NASA Contractor Report 178130

ICASE REPORT NO. 86-40

ICASE

APPROXIMATE ALGORITHMS FOR PARTITIONING
AND ASSIGNMENT PROBLEMS

M. Ashraf Iqbal

Contract Nos. NAS1-17070, NAS1-18107
June 1986

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

Approximate Algorithms for Partitioning and Assignment Problems

M. Ashraf Iqbal

Institute for Computer Applications in Science and Engineering
and
University of Engineering and Technology, Lahore, Pakistan

ABSTRACT

We consider the problem of optimally assigning the modules of a parallel/pipelined program over the processors of a multiple computer system under certain restrictions on the interconnection structure of the program as well as the multiple computer system. We show that for a variety of such programs it is possible to find in linear time if a partition of the program exists in which the load on any processor is within a certain bound. This method, when combined with a binary search over a finite range, provides an approximate solution to the partitioning problem.

The specific problems we consider are partitioning of (1) a chain structured parallel program over a chain like-computer system, (2) multiple chain like programs over a host-satellite system, and (3) a tree structured parallel program over a host-satellite system.

For a problem with m modules and n processors, the complexity of our algorithm is no worse than $O(mnlog(W_T/\varepsilon))$, where W_T is the cost of assigning all modules to one processor and ε the desired accuracy.

Supported by NASA Contracts NAS1-17070 and NAS1-18107 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center.

1. Introduction

With the proliferation of relatively cheap parallel computers in the research as well as the commercial field, it is becoming increasingly important to efficiently utilize the powerful hardware. One important requirement is that the task being executed be partitioned over the multiple computer system in an optimal fashion so as to minimize the total execution time of the job. In general the problem of finding the optimal partition of an arbitrarily connected distributed/parallel program over an arbitrarily connected multiple computer system is very difficult. If, however, the modules of the program communicate in a restricted manner and the multiple computer system has a special structure then it is possible to solve some of the partitioning problems. It has been shown in [1] that if the number of processors is limited to 2 then the partitioning problem can be solved efficiently for a distributed processor system. Similarly if the interconnection structure of the distributed program is tree-like then it is possible to efficiently partition the program over any number of processors [2].

The problem of optimally partitioning a modular program in a parallel processing environment is discussed in [3]. If the interconnection structure of the program is chain or tree-like and the parallel processor is either connected as a chain or is a host-satellite system, then [3] shows how the program may be partitioned optimally in polynomial time. Other related research in this field, which includes sub-optimal or approximate solutions to the partitioning problem, is reported in [4], [5] and [6].

In this paper we describe a *fully polynomial time approximation scheme* which provides approximate solutions to most of the partitioning problems discussed in [3] and already solved by using pure polynomial algorithms. In order to appreciate the usefulness of the approximate solutions, one should bear in mind that data for the problem being solved is often only known approximately. Hence an approximate solution may be as meaningful as an exact solution for many of the practical problems [4] where the extra accuracy of the exact solution is not needed and where the approximate solution can be obtained in a relatively short time [7].

In Section 2 we discuss an algorithm for finding the optimal partition of a chain structured parallel or pipelined program over a chain of identical processors. We assume that the program is made up of m modules numbered $1..m$ and has an intercommunication pattern such that module i can communicate only with modules $i+1$ and $i-1$. Similarly we assume that the multiprocessor of size $n<m$ has also a chain like architecture. We work under the constraint that each processor has a contiguous subchain of program modules assigned to it. Thus the partitions of the chains have to be such that modules i and $i+1$ are assigned to the same or adjacent processors. The optimal partitioning would then be the assignment of subchains of program modules to processors that minimizes the load on the most heavily loaded processor.

The central result of Section 2 is that for a trial weight w, it is possible to find if a partition exists in which the load on each processor is less than or equal to w in time proportional to $O(mn)$. The optimal partition is then found by making a binary search in a given range to find the partition for which w is minimum. The approach we use here is a fully polynomial time approximation scheme [8] and is an extension of the method discussed in [4] for finding the optimal partition of a one dimensional domain over a chain of processors.

In Section 3 and 4 we show that this technique can be used to optimally partition programs composed of multiple chains over a multiple computer system based on a single-host and multiple-satellite architecture. The time required by the entire system to complete the

processing is determined by the greater of (1) the individual load on the most heavily loaded satellite and (2) the sum of the collective loads on the host.

We discuss the partitioning of a chain-like program over a shared memory system in Section 5. In such a system, the total processing time is determined by the greater of (1) the individual load on the most heavily loaded processor and (2) the sum of communication costs between all pairs of processors that communicate through the shared memory [3]. We use Kernighan's approach [10] to design a probing function. The probing function returns *true* if it is possible to partition the program such that the load on any processor and the total communication cost is less than or equal to a trial weight w, and false otherwise. The optimal partition is then found by making a binary search in a finite range.

In Section 6 we discuss an approach to optimally partition a tree-structured parallel or pipelined program over a single-host multiple-satellite system. This algorithm is also based on a probing function.

The paper concludes with a discussion of our results in Section 7.

2. An Algorithm for Partitioning Chains

We describe in this section how a chain structured parallel or pipelined program can be optimally partitioned over a chain of identical processors. We assume that a chain structured program is made up of m modules numbered $1..m$ and has an intercommunication pattern such that module i can communicate only with modules $i+1$ and $i-1$. Similarly we assume that the multiprocessor of size $n<m$ has also a chain like architecture.

We work under the constraint that each processor has a contiguous subchain of program modules assigned to it. Thus the partitions of the chains have to be such that modules i and $i+1$ are assigned to the same or adjacent processors. The optimal partitioning would then be the assignment of subchains of program modules to processors that minimizes the load on the most heavily loaded processor.

It is convenient for us to assume that should the optimal assignment dictate that fewer than the available n processors be used, we can simply ignore the impact of communicating with the outside world through a subchain of unused processors. If desired, it is very simple to account for this overhead by concatenating dummy modules to the chain.

Notation:

w_i time consumed in the execution of module i.

c_i time to communicate between module i and $i+1$ if the two modules are assigned to different processors. We assume that c_0 (c_m), is the time for module 1 (m) to communicate with the outside world.

Definitions:

w_{max} maximum value of w_i for $1 \le i \le m$

W_T load on a processor if all the m modules are assigned to it. Thus $W_T = \sum_{i=1}^{m} w_i + c_0 + c_m$.

$\Omega_{j,k}$ load on a processor if subchain Modules[$j..k$] is assigned to it. It is given by $\sum_{i=j}^{k} w_i + c_k + c_{j-1}$.

Δ_k total remaining load to be assigned given that module k is the last module assigned to a processor. This is given by the following equation: $\Delta_k = \sum_{i=k+1}^{m} w_i + c_k + c_m$.

weight The *weight* of a partition is the weight of its heaviest subchain.

The assignment algorithm based on the function *PROBE*1 (described below), takes advantage of the fact that the load assigned to the most heavily loaded processor in the optimal partition lies somewhere between W_T and W_T/n. This is because at worst all modules are assigned to the same processor, which has load W_T. At best load is uniformly distributed over all processors with W_T/n on each. The algorithm selects a trial weight w in the above range and then uses the function *PROBE*1. The function *PROBE*1(w) returns *true* if it is possible to partition the chain of modules into subchains such that the load on each processor is less than or equal to w, and *false* otherwise. The partition that function *PROBE*1 obtains is called a *conservative partition.*

2.1. The Algorithm

```
function PROBE1( Processors[1..n], Modules[1..m],w):boolean;
begin
   j = 1; k = 0; p = 1; Δmin = WT;
   while p ≤ n do
      begin
         for x = j to m do
            if Ωj,x ≤ w and Δx < Δmin then
               begin
                  Δmin = Δx;
                  k = x;
                  Assign subchain Modules[j..k] to processor p;
                  if k = m then return(true);
               end;
         j = k+1; p = p+1;
      end;
   return(false);
end.
```

2.2. Discussion

In order to understand the working of the function *PROBE*1 it is important to note that:

1. The function assigns Modules[$j..k$] to processor p such that Δ_k is minimum and $\Omega_{j,k} \leq w$ for $j \leq k \leq m$. The corresponding minimum value of Δ_k is denoted by Δ_{min}.

2. If while assigning Modules[k+1..k'] to processor p+1 it is found that $c_k+w_{k+1}>w$ then obviously it is not possible to have a conservative partition of weight w and the function *PROBE*1 returns *false*.
3. If while assigning Modules[k+1..x] to processor p+1, it is found that $\Delta_x>\Delta_{min}$ when $\Omega_{k+1,x}\leq w$ for $k+1\leq x\leq m$ then the function again returns *false*. If instead the function assigns some modules to processor p+1 then either the condition $\Omega_{k+1,x}\leq w$, will be violated or the function will encounter a situation similar to (2). This can be explained with the help of Fig. 1. Suppose $\Omega_{k+1,x}\leq w$ and $\Delta_x>\Delta_{min}$ for $k+1\leq x\leq q$, while $\Omega_{k+1,x}>w$ and $\Delta_x<\Delta_{min}$ for $x>q$. To make sure that the load on processor p+1 is less than or equal to w the value of x should be equal to or less than q. But then $\Omega_{x+1,q+1}$ will become larger than $\Omega_{k+1,q+1}$ because $\Delta_x>\Delta_{min}$ and thus we are led to a situation similar to (2). Thus if k=0, p+1 is the first processor and $\Omega_{1,x}>W_T$ for $x>q$ then the optimal partition will require that all m modules be placed on a processor; otherwise, the load on some processors will become larger than W_T.

In the following paragraphs we shall first examine a detailed example of how the function *PROBE*1 works and then prove that if there exists a partition of weight w then the function *PROBE*1 will always find that or a better assignment.

2.3. An Example

Before proving correctness, let us examine a detailed example of how the function *PROBE*1 tries to find a conservative partition of weight w. Fig. 2 shows a 10 module chain to be mapped on a 4 processor chain with a trial weight w=20. The number below each module is its execution cost while the number above each edge is the communication cost for the two modules at the ends of that edge. Thus for module 4 the value of w_4=6 and c_4=12. The value of W_T for this problem is 54 (we have assumed that $c_0=c_m=0$).

