NASA Contractor Report 178023

ICASE REPORT NO. 85-54

NASA-CR-178023
19860008527

ICASE

PARTITIONING PROBLEMS IN PARALLEL, PIPELINED AND DISTRIBUTED COMPUTING

Shahid Bokhari

Contract Nos. NAS1-17070, NAS1-18107

November 1985

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

ENTER:
17 1 1 RN/NASA-CR-178023
DISPLAY 17/6/1
86N17997**# ISSUE 8 PAGE 1323 CATEGORY 59 RPT#: NASA-CR-178023
ICASE-85-54 NAS 1.26:178023 CNT#: NAS1-17070 NAS1-18107 85/11/00 34
PAGES UNCLASSIFIED DOCUMENT
UTTL: Partitioning problems in parallel, pipelined and distributed computing
TLSP: Final Report
AUTH: A/BOKHARI, S.
CORP: National Aeronautics and Space Administration. Langley Research Center, Hampton, Va. AVAIL.NTIS
SAP: HC A03/MF A01
COI: UNITED STATES Submitted for publication
MAJS: /*ARCHITECTURE (COMPUTERS)/*DISTRIBUTED PROCESSING/*PARALLEL PROCESSING (COMPUTERS)/*PIPELINING (COMPUTERS)/*PROBLEM SOLVING
MINS: / ALGORITHMS/ ALLOCATIONS/ COMPUTER STORAGE DEVICES/ PARTITIONS (MATHEMATICS)
ABA: Author

Partitioning Problems in Parallel, Pipelined and Distributed Computing

Shahid H. Bokhari

Institute for Computer Applications in Science & Engineering,
NASA, Langley Research Center, Hampton, Virginia 23665, USA
and
Department of Electrical Engineering,
University of Engineering & Technology, Lahore-31, PAKISTAN

ABSTRACT

The problem of optimally assigning the modules of a parallel program over the processors of a multiple computer system is addressed. A Sum-Bottleneck path algorithm is developed that permits the efficient solution of many variants of this problem under some constraints on the structure of the partitions.

In particular, the following problems are solved optimally for a single-host, multiple satellite system: Partitioning multiple chain-structured parallel programs, multiple arbitrarily structured serial programs and single tree structured parallel programs. In addition, the problems of partitioning chain structured parallel programs across chain connected systems and across shared memory (or shared bus) systems are also solved under certain constraints. All solutions for parallel programs are equally applicable to pipelined programs.

These results extend prior research in this area by explicitly taking concurrency into account and permit the efficient utilization of multiple computer architectures for a wide range of problems of practical interest.

This research was supported by NASA Contracts NAS1-17070 and NAS1-18107 while the author was in residence at the Institute for Computer Applications in Science & Engineering (ICASE), NASA Langley Research Center, Hampton, Virginia 23665, USA.

1. Introduction

Given a multiple computer system made up of n processors, and a program made up of m modules, each of which can, in general, execute on any processor, one is faced with the problem of partitioning the program over the processors in order to improve performance. If n =2 and the program is serial, (i.e. even though there are m modules, only one module is active on one processor at one time), this problem can be solved efficiently using the network flow approach pioneered by Stone [14]. If the program is serial and the interconnection structure of the modules tree-like, it is possible to solve it for any number of processors n using a shortest tree approach [2]. Several variants of this problem have been solved [1],[11],[16]. Other related research is reported in [5],[10].

If the modules are executable in parallel, it is very difficult to find efficiently the optimal assignment because, depending on the setting, the problem is computationally equivalent to one or the other of the notorious NP-Complete graph partitioning or multiprocessor scheduling problems. At the same time, as commercial multicomputer systems proliferate and cheap, easily interconnectable, 'building block' microcomputer systems become commonly available, it is becoming increasingly important to develop techniques to solve this problem.

In this paper we look at several variants of the partitioning problem for parallel programs and show that, under certain constraints on the structure of the program and/or the multicomputer system, this problem can indeed be solved in polynomial time.

We start the paper by discussing in Section 2 the relatively simple algorithm for finding the optimal partition of a chain-structured parallel or pipelined program over a chain or ring of processors. A chain structured program is made up of m modules numbered 1 .. m and has an intercommunication pattern such that module i is connected only to modules $i+1$ and i-1. Chains or rings of processors have similar structure. We work under the constraint that each processor have a contiguous subchain of program modules assigned to it. That is, partitions of the chains have to be such that modules i and $i+1$ are assigned to the same or to adjacent processors. We call this the *contiguity* constraint.

This problem arises in signal and image processing applications. Our solution technique involves creating a weighted assignment graph and finding the minimum bottleneck path in it. This result is related to prior research for *serial* programs [2] in that both construct a weighted, layered assignment graph and find a path in it. It differs from [2] in that concurrency is explicitly taken into account and a minimum bottleneck weight path, instead of a sum weight path yields the optimal solution. The method for constructing the assignment graph is also different.

The discussion of Section 2 permits us to better appreciate the central result of this paper which is the polynomial time Sum-Bottleneck path algorithm described in Section 3. This algorithm can be applied to a doubly weighted graph (i.e. one which has two kinds of weights associated with each edge) to find the path for which the maximum of (1) the sum of one kind of weight and (2) the bottleneck of the other kind, is minimum. This algorithm can be considered to be a combination of the classical sum weight algorithm and the bottleneck weight algorithms for conventionally (i.e. singly) weighted graphs.

In Section 4 we show how this algorithm may be applied to solve the problem of partitioning chain-structured programs over a single-host, multiple-satellite system under the contiguity constraint. Section 5 discusses the problem of partitioning a single chain structured parallel or pipelined program in a multiprocessor system that uses a shared memory or global bus for communication.

Section 6 we show how to optimally assign several arbitrarily structured *serial* programs over a single-host multiple-satellite system. Each program is associated with a specific satellite and executes serially. We may choose to move some of the modules of a program from its satellite to the host in order to take advantage of the host's greater power. However as more and more modules from *different* satellites are assigned to the timeshared host, its effective power goes down. The problem is to find the assignment that minimizes the time for the slowest program to finish. The efficient solution of this problem is of great relevance to organizations that use a large central timeshared machine connected to a number of

workstations as it allows the central machine's computing capacity to be apportioned fairly among the workstations.

Our solution to this problem uses Stone's network flow algorithm [14] for single-host single-satellite assignments combined with his results on nested assignments under varying load conditions [15] to transform the problem of assigning arbitrarily structured programs into the problem of assigning chains. A layered, doubly weighted assignment graph is created that includes all costs associated with the problem. An optimal Sum-Bottleneck path in this graph yields the optimal assignment of modules between the host and the satellites.

Section 7 discusses how a tree structured parallel or pipelined program may be partitioned optimally over a single-host, multiple-satellite system under the constraint that if a module is assigned to a satellite, all its children modules are also assigned to the same satellite. This problem is of relevance in process monitoring applications where information from many sensors is processed in a hierarchical fashion. The assignment graph for this problem is the most complex of all graphs presented in this paper. However, like all the above mentioned problems, this can also be solved in polynomial time by the application of the Sum-Bottleneck path algorithm.

We conclude with a discussion and tabular summary of our results in Section 8.

2. Mapping chains onto chains

The problem of mapping chains onto chains has applications in the fields of signal processing and image analysis. Certain methods for the parallel solution of partial differential equations can also utilize the techniques presented in this Section to improve running time on multiprocessors. In fact, these techniques are applicable to the parallel processing of any problem in which the communication pattern between different processes is chain-like (either because of the inherent structure of the problem or by deliberate choice of the algorithm designer) and the architecture is a chain or ring.

