Multiprocessor Scheduling with the Aid of Network Flow Algorithms

HAROLD S. STONE, MEMBER, IEEE

Abstract—**In a distributed computing system a modular program must have its modules assigned among the processors so as to avoid excessive interprocessor communication while taking advantage of specific efficiencies of some processors in executing some program modules. In this paper we show that this program module assignment problem can be solved efficiently by making use of the well-known Ford-Fulkerson algorithm for finding maximum flows in commodity networks as modified by Edmonds and Karp, Dinic, and Karzanov. A solution to the two-processor problem is given, and extensions to three and *n*-processors are considered with partial results given without a complete efficient solution.**

Index Terms—**Computer networks, cutsets, distributed computers, Ford-Fulkerson algorithm, load balancing, maximum flows.**

I. Introduction

DISTRIBUTED processing has been a subject of recent interest due to the availability of computer networks such as the Advanced Research Projects Agency Network (ARPANET), and to the availability of microprocessors for use in inexpensive distributed computers. Fuller and Siewiorek [8] characterize distributed computer systems in terms of a coupling parameter such that array computers and multiprocessors are tightly coupled, and computers linked through ARPANET are loosely coupled. In loosely coupled systems the cost of interprocessor communication is sufficiently high to discourage the execution of a single program distributed across several computers in the network. Nevertheless, this has been considered for ARPANET by Thomas and Henderson [17], Kahn [12], and Thomas [16].

In this paper we focus on the type of distributed system that Fuller and Siewiorek treat as systems with intermediate coupling. A primary example of this type of system is the multiprocessor interface message processor for ARPANET designed by Bolt, Beranek, and Newman (Heart *et al.* [10]). Two-processor distributed systems are widely used in the form of a powerful central processor connected to a terminal-oriented satellite processor. Van Dam [18] and Foley *et al.* [7] describe two-processor systems in which program modules may float from processor to processor at load time or during the execution of a program. The ability to reassign program modules to different processors in a distributed system is essential to make the best use of system resources as programs change from one computation phase to another or as system load changes.

Manuscript received November 7, 1975; revised July 9, 1976. This work was supported in part by the National Science Foundation under Grant DCR 74-20025.

The author is with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01002.

In this paper we treat the problem of assigning program modules to processors in a distributed system. There are two kinds of costs that must be considered in such an assignment. In order to reduce the cost of interprocessor communication, the program modules in a working set should be coresident in a single processor during execution of that working set. To reduce the computational cost of a program, program modules should be assigned to processors on which they run fastest. The two kinds of assignments can be incompatible. The problem is to find an assignment of program modules to processors that minimizes the collective costs due to interprocessor communication and to computation. We show how to make such an assignment efficiently by using methods of Ford and Fulkerson [6], Dinic [3], Edmonds and Karp [4], and Karzanov [13] that have been developed for maximizing flows in commodity networks. The two-processor assignment problem can be stated as a commodity flow problem with suitable modifications. We show how to construct a graph for the *n*-processor problem for which a minimal cost cut is a minimal cost partition of the graph into *n* disjoint subgraphs. We give partial results for finding the minimal cost cut but, unfortunately, an efficient solution has not yet been obtained.

In Section II of this paper we discuss the computer model in more detail. Section III reviews the essential aspects of commodity flow problems. The main results of this paper appear in Section IV, in which we show how to solve the two-processor problem, and in Section V, in which we consider the *n*-processor problem. A brief summary with an enumeration of several questions for future research appears in the final section.

II. The Multiprocessor Model

We use a model of a multiprocessor based on the multiprocessor system at Brown University [18], [15]. In this model, each processor is an independent computer with full memory, control, and arithmetic capability. Two or more processors are connected through a data link or high-speed bus to create a multiprocessor system. The processors need not be identical; in fact, the system cited above has a System/360 computer linked to a microprogrammable minicomputer. This much of the description of the model fits many systems. The distinguishing feature of the multiprocessor under discussion is the manner in which a program executes. In essence, each program in this model is a serial program, for which execution can shift dynamically from processor to processor. The two processors may both be multiprogrammed, and may execute concurrently on different programs, but may not execute concurrently on the same program.

A program is partitioned into functional modules some of

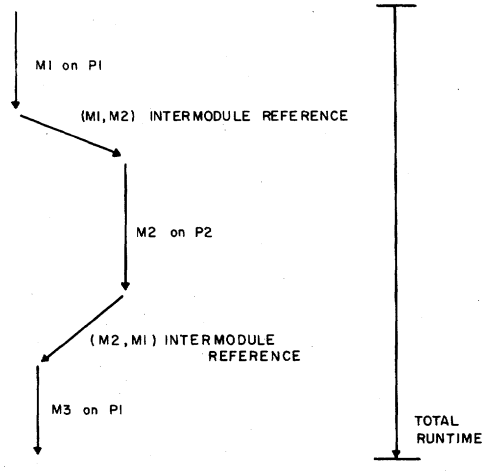

Fig. 1. Module $M1$ calls module $M2$ on a distributed computer system.

which are assigned to particular processors, the remainder of which are permitted to "float" from processor to processor during program execution. Some modules have a fixed assignment because these modules depend on facilitites within their assigned processor. The facilities might be high-speed arithmetic capability, access to a particular data base, the need for a large high-speed memory, access to a specific peripheral device, or the need for any other facility that might be associated with some processor and not with every processor.

