A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System

SHAHID H. BOKHARI, MEMBER, IEEE

Abstract—The problem of optimally assigning the modules of a program over the processors of an inhomogeneous distributed processor system is analyzed. The objective is to assign modules, wherever possible, to the processors on which they execute most rapidly while taking into account the overhead of interprocessor communication. Factors contributing to the cost of an assignment are 1) the amount of computation required by each module, 2) the amount of data transmitted between each pair of modules, 3) the speed of each processor, and 4) the speed of the communication link between each pair of processors.

A shortest tree algorithm is described that minimizes the sum of execution and communication costs for arbitrarily connected distributed systems with arbitrary numbers of processors, provided the interconnection pattern of the modules forms a tree. The algorithm uses a dynamic programming approach to solve the problem for m modules and n processors in $O(mn^2)$ time.

It is shown that this algorithm may also be used to optimally schedule a number of tasks whose precedence relationship forms a tree over a set of processors whose costs of execution and intercommunication vary with time. The motivation in this case is to distribute the task over the processors, delaying the execution of tasks wherever deadlines permit so as to take advantage of periods of light loading of specific processors and communication links.

In this case the algorithm may be used to minimize the sum of execution costs (processor and time dependent), communication costs (dependent on the characteristics of specific links and on time), and the penalties for not meeting deadlines (if any).

Index Terms—Computer networks, distributed processing, precedence trees, program assignment, scheduling, shortest trees.

I. INTRODUCTION

OVER the past few years interests in distributed processing has led to the identification of several challenging problems. One of these is the problem of optimally assigning the modules of a modular program over the processors of an inhomogeneous distributed processor system. Factors contributing to the cost of an assignment are 1) the amount of computation required by each module, 2) the amount of data transmitted between each pair of modules, 3) the speed of each processor, and 4) the speed of the communication link between each pair of processors.

Manuscript received July 5, 1979; revised January 14, 1981. This work was supported by the National Science Foundation under Grant MCS-76-11650 while the author was at the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, and by NASA under Contract NAS-1-14101 while the author was at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA.

The author is with the Department of Electrical Engineering, University of Engineering and Technology, Lahore, Pakistan.

Research by Stone [5], [7] and Bokhari [1], [2] has shown how the optimal assignment may be found efficiently for the case of dual processor systems using a network flow algorithm. While an extension to three processors was developed by Stone [6], algorithms for four or more processors have not been found. Gursky [3] has shown that the problem of finding the optimal assignment for four or more processors is *NP*-complete.

In Section II of this paper we show that if the intermodule communication pattern of the program is constrained to be a tree, then the problem may be solved for an arbitrary number of processors using an efficient dynamic programming approach. Programs that have a tree-like structure form an important class and include programs written as a hierarchy of subroutines. Turner [8] has discussed how a tree-like structure is best suited for large modular programs.

The algorithm takes into account the interconnection structure of the distributed system (i.e., the speeds of links between pairs of processors) a consideration that does not arise in the dual processor case.

The algorithm minimizes the sum of module execution and interprocessor communication costs. The two costs must be expressed in the same units which could be time or dollars or other resource units. If the costs are expressed in dollars, then the algorithm will minimize the total financial cost of executing the program. If expressed in terms of module execution and interprocessor communication time it will minimize the total execution time assuming serial execution of the program. That is, even though there are several modules and processors, only one module is active on one processor at a time. This is the case in some distributed processing applications including the distributed processor at Brown University [4], [9] which has served as a model for prior research by Stone and Bokhari.

In Section III of this paper we show how the dynamic programming approach may also be used to optimally schedule a number of tasks whose precedence relationship forms a tree over a set of processors whose costs of execution and intercommunication vary with time. The motivation in this case is to distribute the tasks over the processors, delaying their execution wherever deadlines permit so as to take advantage of periods of light loading of specific processors and communication links.

In this case the algorithm may be used to minimize the sum of execution costs (processor and time dependent), communication costs (dependent on the characteristics of specific links

0098-5589/81/1100-0583$00.75

and on time), and the penalties for not meeting deadlines (if any).

We assume all processors to be dissimilar. For example, a specific processor may have a hardware floating-point unit and thus be able to carry out floating-point operations with higher speed than a processor without such hardware. Similarly, some processors may be able to do byte manipulation more efficiently than others.

