Proc. 1979 Conf. on Information Sciences and Systems, Johns Hopkins Univ., Mar. 79.

A DISTRIBUTED OPERATING SYSTEM MACHINE*

T. N. Mudge

Systems Engineering Laboratory
Department of Electrical and Computer Engineering
and CICE Program
The University of Michigan
Ann Arbor 48109

ABSTRACT

This paper describes a forwarding algorithm suitable for a distributed multi-microprocessor realization of an operating system machine. A specific operating system will not be described. The algorithm simplifies the design of any system of computational tasks that exhibit a high degree of parallelism, by reducing the inter-task coordination and synchronization problem. It is an improvement of an algorithm given in [To]. Unlike the version in [To] it does not require an associative search, and thus retains its efficiency even when applied to large systems.

1. INTRODUCTION

This paper describes a forwarding algorithm suitable for a distributed multi-microprocessor realization of an operating system machine. A specific operating system will not be described. The algorithm simplifies the design of any system of computational tasks that exhibit a high degree of parallelism, by reducing the inter-task coordination and synchronization problem.

The execution of an operating system results in a collection of computational tasks which exhibit a high degree of parallelism, but whose interactions do not conform to a simple regular structure. Hence, coordination and synchronization become a significant part of the design problem. Our algorithm allows the designer to specify a target system as a list of tasks without concern for the synchronization problem. It is sufficient that if one task is to precede another, it precedes it in the list. At run time the forwarding algorithm automatically detects which tasks can go ahead based on the availability of their input data, thus the algorithm is data driven.

The advantages of the data driven approach to controlling systems can be summarized as follows: In conventional computer systems a computational task, such as the execution of an instruction, is carried out upon the receipt of a control signal from a centralized controller (consider the design of a microprogrammed computer). To make sure that the instruction executes with the correct data requires the controller to have knowledge of the data flow. In the case of parallel processing the number of possibilities for data flow patterns becomes very large, thus making the controller correspondingly complex. By allowing the data to drive instruction execution, and hence in a sense create its own control signals dynamically, this complexity, which eventually becomes overwhelming in large conventional systems, is much reduced in similar data driven ones. (For a discussion of the relationship of data flow concepts to operating systems see [D].)

2. THE FORWARDING ALGORITHM

Figure 1a shows a diagram of a multi-microprocessor system appropriate for our forwarding algorithm. The major components are memory modules, processors, and interconnection logic. The algorithm does not depend on the memories being high speed or low speed, nor on them being random access or sequential access. However, to implement the forwarding algorithm efficiently each memory module needs a limited amount of processing capability for the tag handling that forms the basis of the algorithm. The only constraint on the processors is that they be able to perform the very limited amount of processing required of them by the algorithm, and that there be at least one processor that is capable of distributing tasks from the list of tasks that form the target operating system. This implies that at least one processor have some global knowledge of the system configuration. Otherwise, the algorithm does not depend on the processors being of a particular type. They may be full instruction-

* This work was supported in part by the National Science Foundation under Grant NSF-ENG-78-5779.

A DISTRIBUTED OPERATING SYSTEM MACHINE*

T. N. Mudge

Systems Engineering Laboratory
Department of Electrical and Computer Engineering
and CICE Program
The University of Michigan
Ann Arbor 48109

ABSTRACT

This paper describes a forwarding algorithm suitable for a distributed multi-microprocessor realization of an operating system machine. A specific operating system will not be described. The algorithm simplifies the design of any system of computational tasks that exhibit a high degree of parallelism, by reducing the inter-task coordination and synchronization problem. It is an improvement of an algorithm given in [To]. Unlike the version in [To] it does not require an associative search, and thus retains its efficiency even when applied to large systems.

1. INTRODUCTION

This paper describes a forwarding algorithm suitable for a distributed multi-microprocessor realization of an operating system machine. A specific operating system will not be described. The algorithm simplifies the design of any system of computational tasks that exhibit a high degree of parallelism, by reducing the inter-task coordination and synchronization problem.

The execution of an operating system results in a collection of computational tasks which exhibit a high degree of parallelism, but whose interactions do not conform to a simple regular structure. Hence, coordination and synchronization become a significant part of the design problem. Our algorithm allows the designer to specify a target system as a list of tasks without concern for the synchronization problem. It is sufficient that if one task is to precede another, it precedes it in the list. At run time the forwarding algorithm automatically detects which tasks can go ahead based on the availability of their input data, thus the algorithm is data driven.