2.1 Signal Processing

A common requirement in a communication system is to apply repeatedly a fixed sequence of operations (or transforms) to an essentially unending series of signals. For example, each arriving packet (or 'window' or 'frame') of data may have to be Fourier transformed, multiplied by a fixed frequency, filtered, clipped, inverse transformed etc. This kind of application thus has a serial or chain-like structure to it and naturally lends itself to pipelining[3].

Should we choose to carry out all these processes on a uniprocessor, the maximum rate at which we can process incoming data frames is determined by the time required for the processor to apply all the processing steps to one frame. Clearly, this process can easily be pipelined by putting each process on a separate processor. Since the intercommunication pattern of the processes is serial, the processors need only be connected in a chain. The maximum rate of processing is now determined by the processor that takes the longest amount of time to perform its task (the *bottleneck* processor). This is an expensive solution in that it requires as many processors as processes. It is also inefficient because many processors may be very lightly loaded and spend most of their time waiting for the bottleneck processor to finish.

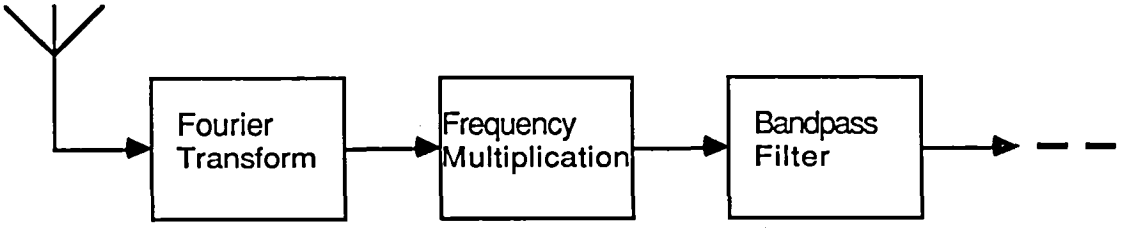

Fig. 1 Typical processing steps in a communication system.

The following problem then emerges. Given a set of m modules connected in a chain-like

fashion and a multiprocessor system of size $n<m$, find the assignment of subchains of processes to processors that minimize the load on the most heavily loaded processor. The contiguity constraint ensures that two modules that communicate with each other lie on directly connected processors.

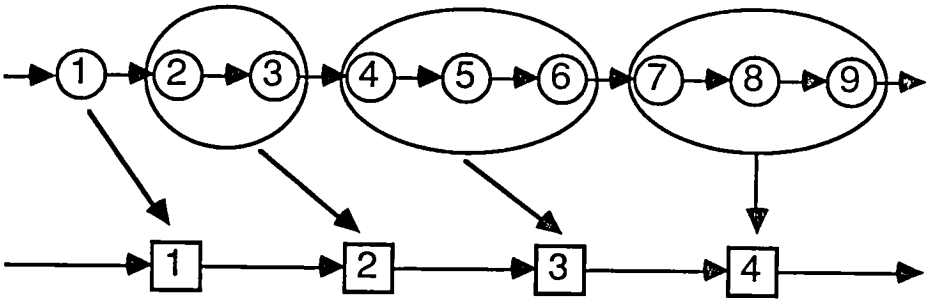

Fig. 2 A 9 module chain mapped onto a 4 processor chain.

The optimal assignment of subchains to processors is influenced by

(1) the time required to run each module (which may vary across processors, in case the processors are dissimilar),
(2) the amount of intermodule communication (which can be non uniform because once a frame of data has been transformed, it may have a different number of data points) and
(3) the speeds of the links between pairs of connected processors.

2.2 Image Analysis

Very similar problems arise in the field of image analysis where the requirement is to take an image or a set of images and apply various operators to it [13]. An interesting variation here is the possibility of obtaining a degree of pipelining greater than the number of different types of operations to be performed. This can be done, for example, if one needs to apply an operator to every 3 x 3 square of pixels in the image. Assuming that data can be transferred between processors at a sufficiently fast rate, it is possible to have as many pipeline stages as there are 3 x 3 squares in the image. The techniques discussed in this paper permit us to find the optimal

degree of pipelining given the processing and communication times of the processors.

2.3 Partial Differential Equations.

A straightforward technique for the parallel solution of certain types of partial differential equations on a possibly non-uniform mesh is to partition the mesh into vertical strips. During each iteration, an estimate is made of the values within a strip. Since strips only influence adjacent strips, the communication pattern required is chain-like and this problem can be run on a chain or ring of processors[12]. There again emerges the problem of optimal assignment of a chain of processes or modules (i.e the strips of the matrix) onto a chain of processors. The structure of this problem is the same as that shown in Fig. 2 except that the edges interconnecting modules or processors are undirected (communication takes place in both directions). The time required to complete one step of the computation is equal to the time required by the most heavily loaded processor to complete that one step. The important difference between this case and the signal or image processing examples described above is that this is *parallel* not pipelined processing. As we will describe later, the assignment algorithm is insensitive to this difference.

2.4 Construction of layered graph

Our approach to the solution of this problem is to first draw up a layered graph that contains all information about the run times of the modules. A path in this graph corresponds to the assignment of subsequences of modules to processors. The weight of the heaviest edge in a path corresponds to the time required to execute the assignment in parallel or pipelined fashion. Thus, having drawn up the graph, all we need do is find the *minimum bottleneck path* in it (i.e. the path for which the weight of the heaviest edge is minimum.)

In Fig. 3, each layer corresponds to a processor and the label on each node corresponds to a subchain of modules. Any path connecting nodes 's and t corresponds to an assignment of modules to processors. For example the thick edges correspond to the assignment of Fig. 2.

To avoid a congested diagram many nodes and edges have been omitted in Fig. 3.

The rule for generating this layered graph for a problem with m modules and n processors is as follows. Each layer contains all subchains of nodes, in other words all pairs $<i,j>$ such that $1<=i<=j<=m$. A node labeled $<i,j>$ is connected to all nodes $<j+1,k>$ in the layer below it for all j except 1 and n. All nodes $<1,i>$ ($<i,m>$) in the first (last) layer are connected to node s (t). As stated above, each path from s to t represents an assignment of subchains to processors under the contiguity constraint. If this path contains the node $<i,j>$ of layer k, this represents the assignment of modules i through j to processor k. Clearly, there is a path in this graph corresponding to every possible contiguous subchain assignment.