Fig. 3 (bottom) shows a plot (grey line) of $\Omega_{1,x}$ for processor 1 against x. It can be seen from the figure that the load on processor 1 increases from zero, when no module is assigned to it, to W_T when all 10 modules are assigned to processor 1. The value of remaining load Δ_x is also plotted (black line) against x. This decreases from W_T, when no module is assigned to processor 1, to zero when all the 10 modules are assigned to it. It is evident from Fig. 3 that the rise of $\Omega_{1,x}$ and the fall of Δ_x with x, are not monotonic. Thus if we initially assign Modules[1..2] to processor 1 and then further assign Modules[3..4] to it then the value of Δ_x instead of decreasing, increases from 40 to 44. The reason for this behavior is non-uniform communication costs.

The subchain Modules[1..x] is assigned to processor 1 such that Δ_x is minimum and $\Omega_{1,x}\leq 20$. The value of x which satisfies the above constraints is 2. It can be seen from Fig. 3 that the condition ($\Omega_{1,x}\leq 20$) will still be satisfied if we further assign module 3 to processor 1, but then the remaining load will increase which will make it impossible to find a conservative partition of weight w=20 for the rest of the module chain in this example. The resulting assignment of modules to processor 1 is shown in Fig. 3 (top).

Having assigned modules to processor 1 we plot the load on processor 2, $\Omega_{3,x}$, (grey line) and the remaining load Δ_x (black line) as a function of x in Fig. 4. The resulting

assignment of modules to processor 2, shown in Fig. 4 (top), is selected on the same basis that $\Omega_{3,x} \leq 20$ and Δ_x should be minimum. Finally we draw $\Omega_{6,x}$ and Δ_x for processor 3 in Fig. 5. For the assignment of modules to processor 3, shown in Fig. 5 (top), the remaining load is just equal to the trial weight and thus it is possible to find a conservative partition of weight w of the module chain in the above example.

2.4. Proof of Correctness

Claim If a problem with m modules and n processors has a partition of weight w, *PROBE*1 will find that or a partition of less weight.

Proof By induction on n.

Consider the case n=2. Suppose the given partition of weight w assigns Modules[1..j_g] to processor 1 and Modules[j_g+1..m] to processor 2.

Apply *PROBE*1 to this problem. Suppose it assigns Modules[1..j_c] to processor 1 and j_c+1..m to processor 2. Because of the way in which *PROBE*1 proceeds, Δ_{j_c} will be minimized under the constraint $\Omega_{1,j_c} \leq w$. But because n=2, $\Delta_{j_c} = \sum_{i=j_c+1}^{m} w_i + c_{j_c} + c_m = \Omega_{j_c+1,m}$ = the weight of the second partition.

Thus the weight of the second partition will be minimized under the constraint that the weight of the first partition is $\leq w$. If there exists a partition in which the weight of both subchains is $\leq w$, *PROBE*1 will clearly find it. The claim is thus true for n=2. Note that the proof is independent of m.

We will now show that if the claim is correct for $n=k$ it is also correct for $n=k+1$. Suppose we are given a chain of m modules which has a partition of weight w. Assume that in this given partition, Modules[1..j_g] are assigned to processor 1 and j_g+1..k_g to processor 2.

Starting with module 1, scan the modules from left to right to identify the module j_c such that $\Omega_{1,j_c} < w$ and Δ_{j_c} is minimum. Delete the nodes[1..j_c].

Three cases are now possible:

Case(1) $j_c = j_g$: the subchain deleted corresponds to the first subchain of the given partition. In this case the remaining nodes[j_c+1..m] *must* have a partition with weight w and $n=k$ subchains (because the original chain was given with $n=k+1$ subchains).

Case(2) $j_c < j_g$: this means that $\Delta_{j_c} < \Delta_{j_g}$ which implies that $\Omega_{j_c,k_g} < \Omega_{j_g,k_g} \leq w$, i.e. the second subchain of the given partition has had its weight reduced below w.

Case(3) $j_c > j_g$: again only possible if $\Delta_{j_c} < \Delta_{j_g}$ which implies that $\Omega_{j_c,k_g} < \Omega_{j_c,k_g} \leq w$, i.e. the second subchain has had its weight reduced.

In all three cases the remaining nodes[j_c+1..m] must have a partition with weight w and $n=k$ subchains.

By applying *PROBE*1 to the remaining chain j_c+1..m, we can obtain a partition of weight w and $n=k$ subchains (since the algorithm is assumed correct for $n=k$).

By concatenating the deleted chain $1..j_c$ with the partition obtained above, we will get a partition of weight w and $n=k+1$ subchains.

Now recall that node j_c was selected under the constraint that $\Omega_{1,j_c} \leq w$ and Δ_{j_c} was minimum. Thus the deletion of $1..j_c$ followed by the application of *PROBE*1 is equivalent to the application of *PROBE*1 for $n=k+1$. This proves that if the claim is true of $n=k$ it is also true for $n=k+1$.

We have already proved the claim to be true for $n=2$. It is therefore true for all n.

The algorithm makes a binary search in the range W_T/n, W_T using the function *PROBE*1 to find the partition for which the weight of the heaviest subchain is minimum. For each trial weight w the function *PROBE1* has to look at each module only once for each processor. Thus for m modules and n processors the function *PROBE*1 will perform $O(mn)$ steps to find a conservative partition, if it exists. If the above range is resolved to an accuracy of ε then the algorithm will find a conservative partition of weight w in time proportional to $O(mn\log_2(W_T/\varepsilon))$ with the assurance that w is no greater than the weight of the heaviest subchain in the optimal assignment by ε. Thus the order of the algorithm is $O(mn\log_2(W_T/\varepsilon))$. It is important to note that the time complexity of the algorithm is proportional to $\log(W_T/\varepsilon)$ unlike other fully polynomial time approximation schemes which are polynomial in $1/\varepsilon$ [8].

3. Partitioning Multiple Chains across a Host-Satellite System

The algorithm presented in the previous section can be used to solve several other partitioning problems in Host-Satellite Systems as shown in Fig. 6. Let us assume that each chain has m modules, there are n satellites and that for each module i of satellite s the time required to run it on the satellite, $e_{i,s}$, and on the host, $h_{i,s}$. For each pair of modules i and $i+1$ from satellite s we have the time required for interprocessor communication, $c_{i,s}$, should i be assigned to the satellite and $i+1$ 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 is determined by the greater of (1) the individual load on the most heavily loaded satellite and (2) the sum of the collective loads on the host.

We represent the load on satellite s by $\Omega_{k,s}$, provided Modules[1..k] are assigned to it. It is given by $\sum_{i=1}^{k} e_{i,s}+c_{k,s}$. The remaining load, due to the rest of the modules of satellite s, is assigned to the host and is denoted by Δ_s, which is equal to $\sum_{i=k+1}^{m} h_{i,s}+c_{k,s}$.

The probing function, while trying to find a conservative partition of weight w, will assign Modules[1..k] to satellite p such that $\Omega_{k,s} \leq w$ and Δ_s is minimum. If the total load on the host which is equal to $\sum_{s=1}^{n} \Delta_s \leq w$ then the function returns *true* and the conservative partition, and *false* otherwise.

3.1. An Example

Let us consider an example of two satellites each having 10 modules. For the sake of simplicity assume that $e_{i,s}=h_{i,s}$. The two satellite chains are shown in Fig. 7 (top). $\Omega_{k,1}$ (the load on satellite 1) and Δ_1 (the remaining load on the host) are plotted against k in grey and black lines respectively in Fig. 7 (middle). In Fig. 7 (bottom) we plot the corresponding values of $\Omega_{k,2}$ and Δ_2 for satellite 2, against k. The total load on the host is the sum of Δ_1 and Δ_2. For example if k=5 for satellite 1, and k=8 for satellite 2 then $\Omega_{k,1}$ and $\Omega_{k,2}$ will be 33 and 40 respectively and the total load on the host will be equal to 29+16=45.

For a trial weight w=44, the probing function will select k=7 for satellite 1. In this case Δ_1 will be 20. Note that for any other value of k either the value of $\Omega_{x,1}$ is larger than w or Δ_1 is not as small as 20 as shown in Fig. 7 (middle). Similarly for satellite 2, the selected value of k is 8 and so Δ_2 will be 16. The total load on the host will then become 20+16=36 which means that a conservative partition with w=44 exists as shown (bold line) in Fig. 7 (top).

Thus for each satellite s, the probing function selects $\Omega_{k,s}$, such that Δ_s is minimum. All satellite chains are independent of each other. Thus if Δ_p is minimum for each p, where $1\leq p\leq n$, then $\sum_{p=1}^{n}\Delta_p$ will also be minimum. Now if there exists a partition in which the total processing time is less than or equal to w then for each individual satellite s, this partition can always be transformed into a conservative partition of weight w by increasing or decreasing k until Δ_s becomes minimum and $\Omega_{k,s}\leq w$. The only result of this transformation will be that the load on the host will either decrease or remain the same. Thus if there exists a partition of weight w then this approach will always find that or an equivalent assignment.

The algorithm makes a binary search in the range W_T, W_T/n, where W_T is given by equation (2), using the probing function to find the conservative partition of weight w for which w is minimum.

$$W_T = \min\left[\left(\sum_{s=1}^{n}\sum_{i=1}^{m}h_{i,s}\right),\left(max_{s=1,n}\left(\sum_{i=1}^{m}e_{i,s}\right)\right)\right] \qquad (2)$$

Note that W_T is the smaller of (1) total processing time if all modules are assigned to the host and (2) total processing time if no module is assigned to the host. Thus if $w_{\max}$ is the maximum value of $h_{i,s}$ for $1\leq i\leq m$ and $1\leq s\leq n$ then $W_T\leq mn(w_{\max})$.