When program execution begins, the floating modules are given temporary assignments to processors, assignments that reflect the best guess as to where they should reside for maximum computation speed. As execution progresses, the floating modules are free to be reassigned so as to improve execution speed and to reflect the changing activity of the program. Here we presume that interprocessor communication incurs relatively high overhead, whereas computations that make no calls to other processors incur zero additional overhead. Fig. 1 shows the sequence of events that occur when module $M1$ on processor $P1$ calls module $M2$ on processor $P2$. The program state shifts from processor $P1$ to processor $P2$, and returns to processor $P1$ when module $M2$ returns control to $M1$. Instead of transferring state to $P2$, it may be better to colocate $M1$ and $M2$ on the same computer. The choice depends on the number of calls between $M1$ and $M2$, and on the relationship of $M2$ and $M1$ to other modules.

The advantage of this type of approach over other forms of multiprocessing is that it takes advantage of program locality. When activity within a program is within one locality, in ideal circumstances all the activity can be forced to reside within one processor, thereby eliminating conflicts due to excessive use of shared resources. The locality need not be known beforehand, because program modules tend to float to processors in which their activity is highest. Another type of multiprocessor is exemplified by C.mmp (Wulf and Bell [19]) in which up to 16 minicomputers and 16 memories are linked together through a crossbar. The coupling among processors in this system is very tight because potentially every memory fetch accesses memory through the crossbar. Conflicts are potentially high because two or more processors may attempt

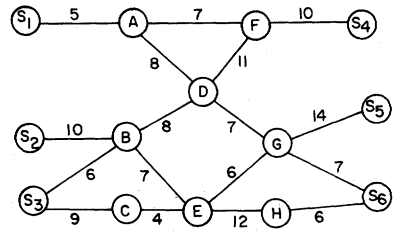

Fig. 2. Commodity flow network.

to access the same memory module simultaneously for an extended period of time.

In the present model, we eliminate the costly crossbar, the degradation on every memory access due to delays through the crossbar, and the potential for conflicts on every memory access to further degrade memory access. If this model is more effective than the crossbar architecture of C.mmp, it is because the costly interprocessor calls incurred in practice occur rarely due to the ability to assign program working sets to a single processor. Whether or not this is the case has not been settled. The question of the relative effectiveness of this architecture and the crossbar architecture has barely been investigated at this writing. The memory assignment algorithm described in the following sections indicates that it is possible to control a distributed computer of the type under study here reasonably efficiently, thereby opening the door to further and deeper studies of this idea.

III. Maximum Network Flows and Minimum Cutsets

In this section we review the commodity flow problem for which Ford and Fulkerson formulated a widely used algorithm [6]. The algorithm is reasonably fast in practice, but the original formulation of the algorithm is subject to rather inefficient behavior in pathological cases. Edmonds and Karp [4] have suggested several different modifications to the Ford-Fulkerson algorithm so as to eliminate many of the inefficiencies, at least for the pathological examples, and to obtain improved worst-case bounds on the complexity of the algorithm. Dinic [3] and Karzanov [13] have derived other ways to increase speed.

In this section we shall describe the problem solved by the Ford-Fulkerson algorithm as modified by Edmonds and Karp, but we shall refer the reader to the literature for descriptions of the various implementations of the algorithm. The important idea is that we can treat an algorithm for the solution of the maximum flow problem as a "black box" to solve the module assignment problem. We need not know exactly what algorithm is contained in the black box, provided that we know the implementation is computationally efficient for our purposes.

The maximum flow problem is a problem involving a commodity network graph. Fig. 2 shows the graph of such a network.

The nodes labeled S_1, S_2, and S_3 are source nodes, and the nodes labeled S_4, S_5, and S_6 are sink nodes. Between these subsets of nodes lie several interior nodes with the entire collection of nodes linked by weighted branches. Source nodes represent production centers, each theoretically capable

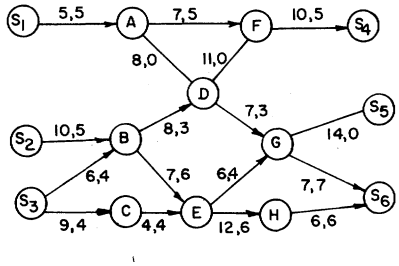

Fig. 3. Flow in a commodity flow network.

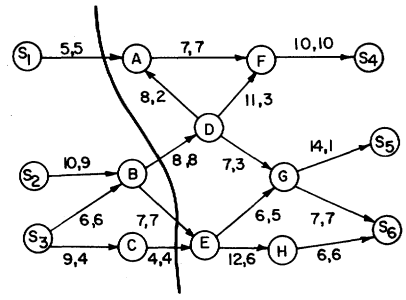

Fig. 4. Maximum flow and minimum cut in a commodity flow network.

of producing an infinite amount of a specific commodity. Sink nodes represent demand centers, each of which can absorb an infinite amount of the commodity. The branches represent commodity transport linkages, with the weight of a branch indicating the capacity of the corresponding link.

A *commodity flow* in this network is represented by weighted directed arrows along the branches of the network, with the weight of the arrow indicating the amount of the flow on that branch, and the direction of the arrow indicating the direction of the commodity flow. Fig. 3 shows a commodity flow for the graph in Fig. 2. Each arrow in Fig. 3 carries a pair of numbers, the first of which is the capacity of that branch and the second of which is the actual flow on that branch. A *feasible* commodity flow in this network is a commodity flow originating from the source nodes and ending at the sink nodes such that: 1) at each intermediate node, the sum of the flows into the node is equal to the sum of the flows out of the node; 2) at each sink node the net flow into the nodes is nonnegative, and at each source node the net flow directed out of the node is nonnegative; and 3) the net flow in any branch in the network does not exceed the capacity of that branch. Note that the flow in Fig. 3 is a feasible flow according to this definition. The three constraints in this definition guarantee that interior nodes are neither sources nor sinks, that source nodes and sink nodes are indeed sources and sinks, respectively, and that each branch is capable of supporting the portion of the flow assigned to it.