Although all processors are dissimilar, each module is able, in general, to execute on any processor. This is possible if all modules are written in a high-level language and separate object versions of these modules are available for each processor. This is the case in the Brown University System.

Alternatively, the distributed system may be made up of machines which have varying computational power but can execute essentially the same instruction set. A system based on the PDP-11 family of machines is one example. The complex of computers at NASA Langley Research Center, where there are several CDC machines varying from a CDC 6400 to a Cyber 175, is another example.

II. Distributing Across Space

In this section we examine the problem of optimally distributing a modular program over the processors of a distributed processing system. We call this the problem of distributing across space (i.e., the space of processors).

A. Formulation of the Problem

Our distributed processor system is assumed to be made up of an arbitrary number n of dissimilar processors. These processors are assumed to be interconnected in an arbitrary fashion, with arbitrary link speeds.

The program to be distributed across the processors is considered to be made up of a number of *modules*. A module is considered to be a portion of a program that can, in general, execute on any processor and could, for example, be a subroutine or coroutine. There are assumed to be m modules in the distributed program.

For each module we have the cost of executing it on each of the n-processors. The cost of executing module i on processor j is denoted e_{ij} and equals the sum of the costs of the various periods of execution of the module throughout the lifetime of the program (since, for example, a subroutine is typically executed several times during a program run).

The e_{ij}'s for an m module, n processor problem form an $m \times n$ matrix. Since the processors are dissimilar, the cost of executing a module varies from processor to processor, that is, $e_{ij} \neq e_{ik}$, in general. A module may be constrained to reside on a subset of available processors (perhaps on only one processor) by making its execution costs on the complementary subset infinite.

Modules will transfer control to each other at various points during the lifetime of the program. If we draw up a directed graph in which each node represents a module and in which there is an edge from node i to node j if and only if module i calls module j during program execution, this would be a *calls graph*. The algorithm for optimal assignments that will be presented in this section assumes that the calls graph of the program is a directed tree. We will call this an *invocation tree* because it describes the way modules invoke other modules during the execution of the program. Fig. 1 shows an invocation tree made up of eight modules.

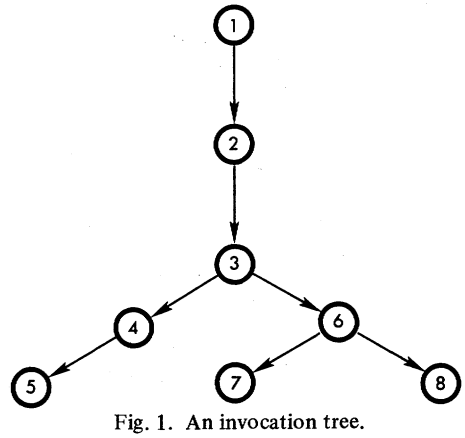

Fig. 1. An invocation tree.

Should a module invoke another module that is not coresident with it, this invocation would have to be transmitted over a communication link and thus incur interprocessor communication cost. This is dependent on the amount of data transmitted from one module to the other and the cost per bit of transmission between the two processors on which the modules are resident.

We will denote by d_{ij} the total amount of data transmitted between the calling module i and the called module j during the lifetime of the program. The cost of transmitting an amount of data d_{ij} between processors p and q is given by a function $S_{pq}(d_{ij})$. In its simplest form this function is $S_{pq}(d_{ij}) = s_{pq} \cdot d_{ij}$ where s_{pq} is the cost of transmitting a unit amount of data between processors p and q.

We assume the communication cost function to be symmetric, i.e., $S_{pq} = S_{qp}$. This assumption permits us to associate the sum of all data flow between modules i and j (whether from i to j or j to i) with the direction i to j. Although we see no motivation for considering the case $S_{pq} \neq S_{qp}$, the algorithm described below can easily be modified to handle this case (for which separate d_{ij}'s and d_{ji}'s will be required.)

The cost of invoking a coresident module is assumed to be zero, i.e., $S_{pp} = 0$. Should this assumption not be valid, it too can be relaxed to account for nonnegligible intraprocessor communication costs.

There is a nonzero d_{ij} associated with each directed edge from node i to node j in the invocation tree of our program.