For each trial weight w the probing function has to look at each module at least once for each satellite before a decision is made to assign this module to the satellite or not. Thus the function will perform $O(mn)$ steps to find a conservative partition of weight w if it exists. If the range W_T, W_T/n is resolved to an accuracy of ε then the algorithm will find a conservative partition of weight w in time proportional to $O(mn\log_2(W_T/\varepsilon))$ with the assurance that w is no greater than the worst load on any satellite and the total load on the host in the optimal assignment by ε.

4. Partitioning Distributed Programs in Host-Satellite System

Stone has solved the problem of partitioning a distributed program over a single-host and single-satellite system in [1]. He has further studied the behavior of the optimal assignment as a function of load on the host and shows that a nesting property holds [9]. As load increases on the host, modules move away from the host and onto the satellite. At no point does an increase in load cause a module to move from the satellite onto the host.

Subsequent work by Bokhari [3] shows how this property can be exploited when finding optimal assignments in a single-host multiple-satellite system. We can consider the individual programs to have chain-like structure, regardless of their actual interconnection. The optimal assignment can be found using a Sum-Bottleneck path algorithm. The complexity of this approach is dominated by the $O(m^4n)$ algorithm that finds the individual chains, for a problem with n satellites, each executing a program with m modules. The partitioning of the chains takes far less time than $O(m^4n)$ time.

We can find the partitioning of the chains using an approach similar to the one discussed in the previous section. This takes $O(mnlog(W_T/\varepsilon))$ time, where W_T and ε are as defined in the previous section. The overall complexity of the algorithm is still $O(m^4n)$.

5. Partitioning Chains in Shared Memory System

Consider a chain of m modules numbered $1..m$. Each module i has an associated execution cost w_i and each edge (i,j) has a communication cost $c_{i,j}$, should modules i and j be placed on different processors in a shared memory system. Under such a system, the total processing time is determined by the greater of (1) the individual execution load on the most heavily loaded processor and (2) the sum of communication costs between all pairs of processors that communicate through the shared memory [3]. It has been shown by Kernighan [10] that it is possible to find a partition of an m module chain into n disjoint subsets such that the size of each subset is less than or equal to a given constant and the sum of costs on edges joining nodes in different subsets is minimum in time proportional to $O(m)$. Using this approach we can design a probing function which can find if a partition of the chain-like program exists in which the load on any processor is less than or equal to a trial weight w. If, in the resulting partition the sum of communication costs on edges joining modules on different processors is less than or equal to w then the probing function returns *true* and *false* otherwise. We can then make a binary search in the range $\sum_{i=1}^{m} w_i$, $\sum_{i=1}^{m} w_i/n$ using the probing function to find the partition for which w is minimum assuming that there are n processors in the shared memory system.

6. Partitioning Trees in Host-Satellite System

We consider the problem of partitioning a tree structured pipelined or parallel program over a single-host, multiple-satellite system as shown in Fig. 10. We assume that there are m nodes in the program tree and there are as many satellites as the number of leaf nodes of the tree. We work under the constraints that (1) individual maximal subtrees of the given tree are assigned to each satellite and (2) that the root is always assigned to the host. The total

processing time under such a system will be the larger of the load on the host and the worst load on any satellite.

Notation:

h_i time consumed in the execution of module i on the host.

e_i time consumed in the execution of module i on any satellite (all satellites are similar).

c_i time required for communication if node i is assigned to a satellite and node *father*(i) to the host.

Definitions:

$w_{\max}$ maximum value of h_i for $1 \le i \le m$.

load(i) load on a satellite if node i and all its children nodes are assigned to the satellite. This is equal to the sum of individual e_i's of the modules assigned to the satellite plus c_i.

cost(i) cost of execution of node i and its children on the host. This is equal to the sum of individual h_i's of node i and its children.

W_T load on the host if all m nodes of the program tree are assigned to the host. This is equal to $\sum_{i=1}^{m} h_i$.

depth(i) distance of node i from the root. The value of *depth*(*root*)=0.

$d_{\max}$ the maximum value of *depth*(i) for $1 \le i \le m$.

Host_Load total load on the host. If the program tree is partitioned over a host and n satellites then for each satellite p there will be a node $\pi(p)$ assigned to the satellite while *father*($\pi(p)$) will be assigned to the host. The value of *Host_Load* will then be the sum of individual h_i's of the modules assigned to the host plus $\sum_{p=1}^{n} c_{\pi(p)}$. In terms of W_T this is given by:

$$Host\_Load = W_T - \sum_{p=1}^{n} cost(\pi(p)) + \sum_{p=1}^{n} c_{\pi(p)}.$$

Δ_i Suppose we have assigned all nodes to the host except the n sons of node i numbered *1..n*. The value of *Host_Load* will then be $W_T - \sum_{k=1}^{n} (cost(k) - c_k)$. If, instead of assigning each son of node i to a separate satellite, node i itself is assigned to a satellite then the new value of *Host_Load* will be $W_T - cost(i) + c_i$. The difference between the two values of *Host_Load* is denoted by Δ_i and is equal to $\Delta_i = (cost(i) - c_i) - \sum_{k=1}^{n} (cost(k) - c_k)$. Thus if Δ_i is positive then the value of *Host_Load* will reduce by Δ_i if we assign a single satellite to node i (and its children) instead of assigning a separate satellite to

each son of node i. For each leaf node i, the value of Δ_i is $cost(i)-c_i$.

The algorithm selects as before a trial weight w and then uses the function *PROBE_TREE* (described below). The function *PROBE_TREE*(w) returns *true* if it is possible to partition the program tree over the host-satellite system such that the load on any satellite and the load on the host is less than or equal to w, and *false* otherwise. The resulting partition, if any, is called the *conservative partition* for a trial weight w.

6.1. The Algorithm

```
function PROBE_TREE(w):boolean;

procedure MERGE(i,father(i));
begin
    e_father(i) = e_father(i)+e_i;
    h_father(i) = h_father(i)+h_i;
    remove edge(i,father(i));
end;

begin
    for level=d_max down to 1 do
        begin
            for each node i at depth(i)=level do
                if load(i)>w then Merge(i,father(i));
        end;
    Host_Load = W_T;
    for level=d_max down to 1 do
        begin
            for each node i at depth(i)=level do
                if Δ_i<0 then Merge(i,father(i))
                        else Host_Load = Host_Load − Δ_i;
        end;
    if Host_Load≤w then return(true) else return(false);
end.
```

6.2. Discussion

1. The function *PROBE_TREE*, while trying to find a conservative partition of weight w, will assign a node i and its children to a satellite if and only if $load(i) \leq w$. Each node j for which $load(j)>w$ is therefore merged with $father(j)$ by combining the execution cost e_j (h_j) with $e_{father(j)}$ ($h_{father(j)}$) and removing the edge(j,$father(j)$). The problem is now reduced to partitioning the new program tree, in which $load(i) \leq w$ for each node i other than the root, in such a fashion that the value of *Host_Load* is minimum.
2. We assume that initially all m nodes are assigned to the host and thus the initial value of *Host_Load* is equal to W_T. The function, while examining each leaf node i at

depth(i)=d_{max}, assigns it to a satellite if $\Delta_i \geq 0$. If, on the other hand, $\Delta_i < 0$ then node i is merged with *father*(i) by combining h_i with $h_{father(i)}$ and removing the edge(i,*father*(i)). In the resulting partition the value of *Host_Load* will reduce by an amount equal to the sum of individual Δ_i's for each leaf node i assigned to a satellite.

3. The function then looks at each node j at a depth one less than d_{max}. Remember that in the previous iteration of the for loop each son of node j has either already been assigned to a satellite or been merged with node j. If $\Delta_j \geq 0$ then the value of *Host_Load* will further reduce by Δ_j if the previous partition is moved up one level by assigning a satellite to node j and to its children instead of keeping the previous partition in which each son of node j has been assigned to a separate satellite. If however $\Delta_j < 0$ then node j is merged with *father*(j) and the previous partition is maintained (assigning node j to a satellite will increase the value of *Host_Load* instead of reducing it).
4. The function *PROBE_TREE* works from bottom to top in the program tree reducing the value of *Host_Load* at each iteration of the for loop by an amount equal to the sum of individual Δ_i's for each node i examined during that iteration and not merged with its father. Thus for each node j examined during an iteration, the decision to keep the previous partition, or to move the partition up one level by assigning a single satellite to node j, is solely dependent upon node j and its sons and is not influenced by any other nodes examined during that iteration. The nodes merged with the root in the last iteration are assigned to the host and the value of *Host_Load* is compared with w.
5. The resulting value of *Host_Load* is equal to W_T minus the sum of individual Δ_k's for each node k of the program tree which is examined by the function and not merged with *father*(k). It is important to note that the policy according to which nodes are assigned to satellites makes sure that the value of *Host_Load* reduces by a maximum amount.

6.3. An Example

Let us now consider an example of a tree structured program consisting of 32 nodes as shown in Fig. 8(a). For the sake of simplicity it is assumed that the execution cost of a module is the same on the satellite as well as on the host and is shown inside each node in Fig. 8(a). The number associated with each edge is the communication cost for the two modules at the ends of that edge. Trial weight w=140.

Each node i for which *load*(i)>140 is merged with *father*(i) and the edge(i,*father*(i)) is removed. The nodes merged and the edges to be removed are shown in bold in Fig. 8(b). The 32 node program tree is thus transformed into a 28 node program tree as shown in Fig. 8(c).

Initially all the remaining 28 nodes are assigned to the host. The value of *Host_Load* will then be equal to W_T=175. The value of Δ_i for each node i at *depth*(i)=5 is shown in bold outside each node in Fig. 8(c). The function assigns each node i to a satellite if $\Delta_i \geq 0$ and merges it with its father if $\Delta_i < 0$, the resulting partition is also shown in the figure. The value of *Host_Load* for this partition is 175-(3+9+5+4+4+5+8)=137. The function goes up one level and examines each node i at *depth*(i)=4. If the value of $\Delta_i > 0$ then node i is assigned to a satellite and the previous partition is moved up one level with the result that the value of *Host_Load* further reduces by Δ_i. The resulting partition is shown in Fig. 8(d) with *Host_Load*=137-(7+5+14+9)=102. In the third iteration of the second outermost for loop, the

value of the *Host_Load* is further reduced by 6+9 as shown in Fig. 8 (e). The function then examines each node at a depth equal to 2. The value of *Host_Load* is further reduced to 87-(7+15+12)=53 with the partition shown in Fig. 8(f). In the last iteration 3 nodes are merged with the root and are thus assigned to the host, the resulting partition is shown in Fig. 8(g). The final value of *Host_Load* is however the same as in the last iteration equal to 53.

