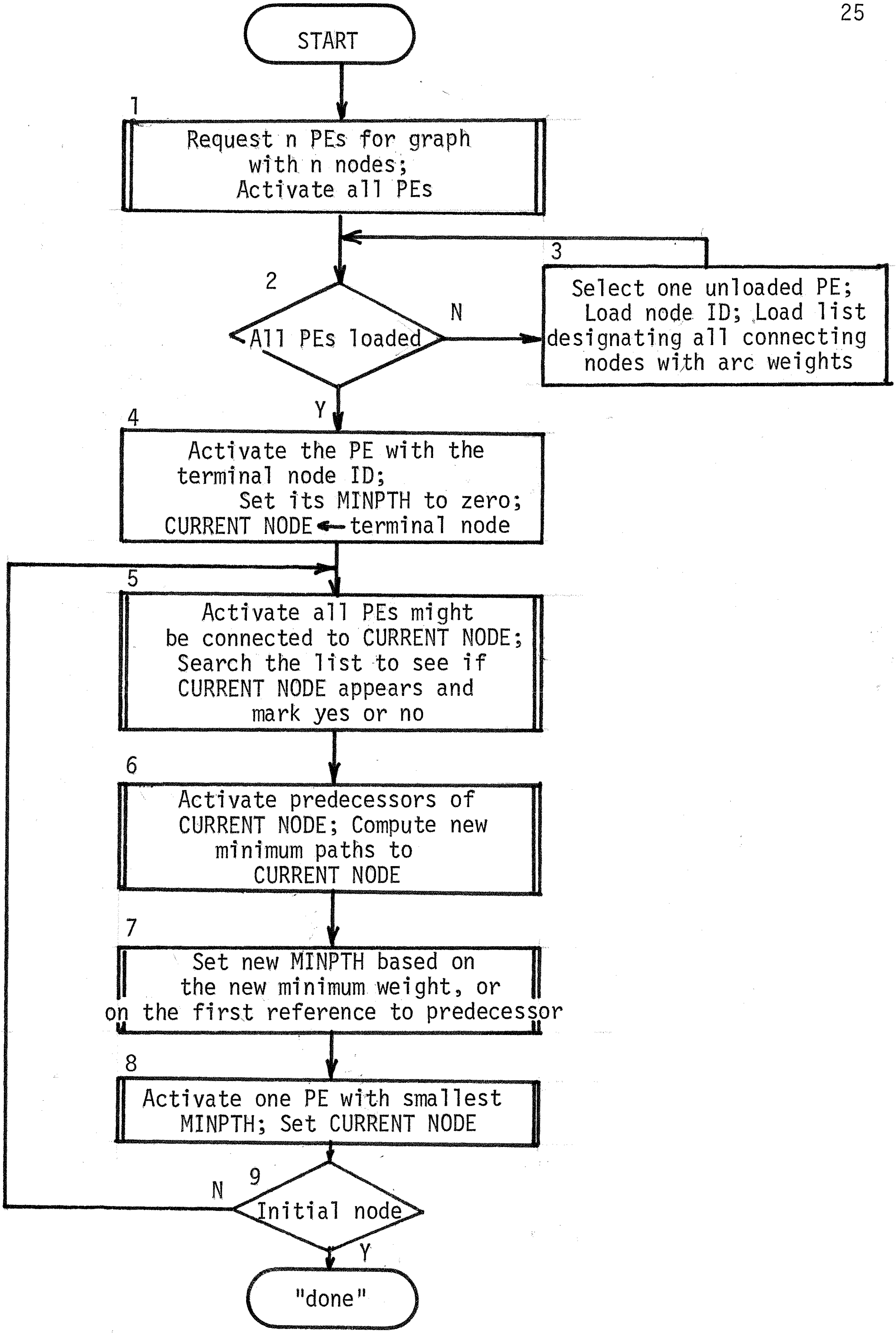
START
1
Request n PEs for graph with n nodes; Activate all PEs
2
All PEs loaded
N
3
Select one unloaded PE; Load node ID; Load list designating all connecting nodes with arc weights
Y
4
Activate the PE with the terminal node ID; Set its MINPTH to zero; CURRENT NODE ← terminal node
5
Activate all PEs might be connected to CURRENT NODE; Search the list to see if CURRENT NODE appears and mark yes or no
6
Activate predecessors of CURRENT NODE; Compute new minimum paths to CURRENT NODE
7
Set new MINPTH based on the new minimum weight, or on the first reference to predecessor
8
Activate one PE with smallest MINPTH; Set CURRENT NODE
9
N
Initial node
Y
"done"