B. Minimum Cost Assignment Across Space

We now show how the minimum cost assignment for our modular program may be found. Such an assignment minimizes the sum of execution costs and interprocessor communication costs.

Given the invocation tree of a modular program, and the execution and interprocessor communication costs, we may

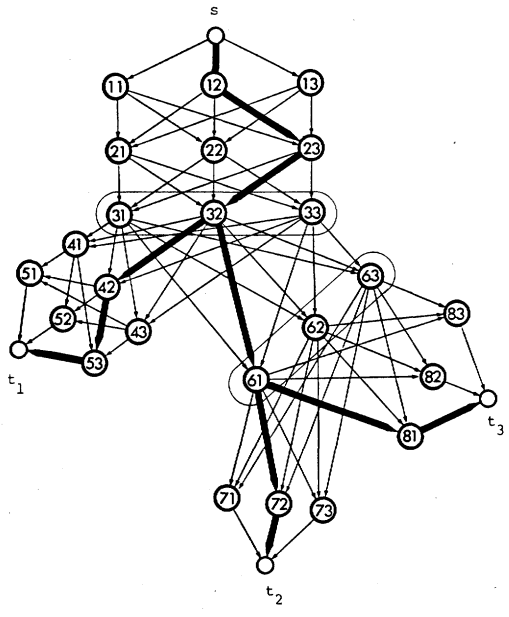

Fig. 2. An assignment graph for the invocation tree of Fig. 1 and a three-processor system. Thick edges form an assignment tree. Ovals have been drawn around the two forksets.

draw up an *assignment graph*. Fig. 2 shows the assignment graph for the invocation tree of Fig. 1 and a three-processor system.

The following definitions apply to this assignment graph.

1) The assignment graph is a directed graph with weighted edges.

2) There is one distinguished node called the *source node*, denoted s.

3) There are several *terminal nodes* $t_1, t_2, \cdots$, one for each leaf node of the invocation tree.

4) In addition to the source and terminal nodes there are $m \times n$ further nodes in the assignment graph (for a problem involving m modules and n processors). Each node is labeled with a pair of numbers (i,j) and represents the assignment of module i to processor j.

5) Each *layer* of the assignment graph corresponds to a node of the invocation tree. For example, the layer comprising nodes (2, 1), (2, 2), and (2, 3) corresponds to node 2 of the invocation tree.

6) Nodes in layers corresponding to nodes in the invocation tree having outdegree greater than 1 are called *forknodes*. Each layer of forknodes is called a *forkset*.

The edges have weights on them according to the following rules.

7) All edges incident on the terminal nodes t_1, t_2, etc. have zero weight on them.

8) Edges joining sourcenode S to nodes $(1, 1), (1, 2), \cdots$ have weights $e_{11}, e_{12}, \cdots$.

9) The edge joining node (i,p) to node (j,q) has weight $e_{jq} + S_{pq}(d_{ij})$. For example, the weight on the edge joining node $(1, 3)$ to $(2, 1)$ is $e_{21} + S_{13}(d_{12})$. This equals the cost of assigning module 2 to processor 1, given that module 1 has been assigned to processor 3.

It follows from property 4) that to each assignment of the m modules to the n processors there corresponds some subset of nodes of the assignment graph. The subgraph generated by these nodes plus the source and terminal nodes is called an *assignment tree* and has the following properties.

1) It is a tree.

2) It connects the source node s to all terminal nodes $t_1, t_2, \cdots$.

3) It contains one and only one node from each layer of the assignment graph.

There is a one to one correspondence between assignment trees and module assignments. Furthermore, the weight of each assignment tree (i.e., the sum of the weights of all edges in it) equals the cost of the corresponding assignment. This follows from property 9) of assignment graphs. The thick edges in Fig. 2 represent an assignment tree which is shown in isolation in Fig. 3.

To find a minimum cost assignment we need to find the minimum weight assignment tree in the assignment graph. This may be done using the dynamic programming approach described in the Appendix in $O(mn^2)$ time.

III. Distribution Across Space and Time

A. Motivations

In this section we discuss how a set of tasks that must be executed according to a tree-like precedence relationship may be optimally scheduled over a distributed processor system in which costs vary with time.