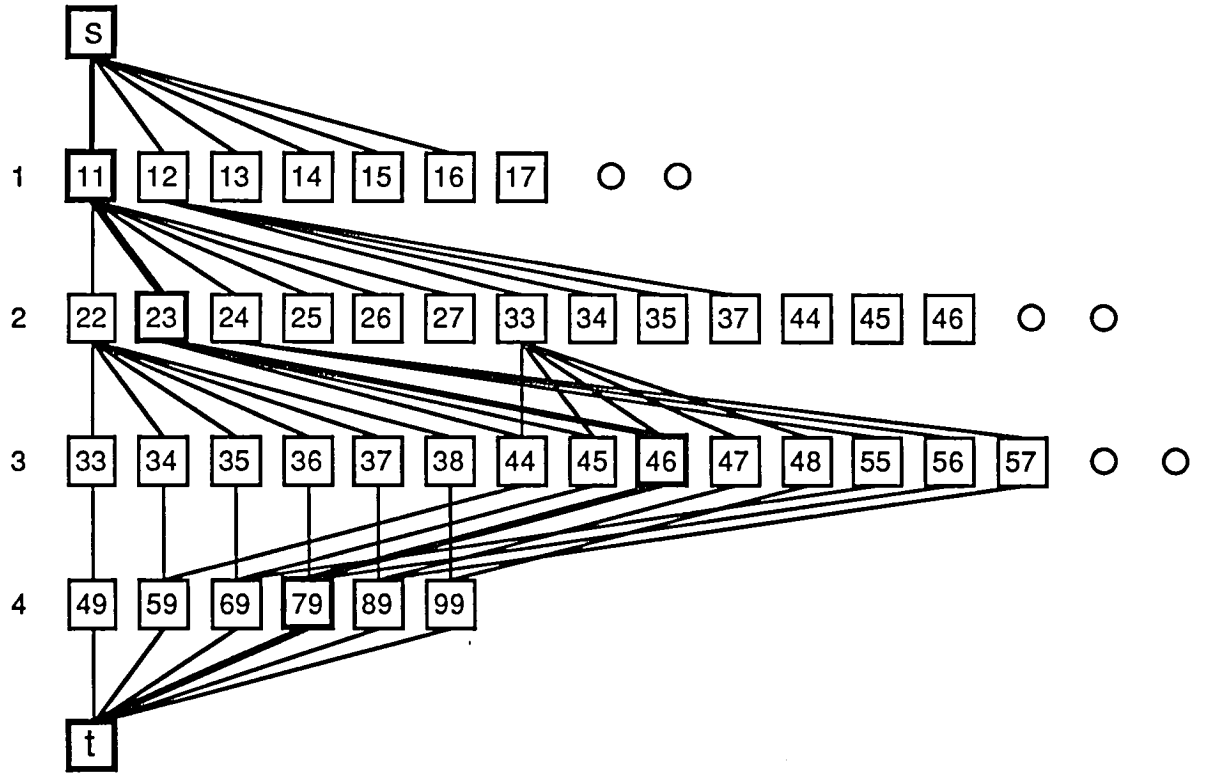

Fig. 3 The layered graph for a problem with 9 modules and 4 processors.

Weights can now be added to the edges of this layered graph as follows. In layer k, each edge emanating downwards from node $<i,j>$ is first weighted with the time required for

processor k to process nodes i through j. This accounts for the computation time. The communication time is now included in the graph: to the weight of the edge joining node $<a,b>$ in layer k to node $<b+1,d>$ in layer $k+1$ is added the time to communicate between modules b and $b+1$ over the link connecting processors k and $k+1$. It is clear that the influence of both the amount of data transmitted between modules b and $b+1$ as well as the speed of the link between processors k and $k+1$ can be included in the graph.

2.5 Finding the Optimal Assignment

A path in the layered graph in which the heaviest edge has minimum weight -- the *bottleneck* path -- can be found using a simple labeling procedure. Each node i is given a *Label L(i)*. Initially all nodes are given infinite labels except in the first layer, where the nodes are labeled zero. The following procedure is applied to all layers of this graph, starting at the top and working downwards.

Labeling Procedure for Minimum Bottleneck Path

Examine each edge e emanating downwards from a layer. Suppose it connects node a (above) to node b (below). Let the weight on this edge be $W(e)$. Then replace $L(b)$ by $\min(L(b), \max(W(e),L(a))$.

The number of nodes per layer is $O(m^2)$. The total number of nodes is $O(m^2n)$ since there are n layers in all. Since each node has at most m edges connected to it, there are $O(m^3n)$ edges in all. The labeling algorithm looks at each edge once. Therefore the space as well as time required by this algorithm is $O(m^3n)$ for a problem with m modules and n nodes.

2.6 Memory Constraints

To take memory constrains on individual processors into account it suffices to add up the memory requirements of all modules in every subchain. If the sum of memory requirements for nodes i through j exceeds the capacity of processor k, node $<i,j>$ in layer k is deleted, along with all edges incident on it.

3. Sum-Bottleneck paths in Doubly Weighted Graphs.

In the previous section we found the bottleneck path in the assignment graph in time proportional to the number of edges. This was possible because the assignment graph had a layered structure. In an arbitrary graph with n nodes the time required to find the bottleneck path is $O(n^2)$ [6]. The more familiar shortest path between two nodes can also be found in $O(n^2)$ time using Dijkstra's algorithm [4]. In this section we discuss the notion of Doubly Weighted graphs and optimal Sum-Bottleneck (SB) paths in them. We describe an efficient algorithm for finding the optimal SB path. This algrorithm is very useful for solving a wide range of assignment problems.

3.1 Definitions

A doubly weighted graph $D=<N,E>$ has two weights associated with each edge e from E : a *SumWeight* $\sigma(e)$ and a *Bottleneck Weight* $\beta(e)$. So, instead of a single weight on each edge as in the traditional weighted graph, we have an ordered pair of weights on each edge.

As usual, a path between any two nodes in this graph will be composed of a sequence of edges e_1 , e_2 , e_3 , ... The Sum Weight of this path, S , is the familiar *sum* of all $\sigma(e_i)$. The Bottleneck weight, B, is the *largest* of all $\beta(e_i)$.

The Sum-Bottleneck weight (SB weight) of this path is defined to be max(S,B). The optimal Sum-Bottleneck path (SB path) between two nodes in a doubly weighted graph is the path for which the the Sum-Bottleneck weight is *minimum*. In Fig. 4 the labels on each edge represent $<\sigma,\beta>$; the optimal SB path between nodes s and t has weight 8.

The notion of doubly weighted graphs is due to Lawler [9] who uses them for certain types of combinatorial optimization problems (e.g. shortest paths in networks with specified transit times). The contributions of the present paper are (1) the interpretation of these weights as sum

and bottleneck, (2) definition of the SB weight criterion for paths, (3) the specific polynomial time algorithm for finding the optimal SB path which follows, and (4) subsequent application of this algorithm to several partitioning problems.

3.2 An algorithm for finding the optimal SB path.

Assume we are given a Doubly Weighted graph D with n nodes and e edges and that the Dijkstra algorithm for shortest paths in a conventional graph is available to us. We wish to find the optimal SB path between two distinguished nodes s and t in the graph.

1. Create a list of all unique β weights sorted in descending order. Insert pointers from each entry in this list to the edges in D that have the corresponding weight. Let FIRST (LAST) be pointers to the first (last) elements of this list.
2. Select the midpoint, M, of list FIRST..LAST. Suppose the entry at that point is B_m. In graph D, remove all edges in which the Bottleneck weight is greater than B_m. Delete (or ignore) all Bottleneck weights less than or equal to B_m. The Doubly weighted graph D has now been transformed into a conventional (singly) weighted graph.
3. Apply Dijkstra's algorithm to this conventional graph to obtain the shortest path between s and t. Let the weight of this path be W.
4. If $W = B_m$ then stop; the path found by Dijkstra's algorithm is the optimal SB path.
5. If $W > B_m$ then FIRST:=M; restore D to its original form; go to step 2.
6. If $W < B_m$ then LAST:=M; restore D to its original form; go to step 2

To understand the working of this algorithm, refer to Fig. 5 which shows plots (as functions of B_m) of Sum weight S and Bottleneck weight B for a path in a Doubly weighted graph in which all edges with β weight greater than B_m have been removed.

The plot labelled B shows an upper bound for the bottleneck weight. As we travel from right to left (B_m decreases), B is a non-increasing curve, because this line represents the

weight of the bottleneck path, given that all edges with β weight greater than B_m have been removed. If a path exists between the given nodes after all edges e_i with $\beta(e_i)>B_m$ have been removed, the bottleneck weight of this path is clearly less than or equal to B_m. As B_m gets smaller, more and more edges get removed from the graph and it eventually gets disconnected. This is indicated by the shaded region in the graph.

The plot labelled S, on the other hand, shows the value of the minimum Sum weight of a path in the graph, given that all edges with β weight greater than B_m have been removed. As we decrease B_m, this curve is non-decreasing. This is because as we delete more and more edges from a graph, the weight of the minumum sum weight path can either remain undisturbed or increase. This curve also stops at the point the graph gets disconnected.