6.4. Proof of Correctness

Claim If a tree structured program has a partition over a host satellite system in which the worst load on any satellite and the load on the host is w, then the function *PROBE_TREE* will find that or an equivalent assignment.

Definitions:

height d_{max}+1 minus *depth*(i) for a node i. Thus *height* of root is d_{max}+1.

$\pi_{w,h}$ a partition which guarantees that the load assigned to the host is minimum provided only nodes with *height*$\leq h$ are permitted to reside on the satellites.

Proof In a tree structured program, where each node i is already merged with *father*(i), if *load*(i)>w, the above claim will be true if the function *PROBE_TREE* can find $\pi_{w,d_{max}}$.

By induction on *height.*

Consider the case *height*=1. The function initially assigns all m nodes to the host and thus the starting value of *Host_Load* is W_T. During the execution of the first iteration of the for loop, the function examines each leaf node i at *height*=1 and assigns it to a satellite if $\Delta_i \geq 0$. If on the other hand $\Delta_i < 0$ then node i is merged with the *father*(i). In the resulting partition the value of *Host_Load* will reduce by an amount equal to the sum of individual Δ_i's for each node i assigned to a satellite. Obviously, the partition found after the first iteration will be $\pi_{w,1}$.

We will now show that if the function *PROBE_TREE* can find $\pi_{w,k}$ after the kth iteration then it can also find $\pi_{w,k+1}$ after the k+1 th iteration of the for loop.

After finding $\pi_{w,k}$ the function looks at each node i at *height* equal to k+1. If the value of $\Delta_i \geq 0$ then for each node i the previous partition $\pi_{w,k}$ is moved up one level by assigning a satellite to node i and its children instead of keeping the previous partition, and thus the value of *Host_Load* further reduces by Δ_i. If however $\Delta_i < 0$ then the previous partition $\pi_{w,k}$ is maintained. For each node i examined at *height*=k+1, the decision to keep the previous partition or to move the partition up one level is solely dependent upon Δ_i and is not influenced by any other node examined during that iteration. Note that the partition $\pi_{w,k+1}$ will either be the previous partition $\pi_{w,k}$ or the new partition in which node i at *height*=k+1 is assigned to a satellite (along with its children). It can not be any other partition because by definition, $\pi_{w,k}$ guarantees

that the value of *Host_Load* is minimum under the constraint that only those nodes with *height*$\leq k$ are permitted to reside on the satellites.

Thus if the function can find $\pi_{w,k}$ after the kth iteration of the for loop then it will be transformed into $\pi_{w,k+1}$ after the k+1 th iteration and into $\pi_{w,d_{max}}$ after the last iteration.

The algorithm makes a binary search in the range W_T, W_T/m, and it uses the function *PROBE_TREE* to find a conservative partition of weight w for which w is minimum. For each trial weight w, the function *PROBE_TREE* has to examine each node only once to decide whether to assign the node and its children to a satellite or to merge it with its father. Thus the function performs $O(m)$ steps to find a conservative partition of weight w, if it exists. If the range W_T, W_T/m is resolved to an accuracy of ε then the algorithm will find a conservative partition of weight w in time proportional to $O(m\log_2(W_T/\varepsilon))$ with the assurance that w is no greater than the larger of the load on the host and the worst load on any satellite in the optimal assignment by ε.

7. Conclusions.

We have discussed a number of partitioning problems in the field of parallel, pipelined and distributed computing. We have demonstrated that in a variety of such problems it is possible to design a probing function which can find out if a partition of a parallel program over a multiple computer system exists in which the load on any processor is less than or equal to a given weight w. It has been shown that for a parallel program with a chain or tree-like interconnection structure, the probing function provides a *true/false* answer in linear time provided the processor system is also limited to a chain of processors or is a host satellite system. The optimal partition is then found approximately by making a binary search in a finite range to find the partition for which w is minimum.

In order to extend this approach to other problems, it is essential to find an efficient probing function. The rule on which the probing function is based is dependent upon the nature of the partitioning problem to be solved. For example, in the partitioning of a one dimensional domain over a chain of processors, the probing function was simply a greedy method [4] while in case of partitioning a chain-like program over a shared memory system it was based on a dynamic programming approach described in [10]. Once an efficient probing function is found for a problem, the optimal partitioning can be found by making a binary search in a given range using the probing function.

Future work in this field requires that this approach be extended to multiple computer systems with a richer interconnection structure like a binary tree, hypercube or a mesh. It will be interesting to find if this approach can be efficiently applied to multiple computer systems composed of dissimilar processors. It also remains to be seen how efficiently this type of approach can be applied to a two dimensional domain, with non-uniform work loads, which is to be partitioned into areas requiring equal computational effort [11].

8. References

[1] 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.

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

[3] S. Bokhari, "Partitioning Problems in Parallel, Pipelined and Distributed Computing," ICASE Report No. 85-54, November, 1985.

[4] M. A. Iqbal, J. H. Saltz and S. H. Bokhari, "Performance Tradeoffs in Static and Dynamic Load Balancing Strategies," ICASE Report No. 86-13, March, 1986.

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

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

[7] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms,* Computer Science Press, Inc., 1978.

[8] M. Garey and D. Johnson, *Computers and Intractability,* W. H. Freeman and Company, 1979.

[9] H. S. Stone, "Critical load factors in distributed computer systems," *IEEE Trans. Software Engineering,* vol. SE-4, No. 3, pp. 254-258, May 1987.

[10] B. Kernighan, "Optimal Sequential Partitions of Graphs," *JACM,* vol. 18, No. 1, pp. 34-40, January 1971.

[11] M. Berger and S. Bokhari, "A Partitioning Strategy for Non-Uniform Problems across Multiprocessors," ICASE Report No. 85-55, November, 1985. To appear in IEEE Trans. Computers.

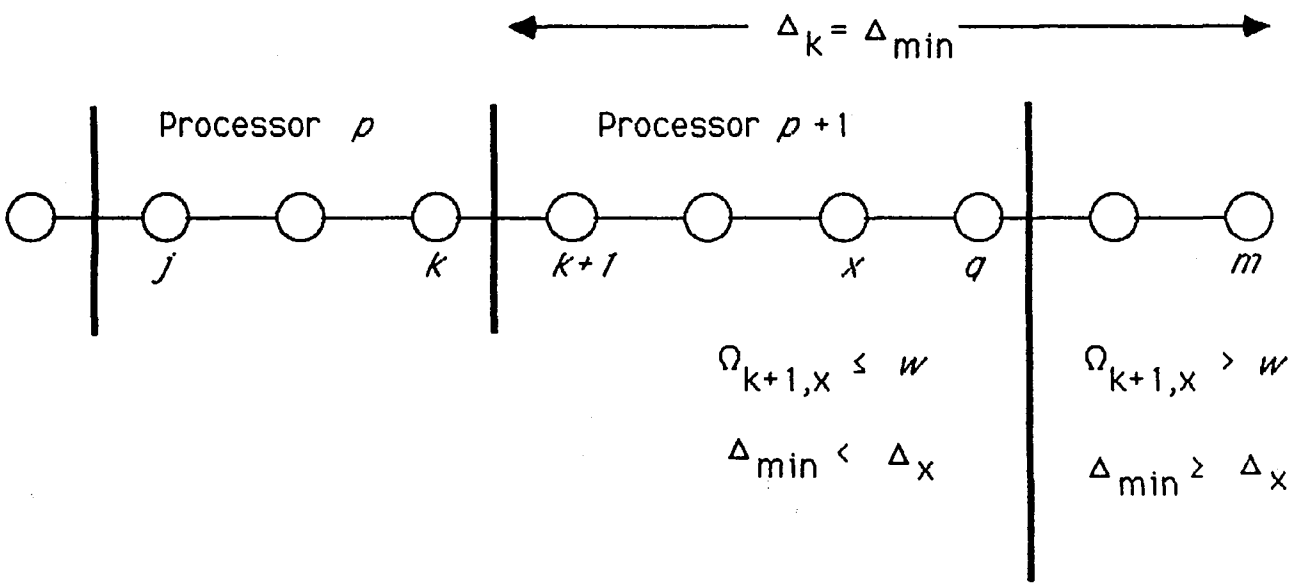


Fig. 1 Subchain Modules[$j..k$] are assigned to Processor p while Modules[$k+1..x$] are assigned to Processor $p+1$. If x is smaller than q then $\Omega_{x+1,q+1} > \Omega_{k+1,q+1}$ because $\Delta_x > \Delta_k$.