The computational resources of many organizations are in the form of a distributed computer network with each computer servicing one or more high priority local tasks and retaining the capability of servicing remotely submitted tasks at a low priority. In a production environment, the loads on the computers are fairly predictable as they depend heavily on the specific local loads on the machines.

In such an environment we would be interested in distributing a large set of tasks over the system such that it utilizes whatever processors are lightly loaded. Since the loads on the processors are time dependent, we may wish to consider suspending some tasks until a time when the load on a particular processor is very light. Of course, some tasks may be time critical and not admit any such suspension. Our problem thus becomes one of distributing a set of tasks over a set of processors, taking into account the run cost of each task on each processor (which will, in general, be time-dependent), the intercommunication costs between pairs of tasks (which will depend on the pairs of processors on which the two tasks are executed and may also be time dependent) and the penalties for not meeting deadlines for tasks (which may be set to infinity if a task must be executed by a certain time).

As a concrete example of a situation where such scheduling may be useful, consider the setting up of an appointment for a student to see a specific doctor at a University Infirmary. This task is to be run at the University's central administrative computing center (which is made up of one or more time-shared machines). The task involves the updating of a specific

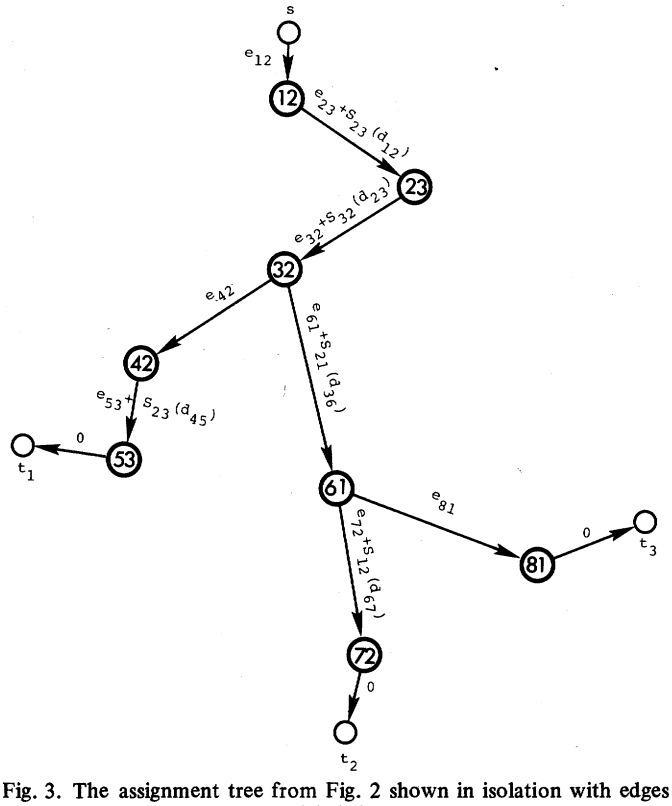

Fig. 3. The assignment tree from Fig. 2 shown in isolation with edges labeled.

doctor's appointment file, the scheduling of an examination room, mailing a remainder to the student, updating the student's account file, etc. Some tasks (e.g., updating the doctor's appointment file) must be done immediately, while others (updating the student's account) may be deferred until a later time.

In an airlines reservation computer system, the prime objective is to provide fast reliable service to the numerous ticketing terminals. However, there is no reason why such a system cannot be used for other functions as well. Tasks other than ticketing need to be scheduled over this system so as to minimally impact the efficiency of ticketing by deferring as much as possible such tasks to off-peak load hours.

Another example is where a long engineering computation (comprising several steps to be executed in a sequence) is to be carried out at a large computation laboratory with several computers, some of which are used for real-time simulation during the day. An engineer carrying out such a calculation will often do the initial data preparation during the day and then suspend his task until night time when it may be run at very low cost on a lightly loaded machine. The final interactive examination of the results is done on the following day.

The complex of computers at NASA Langley Research Center is an example of this kind of environment. There are at this center several CDC Cyber machines with varying computational power but capable of executing essentially the same instruction set. Some powerful machines are completely dedicated to real-time simulation during the day but may be used at lower cost during the evenings and at night.

B. Formulation of the Problem

We assume there to be n processors in our distributed system. The costs of executing tasks on processors vary with time but remain constant over specified periods of time called *phases*. This is illustrated in Fig. 4 where the vertical axis represents the space of processor and the horizontal axis represents time. During some phases a processor may be totally unavailable because of complete dedication to a real-time task or perhaps because of scheduled maintenance.