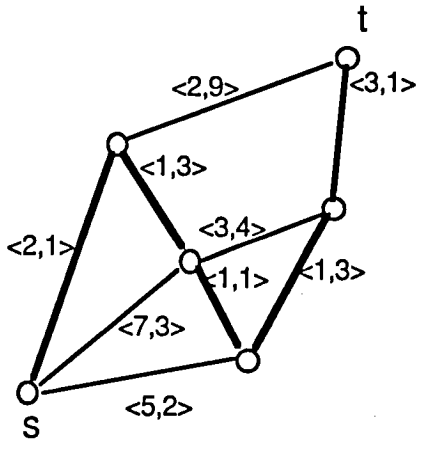

Fig. 4

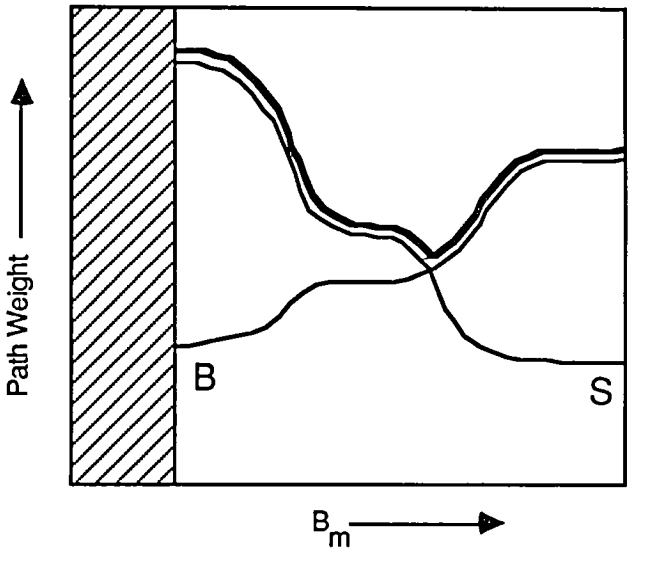

Fig.5

It should now be clear that steps 2 and 3 of the algorithm serve as a probe into this plot at a

fixed value of B_m. The SB weight (i.e. max(S,B)) is given by the thick line in Fig. 5. The optimal SB weight occurs at the minimum point of this curve. Because $S(B)$ is non-decreasing (non-increasing) with decreasing B_m, there is one unique minimum, which can be found using a binary search, and that is what the remaining steps of the algorithm achieve.

The number of distinct values B_m can have is no more than the total number of edges in the graph (the x-axis in Fig. 5 is a sorted list of unique β edge weights--it is *not* a continuous range of real numbers). The complexity of this algorithm is thus $O(n^2 log\ e)$, since each application of Dijkstra's algorithm takes $O(n^2)$ time.

4. Partitioning Multiple Chains across a Host-Satellite System

The algorithm presented in the previous section can be used to solve several difficult partitioning problems in Host-Satellite Systems of the sort shown in Fig. 6. Here we have a large host computer connected to several satellite computers which receive data from a real-time environment (for example, an aircraft.) The data streams entering each satellite have to be processed in a pipelined fashion (as shown in Fig. 1). The individual satellites may have different computational capabilities, the data streams could have different arrival rates (in frames per second), and the chains of computations to be performed on each stream need not be identical.

Each relatively small satellite computer has the capability of partitioning its workload between itself and the larger, more powerful, host to improve its individual processing time. However the act of moving some modules to the host would adversely impact the performance of other satellites. It is the complex interaction between the loads of the satellites via the shared host which makes this a difficult problem.

The factors influencing the cost of an assignment are the same as those enumerated in Section 2.1, except that there is a different set of costs for each satellite. We have for each module i of satellite j the time required to run it on the satellite, e_{ij}, and on the host, h_{ij}. For each pair of modules i and $i+1$ from satellite j we have the time required for interprocessor communications, c_{ij}, should i be assigned to the host and $i+1$ to the satellite.

Since all processing is to be done in a pipelined fashion, the times for execution and interprocessor communication are the times to pass through one frame of data. We assume that the data streams flowing into each satellite are all to be treated equally. Our assignment should be such that the time required to process one frame of data each from *all* streams is minimized. As an alternative, we may wish to give more importance to some data streams at the expense of others. For example, we may wish to process 5 frames of stream 1 for every 3 frames of stream 2. This is easily done by multiplying the cost figures for these streams by 5 and 3 respectively.

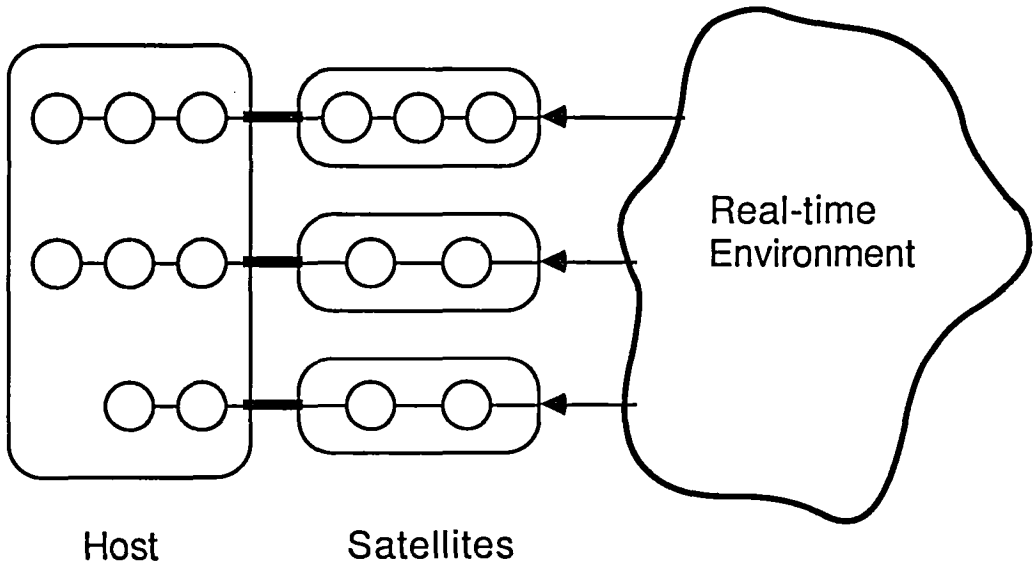

Fig. 6 A host satellite system processing real time data.

Without loss of generality, assume that each chain has m modules and that there are n satellites in all. Let us number the modules from left to right and define the partition point of each chain by the highest numbered module in that chain that is assigned to the host. When these n chains are partitioned between the host and the n satellites, the time required by the entire system to complete the processing of one frame of data from each stream is determined by the greater of (1) the individual load on the most heaviliy loaded satellite and (2) the sum of the collective loads on the host. Of all the satellites, the one which has is the most heavily loaded--the *bottleneck* satellite--determines the processing time as far as the satellites are concerned. On the other side, the *sum* of all loads on the host determines the time it will take. The greater of these two is the actual time since either the host waits for the slowest satellite to finish, or vice-versa.

4.1 Construction of Assignment Graph

Armed with the Sum-Bottleneck algorithm of the previous section, we can proceed to capture all this information in a new kind of layered assignment graph. This graph has n layers, one for each satellite. Each layer has m nodes, one for each module. An edge extends from each node in layer k to *all* nodes in layer k+1. There is a start node s above the first layer and a terminating node t after the last layer.

The assignment graph for a problem with 5 satellites and 5 modules on each satellite is shown in Fig.7. It is clear that a path from s to t represents a partitioning of the 5 chains between the host and the satellites. The path shown by thick edges in Fig. 7 represents the assignment of modules 1-4 of chain 1, 1 of chain 2, 1-3 of chain 3 etc. to the host and the remainder of each chain to the corresponding satellite.

