NASA Contractor Report 178073

ICASE REPORT NO. 86-13

ICASE

PERFORMANCE TRADEOFFS IN STATIC AND DYNAMIC LOAD BALANCING STRATEGIES

M. Ashraf Iqbal

Joel H. Saltz

Shahid H. Bokhari

Contract Nos. NAS1-17070, NAS1-18107

March 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

Performance Tradeoffs in Static and Dynamic Load Balancing Strategies

M. Ashraf Iqbal
Institute for Computer Applications in Science and Engineering
and
University of Engineering and Technology, Lahore, Pakistan

Joel H. Saltz
Institute for Computer Applications in Science and Engineering

Shahid H. Bokhari
Institute for Computer Applications in Science and Engineering
and
University of Engineering and Technology, Lahore, Pakistan

ABSTRACT

We consider the problem of uniformly distributing the load of a parallel program over a multiprocessor system. We analyze a program whose structure permits the computation of the optimal static solution. We then describe four strategies for load balancing and compare their performance.

The four strategies are (1) the optimal static assignment algorithm which is guaranteed to yield the best static solution, (2) the static binary dissection method which is very fast but sub-optimal (3) the greedy algorithm, a static fully polynomial time approximation scheme, which estimates the optimal solution to arbitrary accuracy and (4) the predictive dynamic load balancing heuristic which uses information on the precedence relationships within the program and outperforms any of the static methods.

It is also shown that the overhead incurred by the dynamic heuristic (4) is reduced considerably if it is started off with a static assignment provided by either (1), (2) or (3).

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

1. Introduction

Efficient utilization of parallel computer systems requires that the task or job being executed be partitioned over the system in an optimal or near optimal fashion. In the general partitioning problem, one is given a multicomputer system with a specific interconnection pattern as well as a parallel task or job composed of modules that communicate with each other in a specified pattern. One is required to assign the modules to the processors in such a way that the total execution time of the job is minimized.

An assignment is said to be *static* if modules stay on the processors to which they are assigned for the lifetime of the program. A *dynamic* assignment, on the other hand, moves modules between processors from time to time whenever this leads to improved efficiency.

Given an arbitrarily interconnected multicomputer system and an arbitrarily interconnected parallel task, the problem of finding the optimal static partition is very difficult and can be shown to be computationally equivalent to the notoriously intractable NP-Complete problems [1]. However, many practical problems have special structure that permits the optimal solution to be found very efficiently.

In this paper we will compare the performance obtained through the use of a dynamic load balancing method, a suboptimal but very inexpensive static load balancing method and the optimal static load balancing on a problem with a structure that permits the computation of the optimal balance. We also consider a fully polynomial time approximation scheme, the solution of which can be made to approach the optimal load balance. These methods for balancing load are suitable for distinct but overlapping varieties of problems. These problems can arise, among other places, in the solution of systems of linear equations using point or block iterative methods, in problems of adaptive mesh refinements, as well as in time driven discrete event simulation. We describe our experience with four different algorithms that we have used to solve a problem for which all these methods are applicable.

The first method finds the optimal static assignment using the bottleneck path algorithm described in [2]. This algorithm captures the execution costs of the modules or processes of the task as edge weights in an *assignment graph*. A minimum bottleneck path in this graph then yields the optimal assignment. This algorithm has moderate complexity and is guaranteed to yield the optimal static assignment.

The second method that we evaluate is the binary dissection algorithm which is derived from the work of Berger and Bokhari [3],[4]. This algorithm is very fast but does not always yield the optimal static solution.

The third scheme that we consider is based on a widely used *greedy method* described in [5], which when combined with a binary search yields an approximate solution to the static partitioning problem.

Finally we evaluate the predictive dynamic load balancing method developed by Saltz[6]. This is a dynamic algorithm in that modules are reassigned form time to time during the course of execution of the parallel program. This heuristic takes the precedence relationships of the subtasks into account when deciding whether and when to

relocate modules. This additional information and the capability to relocate dynamically permits this algorithm to usually outperform the optimal static algorithm.

The following section discusses in detail the problem addressed in this research. Section 3 contains a brief description of the optimal static algorithm. In Section 4 we describe the binary dissection algorithm. The greedy algorithm is described in section 5. Section 6 contains a description of the heuristic dynamic algorithm and Section 7 compares the performance of these four algorithms.