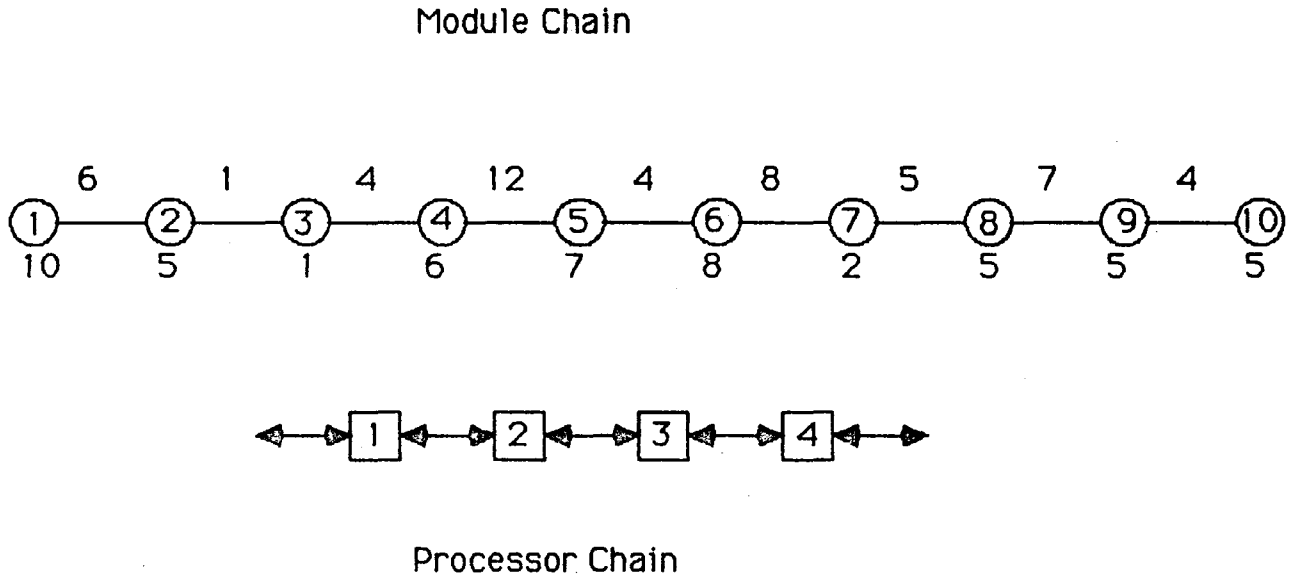
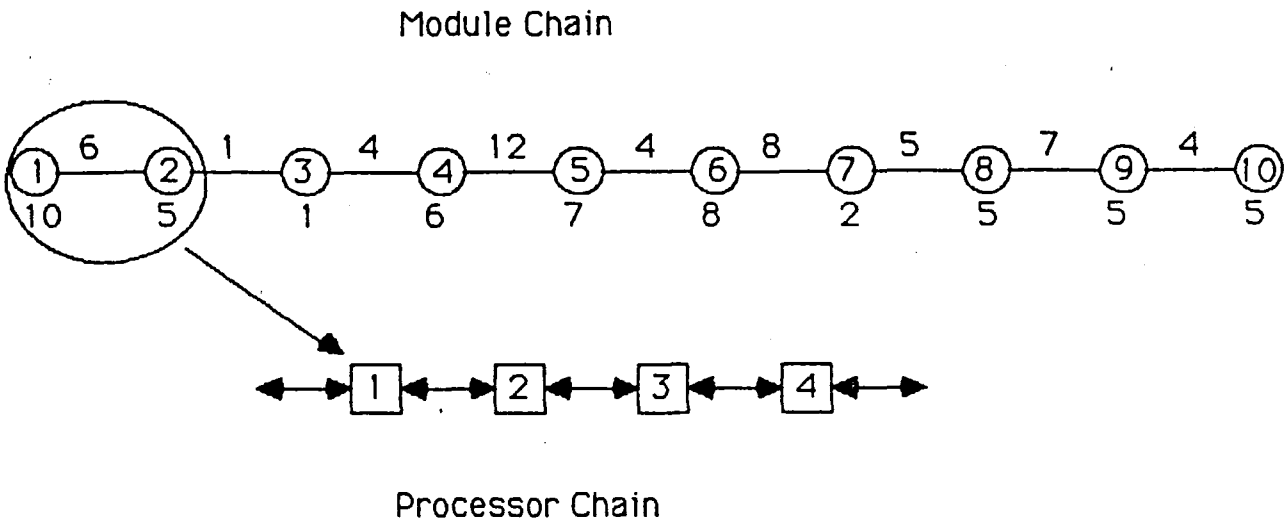


Fig. 2 A 10 module chain to be mapped on a 4 processor chain. The number below each module is its execution cost. The number above each edge is the communication cost for the two modules at the ends of that edge.



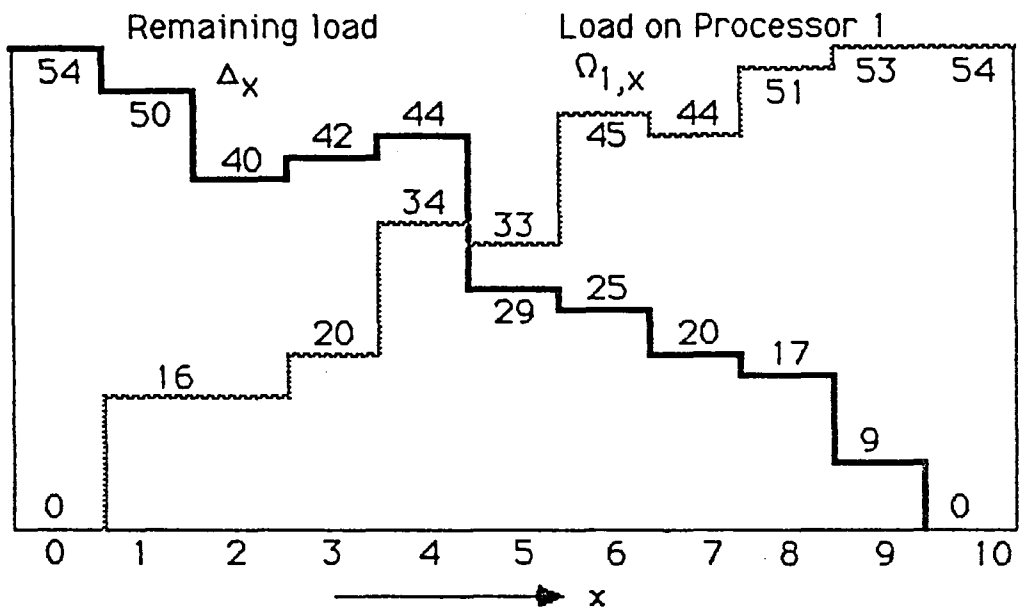
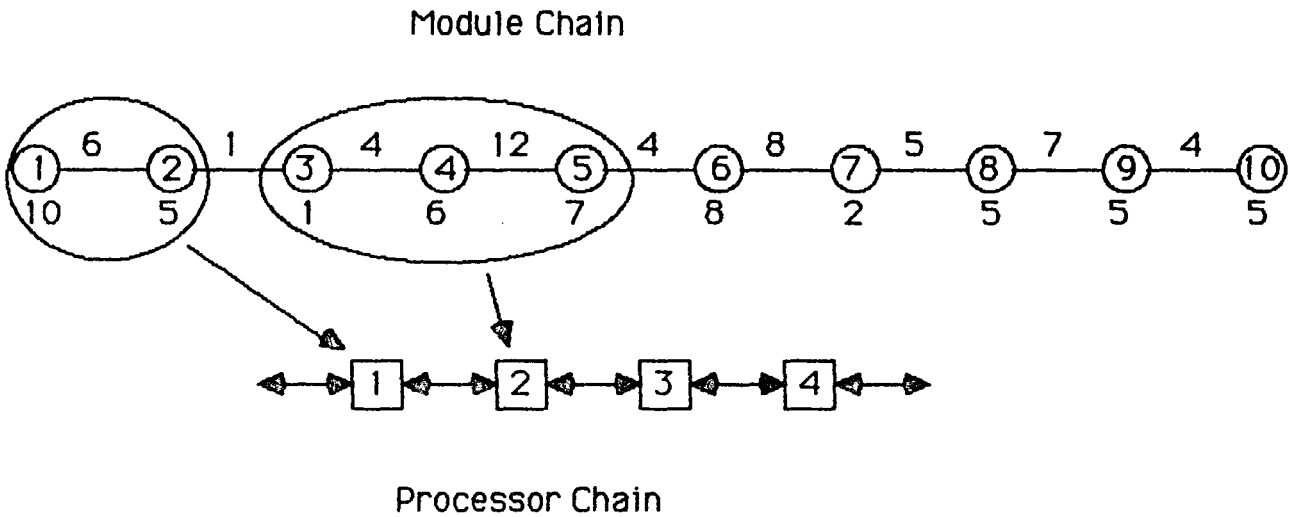
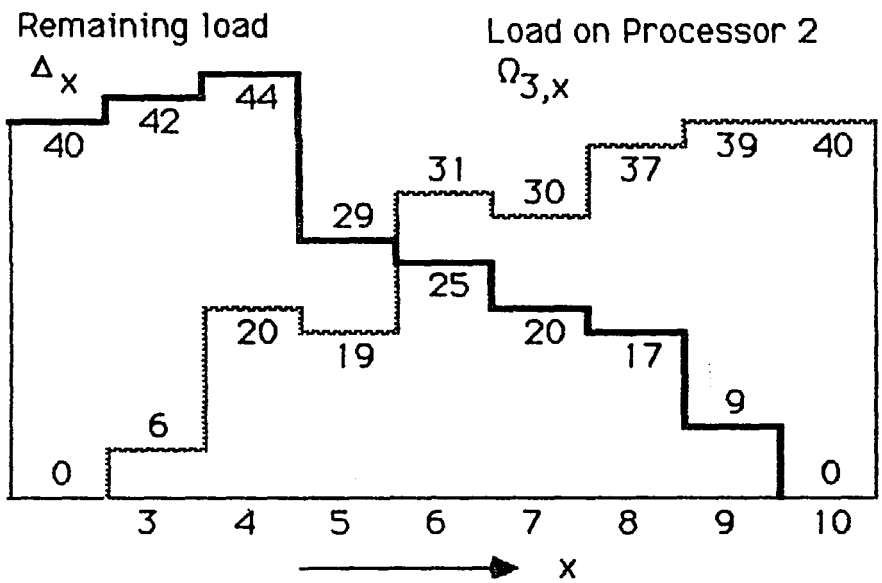


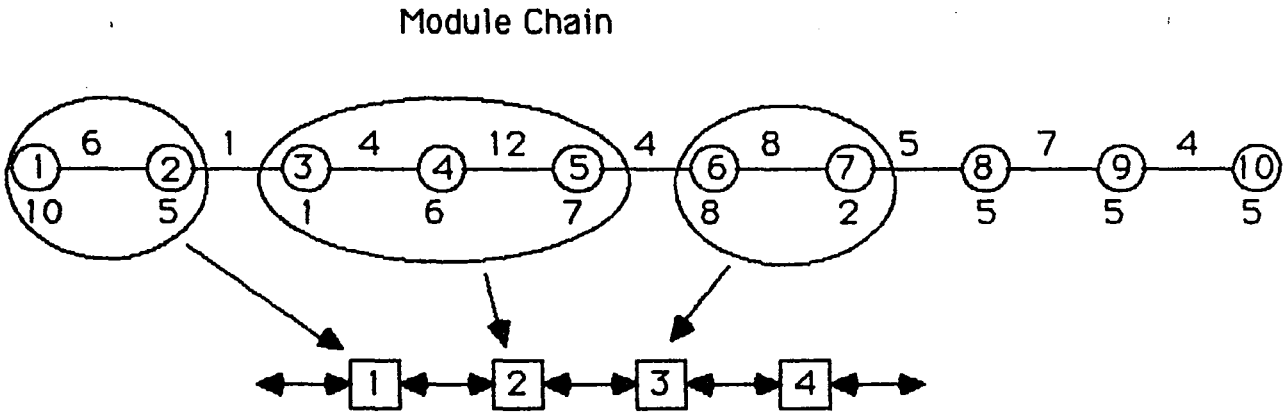
Fig. 3 The load on processor 1, $\Omega_{1,x}$, (grey line) and the remaining load Δ_x (black line). Trial weight $w = 20$.