Once a task starts executing during a particular phase, it is allowed to run to completion even if the phase ends during task execution. The time to execute a task is considered to be small compared to the length of a phase. A task that is initiated near the end of a phase is treated like a customer who arrives at a bank just before closing time—he is allowed to complete his transaction even though closing time is past.

The graph superimposed on Fig. 4 shows one possible way of scheduling a precedence tree of tasks over the two dimensions of space (processors) and time (phases).

In the usual fashion, each node of the precedence tree represents a task and a directed edge from node i to node j implies that task i must be completed before task j is started.

With each node i is associated e_{ijk}, the cost of executing task i on processor j during phase k. This cost will, in general, vary across the processors and phases. It may be set to infinity for some processors during certain phases if these processors are not available during these specific phases.

Also associated with each node i is F_{ik}, the penalty for not completing task i by the end of phase k. This penalty F_{ik} may be set to infinity if the task must finish by phase k.

With the edge connecting nodes i and j in the precedence tree, we have d_{ij}—the amount of data that must be transmitted between tasks when invoking task j at the end of task i. The overhead of invoking task j in phase ϕ_w on processor p_y after completing task i in phase ϕ_v on processor p_x is a function $T(d_{ij}, p_x, p_y, \phi_v, \phi_w)$. This function can take into account 1) the amount of data transmitted, 2) the cost per bit of the link between p_x and p_y, and 3) the overhead of suspending a task when it finishes in ϕ_v and resumes in ϕ_w. A simple function, which ignores 3) above is $T = S_{xy} \cdot d_{ij}$ where S_{xy} is the cost of transmitting a unit of data between processors p_x and p_y.

C. Solution

The precedence tree of tasks may be scheduled to minimize the sum of execution costs, interprocessor communication costs, penalties for not meeting deadlines, and costs of suspending and resuming tasks. The solution techniques are very similar to the optimal assignment approach of Section II.

Fig. 5 shows a scheduling graph that has $m \cdot n \cdot \phi$ nodes for a problem based on m modules, n processors, and ϕ phases. This is a weighted directed graph. (Fig. 5 omits the arrow heads on the edges for clarity—there is no ambiguity of direc-