The *value* of a commodity flow is the sum of the net flows out of the source nodes of the network. Because the net flow into the network must equal the net flow out of the network, the value of a commodity flow is equal to the sum of net flows into the sink nodes. The value of the flow in Fig. 3 is 18. A *maximum flow* is a feasible flow whose value is maximum among all feasible flows. Fig. 4 shows a maximum flow for the network in Fig. 2.

The maximum flow in Fig. 4 is related to a cutset of the network. A *cutset* of a commodity network is a set of edges which when removed disconnects the source nodes from the sink nodes. No proper subset of a cutset is a cutset. Note that branches (S_1, A), (B, E), (B, D), and (C, E) form a cutset of the graph in Fig. 4. The *weight* of a cutset is equal to the sum of the capacities of the branches in the cutset. The weight of the above cutset is 24, which is equal to the value of the maximum flow. This is not a coincidence, because central to the Ford-Fulkerson algorithm is the following theorem.

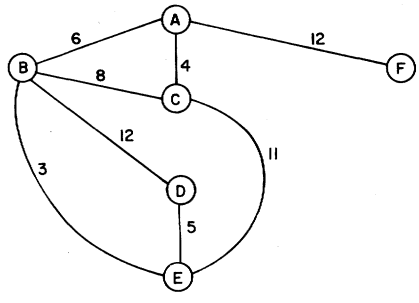

Fig. 5. Intermodule-connection graph.

Max-flow, Min-cut Theorem (Ford and Fulkerson [6]): The value of a maximum flow in a commodity network is equal to the weight of a minimum weight cutset of that network.

The proof of this theorem and a good tutorial description of algorithms for finding a maximum flow and minimum weight cutset appear in Ford and Fulkerson [6]. A very clear treatment of both the Ford-Fulkerson algorithm and the Edmonds-Karp modifications appear in Even [5]. At this point we note that the commodity flow problem as described can be solved in a time bounded from above by the fifth power of the number of nodes in the graph. All of the variations of the maximum flow algorithm of interest compute both a maximum flow in the graph and find a minimum weight cutset. We shall make use of this fact in solving the module assignment problem because each possible assignment corresponds to a cutset of a commodity graph, and the optimum assignment corresponds to a minimum weight cutset.

IV. The Solution to the Two-Processor Assignment Problem

In this section we show how the maximum-flow algorithm finds an optimal partition of a modular program that runs on a two-processor system. In the following section we show how this algorithm generalizes to systems with three or more processors.

To use the maximum flow algorithm we develop a graphical model of a modular program. Fig. 5 shows a typical example in which each node of the graph represents a module and the branches of the graph represent intermodule linkages. The modules should be viewed as program segments which either contain executable instructions plus local data or contain global data accessible to other segments. The weights on the

TABLE I

Module	P_1 Run Time	P_2 Run Time
A	5	10
B	2	∞
C	4	4
D	6	3
E	5	2
F	∞	4

branches indicate the cost of intermodule references when the modules are assigned to different computers. We assume that the cost of an intermodule reference is zero when the reference is made between two modules assigned to the same computer.

The cost function used to compute the branch weights may vary depending on the nature of the system. Initially we choose to minimize the absolute running time of a program, without permitting dynamic reassignments. We modify the constraints later. The weight of a branch under this assumption is the total time charged to the intermodule references represented by the branch. Thus if k references between two modules occur during the running of a program and each reference takes t seconds when the modules are assigned to different computers, then the weight of the branch representing these references is kt.

The graph in Fig. 5 captures the notion of the costs for crossing boundaries between processors. This is only part of the assignment problem. We must also consider the relative running time of a module on different processors. One processor may have a fast floating-point unit, or a faster memory than another processor, and this may bias the assignment of modules. Table I gives the cost of running the modules of the program in Fig. 5 on each of two processors. The symbol ∞ in the table indicates an infinite cost, which is an artifice to indicate that the module cannot be assigned to the processor. Since our objective in this part of the discussion is to minimize the total absolute running time of a program, the costs indicated in Table I give the total running time of each module on each processor.

In this model of a distributed computing system, there is no parallelism or multitasking of module execution within a program. Thus the total running time of a program consists of the total running time of the modules on their assigned processors as given in Table I plus the cost of intermodule references between modules assigned to different processors. Note that an optimum assignment must take into consideration both the intermodule reference costs and the costs of running the modules themselves. For the running example, if we assume that B must be assigned to P_1, and F must be assigned to P_2, then the optimum way of minimizing the intermodule costs alone is to view B and F as source and sink, respectively, of a commodity-flow network. The minimum cut is the minimum intermodule cost cut, and this assigns B, C, D, and E to P_1, and A and F to P_2. On the other hand, an optimum way to minimize only the running time of the individual modules is to assign A, B, and C to P_1 and D, E, and F to P_2. But neither of these assignments minimize the total running time.