2. Formulation of Problem

We consider the partitioning on a multiprocessor system of a problem which is composed of a number of processes or modules with a predictable, repetitive pattern of inter-module data dependencies. The computation is divided into steps, and each module requires data from a set of other modules at step $s-1$ to begin the computations required for step s.

Problems that exhibit this pattern of data dependence include explicit schemes for solving partial differential equations [7], iterative and block iterative methods such as Jacobi and multicolor SOR for the solution of systems of linear equations [8] & [9], and problems in discrete event simulation [10] and time driven discrete event simulation*.

The importance of good load balancing strategies is accentuated when the work involved in solving a problem separates naturally into a number of subunits that is relatively small compared to the number of processors utilized, and when partitioning any one of these subunits across several processors is inconvienient or expensive.

For example, consider the solution of an elliptic partial differential equation through the use of a block iterative method. The factored submatrices that represent portions of the domain of the partial differential equation are used repeatedly to iteratively improve an approximate solution of the equation. The computations that must be performed using each factored submatrix are forward and back substitution. If there are more factored submatrices than processors, it may be computationally more efficient not to spread the forward and back substitutions across processors. If the work required to iterate using the factored submatrices cannot be evenly divided amongst the processors, dynamic balancing of load may be useful in preventing processors from becoming idle due to load imbalances.

Dynamic load balancing becomes particularly desirable in problems in which the time needed for a process to complete one step is difficult to determine before the problem is mapped onto a machine, or when the time required to complete a step changes during the problem's execution.

Consider the simulation of physical processes, either by means of solving a partial differential equation or by means of a discrete event simulation. The computations relating to a particular spatial region may be assigned to a specific process which handles all computations describing events occurring in that region. In the case of discrete event simulations and methods that solve time dependent partial differential equations using

an adaptive grid as part of an explicit timestepping scheme, the activity in a given region may vary during the course of the solution of the problem.

In this paper a method for dynamic load balancing that exploits the repetitive pattern of data dependencies is presented, and is compared with two static load balancing methods. The first finds the optimal solution exactly using the computationally expensive optimal algorithm or approximately by means of the greedy algorithm and the second is an inexpensive heuristic.

The static load balancing methods yield a mapping of modules to processors. The time required to complete a problem is determined by the processor with the heaviest load. With the dynamic load balancing method, each module may proceed at a rate constrained only by the local availability of computational resources and its data dependence on other modules. Load balancing is performed in a way that is explicitly designed to prevent processor inactivity due to a lack of data availability.

The performance of the dynamic load balancing method may be expected to depend to some extent on the initial balance of load at the time dynamic load balancing is initiated. One would expect the performance of the dynamic load balancing method to be favorably influenced by the use of static load balancing to improve the initial load balance.

3. The Optimal Static Algorithm

In this section we discuss briefly Bokhari's algorithm for optimally partitioning a chain structured parallel or pipelined program over a chain of processors [2]. 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 $i-1$ as shown in Fig. 1. Similarly, we assume that the multiprocessor of size $n<m$ also has 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. This is known as the *contiguity* constraint. 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 above problem is solved by first drawing a layered graph (Fig. 2) in which every layer corresponds to a processor and the label on each node corresponds to a subchain of modules. Every layer in this graph contains all subchains of modules i.e. all pairs $<i,j>$ such that $1 \leqslant i \leqslant j \leqslant m$. 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 $<1,i>$ in the first layer are connected to node s while all nodes $<i,m>$ in every layer are connected to node t. Any path connecting nodes s and t corresponds to an assignment of modules to processors. For example the thick edges in Fig. 2 corresponds to the assignment of Fig. 1.

---

* D. Nicol and J. Saltz, "A Statistical Methodology for the Control of Dynamic Load Balancing," to be published as an ICASE Report.

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 which accounts for the computation time. Now we add the time to communicate between modules b and $b+1$ over the link connecting processors k and $k+1$ to the weight of the edge joining node $<a,b>$ in layer k to node $<b+1,d>$ in layer $k+1$. It is clear now that there is a path in this graph corresponding to every possible contiguous subchain assignment and the weight of the heaviest edge in a path corresponds to the time required by the most heavily loaded processor to finish. Thus to find the optimal assignment, we have to find the path in the layered graph in which the heaviest edge has minimum weight – the bottleneck path.