The advantages of the data driven approach to controlling systems can be summarized as follows: In conventional computer systems a computational task, such as the execution of an instruction, is carried out upon the receipt of a control signal from a centralized controller (consider the design of a microprogrammed computer). To make sure that the instruction executes with the correct data requires the controller to have knowledge of the data flow. In the case of parallel processing the number of possibilities for data flow patterns becomes very large, thus making the controller correspondingly complex. By allowing the data to drive instruction execution, and hence in a sense create its own control signals dynamically, this complexity, which eventually becomes overwhelming in large conventional systems, is much reduced in similar data driven ones. (For a discussion of the relationship of data flow concepts to operating systems see [D].)

2. THE FORWARDING ALGORITHM

Figure 1a shows a diagram of a multi-microprocessor system appropriate for our forwarding algorithm. The major components are memory modules, processors, and interconnection logic. The algorithm does not depend on the memories being high speed or low speed, nor on them being random access or sequential access. However, to implement the forwarding algorithm efficiently each memory module needs a limited amount of processing capability for the tag handling that forms the basis of the algorithm. The only constraint on the processors is that they be able to perform the very limited amount of processing required of them by the algorithm, and that there be at least one processor that is capable of distributing tasks from the list of tasks that form the target operating system. This implies that at least one processor have some global knowledge of the system configuration. Otherwise, the algorithm does not depend on the processors being of a particular type. They may be full instruction-

* This work was supported in part by the National Science Foundation under Grant NSF-ENG-78-5779.

set processors, I/O processors, or special purpose processors such as floating point units. Also, since the algorithm facilitates a high degree of parallelism among the target system's tasks, the interconnection logic should be capable of supporting high bandwidth intra-system communication. In general, this rules out a single bus. Furthermore, since this intra-system communication is determined by destination addresses (see later), some form of interconnection logic that routes data by address is appropriate. Thurber gives a survey of such logic in [Th].

In order to explain the operation of the forwarding algorithm it is first necessary to clarify what is meant by a task. For our purposes the general form of a task is as follows:

R <- P(D)

Where, R is the set of range memory locations used by the task, i.e. the set of locations that the task writes to.
P is the (virtual) processor that runs the task.
D is the set of domain memory locations of the task, i.e. the set of locations that the task reads from.

In the simplest case a task can be a register transfer operation, in more complicated cases a task may be a program or collection of programs.

The following list of tasks will be used to illustrate the algorithm.

1. A <- P1(B)
2. C <- P2(A)
3. D <- P3(C)
4. C <- P4(E)

For the sake of this explanation we will consider that one of the processors issues these tasks sequentially, as follows:

Task 1 is issued: The contents of memory region B is moved to processor P1 and P1 is set to work on it (see Figure 1a). A tag (①) is placed at the beginning of memory region A to reserve it for the result of P1(B). At the same time a list is created with A as the first element of the list (here A denotes the necessary information to define the region in memory named A), so that P1 has a record of where to send its output. This list is called the destination list of P1. Task 2 may now be issued. Note that, in general, task 1 will still be in operation.

Task 2 is issued: The contents of memory region A is moved to processor P2 (see Figure 1b), i.e. the tag ① is move to P2. Since a tag, and not data, was found it is necessary to go to the processor identified by ① (i.e. P1) and append to its destination list the name P2, so that the as-yet-uncomputed data represented by ① will be sent to P2. In other words, when P1 has finished the first task, it will now know to send its output to both A and P2. At the same time a tag (②) is placed at the beginning of memory region C to reserve it for the result of P2(A), and C (here C denotes the necessary information to define the region in memory named C) is appended to the destination list of P2, so that P2 has an up to date record of where to send its output. Task 3 may now be issued.

Since the only data moved to P2 is a tag (①), P2 is blocked from further use until P1 completes. In general, this inefficiency can be overcome by using virtual processors. These can be implemented by having each real processor maintain several sets of the following: a request register (suitable for holding a tag), and some memory for a destination list. Each set corresponds to a virtual processor. When a virtual processor has real data to work on, the task is accomplished by the real processor. In a properly designed system, each real processor should be busy most of the time without being a bottleneck to system performance.

Task 3 is issued: The contents of memory region C is moved to processor P3 (see Figure 1c). Since this is the tag ② the name P3 is appended to the destination list of P2, so that P2 can forward its result directly to P3 as well as C. At the same time a tag (③) is placed at the beginning of memory region D to reserve it for the result of P3(C), and D is appended to the destination list of P3. Task 4 may now be issued.