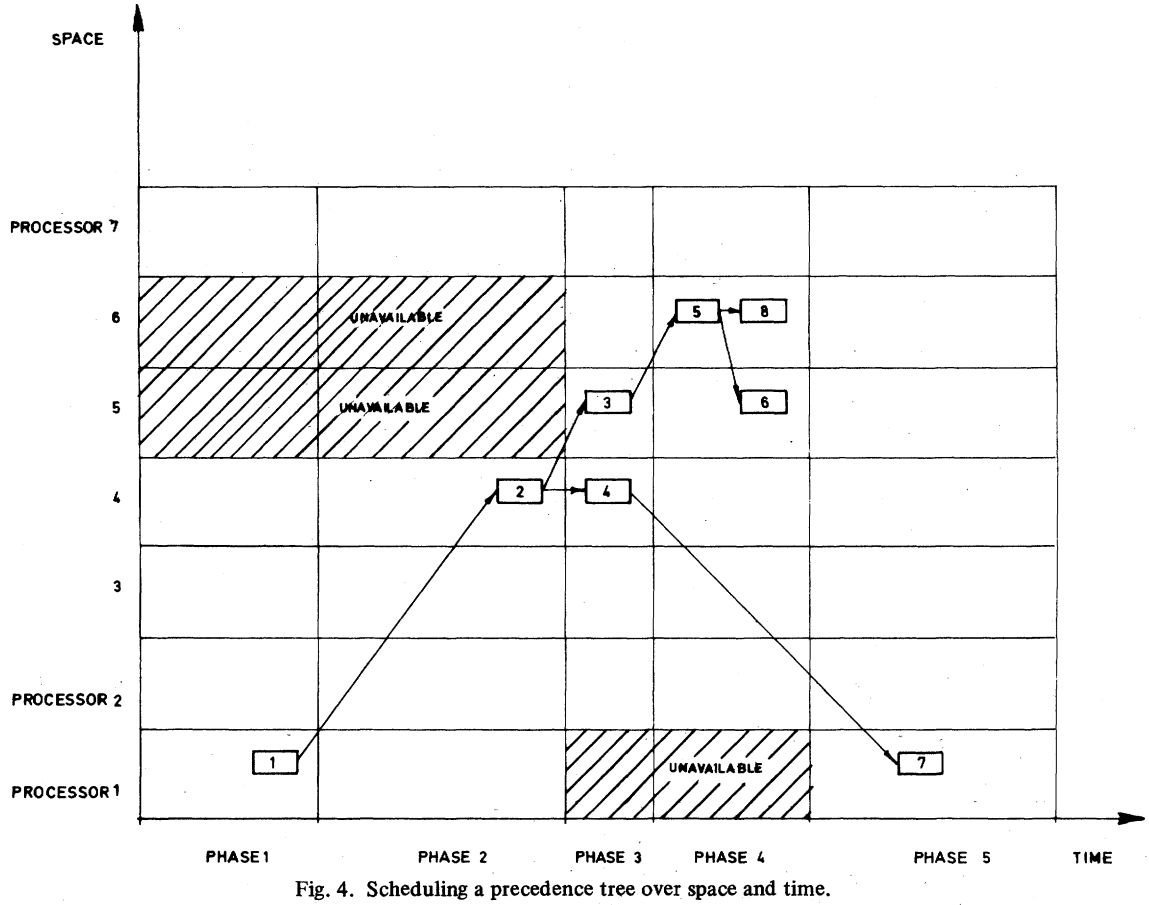

Fig. 4. Scheduling a precedence tree over space and time.

tion since all edges consistently point away from node s.) This graph is drawn up according to the following rules.

1) Node (i,j,k) represents the execution of task i on processor j during phase k.

2) Edges joining node s to nodes 111, 121, 112, etc. have weights $e_{111}, e_{121}, e_{112}$, etc.

3) Edges incident on nodes $t_1, t_2, \cdots$ have weight zero.

4) There is an edge joining node (i, p_x, ϕ_v) to node (j, p_y, ϕ_w) if node i precedes node j in the precedence tree and if $\phi_w \geqslant \phi_v$.

5) The weight on the edge described above is $e_{j,p_y,\phi_w} + T(d_{ij}, p_x, p_y, \phi_v, \phi_w) + F_{j,\phi_{w-1}}$. The first term takes into account the cost of execution, and the second term is the interprocessor and interphase communication overhead. The last term represents the penalty for not completing this task by the end of the previous phase.

The similarity between this scheduling graph and the assignment graph of the previous section is obvious. In fact the scheduling graph may be considered to be an assignment graph based on m modules and $n \cdot \phi$ processors. Thus, the shortest tree in the scheduling graph, which would correspond to the optimal schedule, may be found using the algorithm described in the Appendix in $O(mn^2\phi^2)$ time.

IV. Conclusions

A dynamic programming algorithm has been presented that may be used 1) to optimally assign a modular program that has a tree-like structure over a distributed processor system and 2) to optimally schedule a set of tasks that have a tree-like precedence relationship over a distributed processor system in which costs vary with time but are constant over contiguous periods of time called phases.

For the first case the algorithm has $O(mn^2)$ complexity where m is the number of modules and n is the number of processors. We have thus shown the unsolved problem of optimally assigning a modular program over more than three processors to be solvable for the important class of programs in which the module interconnection structure is a tree.

In the second case the algorithm takes $O(mn^2\phi^2)$ time where ϕ is the number of phases.

This algorithm does not take into account the capacity of each processor which may be a consideration in practice.

Appendix
The Shortest Tree Algorithm

An algorithm to find the shortest assignment tree in an assignment graph is presented in this section. At the heart of this algorithm is a procedure that will find the shortest paths from a terminal node of the assignment graph to all nodes in the nearest forkset (Fig. 6). This may be done using dynamic programming in $O(kn^2)$ time, where k is the number of layers between the terminal node and the forkset involved. (From each node in a layer we label all nodes in the preceding layer—this takes $O(n^2)$ time. This labeling is repeated k times.)