The bottleneck path can be found by using the following labeling procedure. Initially all nodes are given labels $L(i)=\infty$ except in the first layer, in which all nodes are labeled zero. Then starting at the top and working downwards we examine each edge e emanating downwards from a layer. If this edge connects node a (above) to node b (below) then replace $L(b)$ by $min(L(b),max(W(e),L(a))$ where $W(e)$ is the weight associated with edge e. Once the graph has been labeled, we then find the edge incident on node t which has maximum weight. Suppose the edge joining node $<i,m>$ of layer k with node t has maximum weight, then it means that the bottleneck path would contain the node $<i,m>$ of layer k and thus modules i through m would be assigned to processor k. The rest of the bottleneck path can be found in the same manner by working upwards from layer k to the top.

The number of nodes per layer in the layered graph is $O(m^2)$ and thus the total number of nodes in the graph is $O(m^2n)$. The number of edges emanating from a node is at the most m, thus the total number of edges would be $O(m^3n)$. As the labeling algorithm looks at each edge once, therefore the space as well as time required by this algorithm is $O(m^3n)$.

4. The Binary Dissection Method

The binary dissection approach to the solution of the basic partitioning problem addressed in this paper is very efficient in terms of run time and gives solutions that are very close to optimal. This algorithm is a simplified version of the two dimensional partitioning strategy developed by Berger and Bokhari [4],[5].

The algorithm proceeds as follows. The given chain of m modules is split up into two halves such that the difference of the sums of execution costs in each half is minimum. The two halves are then recursively subdivided as many times as desired. Clearly, the number of pieces into which the chain can be partitioned must be exactly 2^k where the integer k represents the depth of partitioning.

Thus this algorithm is useful for problems in which the number of processors is a power of 2. The time required by this algorithm is $O(m\log n)$ for a problem with m modules and n processors since there can be no more than $\log n$ levels of partitioning with each level requiring at most one access to each module weight.

At first sight this algorithm may seem capable of yielding the optimal solution. This is not always so, as the example in Fig. 3 demonstrates. In the next paragraph we will find an upper bound on the difference between the optimal solution and the solution yielded by the binary dissection method.

Let W_T represent the sum of the weights of all m modules. A lower bound on the weight of the heaviest subchain W_{OPT} in the optimal partition will be W_T/n under the special case when all the n processors are uniformly loaded. Let us designate the weight of the heaviest module by $w_{\max}$ and the weight of the heaviest subchain assigned to a processor using the techniques of binary dissection by W_{MAX}. Then whenever a chain is divided into two parts, the maximum difference between the two halves will be bounded by $w_{\max}$. Thus if $n=2$ then $W_{MAX} \leqslant W_T/2 + w_{\max}/2$. Similarly if there are n processors then an upper bound on W_{MAX} will be:

$$W_{MAX} \leqslant W_T/n + w_{\max}(\frac{1}{2}+\frac{1}{4}+\ldots+\frac{1}{n})$$

$$\leqslant W_T/n + w_{\max}(n-1)/n$$

Thus the maximum difference between W_{MAX} and W_{OPT} will be given by the following equation under the assumption that $m>n$.

$$W_{MAX} - W_{OPT} \leqslant w_{\max}(n-1)/n \qquad (1)$$

5. The Greedy Algorithm

This algorithm is based on a greedy method, which is a widely used technique and is applied to a variety of problems [3]. Sahni [1] has devised a polynomial time approximation scheme to solve the knapsack problem using a greedy method while Kernighan uses a similar approach [11] for finding optimal sequential partitions of graphs. Utilizing this method one can devise an algorithm which works in stages and at each stage a decision is made regarding whether or not the next input be included in the partially constructed solution. If the inclusion of the next input will result in an infeasible solution then this is not added to the partial solution. Greedy methods may not necessarily provide optimal answers. For example consider the binpacking problem: Given a finite set $W=\{w_1,w_2,\ldots,w_m\}$ of m different weights, find a partition of W into n disjoint subsets $W_1, W_2,\ldots, W_n$, such that n is minimum and the sum of the weights in each subset W_i is no more than a fixed constant. The First Fit algorithm for the above problem is essentially a greedy method in the sense that it tries to place each weight in the lowest indexed subset as far as possible, but this does not result in the optimal solution [1]. If however we put an extra condition on the problem that weights w_i and w_{i+1} are to be placed in either the same subset or subsets W_j and W_{j+1} respectively then the same greedy approach will be able to find the optimal solution.