To minimize the total running time, we modify the module interconnection graph so that each cutset in the modified graph corresponds in a one-to-one fashion to a module assignment and the weight of the cutset is the total cost for that assignment. With this modification, we can solve a maximum flow problem on the modified graph. The minimum weight cutset obtained from this solution determines the module assignment, and this module assignment is optimal in terms of total cost.

We modify the module interconnection graph as follows.

1) Add nodes labeled S_1 and S_2 that represent processors P_1 and P_2, respectively. S_1 is the unique source node and S_2 is the unique sink node.

2) For each node other than S_1 and S_2, add a branch from that node to each of S_1 and S_2. The weight of the branch to S_1 carries the cost of executing the corresponding module on P_2, and the weight of the branch to S_2 carries the cost of executing the module on P_1. (The reversal of the subscripts is intentional.)

The modified graph is a commodity flow network of the general type exemplified by the graph in Fig. 2. Each cutset of the commodity flow graph partitions the nodes of the graph into two disjoint subsets, with S_1 and S_2 in distinct subsets. We associate a module assignment with each cutset such that if the cutset partitions a node into the subset containing S_1, then the corresponding module is assigned to processor P_1. Thus the cut shown in Fig. 6 corresponds to the assignment of A, B, and C to P_1 and D, E, and F to P_2.

With this association of cutsets to module assignments it is not difficult to see that module assignments and cutsets of the commodity flow graph are in one-to-one correspondence. (The one-to-one correspondence depends on the fact that each interior node is connected directly to a source and sink, for otherwise, a module assignment might correspond to a subset of edges that properly contains a cutset.) The following theorem enables us to use a maximum flow algorithm to find a minimum cost assignment.

Theorem: The weight of a cutset of the modified graph is equal to the cost of the corresponding module assignment.

Proof: A module assignment incurs two types of costs. One cost is from intermodule references from processor to processor. The other cost incurred is the cost of running each module on a specific processor. The cutset corresponding to a module assignment contains two types of branches. One type of branch represents the cost of intermodule references for modules in different processors, and a particular assignment. All costs due to such references contribute to the weight of the corresponding cutset, and no other intermodule references contribute to the weight of the cutset.

The second type of branch in a cutset is a branch from an internal node to a source or sink node. If an assignment places a module in processor P_1, then the branch between that node and S_2 is in the cutset, and this contributes a cost equal to the cost of running that module on P_1, because the

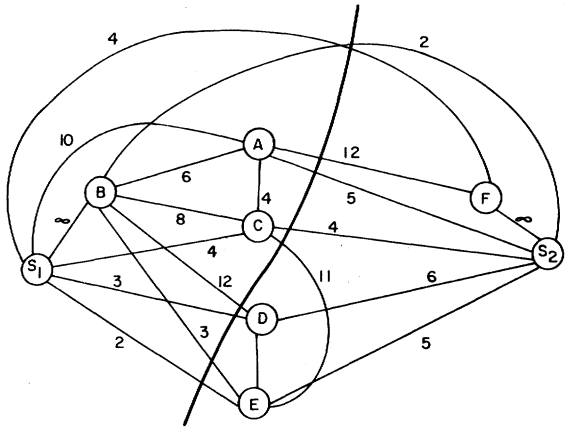

Fig. 6. Modified module-interconnection graph and a cut that determines a module assignment.

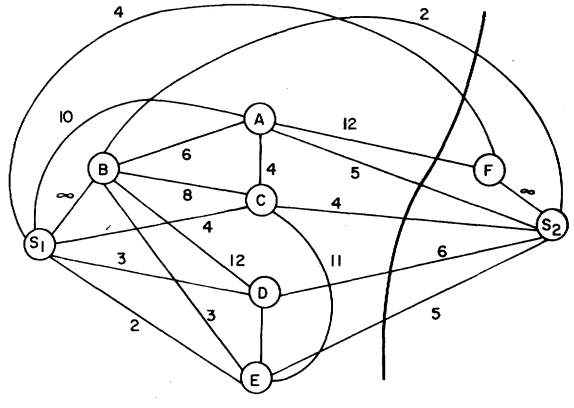

Fig. 7. Minimum cost cut.

branch to S_2 carries the P_1 running time cost. Similarly, the cost of assigning a module to P_2 is reflected by adding the cost of running that module on P_2 to the weight of the cutset.

Thus the weight of a cutset accounts for all costs due to its corresponding assignment, and no other costs contribute to the weight. This proves the theorem.

The theorem indicates that an optimal assignment can be found by running a maximum flow algorithm on the modified graph. A maximum flow algorithm applied to the graph shown in Fig. 6 produces the minimum weight cutset shown in Fig. 7. The corresponding module assignment is different from the assignment that minimizes the cost of intermodule references and the assignment that minimizes the individual running time costs, and its total cost is lower than the total cost of either of the two other assignments.

At this point the basic essentials for treating the module assignment problem as a maximum flow problem are clear. There remain a number of details concerning the practical implementation of the algorithm that merit discussion.

The running example indicates how to select an optimum *static* assignment that minimizes total running time. We mentioned earlier that it makes sense to change assignments dynamically to take advantage of local behavior of programs and the relatively infrequent changes in program locality. To solve the dynamic assignment problem we essentially have to solve a maximum flow problem at each point in time during which the working set of a program changes. Fortunately, the dynamic problem is no harder than the static one, except for the need to solve several maximum flow problems instead of a single one. The only additional difficulty in dynamic assignments is detecting a change in the working set of a program. Since a control program must intervene to perform intermodule transfers across processor boundaries, it should monitor such transfers, and use changes in the rate and nature of such transfers as a signal that the working set has changed. Thus we can reasonably expect to solve the dynamic assignment problem if it is possible to solve a static assignment problem for each working set.