Modules[3..x] assigned to Processor 2

Fig. 4 The load on processor 2, $\Omega_{3,x}$,(grey line) and the remaining load Δ_x (black line).



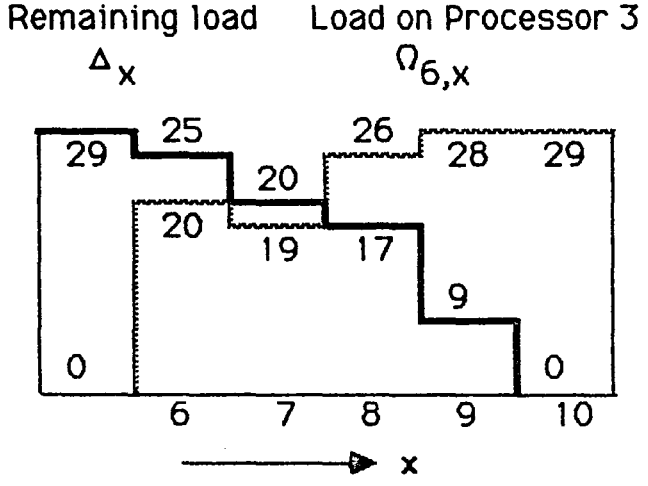


Fig. 5 The load on processor 3, $\Omega_{6,x}$, (grey line) and the remaining load Δ_x (black line).

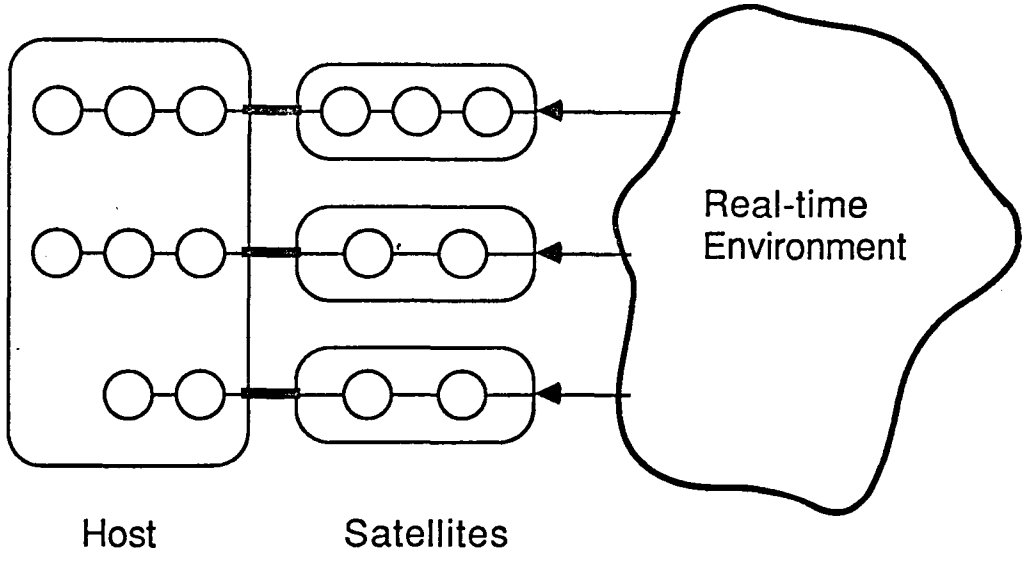


Fig. 6 A host satellite system processing real time data.

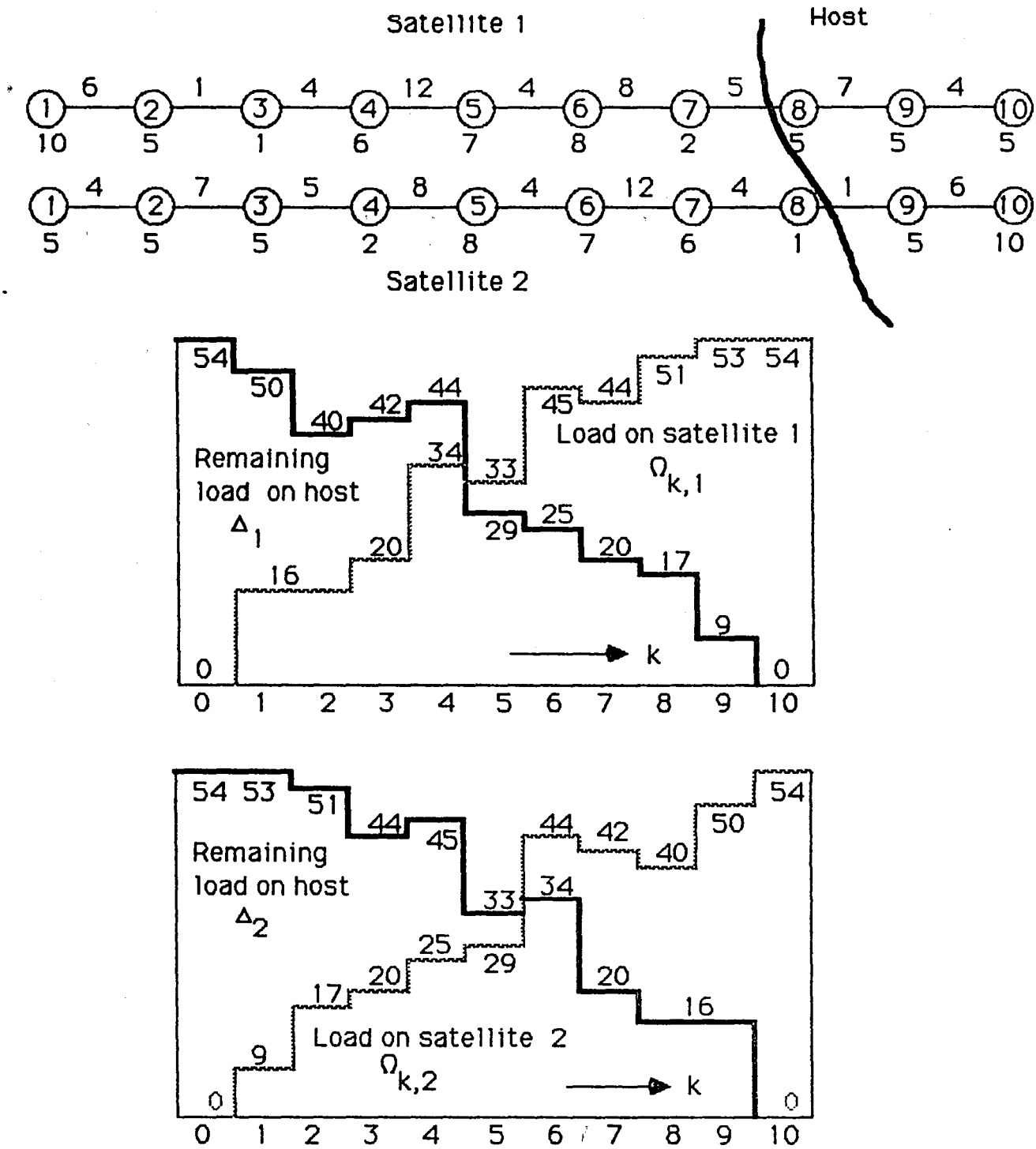


Fig. 7 (top) The two satellite chains each having 10 modules. The number below each module is its execution cost. The number above each edge is the communication cost for the two modules at the ends of that edge. (middle) The load on satellite 1, $\Omega_{k,1}$, (grey line) and the remaining load on Host Δ_1 (black line). (bottom) The load on satellite 2 $\Omega_{k,2}$, (grey,line) and the remaining load on Host Δ_2 (black line). Trial weight w =44.