FIGURE 4

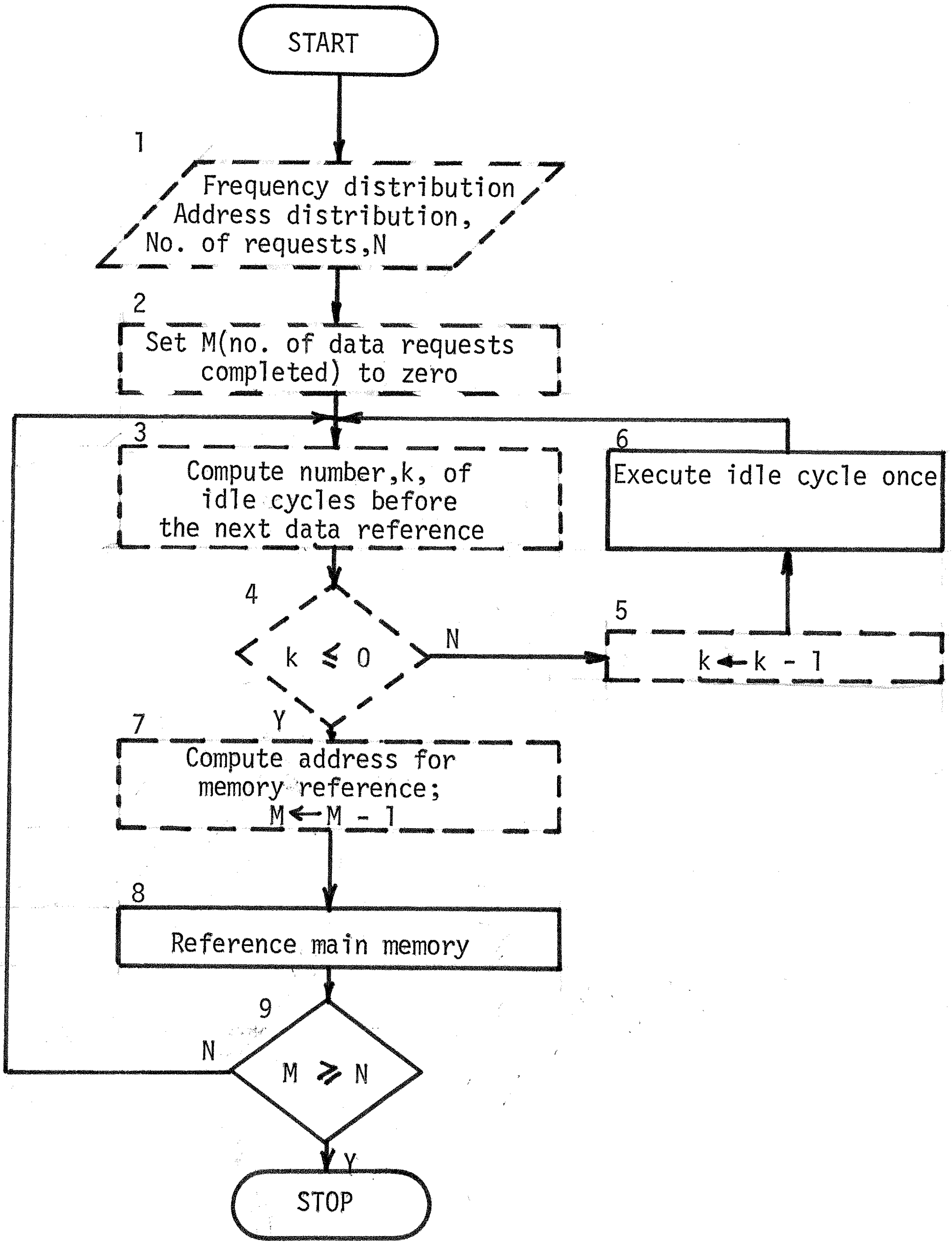
START
1
Frequency distribution
Address distribution,
No. of requests,N
2
Set M(no. of data requests
completed) to zero
3
Compute number,k, of
idle cycles before
the next data reference
6
Execute idle cycle once
4
k ≤ 0
N
5
k ← k - 1
Y
7
Compute address for
memory reference;
M ← M - 1
8
Reference main memory
9
N
M ≥ N
Y
STOP

FIGURE 5