The greedy algorithm is based on the function *PROBE* (described below) and takes advantage of the fact that the weight assigned to the most heavily loaded processor in

the optimal partition lies somewhere between W_T/n and $W_T/n + w_{\max}$ as discussed in the previous section. The algorithm selects a trial weight w in the above range and then uses the function *PROBE*. The function *PROBE(w)* returns *true* if it is possible to partition the chain of modules into subchains such that the weight of each subchain is less than or equal to w, the resulting partition is called the *greedy partition(w)* and *false* otherwise.

```
function PROBE(Processors[1..n], Modules[1..m], w):boolean;
begin
  i = 1; j = 1; p = 1;
  while p≤n do
    begin
      Assign the subchain Modules[i..j] to processor p;
      repeat
        j=j+1;
      until weight of subchain Modules[i..j] > w or j=m;
      If j = m (all modules have been assigned) then return(true);
      i = j+1; j = i; p = p+1;
    end;
  return(false);
end.
```

The greedy algorithm then makes a binary search in the range W_T/n, $W_T/n+w_{\max}$ using the above function to find the partition for which the weight of the heaviest subchain is minimum. For each trial weight w the function *PROBE* has to look at each module only once. If the above range is resolved to an accuracy of ϵ then the greedy algorithm will find a *greedy partition(w)* in time proportional to $O(m\log_2(w_{\max}/\epsilon))$ with the assurance that w is no more greater than the weight of the heaviest subchain in the optimal assignment by ϵ. It is important to note that the order of the greedy algorithm is proportional to $\log(w_{\max}/\epsilon)$ unlike other *fully polynomial time approximation schemes* in which the time complexity is polynomial in $1/\epsilon$ as described in [1].

In the following paragraphs we will prove that if there exists an assignment with the weight of its heaviest subchain equal to w then the procedure PROBE will always find that or an equivalent assignment assuming that subchains with no modules in them (empty subchains) are allowed.

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

Notation:

$\pi_{w,n}$

a partition with weight w and n subchains.

$\gamma_{w,n}$

a greedy partition with weight w and n subchains.

$\mu_{w,n,k}$

a mixed partition with weight w and n subchains in which the partition up to the first k subchains is greedy and the remaining partition may or may not be greedy.

Observe that $\mu_{w,n,0}=\pi_{w,n}$ and $\mu_{w,n,n}=\gamma_{w,n}$.

Claim 1: $\mu_{w,n,k}$ can always be transformed into $\mu_{w,n,k+1}$

Proof: Move the right hand partition of subchain $k+1$ to the right until any further movement would cause the weight of subchain $k+1$ to exceed w or exhaust the modules.

1. If this is possible without disturbing the right hand partition of subchain $k+2$ then $\mu_{w,n,k}$ been transformed into $\mu_{w,n,k+1}$ and the claim is correct.
2. If during the course of this movement the r.h. partition of subchain $k+1$ coincides with the r.h. partition of subchain $k+2$, this means that subchain $k+2$ is now empty (which is permitted). Continue movement of both partitions together, combining with any further partitions that may be encountered. When the threshold point is reached, $\mu_{w,n,k}$ been transformed into $\mu_{w,n,k+1}$, one or more subchains to the right of $k+1$ are empty but the claim is still correct.

Claim 2: If there exists a $\pi_{w,n}$ then there must also exist a $\gamma_{w,n}$.

Proof: Recall that $\pi_{w,n}=\mu_{w,n,0}$.

By repeatedly applying transformation (1) above we can transform:

$$\pi_{w,n}=\mu_{w,n,0}\rightarrow\mu_{w,n,1}\rightarrow\cdots\rightarrow\mu_{w,n,k}\rightarrow\cdots\rightarrow\mu_{w,n,n}=\gamma_{w,n}$$

Result: If there exists an assignment of weight w then the procedure PROBE will find that or an assignment of equal weight.