The major difficulty in solving a static assignment problem is obtaining the requisite data for driving the maximum flow algorithm. It is not usually practical to force the user to supply these data, nor is it completely acceptable to use compiler generated estimates. Some of the data can be gathered during program execution. The cost of each invocation of a module on a particular processor can be measured by a control program, and it usually measures this anyway as part of the system accounting. However, we need to know the cost of running a module on each processor to compute an optimal assignment, and we certainly cannot ship a module to every other processor in a system for a time trial to determine its relative running time.

A reasonable approximation is to assume that the running time of a module on processor P_1 is a fixed constant times the running time of that module on processor P_2 where the constant is determined in advance as a measure of the relative power of the processors without taking into consideration the precise nature of the program to be executed. Under these assumptions if we can gather data about intermodule references, we can obtain sufficient data to drive the maximum flow algorithm. If after making one or more assignments a module is executed on different computers, sufficient additional information can be obtained to refine initial estimates of relative processor performance. The refined data should be used in determining new assignments as the data are collected.

How do we collect data concerning intermodule references? Here the collection of data follows a similar philosophy as for running time measurements. Initially an analysis of the static program should reveal where the intermodule references exist, and these in turn can be assumed to be of equal weight to determine an initial assignment. We assume that we automatically measure the intermodule references across processor boundaries because all such references require control program assistance. In measuring these references we obtain new data that refine the original estimates. If as a result of the refinement of these data we reassign modules, then we obtain new data about intermodule links that further refine the data, which in turn permit a more accurate appraisal of a minimum cost assignment.

At this writing the control methodology described here has been implemented by J. Michel and R. Burns on the system described by Stabler [15] and van Dam [18]. That system gathers the requisite statistics in real time in a suitable form

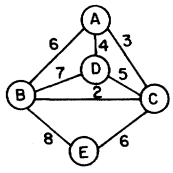

Fig. 8. Module-interconnection graph.

for input to the algorithm. It is too early to say how effective the statistics gathering and automatic reassignment processes are in performing load balancing, but it is safe to say that the ability to gather suitable statistics automatically has been demonstrated.

In closing this section we take up one last subject, that of considering objective functions other than total running time. There are many suitable cost functions one may wish to use. For example, instead of absolute time, one may choose to minimize dollars expended. For this objective function, the intermodule reference costs are measured in dollars per transfer, and the running time costs of each module are measured in dollars for computation time on each processor, taking into account the relative processor speeds and the relative costs per computation on each processor. Many other useful objective functions can be met by choosing the cost function appropriately. We should also point out that generalizations of the maximum flow algorithm are applicable to module assignment problems under more complex cost measures. The most notable maximum flow problem generalization is the selection of a maximum flow with minimum cost. For this problem each flow is associated with both a flow value and a cost. The algorithm selects among several possible maximum flows to return the one of minimum cost. The equivalent problem for the module assignment problem is to find an assignment that achieves fastest computation time and incurs the least dollar cost of all such assignments.

The fact that the two-processor assignment problem is mapped into a commodity flow problem for solution suggests that the flow maximized in the commodity flow problem corresponds to some physical entity flowing between the two-processors in the two-processor assignment problem. There is no such correspondence, however, since the maximal flow value corresponds to time in a two-processor assignment, and not to information flow.

V. Extension to Three or More Processors

In this section we show how the module assignments to three or more processors can be accomplished by using the principles described in the previous section. We first obtain a suitable generalization of the notion of cutset. Then we describe a procedure to construct a modified graph of a program such that its cutsets are in one-to-one correspondence with the multiprocessor cutsets, and the value of each cutset is equal to the cost of the corresponding assignment. Finally we consider how to solve the generalized multiprocessor flow problem, and obtain partial results but no efficient solution.

Let us take as a running example the three-processor program shown in Fig. 8 with the running times for each processor given in Table II. Since we have to assign the nodes to three rather than to two processors, we need to generalize the notion of cutset. Let us designate the source and sink nodes of a commodity network to be distinguished nodes, and we say they are distinguished of type *source* or type *sink*. For the n-processor problem we shall have n types of distinguished nodes. This leads naturally to the following definition.

TABLE II

Module	P_1 Time	P_2 Time	P_3 Time
A	4	∞	∞
B	∞	6	∞
C	∞	∞	7
D	10	7	5
E	4	7	3

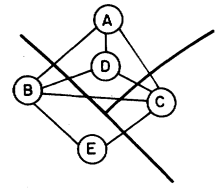

Fig. 9. Cut representing a module assignment to three processors.

Definition: A *cutset* in a commodity flow graph with n types of nodes is a subset of edges that partitions the nodes of the graph into n disjoint subsets, each of which contains all of the distinguished nodes of a single type plus a possibly empty collection of interior nodes. No proper subset of a cutset is also a cutset.

A cutset for the network of Fig. 8 appears in Fig. 9. We shall deal with networks in which there is a single distinguished node of each type, and this node represents the processor to which the interior nodes associated with it are assigned.

Proceeding as before we modify the intermodule reference graph of a program to incorporate the relative running time costs of each module on each processor. Again we add a branch from each interior node to each distinguished node as in the two-processor case. The weight on each such branch is computed by a formula explained below and exemplified in Fig. 10. The weights are selected so that, as in the two-processor case, the value of a cutset is equal to total running time.