To capture all information about run times, we proceed to doubly weight this graph as follows. Each edge leaving node j in layer k is first given a σ weight equal to the cost of the sum of the execution times of modules 1 through j of chain k. The β weight of this edge is the

sum of execution times of modules $j+1$ through m of chain k. To *both* these weights is added the communication time for modules j and $j+1$ over the link connecting the host to satellite k (this is because the communication overhead is incurred on both sides of the link.) Edges emanating from node s have all zero weights.

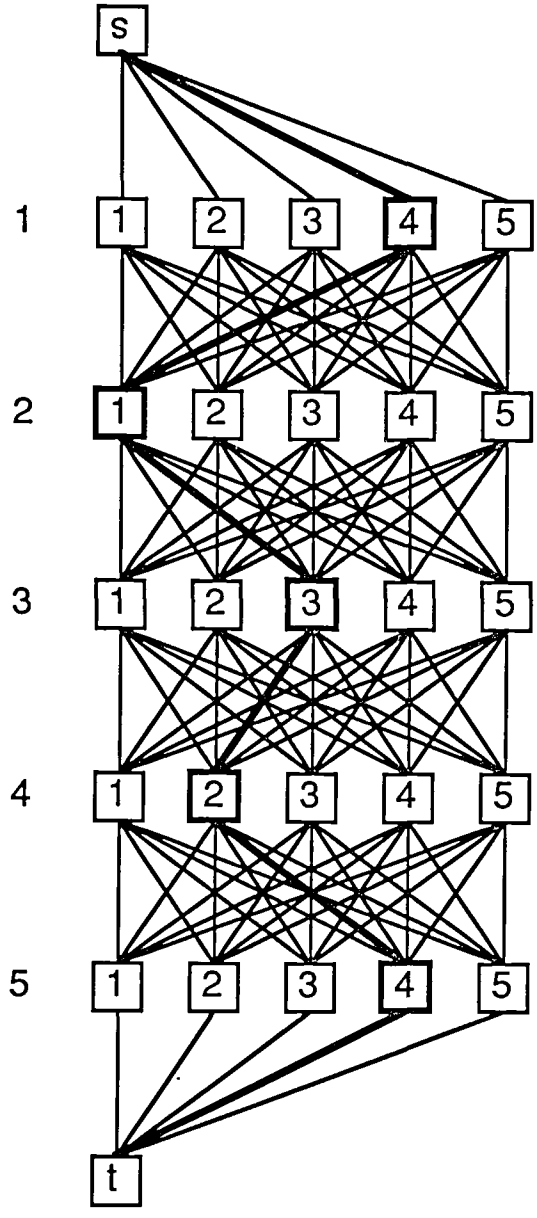

Fig. 7 Assignment graph for Host-Satellite assignment problem.

4.2 Solution of Problem

Because of our method of adding σ and β weights to the edges of this graph, the SB weight of any s to t path corresponds to the time required by the equivalent assignment of modules and therefore the optimal SB path corresponds to the best assignment. This optimal path may be found using the algorithm described in the preceeding section. However, the layered structure of the assignment graph means that we do not have to use Dijkstra's algorithm to find the shortest paths at step 3 (Section 3.2). The labeling process of Section 2.5 can find the shortest path in time proportional to the number of edges in the graph. For a problem with n satellites and m modules, the assignment graph of Fig. 7 has $O(m^2n)$ edges.The entire algorithm thus takes $O(m^2n \log m)$ time assuming $m>n$ and $O(m^2n \log n)$ otherwise.

5. Partitioning Chains in Shared Memory Systems.

The SB algorithm of Section 3 may also be used to solve the problem of partitioning chains in shared memory or bus interconnected systems. In such systems, all communication is through an area of shared memory or through a shared bus. In this case the sum of communication costs between all pairs of communicating processors--not the worst communication cost between a pair of processors--determines the degradation in performance due to interprocessor communication (Fig. 8). For an n processor problem, the partition is constrained to be composed of n subchains.

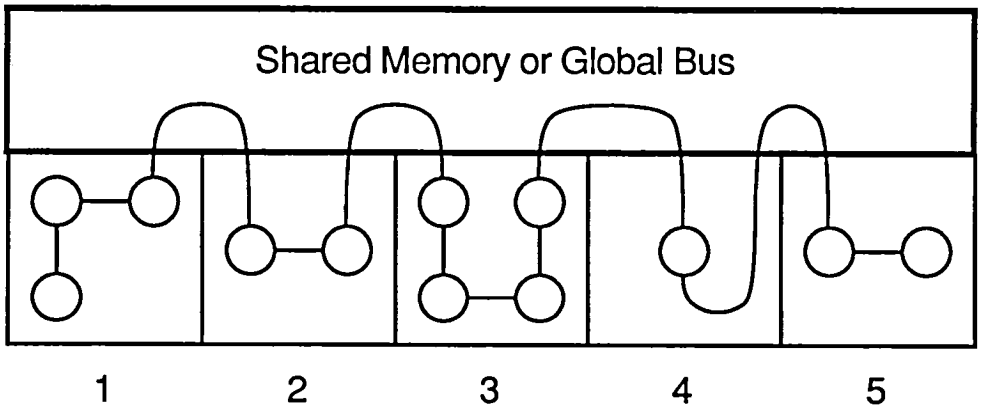

Fig. 8 A 12 module chain partitioned over 5 processors that use a shared memory or global bus for communication.

This problem is solved by constructing an assignment graph of the type shown in Fig.3 and doubly weighting it as follows. The communication costs are inserted as σ weights and the execution costs as β weights. Application of the optimal SB path algorithm to this graph yields the assignment of subchains to processors that minimizes the maximum of (1) the worst execution time of any processor and (2) the sum of interprocessor communication overheads.

6. Partitioning arbitrary programs in a Host-Satellite system.

In Section 4 we showed how multiple independent chains could be partitioned across a host-satellite system so as to minimize the time for execution for the most heavily loaded satellite. The chains could be streams of pipelined signal processing tasks as discussed in Section 2.1 or parallel programs with a chain like interconnection, as described in Section 2.3.

If we constrain the execution of a program in a host-satellite system to be *serial*, we can solve the problem of optimally assigning programs with arbitrary interconnection structure. In this setting we have one large host connected to multiple satellites or workstations. Each satellite is assumed to have a single program associated with it. The satellites are dissimilar, and the programs running on them could be composed of dissimilar modules. For simplicity, but without loss of generality, the *number* of modules on each program is assumed to be m.We will show how the SB path algorithm can be used to optimally partition arbitrary serial programs across a single-host, multiple-satellite system.

6.1 Stone's solution to the Single-Host, Single-Satellite problem.

In 1975 Stone showed how a distributed program could be optimally assigned over a single-host, single-satellite system, to minimize either serial execution time or total cost of computation (*e.g.* the finiancial cost of executing on both processors). The motivation in distributing computation in this case is to take advantage of specific efficiencies of the two processors in executing specific parts of the computation [14]. If two modules (or subroutines or coroutines) of a program are assigned to different processors, interprocessor communications cause an overhead which must be added to the total cost of executing the program.

Stone's method involves the construction of a network flow graph in which edge capacities represent computation and communication costs in such a fashion that the minimum weight cut separating two distinguished nodes in the graph corresponds to the optimal assignment (*i.e.* the assignment that minimizes the sum of computation and communication costs). This minimum

weight cut can be found using any one of several available network flow algorithms, in time no worse than $O(m^3)$ for a problem with m modules.