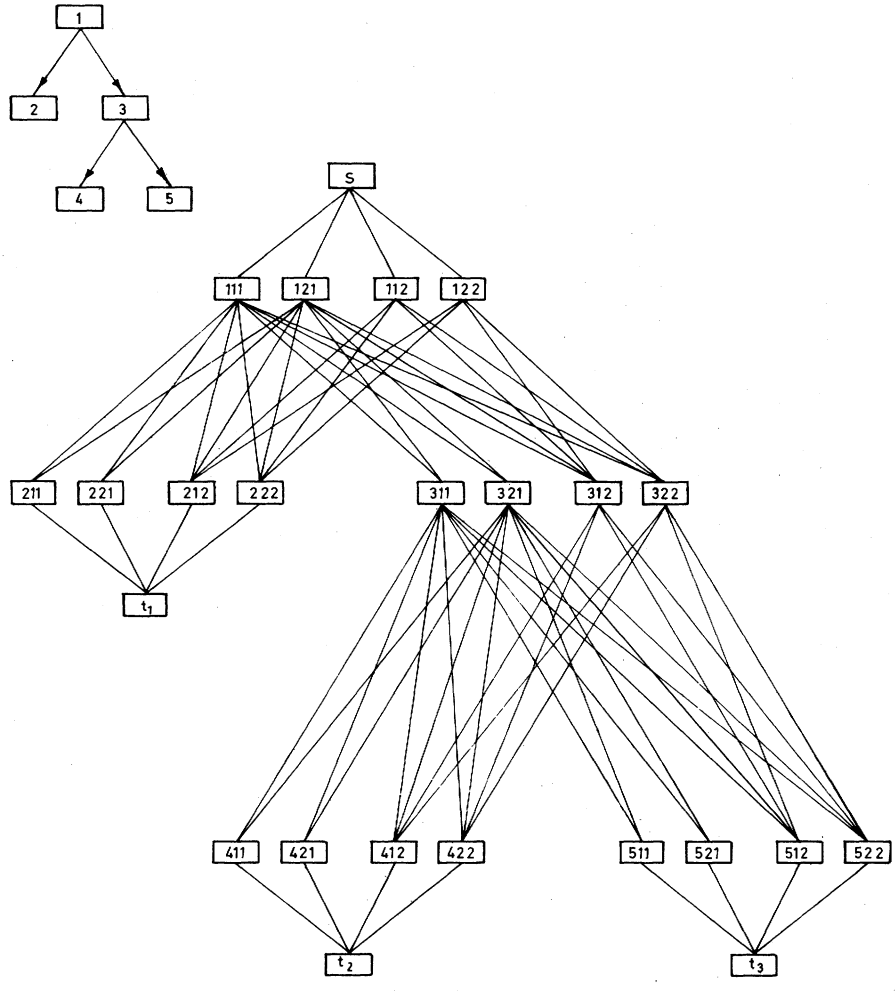

Fig. 5. A precedence tree and its scheduling graph for a two-processor two-phase problem.

Let us call this procedure SHORT and assume that it leaves pointers from each node to the next node in the shortest path to the terminal node.

We will call a forkset "exposed" when the shortest paths from its nodes to all possible nodes have been found.

```
begin
    input graph;
    (* TSET is the set of all terminal nodes *)
    (*FSET is the set of all forksets *)
    while |TSET| > 1 do
      begin
        to each terminal node t in TSET apply procedure
          SHORT and remove t from TSET;
        for each exposed forkset f in FSET do
          begin
            temporarily disconnect all outgoing edges;
            create a pseudoterminal node t_f;
            join all nodes in f to t_f with edges that
              have weights equal to the sum of the
              several shortest paths to the several
              discarded terminal nodes;
            remove f from FSET;
            add t_f to TSET;
          end;
      end;
    find the shortest path from the last terminal node to s;
    (* the length of this path equals the weight of the
      shortest tree *)
    reconnect all disconnected edges;
    traverse graph from s to all terminal nodes by following
      pointers set up by procedure SHORT; (* each node
      encountered is part of the shortest tree *)
end.
```

Fig. 6 shows an assignment graph just after the application of procedure SHORT to terminal nodes t_1 and t_2. In Fig. 7 we have temporarily removed the two limbs of the graph and created a pseudoterminal node t_0. The weight on the edge joining t_0 to a node in the forkset equals the sum of the shortest paths from that node to t_1 and t_2 from Fig. 6. After finding the shortest path from s to t_0 we reconnect the two limbs of the graph to obtain the shortest tree as shown in Fig. 8.