Task 4 is issued: The contents of memory region E is moved to P4, and P4 is set to work on it (see Figure 1d). A tag (④) is placed at the beginning of memory region C to reserve it for the result of P4(E). However, C already contains a tag (②), so the following additional steps must be taken: Before writing ④ over ②, the old tag (i.e. ②) is used to indicate the processor (in this case P2) whose destination list now contains an entry, C, that no longer needs that processor's result. In response to this the entry C in P2's destination list is deleted. In Figure 1d the

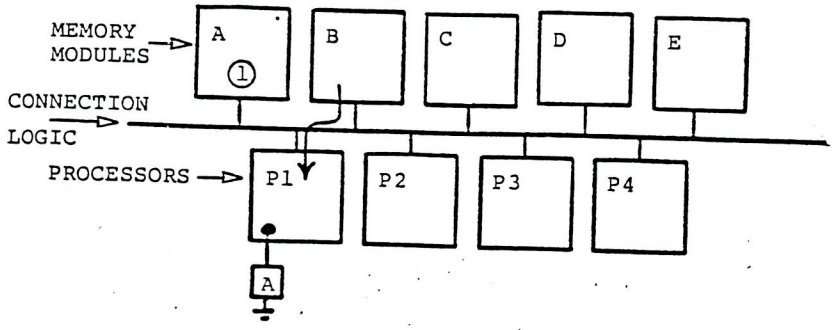


Figure 1a: Task 1 is issued.

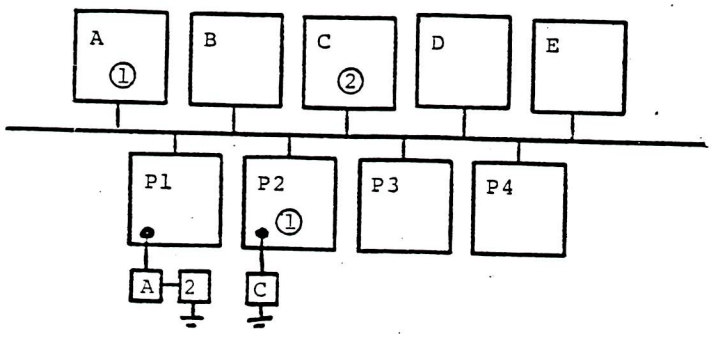


Figure 1b: Task 2 is issued.

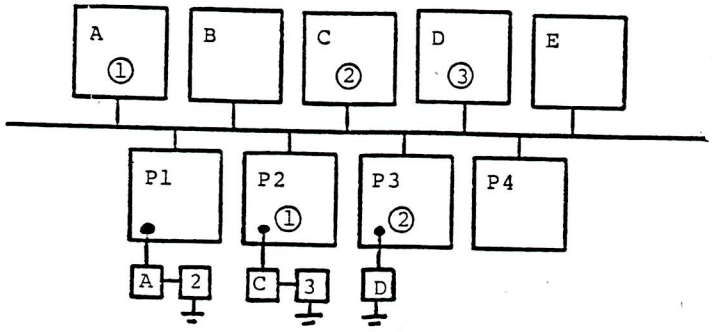


Figure 1c: Task 3 is issued.

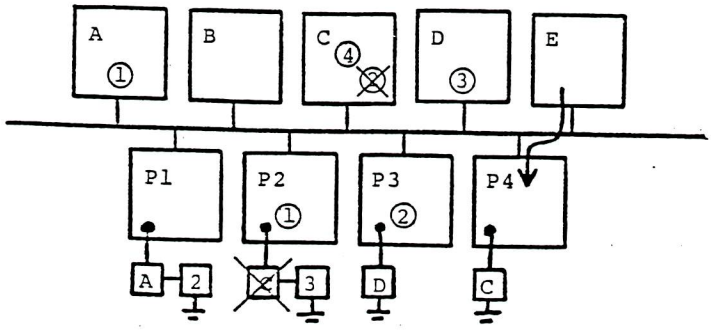


Figure 1d: Task 4 is issued.

Figure 1: Task Issuing.