For simplicity Fig. 10 does not show the branches from nodes A, B, and C to nodes to which they are not assigned. Suppose that module D runs in time T_i on processor P_i, $i = 1, 2, 3$. Then the branch from node D to distinguished node S_1 carries the weight $(T_2 + T_3 - T_1)/2$, and likewise the branches to nodes S_2 and S_3 carry the weights $(T_1 + T_3 - T_2)/2$, and $(T_1 + T_2 - T_3)/2$, respectively. Under this weight assignment, if D is assigned to processor P_1, the arcs to S_2 and S_3 are cut, and their weights total to T_1, the running time of D on P_1.

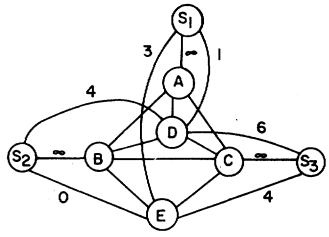

Fig. 10. Modified module-interconnection graph for a three-processor assignment.

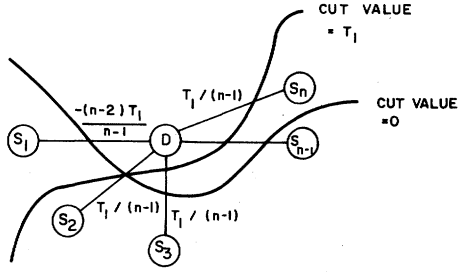

Fig. 11. Two possible assignments of module D in an n-processor system.

This idea generalizes naturally to n-processors. The running time of D on P_1 contributes to the weight of the branches from D to every distinguished node S_i, $i = 1, 2, \cdots, n$. Its contribution to the branch to S_1 is $-(n-2)T_1/(n-1)$. Its contribution to the branch to S_i, $i \neq 1$, is $T_1/(n-1)$. If D is assigned to S_1, then $n-1$ branches are cut, each of which contributes $T_1/(n-1)$ to the cutset, giving a net contribution of T_1. If D is assigned to S_i, $i \neq 1$, then n-2 branches contribute $T_1/(n$-1) to the cutset weight and the branch to S_1 contributes $-(n$-2)$T_1/(n$-1) for a net contribution of zero. This is shown graphically in Fig. 11. Consequently, under the graph modification scheme described here, we obtain the desired property that the weight of a cutset is equal to the cost of the corresponding module assignment.

One problem with this scheme is that some individual edges of the graph may have a negative capacity, and this cannot be treated by maximum flow algorithms. This problem can easily be overcome, however. Suppose that among the branches added to node D is an arc with negative weight $-W$, and this is the most negative of the weights of the branches added to node D. Then increase the weights of all branches added to D by the amount $W+1$, and we claim that no branch has negative weight after this change. Moreover, every cut that isolates D from a distinguished node breaks precisely $(n-1)$ of the branches added to D, contributing a value to the cutset of $(n-1)(W+1)$ plus the run time of D on the processor corresponding to the cut. The term $(n-1)(W+1)$ is a constant independent of the assignment so that an assignment that minimizes the cutset weight in the present graph is an assignment that finds a minimum time solution of the original problem.

At this point we come to the question of finding a minimum cutset in an n-processor graph. The solution can be found by exhaustive enumeration but the computational complexity of such an approach is quite unattractive. It seems natural that the n-processor problem can be reduced to several two-processor flow problems, and can be attacked efficiently in this way. In the remainder of this section we show that a two-processor flow can give information about the minimal cut in an n-processor graph, but we are unable to produce a complete efficient algorithm.

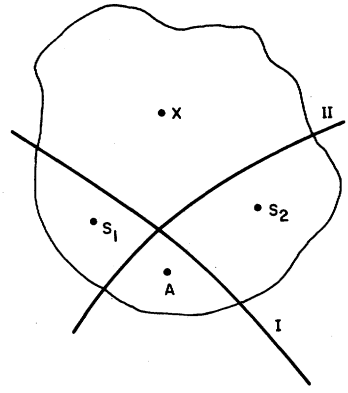

Fig. 12. Two cutsets in a graph.

Specifically, consider what happens when we run a two-processor flow from S_1 to the subset $\{S_i: 2 \leq i \leq n\}$, where the nodes in the subset are all sink nodes for the flow. This two-processor flow produces a cutset that associates some nodes with S_1, and the remainder of the nodes in the graph to the subset of $n-1$ distinguished nodes. Does this flow give any information about the minimum cut in the n-processor graph? Indeed it does, as is shown by the following theorem. The proof technique is similar to the proof technique used by Gomory and Hu [9] and Hu [11].

Theorem: Let node A be associated with distinguished node S_1 by a two-processor flow algorithm. Then A is associated with S_1 in a minimum cost partition of the n-processor graph.

Proof: Without loss of generality, suppose that A is associated with S_1 by the two-processor flow algorithm, and that it must be associated with S_2 in a minimum cost partition. We prove the theorem by showing that A can be moved from S_2 to S_1 in the minimum cost partition without altering the cost of the partition. Fig. 12 shows the cutset I that associates A with S_1 when the two-processor flow is run, and the cutset II that associates A with S_2 in the minimum cost partition. These cutsets cross each other, thus dividing the graph into four disjoint regions. Let us denote the four regions as S_1, S_2, A, and X according to the nodes appearing in these regions in Fig. 12. (Region X may be empty, but this will not upset the proof.) Let $c(U, V)$ denote the sum of the weights of all branches between two regions U and V.