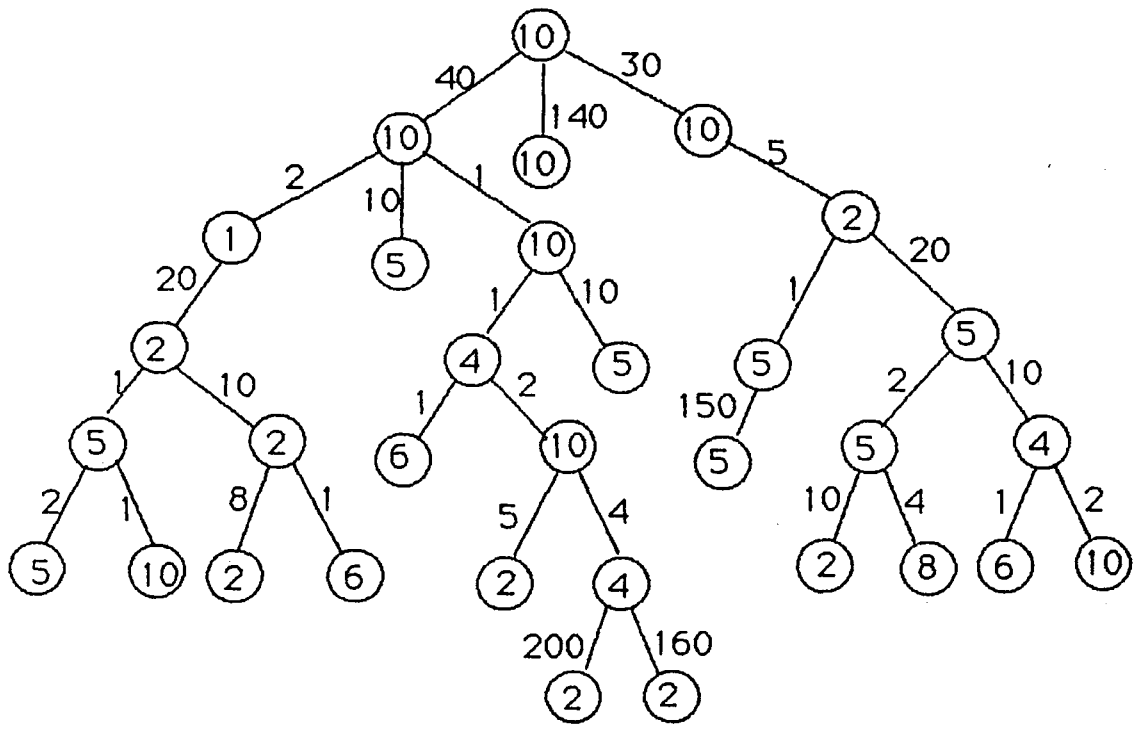


Fig. 8(a) A 32 node tree-structured program tree to be partitioned over a host-satellite system. The number inside each module is the execution cost on the host as well as on a satellite. The number associated with each edge is the communication cost for the two modules at the ends of that edge. Trial weight $w = 140$.

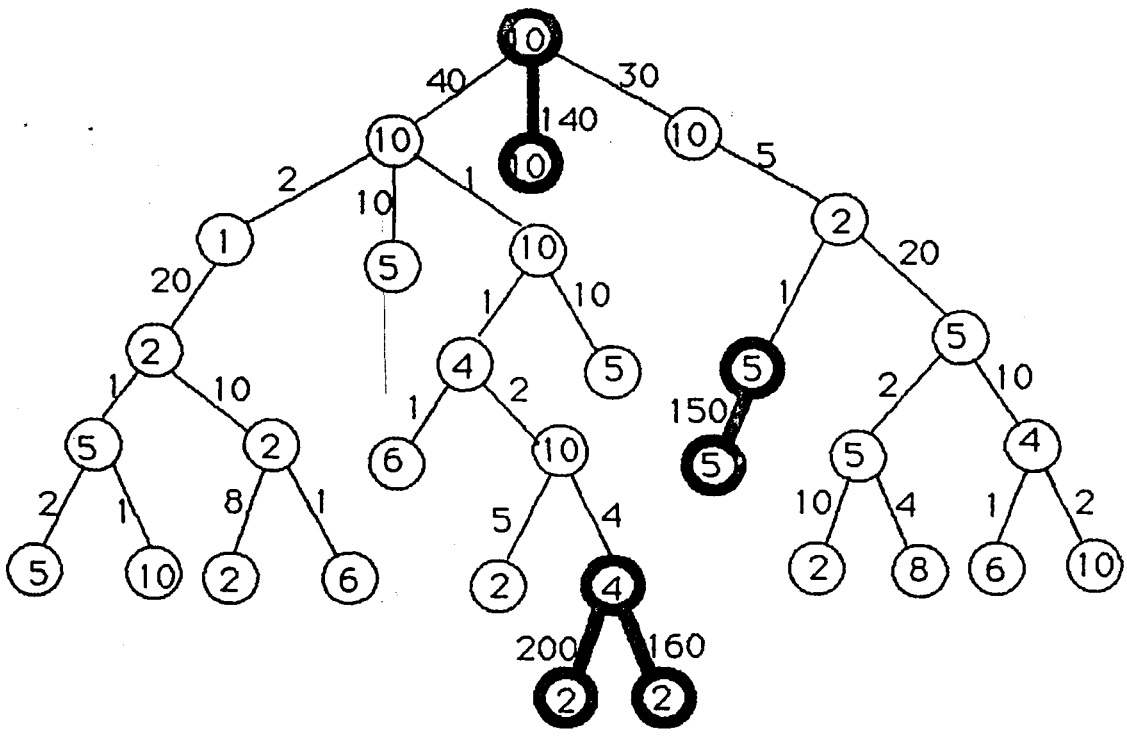


Fig. 8(b) Each node *i*, for which *load(i)*>140 is merged with *father(i)* and the edge(*i, father(i)*) is removed. The nodes merged and the edges removed are shown in bold. The 32 node program tree is thus transformed into a 28 node program tree.

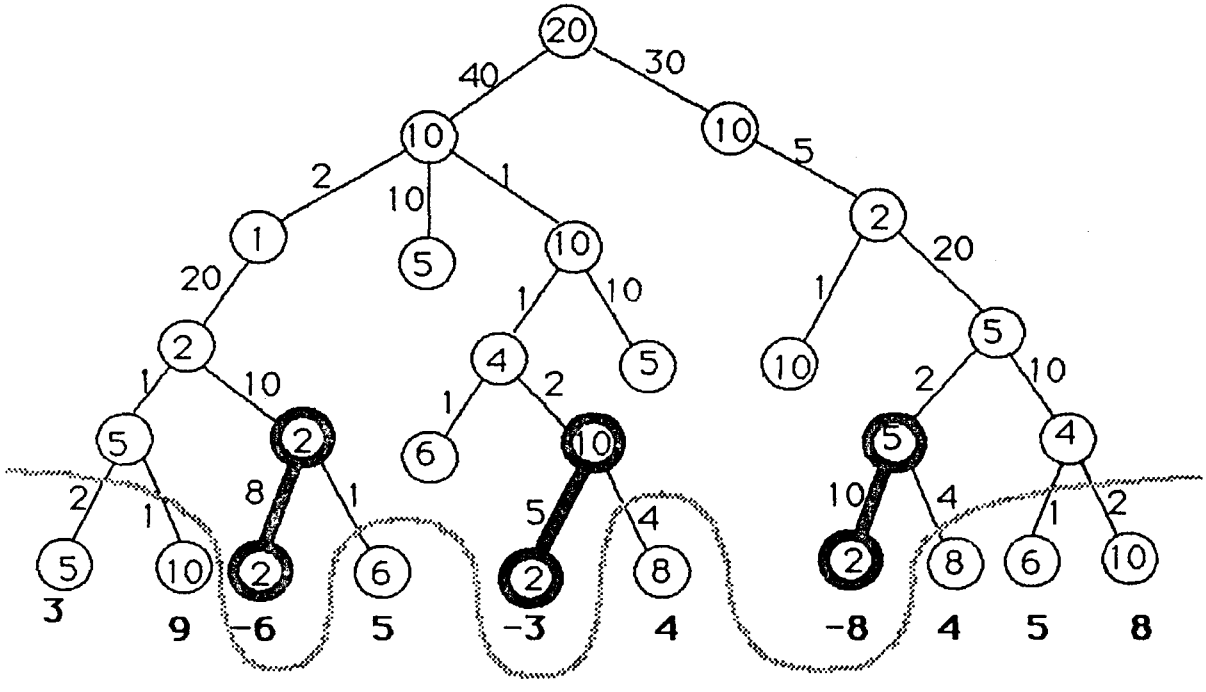


Fig. 8(c) The partition (shown by a grey line) is generated by the function when *level* = 5. For each node i at that level the value of Δ_i is shown in bold below each node. The nodes merged are shown in bold and are above the grey line. Each node below the grey line is assigned to a satellite. Thus net reduction in Host_Load is 3+9+5+4+4+5+8=38.

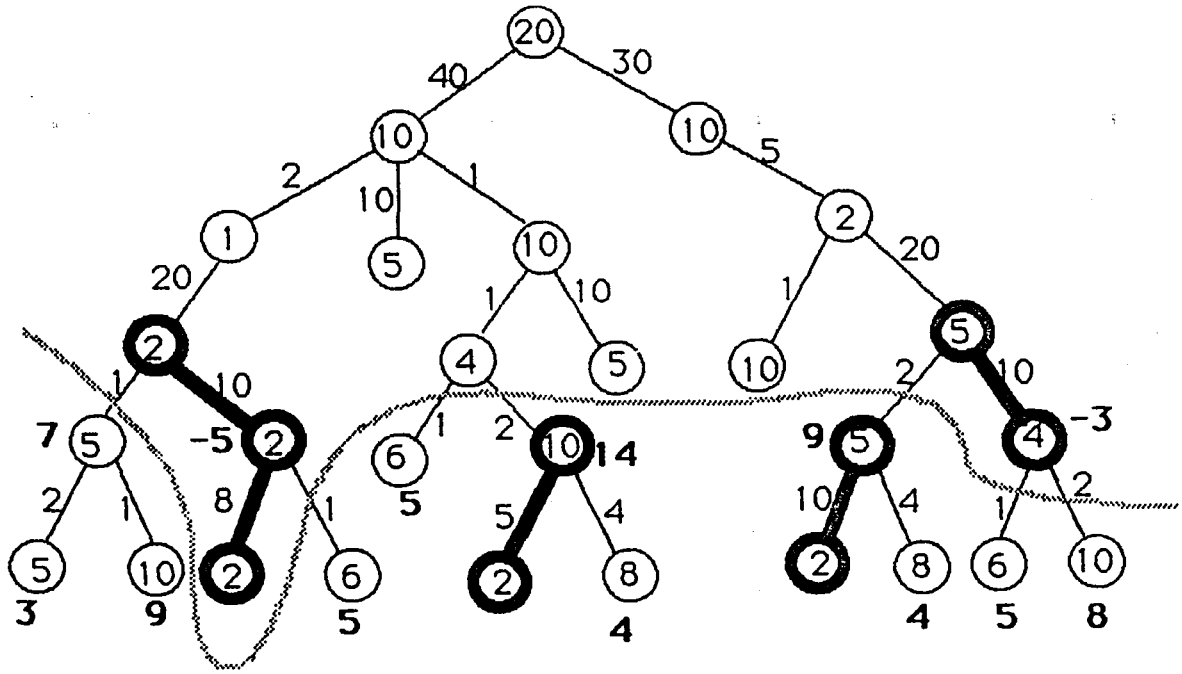


Fig. 8(d) The partition generated by the function when *level* = 4. The value of Δ_1 for each node *i* at that level is shown in bold. Only those nodes which are above the grey line are assigned to the host. The new value of Host_Load is 137-(7+5+14+9)=102.