In later research, Stone analyzed the behavior of the optimal assignment as a function of the load on the host [15]. The crucial result here is the *Nesting Theorem* which states that as the load on the host increases the optimal assignment is always such that modules move away from the host and on to the satellite. It is never necessary during the course of an increase in load for the host and satellite to exchange two modules. Thus successive optimal assignments for successively increasing loads are nested inside each other, as illustrated in Fig. 9. There exist values of load which, once exceeded, cause one or more modules to move away from the host onto the satellite. These *critical load factors* may be found very efficiently (in no more than m applications of the network flow algorithm) using the method developed by Eisner and Severance[8].

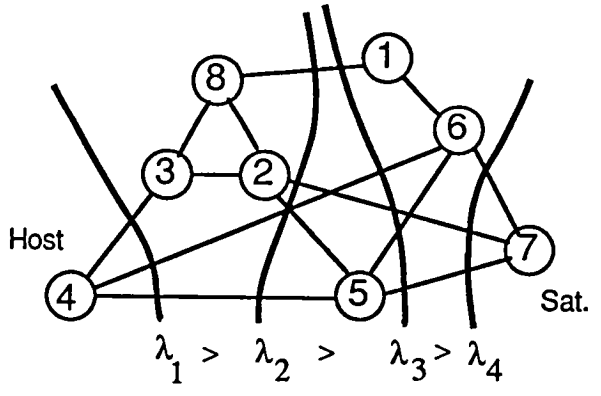

Fig. 9

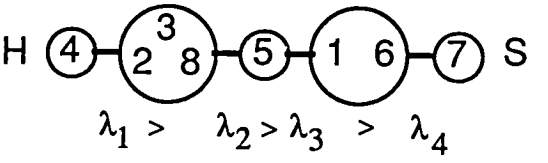

Fig. 10

We will show how these results allow us to *view* the interconnection of the modules as chain-like, regardless of the actual interconnection. We can then attack successfully the problem of optimally assigning or partitioning multiple distributed programs across a single-host, multiple-satellite system, which has remained unsolved until now.

6.2 Transformation into Chains

The program graph of Fig. 9 may be transformed into the *Loading Chain* shown in Fig. 10. All program modules lying between two adjacent cuts in Fig. 9 are clumped together into one super node in Fig. 10. The critical load factor property states that, in this specific case, if the load on the host is less than λ_1 modules 2, 3 and 8 will lie on the host. If the load is more than λ_1, they will lie on the satellite. These modules move as a group or clump--there is no value of load for which this group is split up. It is, however, possible to contrive communication and execution costs where the Loading Chain is made up of individual modules (i.e. there is no clumping). We will assume the worst case in subsequent analysis: a program graph of m modules gives rise to a Loading Chain of m nodes. We will lose no information by renumbering these nodes in a left to right order.

6.3 Construction of the Assignment Graph

Suppose we are given a single host connected to n independent satellites each of which has an arbitrarily connected program of m modules associated with it. We consider each particular program to go through an unending series of iterations. For each program i and module j we have h_{ij} (s_{ij}), the number of time units per iteration that the program spends in module j should this module be assigned to the host (satellite). In general h_{ij} is not equal to s_{ij} since the host (satellite) may be more efficient than the satellite (host) in executing certain types of computations. For example, the host may have a powerful floating point unit which will cause h_{ij} to be far less than s_{ij} for a module that does intensive arithmetic. This is in fact the motivation for distributing the computation of a serial program.

To account for interprocessor communication costs, we have for each pair of modules, i and

j, the number of time units per iteration spent in communication between the modules, c_{ij}, should they not be resident on the same processor.

We first independently find the n loading chains for each individual host-satellite combination. This takes no more than $O(m^4 n)$ time and yields n Loading Chains of size no more than m each. Each node p of a loading chain corresponds to the assigment of nodes $1..p$ to the host and $p+1..m$ to the satellite. We can therefore compute for each node of every Loading Chain:

(1) Ψ_p, the number of time units of satellite time that it requires per iteration. This is the sum of the individual s_{ij}'s of the modules assigned to the satellite.

(2) H_p, the number of time units of host time that it requires per iteration. This is the sum of the individual h_{ij}'s of the modules assigned to the host.

(3) C_p, the number of time units of interprocessor communication time that it requires. This is the sum of all communication times c_{ij} such that $i<=p$ and $j>p$, *i.e.* all pairs of modules i,j that are not coresident.

The global assignment of modules from n programs in this system is given by a vector $\pi[i]$ of separate assignments. The time required by an assignment is

$$\max\left(\Sigma_{i=1,n}\{H_{\pi[i]} + C_{\pi[i]}\}, \max_{i=1,n}\{\Psi_{\pi[i]} + H_{\pi[i]} + C_{\pi[i]}\}\right).$$

For a given $\pi[i]$ the sums $\Psi_{\pi[i]} + H_{\pi[i]} + C_{\pi[i]}$ represent the times for the n individual assignments as if the n programs were running on n isolated host-satellite systems. On the single-host system, however, the time for every program to complete one iteration each is determined by the slowest, hence the selection of the maximum of these. The time for the host to complete its share of the work is the sum of all $H_{\pi[i]} + C_{\pi[i]}$. The time for the entire system to complete one iteration of every program is the maximum of these quantities.

These times can be used to create a doubly weighted layered graph similar to the one in Fig. 7. Each layer corresponds to a Loading Chain. Each path from s to t stands for a $\pi[i]$. All possible $\pi[i]'s$ exist in this graph. An edge emanating downward from node p in layer k has σ weight equal to $H_p + C_p$ and β weight equal to $\Psi_p + H_p + C_p$. It can be verified that the SB weight of each path equals the time required for the corresponding global assignment. It follows that appication of the optimal SB path algorithm will yield the optimal assignment.

The time required to solve this problem is dominated by the time required to find the individual loading chains i.e. $O(m^4 n)$.

7. Partitioning Trees in a Host-Satellite System.

The final problem we solve in this paper is that of partitioning a tree structured program over a single-host, multiple-satellite system. The tree represents a parallel or pipelined computation and we assume the satellites to be similar. For example, external information may be input to the leaves of the tree which then process and pass this information up to their parents.

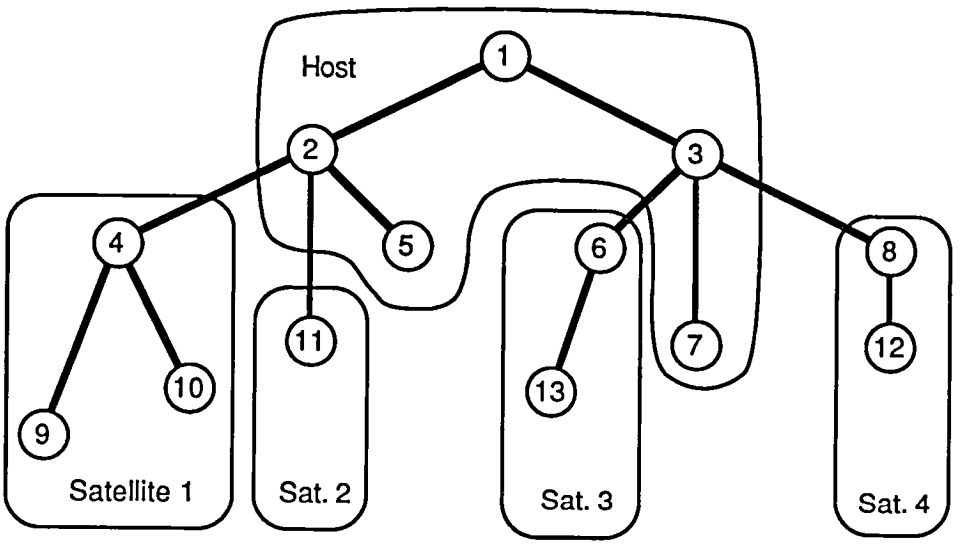

Fig. 11 A 13 node tree-structured parallel or pipelined program partitioned over a host-satellite system.