Since II is a minimal cut the value of II does not exceed the value of a cut that fails to include A with S_2. Thus,

$$c(S_2,X) + c(S_2,S_1) + c(A,X) + c(A,S_1) \leq c(S_2,X) + c(S_2,S_1) + c(S_2,A). \quad (1)$$

From this we find that $c(A,S_1) \leq c(S_2,A) - c(A,X)$, and since all costs are nonnegative we may add $2c(A,X)$ to the

right-hand side of the inequality and obtain

$$c(A,S_1) \leqslant c(S_2,A) + c(A,X). \quad (2)$$

By hypothesis the algorithm associates node A with S_1, with a cost for cut I equal to $c(S_1,X) + c(S_1,S_2) + c(A,X) + c(S_2,A)$. However, if cut I veers slightly to exclude node A, and otherwise remains the same, the cost of the new cut is $c(S_1,X) + c(S_1,S_2) + c(A,S_1)$. Since I is a minimal cost cut, we must have

$$c(A,S_1) \geqslant c(S_2,A) + c(A,X). \quad (3)$$

From (2) and (3) we find that the inequality in (3) can be changed to equality. Since (3) holds with equality, by substituting (3) into (1) we find $2c(A,X) \leqslant 0$, which must hold with equality since all costs are nonnegative. Then (1) holds with equality. Thus cut II can be altered to exclude node A from S_2's subset and include A with S_1's subset without changing the cost of partition. This proves the theorem.

The previous theorem suggests that a two-processor algorithm may be used several times to find a minimum n-processor cutset. There are some difficulties in extending the theorem, however, that have left the n-processor problem still unsolved. Among the difficulties are the following.

1) The theorem states that a node associated with a distinguished node by a two-processor flow belongs with that node in the minimum n-processor cutset. Unfortunately, it is easy to construct examples in which a node that belongs with a particular distinguished node in a minimum n-processor cutset fails to be associated with that node by a two-processor flow.

2) Suppose a two-processor flow is run that results in the assignment of one or more nodes to a particular distinguished node. Let these nodes and the corresponding distinguished node be removed from the graph, and run a new two-processor flow from some other distinguished node S_i to all of the other distinguished nodes. In the cut found by this algorithm the nodes associated with S_i need not be associated with S_i in a minimum cost n-processor partition. In other words, the theorem does not apply when graph reduction is performed.

We conjecture that at most n^2 two-processor flows are necessary to find the minimum cost partition for n-processor problems, since there are only $n(n\text{-}1)/2$ different flows from one distinguished node to another distinguished node. These flows should somehow contain all the information required to find a minimal cost n-processor partition. It is possible that only n two-processor flows are required to solve the problem since Gomory and Hu [9] have shown that there are only n-1 independent two-terminal flows in an n-terminal network. This problem remains open at present.

VI. Summary and Conclusions

The two algorithms presented here provide for the assignment of program modules to two processors to minimize the cost of a computation on a distributed computer system. The algorithm uses a maximum flow algorithm as a subroutine so the complexity of the module assignment is dependent upon the implementation of the maximum flow algorithm used. Fortunately, the maximum flow algorithm is generally efficient and there are various modifications of the algorithm that take advantage of special characteristics of the module to obtain increased efficiency (see Dinic [3], Karzanov [13]). To obtain truly optimal assignments, the costs for intermodule transfers and relative running times have to be known. However, if good estimates are available, these can be used to obtain near-optimal assignments that are satisfactory in a pragmatic sense.

One may choose to use the assignment algorithm in a static sense, that is, to find one assignment that holds for the lifetime of a program, and incurs least cost. We believe it is more reasonable to reassign modules dynamically during a computation at the points where working sets change. Each dynamic assignment then is chosen to be optimal for a given working set. Dynamic identification of working sets and the identification of times at which a working set changes is still a subject of much controversy with proposals favoring particular schemes [2] or disfavoring those schemes [14]. Progress in this area will in turn lead to progress in the ability to make dynamic module assignments in a distributed processor system.

The model presented here is highly simplified and idealized, but it is useful in real systems. Foley *et al.* [7] can use our algorithms in place of their backtracking algorithms to do module reassignment in their distributed computer systems. We suspect that the maximum flow approach is more efficient than backtracking because worst-case performance of backtracking has a much greater complexity than maximum flow algorithm complexity. However, the actual performance of their algorithm may be quite different from worst-case performance, and could have a small average complexity, perhaps lower than the average complexity of maximum flow algorithms. No data on actual running times appears in Foley's report, so there is no information on which to base estimates of relative running times.

There are a number of open problems related to the research reported here. We mention just a few of them here.

1) If the minimum cost module assignment is not unique, then what additional criteria are useful in selecting the most desirable minimum cost assignment? Given such criteria, this form of the problem can be solved by using efficient algorithms to find a maximum flow of minimal cost. Such algorithms are described by Ford and Fulkerson [6] and Edmonds and Karp [4].

2) If a program is divided into tasks that can execute simultaneously under various precedence constraints, how can modules be assigned so as to minimize the cost of computation? This differs from the multiprocessor scheduling problem studied by Coffman and Graham [1] and by others in that there is no cost incurred in that model for interprocessor references.

3) If the various processors in distributed computer systems are each multiprogrammed, and queue lengths become excessive at individual processors, how might modules be reassigned to minimize costs of computation over several programs?