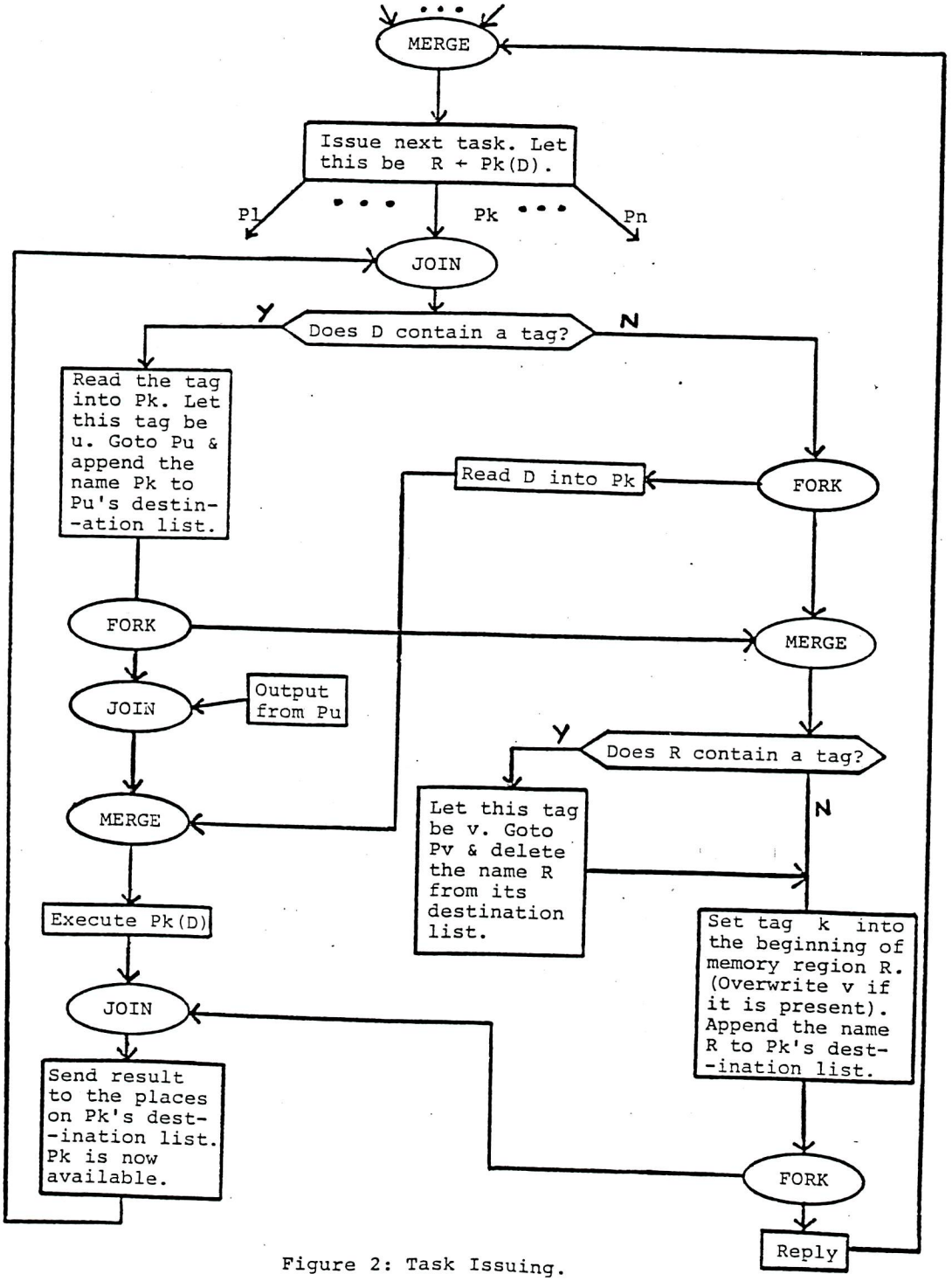
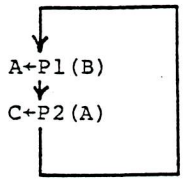


Figure 2: Task Issuing.



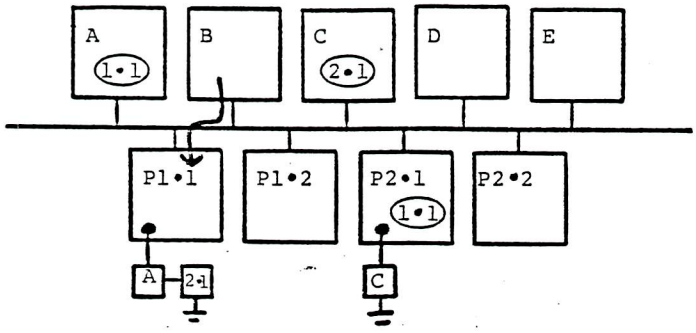


Figure 3a:
Situation after
the first iteration.