This algorithm is applicable to arbitrary assignment graphs.

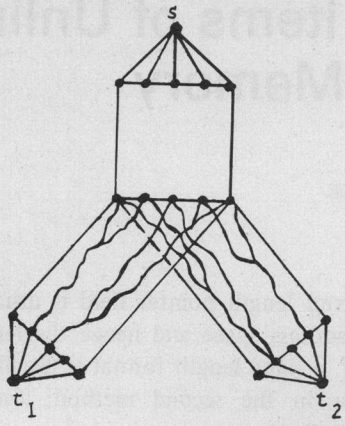

Fig. 6. Shortest paths from t_1 and t_2 to all the nodes in the forkset.

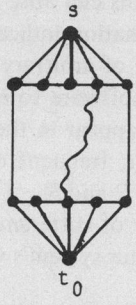

Fig. 7. Transformed graph with shortest path from pseudoterminal node t_0 to s.

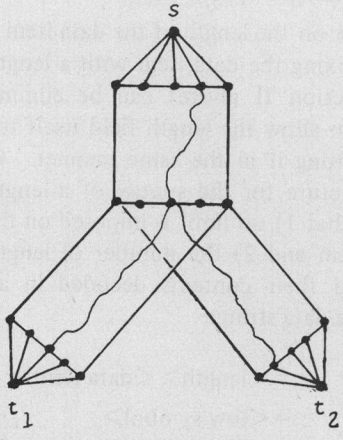

Fig. 8. The shortest assignment tree.

Each application of procedure SHORT takes $O(kn^2)$ time. The total time is $\Sigma_i\, O(k_i n^2)$, where i represents each application of procedure SHORT. This equals $O(n^2)\, \Sigma_i\, k_i = O(mn^2)$ since the total number of layers in the graph is m.

Acknowledgment

The author wishes to thank Prof. H. Stone for his encouragement of this research. Comments of the referees helped to reshape this paper significantly.

References

[1] S. H. Bokhari, "Dual processor scheduling with dynamic reassignment," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 341–349, July 1979.

[2] —, "Optimal assignments in dual processor distributed systems under varying load conditions," *IEEE Trans. Software Eng.*, to be published.

[3] M. Gursky, private communication.

[4] J. Michel and A. van Dam, "Experience with distributed processing on a host/satellite graphics system, in *Proc. SIGGRAPH '76*; available as *Computer Graphics* (SIGGRAPH Newsletter), vol. 10, no. 2, 1976.

[5] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 85–93, Jan. 1977.

[6] —, "Program assignment in three-processor systems and tricutset partitioning of graphs," Dep. Elec. Comput. Eng., Univ. Massachusetts, Amherst, Tech. Rep. ECE-CS-77-7, 1977.

[7] —, "Critical load factors in distributed systems," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 254–258, May 1978.

[8] J. Turner, "The structure of modular programs," *Commun. Ass. Comput. Mach.*, vol. 23, pp. 272–277, May 1980.

[9] A. van Dam, G. Stabler, and R. Harrington, "Intelligent satellites for interactive graphics," *Proc. IEEE*, vol. 62, pp. 83–92, Apr. 1974.

Shahid H. Bokhari (S'75–M'78) was born in Lahore, Pakistan, in 1953. He received the B.Sc. degree in electrical engineering from the University of Engineering and Technology, Lahore, Pakistan, in 1974, and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Massachusetts, Amherst, in 1976 and 1978, respectively.

In 1974 he worked as a Research Associate in the Department of Electrical Engineering at the Pakistan University of Engineering and Technology. From 1975 to 1978 he was a Research Assistant at the Department of Electrical and Computer Engineering, University of Massachusetts. He was a Research Scientist at the Institute for Computer Applications in Science and Engineering (ICASE) at NASA Langley Research Center, Hampton, VA, from July 1978 to December 1979. He is currently an Assistant Professor in the Department of Electrical Engineering, University of Engineering and Technology, Lahore, Pakistan. His research interests include computer architecture, distributed processing, and performance evaluation.

Dr. Bokhari is a member of the Association for Computing Machinery.