6. The Predictive Dynamic Load Balancing Method

We assume that a computation is composed of a fixed number of computational processes or modules. The computation is divided into steps, and each module requires data from a set of other modules at step $s-1$ to begin the computations required for step s. Each module may proceed at a rate constrained only by the time required for the processor to perform the computations required by the module, the local availability of computational resources and data dependence on other modules. Load balancing is performed in a way that is explicitly designed to prevent processor inactivity due to a lack of data availability.

The *potential work* of a processor is defined as the amount of time that will be required to advance all modules in a processor as many steps as possible given the data currently available from other processors. The parallel efficiency of a processor may be

defined as the percentage of time a processor spends performing the computations required by the modules assigned to it. Transfers of modules between processors impact parallel efficiencies in a machine dependent way. The communication time required to transfer a module from one processor to another along with the degree to which that communication can be masked with computation are essential factors in this dependency.

In the predictive dynamic load balancing method to be discussed here, load is shifted between processors in a way that attempts to equalize the potential work in each processor. When the potential work of a processor falls below a predetermined threshold, load balancing is considered. A module is shifted from a neighboring processor when the neighboring processor has stored an amount of potential work greater than or equal to the threshold plus a pre-determined safety factor. If more than one neighboring processor fits this criterion, the processor with the largest potential work contributes a module.

The ability to efficiently calculate the potential work in a processor is central to the usefulness of this method. Simple and inexpensive methods for calculating potential work will now be described. The potential work stored in a processor may have to be calculated from scratch in some situations. When the computations involved in solving a problem are initiated or when modules are shifted in or out of a processor after load balancing, one must take into account both the pattern of data dependencies within a processor and the availability of data from other processors in order to calculate potential work. Given a processor which has assigned to it a value for potential work, a simpler set of computations can be performed to update the value of potential work in response to the receipt of a new datum from another processor.

It is useful at this point to describe in more detail the interaction between step numbers achievable by the modules assigned to a processor and the external data available to the processor. A linked data structure representing an undirected graph DEPEND, with weighted vertices is defined for each processor P. The vertices represent the modules in P as well as the modules in other processors directly coupled to modules in P. Let z_i $1 \leqslant i \leqslant B$ represent boundary vertices and let v_i $1 \leqslant i \leqslant I$ represent vertices within the processor. The *weight* w_i of each vertex v_i represents the largest step reachable by each module, given the currently available boundary information. The weight q_i of each of the vertices z_i, represents the step of the largest available boundary variable data for the module.

The largest step reachable by a vertex v_i in the processor given currently available boundary data is determined by adding one to the minimum of: (1) the largest steps reachable by all internal vertices v_j linked to v and (2) the step number of the latest available boundary data for the boundary vertices z_l linked to v. The weight assigned to v_i may be written as

$$w_i = \min_{j,\,l}(w_j, q_l) + 1 \tag{2}$$

where v_j and z_l are linked to v_i.

Denote the current step number of v_i as s_i and the time required to advance v_i one step t_i. The potential work associated with P at a given point in the computations may be written as

$$\sum_i (w_i - s_i) t_i \tag{3}$$

where the sum is over all i corresponding to v_i in P. For each boundary vertex z_i the graph DEPEND may be divided into equivalence classes based on the minimum number of edges that have to be traversed to get to z_i. We define $r_{k,j}$ as the equivalence class of z_k to which v_j belongs. Note that each internal vertex belongs to B different equivalence classes, one corresponding to each boundary vertex z_k, $1 \leqslant k \leqslant B$. The proposition below states a sort of superposition principle that holds for the determination of the maximum achievable cumulative microstep for internal vertices in response to constraints arising from boundary vertices.

Proposition: The weight of v_i is given by

$$w_i = \min_{1 \leqslant k \leqslant B} (q_k + r_{k,i}) \tag{4}$$

The proof is carried out by substituting the postulated solution into (2). Fix attention on an internal vertex v_i. Corresponding to each $r_{k,i}$ where $r_{k,i} \geqslant 2$ there must be an internal vertex v_j linked to v_i with $r_{k,j} = r_{k,i} - 1$. If there were not, it would not be possible to find a shortest path from v_i to z_k consisting of $r_{k,i}$ edges. Moreover, there cannot be an internal vertex v_j connected to v_i with $r_{k,j} < r_{k,i} - 1$, if there were, then v_i would have a shortest path to z_k consisting of fewer than $r_{k,i}$ edges. Corresponding to each $r_{k,i}$ where $r_{k,i} = 1$ there is a direct edge from v_i to z_k.