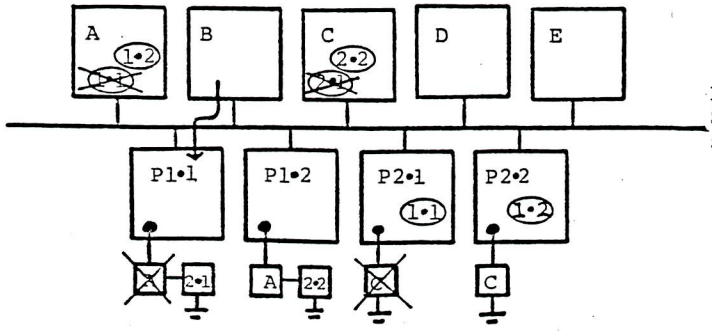


Figure 3b:
Situation after the
second iteration

Figure 3: Iteration.

overwriting and deletion of an item are denoted by an X through that item. Finally, the destination list of P4 is updated by appending C to it.

Concurrently with this task issuing, processors may be completing tasks that were previously issued to them. When a processor completes a task its result is output (forwarded) to all those places on its destination list. The list is deleted and the tags at the result destinations are deleted also. Figure 1 does not show any of the four tasks completing. It assumes all four are issued before any of them get done.

A flowchart of the decision process associated with the issuing of tasks is shown in Figure 2. It is self explanatory except for the following operators: FORK, JOIN, and MERGE. The FORK operator transmits control on both of its outputs upon receiving it on its input. The JOIN operator transmits control on its output only after it has received it on both of its inputs. Lastly, the MERGE operator transmits control on its output whenever it receives it on one of its inputs. In a well-formed flowchart, control can pass along at most one input of a MERGE module at any one time.

The operation of the forwarding algorithm can be summarized by noting the following points. During its progress the algorithm dynamically lays out a linked list over the processors. This list represents the necessary precedence between tasks. It achieves this by a tagging technique which allows memory regions and processors to be reserved until their intended contents and operands are available. In this way potential parallelism among the tasks is scheduled dynamically. The processors start their operations only upon the receipt of their operands, thus the dynamic scheduling is effected by the availability of data - hence the target system is data driven.

Figure 3 shows the situation that arises when tasks are executed iteratively. The iterations unwind quite naturally to fill the number of virtual processors available (assume P1.1 P1.2, etc., are virtual P1 processors). Issuing is held up by the lack of suitable processors. As with Figure 1, no tasks are shown completing.

3. DISCUSSION OF THE ALGORITHM

The above algorithm is similar to the forwarding algorithm found in the floating point unit of the S/360 model 91 [To]. This last algorithm was shown to be optimal by Keller (see [K]), in the sense that it ensures its sequence of tasks will execute with maximal parallelism. Because of the similarity to our forwarding algorithm, this result holds for our algorithm also. Thus any target systems designed using our algorithm will exhibit maximal parallelism among the tasks that make up that system.

The key difference between our algorithm and Tomasulo's is that our algorithm does not require processors to make an associative search to determine where to send their output. Instead the processors know exactly where to send their output by refering to the entries in their destination lists. The idea of using a destination list to avoid having to make an associative search was first suggested by Keller in [K]. However, he did not incorporate it into an algorithm, because at that time there was no solution to the problem of updating destination lists when tags are overwritten. Our algorithm solves this problem by first examining the tag to be overwritten (see Figure 2) and then using it to point to the processor whose destination list must be modified. Once the associative search component of Tomasulo's algorithm has been eliminated it can be applied efficiently across a complete system. A precursor to the forwarding algorithm presented in this paper can be found in [M1]. It was first suggested in [M2].

4. REFERENCES

[D] Denning, P.J., "Operating Systems Principles for Data Flow Networks", Computer Magazine, pp. 86-96, Jul. 78.

[K] Keller, R.M., "Look-Ahead Processors," Computing Surveys, Vol. 7, No. 4, pp. 177-195, Dec. 75.

[M1] Mudge, T.N.,"A Data Driven Computer Architecture", Proc. 1978 Conf. on Information Sciences and Systems, pp. 365-370, Mar. 78.

[M2] Mudge, T.N.,"A Distributed Operating System Machine", Proc. Louisiana Computer Exposition pp. 143-166, Mar. 79.

[Th] Thurber, K.J.,"Interconnection Networks - A Survey and Assessment", Proc. NCC, pp. 909-914, Jun. 74.

[To] Tomasulo, R.M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Jour. R&D, Vol. 11, No. 1, pp. 25-33, Jan. 67