This is a good model of many industrial process monitoring systems where information from several sensors is collected by small satellite computers and transmitted to a large central host for processing. Depending on the volume of information being received from each processor and the type and amount of processing to be done, part of the work can be done in the satellites. By offloading work to the satellites, we reduce the load on the host and improve the response time of the system. The amount of work that can be assigned to the satellites is constrained by their lower computational power and small memories. The amount of interprocessor communication, which depends on the amount of data being transmitted and the speed of the links, also has to be taken into account when making the assignment.

The solution we present yields the partition that optimizes *pipelined* or *parallel* execution time under the constraints that

(1) the root is always assigned to the host,

(2) once a node is assigned to a satellite, all its children nodes are also assigned to the same satellite and

(3) if two nodes are assigned to a satellite their lowest common ancestor is also assigned to that satellite.

Informally speaking, this constraint means that individual maximal subtrees of the given tree are assigned to each satellite. Fig. 11 shows a 13 node tree that has been partitioned under these constraints. It is assumed that we have available as many satellites as there are leaf nodes in the tree and that we may choose not to use some of them if the optimal assignment so dictates.

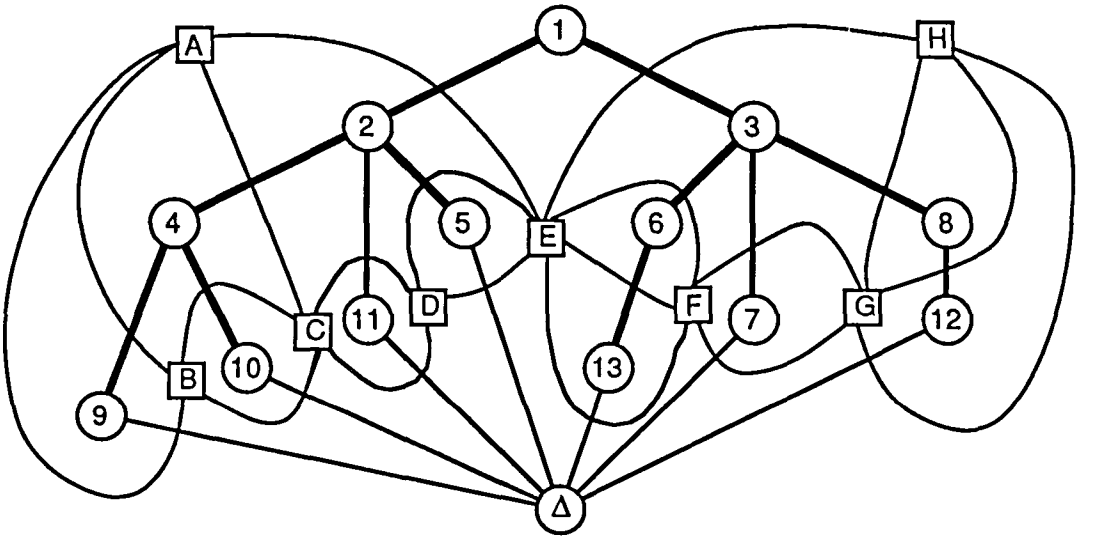

Fig. 12

To solve this problem we draw up an assignment graph as shown in Fig. 12. A dummy

node Δ is placed below the tree and connected to all the leaf nodes. This creates several regions or faces in the graph. Assignment graph nodes (squares in Fig. 12) are inserted in each region and on the left and right sides of the tree. There is an unambiguous left to right ordering of these nodes (indicated by the sequence $A, B, \ldots, H$ in Fig. 12.) A *directed* dual graph of this modified tree is now drawn by adding a directed edge between every pair of nodes that belong to regions that have a common edge. The direction of the edge is from the lower ordered node to the higher. To avoid a congested diagram we have omitted the arrowheads in Fig. 12 (their direction is evident from the node labels).

As before, we assume that we have available for each module i the time required to execute it on the host, h_i and on a satellite s_i (recall that all satellites are similar in this case.) For each edge in the tree connecting parent node i to child node j, we have the time required for interprocessor communication c_{ij}, should i be assigned to the host and j to a satellite.

The dual graph can now be doubly weighted. Suppose an edge of the dual graph separates a subtree τ from the program tree (i.e. removal of the tree edge that the dual edge crosses separates τ from the tree.) Then the β weight of this edge is the sum of all s_i for all $i \varepsilon \tau$, plus the communication cost between the root of the subtree τ and the node in the program tree to which it is connected. For example, consider the assignment graph edge *E-F* that crosses tree edge 3-6. The β weight on this edge is $s_6 + s_{13} + c_{36}$.

The procedure for inserting σ weights is somwhat involved. First move back to the original program tree augmented with dummy node Δ. Split Δ into as many nodes as there are leaf nodes in the original tree. The resultant modified tree (Fig. 13) is equivalent to the original program tree of Fig. 11 with an additional pendant vertex attached to each leaf. Give all edges

e_{ij} connecting parent i to child j an initial weight w_{ij}=0.

Traverse the nodes of this tree in preorder. When visiting node j, which has parent i and leftmost child k, give edge e_{jk} the weight $w_{jk}= w_{ij} + h_j$. The root is assumed to have an incoming edge of weight zero so that the edge connecting the root to its leftmost child has weight h_{root} and the edges to the remaining children have weight zero. The resulting weighting of edges is shown in Fig. 13.

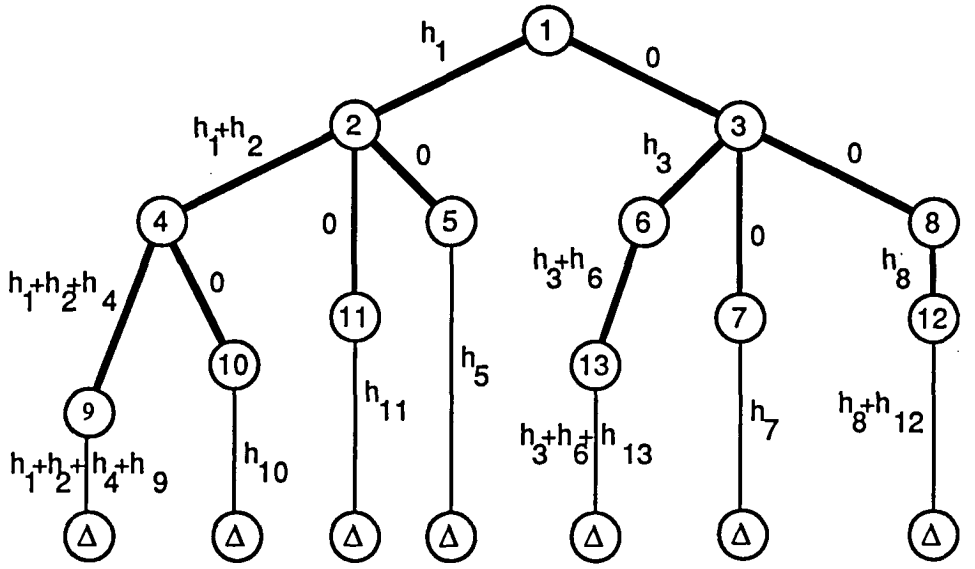

Fig. 13 Host execution times have been added as edge weights to the modified program tree. Communication times will be added to these before they are transferred as σ weights to corresponding edges of the assignment graph.

Now add the communication costs. For all edges connecting parent i to child j, replace w_{ij} by $w_{ij} + c_{ij}$. The weights of Fig. 13 are now copied onto corresponding tree edges in Fig. 12.

Now each edge of the assignment graph is given a σ weight equal to the weight of the tree edge that it crosses. For example, the assignment graph edge A-C crossing tree edge 2-4 is

given σ weight $h_1 + h_2 + c_{24}$. The assignment graph edge *D-E* crossing tree edge 2-5 is given σ weight c_{25}.