4) Given that a module reassignment incurs a processor-to-processor communication cost, how might the cost of

reassigning a module be factored into the module assignment problem?

Since distributed computer systems are still in early stages of development it is not clear which one of the research questions listed here will emerge to become important questions to solve for distributed computer systems as they come of age. The implementation of the methodology described here on the system at Brown University suggests that automatic load balancing among different processors can be done. We hope that the present and future research will show not only the possibility of load balancing but that it can be done efficiently and that load balancing is an efficient method for tapping the power of a distributed computer system.

Acknowledgment

The author is deeply indebted to Professor Andries van Dam for providing the inspiration and motivation for the research, and to J. Michel and R. Burns for implementing the ideas described here on the ICOPS system at Brown University. Discussions with Professor W. Kohler, S. Bokhari, and P. Jones provided additional stimulation that contributed to the research.

References

[1] E. G. Coffman, Jr., and R. L. Graham, "Optimal scheduling for two-processor systems," *Acta Informatica*, vol. 1, pp. 200–213, 1972.

[2] P. J. Denning, "Properties of the working set mode," *Commun. Ass. Comput. Mach.*, vol. 11, pp. 323–333, May 1968.

[3] E. A. Dinic, "Algorithm for solution of a problem of maximum flow in a network with power estimation," *Soviet Math. Doklady*, vol. 11, no. 5, pp. 1277–1280, 1970.

[4] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithm efficiency for network flow problems," *J. Ass. Comput. Mach.*, vol. 19, pp. 248–264, Apr. 1972.

[5] S. Even, *Algorithmic Combinatorics*. New York: Macmillan, 1973.

[6] L. R. Ford, Jr., and D. R. Fulkerson, *Flows in Networks*. Princeton, NJ: Princeton Univ. Press, 1962.

[7] J. D. Foley *et al.*, "Graphics system modeling," Rome Air Development Center, Final Rep. Contract F30602-73-C-0249, Rep. RADC-TR-211, Aug. 1974.

[8] S. H. Fuller and D. P. Siewiorek, "Some observations on semiconductor technology and the architecture of large digital modules," *Computer*, vol. 6, pp. 14–21, Oct. 1973.

[9] R. E. Gomory and T. C. Hu, "Multiterminal network flows," *J. SIAM*, vol. 9, pp. 551–570, Dec. 1961.

[10] F. E. Heart *et al.*, "A new minicomputer/multiprocessor for the ARPA network," in *Proc. 1973 Nat. Comput. Conf.*, *AFIPS Conf. Proc.*, vol. 42. Montvale, NJ: AFIPS Press, 1973.

[11] T. C. Hu, *Integer Programming and Network Flows*. Reading, MA: Addison-Wesley, 1970.

[12] R. E. Kahn, "Resource-sharing computer communications networks," *Proc. IEEE*, vol. 60, pp. 1397–1407, Nov. 1972.

[13] A. V. Karzanov, "Determining the maximal flow in a network by the method of preflows," *Soviet Math. Doklady*, vol. 15 no. 2, pp. 434–437, 1974.

[14] B. G. Prieve, "Using page residency to select the working set parameter," *Commun. Ass. Comput. Mach.*, vol. 16, pp. 619–620, Oct. 1973.

[15] G. M. Stabler, "A system for interconnected processing," Ph.D. dissertation, Brown Univ., Providence, RI, Oct. 1974.

[16] R. H. Thomas, "A resource sharing executive for the ARPANET," in *Proc. 1973 Nat. Comput. Conf.*, *AFIPS Conf. Proc.*, vol. 42. Montvale, NJ: AFIPS Press, 1973.

[17] R. H. Thomas and D. Henderson, "McRoss–A multi-computer programming system," in *Proc. 1972 Spring Joint Comput. Conf.*, *AFIPS Conf. Proc.*, vol. 40. Montvale, NJ: AFIPS Press, 1972.

[18] A. van Dam, "Computer graphics and its applications," Final Report, NSF Grant GJ-28401X, Brown University, May 1974.

[19] W. A. Wulf and C. G. Bell, "C.mmp–A multi-miniprocessor," in *Proc. 1972 Fall Joint Comput. Conf.*, *AFIPS Conf. Proc.*, vol. 41, Part II. Montvale, NJ: AFIPS Press, 1972, pp. 765–777.

Harold S. Stone (S'61–M'63) received the B.S. degree from Princeton University, Princeton, NJ, in 1960 and the M.S. and Ph.D. degrees from the University of California, Berkeley, in 1961 and 1963, respectively.

While at the University of California he was a National Science Foundation Fellow and Research Assistant in the Digital Computer Laboratory. From 1963 until 1968 he was with Stanford Research Institute, and from 1968 until 1974 he was Associate Professor of Electrical and Computer Science at Stanford University. He is currently Professor of Electrical and Computer Engineering at the University of Massachusetts, Amherst. He has recently been engaged in research in parallel computation, computer architecture, and advanced memory system organization. In the past he has performed research in several areas including combinatorial algorithms, operating systems, and switching theory. He has authored over thirty technical publications, and is an author, coauthor, or editor of five text books in computer science. He has also been a Visiting Lecturer at the University of Chile, the Technical University of Berlin, and the University of Sao Paulo, and has held a NASA Research Fellowship at NASA Ames Research Center.

Dr. Stone is a member of Sigma Xi and Phi Beta Kappa. He has served as Technical Editor of *Computer Magazine*, and as a member of the Governing Board of the IEEE Computer Society.