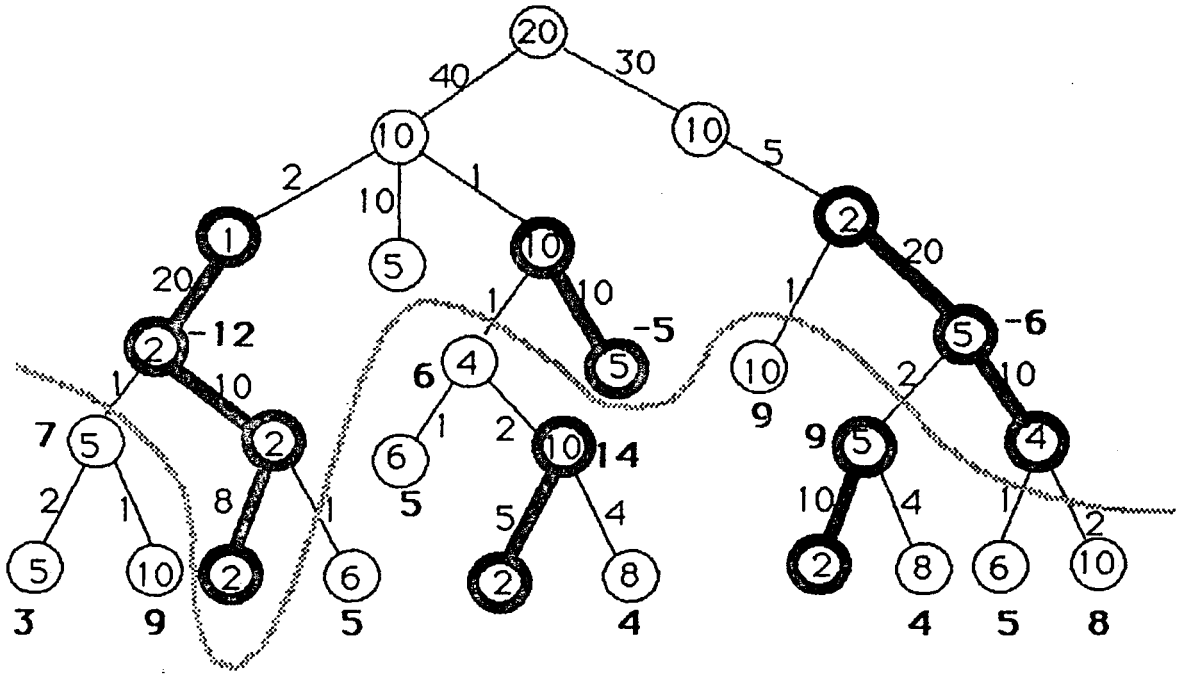


Fig. 8(e) The partition generated by the function when *level* = 3. The value of Host_Load reduces further by 6+9=15.

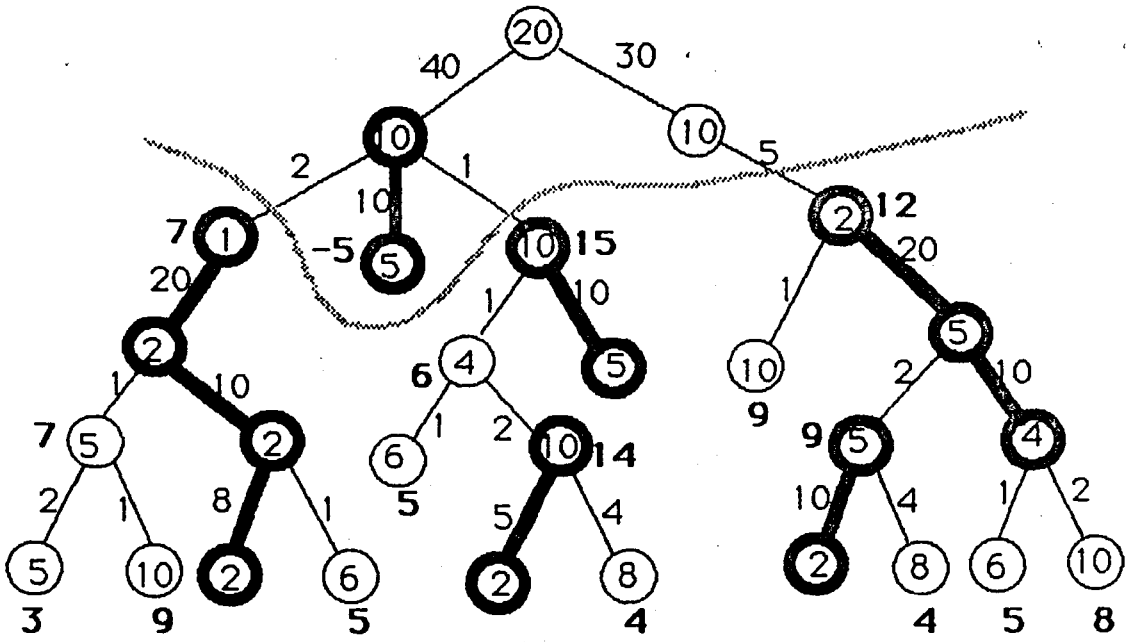


Fig. 8(f) The partition generated by the function when *level* = 2. The value of Host_Load further reduces by 7+15+12=34.

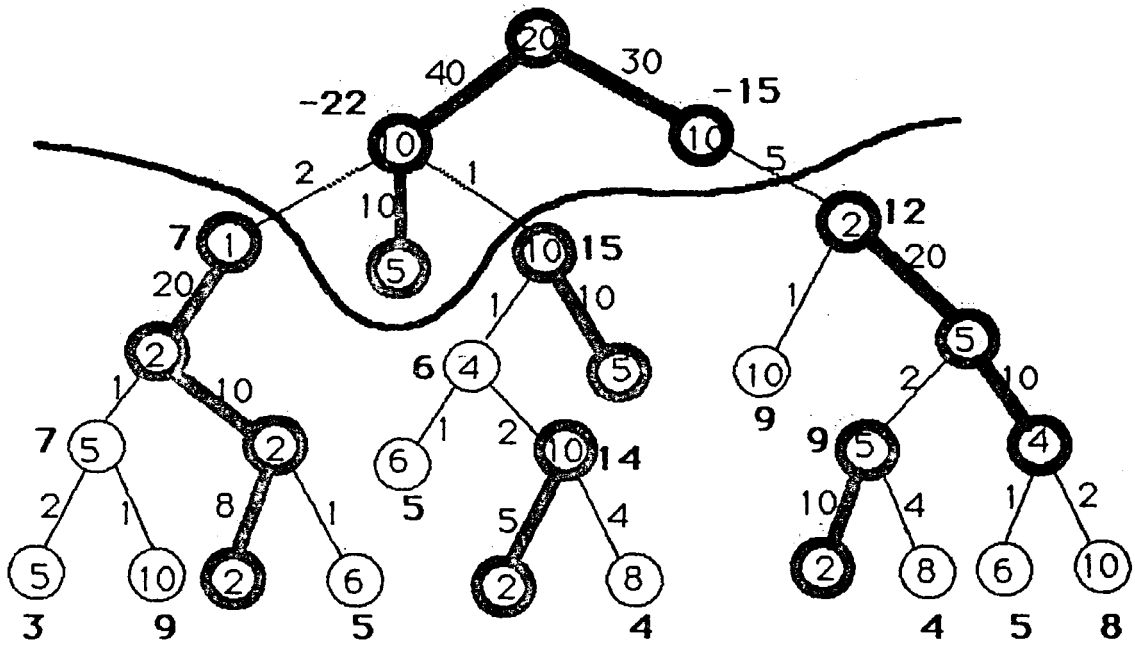


Fig. 8(g) The final partition (shown by black line) generated when *level* = 1. The value of Host_Load = 53.

Standard Bibliographic Page

1. Report No. NASA CR-178130 ICASE Report No. 86-40	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle APPROXIMATE ALGORITHM FOR PARTITIONING AND ASSIGNMENT PROBLEMS		5. Report Date June 1986 6. Performing Organization Code
7. Author(s) M. Ashraf Iqbal		8. Performing Organization Report No. 86-40
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		10. Work Unit No. 11. Contract or Grant No. NAS1-17070, NAS1-18107
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		13. Type of Report and Period Covered Contractor Report 14. Sponsoring Agency Code 505-31-83-01

15. Supplementary Notes

Langley Technical Monitor: J. C. South

Submitted to IEEE Trans. Computers

Final Report

16. Abstract

We consider the problem of optimally assigning the modules of a parallel/pipelined program over the processors of a multiple computer system under certain restrictions on the interconnection structure of the program as well as the multiple computer system. We show that for a variety of such programs it is possible to find in linear time if a partition of the program exists in which the load on any processor is within a certain bound. This method, when combined with a binary search over a finite range, provides an approximate solution to the partitioning problem.

The specific problems we consider are partitioning of (1) a chain structured parallel program over a chain-like computer system, (2) multiple chain-like programs over a host-satellite system and (3) a tree structured parallel program over a host-satellite system.

For a problem with m modules and n processors, the complexity of our algorithm is no worse than $O(mn\log(W_T/\epsilon))$, where W_T is the cost of assigning all modules to one processor and ϵ the desired accuracy.

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
parallel processing, pipeline processing, distributed computing, partitions	61 - Computer Programming and Software 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	30	A03

For sale by the National Technical Information Service, Springfield, Virginia 22161

NASA Langley Form 63 (June 1985)

End of Document