The dual assignment graph has now been doubly weighted. It can be verified that each path between *A* and *H* corresponds to an assignment and the SB weight of the path is the time required for the assignment. It remains to apply the optimal SB path algorithm of Section 3 to this graph between nodes *A* and *H* to obtain the optimal SB path and hence the assignment that minimizes the larger of the load on the host and the worst load on any satellite.

To analyze the running time of this algorithm it must be observed that the assignment graph in this case is a *multigraph* (i.e. more than one edge connects the same pair of nodes.) For a program tree with m nodes and f leaf nodes, this graph has $f+1$ nodes and m edges. With the addition of dummy nodes and edges, this multigraph can easily be transformed into a conventional graph with no more than $2m$ nodes and m edges to which the optimal SB path algorithm can be applied in time $O(m^2 \log m)$ time.

Limited memory on the (identical) satellites can be accounted for by deleting all assignment graph edges that separate subtrees with total memory requirements greater than the capacity of the satellites.

8. Conclusions.

We have addressed a variety of problems in parallel, pipelined and distributed processing and shown how to solve them using the optimal SB path algorithm for doubly weighted graphs of Section 3. This is an efficient polynomial time algorithm and contains the traditional shortest path and bottleneck path algorithms as special cases. For example, the 'pure' bottleneck problem in Section 2 can be solved by the SB algorithm by setting all σ weights to zero. The shortest path problems presented in earlier research on distributed processing [2] are solvable by the SB algorithm by setting all β to zero.

However, the mere existence of this algorithm is not enough to solve other problems in this field--an appropriate assignment graph must first be discovered. We have described assignment graphs for a range of problems. Each assignment graph has to be designed very carefully in order to capture all information about the problem. The structure of the graph is related to the structure of the problem and the method of adding the double weights has to reflect the cost being minimized. The graphs have to be polynomial in size so that the time required to find the optimal SB path will also be polynomial.

The assignment graphs presented in this paper range from the relatively simple ones for mapping chains on chains, discussed in Section 2, to the fairly complex graphs for partitions in host-satellite systems, described in Sections 6 and 7. All are polynomial in size and permit the solution of a particular assignment problem in polynomial time. We believe that the SB path algorithm is a powerful tool that can be used to solve efficiently many other problems in the field of multiple computing.

Table I (following page) is a summary of the results presented in this paper. Most of the column headings are self explanatory except perhaps the second last which lists if a memory constraint can be taken into account by the given algorithm.

Table I
Summary of Results

Section	Problem Structure	Processor Structure	Processing	Partition Constraint	Mem. Limit.	Time
2	Single chain m nodes	Chain n nodes	Pipelined/ Parallel	n contiguous subchains	Yes	m^3n
4	n chains of m nodes each	Single host n dissimilar satellites	Individual pipelined/parallel chains executing in parallel	2 contiguous subchains of each chain	On satellites	$m^3n \log m$
5	Single Chain m nodes	n identical commun. via memory/bus	Pipelined/ parallel	n subchains	Yes	$m^3n \log m$
6	Arbitrary n programs m modules each	Single host n dissimilar satellites	Individual serial progs. executing in parallel	None	No	m^4n
7	Single tree m modules	Single host $n<m$ identical satellites	Pipelined/ parallel	Maximal subtrees on satellites	On satellites	$m^2 \log m$

9. Acknowledgements

I am indebted to M. E. Rose and R. G. Voigt for their constant encouragement of this research. G. Erlebacher was generous with his time and patience in teaching me the use of the Macintosh. Discussions with J. Saltz, M. Berger and A. Iqbal have also been very useful.

10. References

[1] S. H. Bokhari, "Dual processor scheduling with dynamic reassignment," *IEEE Trans. Software Engineering,* vol. SE-5, No. 4, pp. 341-349, July 1979.

[2] S. H. Bokhari, "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system," *IEEE Trans. Software Engineering,* vol. SE-7, No. 6, pp. 583-589, November 1981.

[3] G. Bolch, F. Hofman, B. Hoppe, H. J. Kolb, C. U. Linster, R. Polzer, W. Schussler, G. Wackersreuther and F. X. Wurm, "A multiprocessor system for simulating data transmission systems (MUPSI)," *Microprocessing and Microprogramming,* vol. 12, pp. 267-277, 1983.

[4] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, pp. 269-271, 1959.

[5] K. W. Doty, P. L. McEntire and J. G. O'Reilly, "Task allocation in a distributed computer system," *Proc. IEEE Infocom*, pp. 33-38, 1982.

[6] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow algorithms," *JACM*, vol.19, pp. 248-264, April 1972.

[8] M. J Eisner and D. G. Severance, "Mathematical techniques for efficient record segmentation in large databases," *JACM*, vol. 23, No. 4, pp. 619-635, October 1976.

[9] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*. New York: Holt, Rinehart and Winston, 1976.

[10] V. M. Lo, "Heuristic algorithms for task assignments in distributed systems," *Proc. 4th. Int. Conf. Distributed Proc. Systems*, pp. 30-39, May 1984.

[11] G. S. Rao, H. S. Stone and T. C. Hu, "Assignment of tasks in a distributed processor system with limited memory," *IEEE Trans. Computers*, vol. C-28, No. 4, pp. 291-299, April 1979.

[12] J. H. Saltz, "Parallel and adaptive algorithms for problems in scientific and medical programming," Ph.D. Thesis, Dept. of Computer Science, Duke University, 1985.

[13] S. R. Sternberg, "Biomedical image processing," *Computer*, vol. 16, No. 1, pp. 22-34, January 1983.

[14] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. Software Engineering*, vol. SE-3, No. 1, pp. 85-93, January 1977.

[15] H. S. Stone, "Critical load factors in distributed computer systems," *IEEE Trans. Software Engineering*, vol. SE-4, No. 3, pp. 254-258, May 1978.

[16] H. S. Stone and S. H. Bokhari, "Control of distributed processes," *Computer*, vol. 11, No. 7, pp. 97-106, July 1978.

1. Report No. NASA CR-178023 ICASE Report No. 85-54	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subtitle	5. Report Date
PARTITIONING PROBLEMS IN PARALLEL, PIPELINED AND DISTRIBUTED COMPUTING	November 1985
	6. Performing Organization Code
7. Author(s) Shahid Bokhari	8. Performing Organization Report No. 85-54
	10. Work Unit No.
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	11. Contract or Grant No. NAS1-17070, NAS1-18107
	13. Type of Report and Period Covered Contractor Report
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546	14. Sponsoring Agency Code 505-31-83-01

15. Supplementary Notes
Langley Technical Monitor: J. C. South Jr.
Final Report
Submitted to IEEE Trans. Comput.

16. Abstract

The problem of optimally assigning the modules of a parallel program over the processors of a multiple computer system is addressed. A Sum-Bottleneck path algorithm is developed that permits the efficient solution of many variants of this problem under some constraints on the structure of the partitions.

In particular, the following problems are solved optimally for a single-host, multiple satellite system: Partitioning multiple chain-structured parallel programs, multiple arbitrarily structured serial programs and single tree structured parallel programs. In addition, the problems of partitioning chain structured parallel programs across chain connected systems and across shared memory (or shared bus) systems are also solved under certain constraints. All solutions for parallel programs are equally applicable to pipelined programs.

These results extend prior research in this area by explicitly taking concurrency into account and permit the efficient utilization of multiple computer architectures for a wide range of problems of practical interest.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
optimal Sum-Bottleneck paths parallel processing pipeline processing distributed computing partitions, assignments	59 - Mathematical & Computer Sciences (General) 66 - Systems Analysis Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	33	A03