Now substituting (4) for each v_j into (2) yields

$$w_i = \min_{j,\,l} [\min_{1 \leqslant k \leqslant B} (q_k + r_{k,j}), q_l] + 1 \tag{5}$$

for all j,l such that v_j and z_l are linked to v_i. Equation 5 may be rewritten as

$$w_i = \min_{1 \leqslant k \leqslant B} [\min_{j,\,l} ((q_k + r_{k,j}), q_l)] + 1$$

For each k, there exists an internal vertex v_j with $r_{k,j} = r_{k,i} - 1$ connected to v_i and there cannot be a vertex v_j where $r_{k,j} < r_{k,i} - 1$. Hence from (5) we obtain (6)

$$w_i = \min_{1 \leqslant k \leqslant B, l} [q_k + (r_{k,i} - 1), q_l] + 1 \tag{6}$$

For boundary vertices z_l to which v_i is directly connected, $r_{l,i} = 1$. Since all quantities involved are positive in sign, we obtain from (6) the equation (4) for v_i as desired.

We are now in a position to calculate the potential work from scratch, given values of s_i and t_i corresponding to all vertices v_i in P. For each v_i in P one may calculate w_i from (4) in O(B) operations per vertex. Since there are I vertices the calculation of potential work from scratch requires O(IB) operations.

If a processor has a value of potential work assigned to it the potential work may be updated in response to the receipt of a boundary datum. One finds the weights for each vertex v_i in P in the following way. By equation (4) incrementing the weight of a single boundary vertex can either leave the weight of interior vertices unchanged or increase the weight by one unit. Moreover, only interior vertices currently constrained by the incremented boundary vertex will have their weights incremented.

In response to an increment in a boundary vertex z_k, the weights in equivalence classes may be adjusted in order of increasing equivalence class number with only one pass necessary. Assume that z_k has had its weight incremented from q_k-1 to q_k. Before z_k was incremented, the constraint on the weight of vertices in equivalence class $r_k=n$ was q_k-1+n. The constraint on the weight of vertices in equivalence class $r_k=n-1$ after z_k is incremented is $q_k+(n-1)$. The adjusting of equivalence class $r_k = n$ will have no effect on the adjustment of equivalence class $r_k = n-1$.

If a vertex in equivalence class $r_k = n$ has a weight of less than q_k+n-1 before being considered for readjustment, it is not being constrained by z_k. Incrementing z_k's weight will consequently not affect the vertex. Since the only vertices which can possibly have their weights incremented have weights q_k+n-1, the order in which vertices in an equivalence class are considered is unimportant.

Updating DEPEND may proceed as follows. The weight of the vertex in DEPEND representing z_k is first incremented. In a breadth first manner beginning with the vertex representing z_k, DEPEND is searched for vertices whose weights must be incremented. When a vertex v is found that does not require a weight increment, the search does not continue to examine other vertices linked to v.

In the model problem, the time and space requirements of this updating algorithm algorithm are $O(nm)$ and $O(m)$ where n is the number of modules in the problem and m is the number of steps over which advancement is to proceed.

7. Comparison of Results

We have compared the performance of both the static load balancing methods and the predictive dynamic method through a variety of simulations. Note that with minimal computational effort, on a set of weights consisting of single precision floating point numbers numbers, the greedy approximation scheme produces a balance identical to the optimal load balance. Thus, the performance obtained through the use of the optimal method and the greedy approximation scheme were identical, and in this section we shall simply refer to the performance of the optimal load balancing method.

Static and dynamic methods can be combined; a static load balancing may be performed before beginning work on a problem, and a dynamic load balancing policy may be utilized once work on the problem has begun. It is found that the initial use of static load balancing policies can enhance the performance of the dynamic policy and that both the optimal and the binary dissection static load balancing methods yield rather comparable performance when used with the dynamic predictive load balancing method.

Used without a dynamic load balancing method, the optimal load balance was found to be notably superior to binary dissection, while there was hardly any difference between the optimal load balance and the greedy load balance on the test problems described here.

We consider a system with 16 processors and a fixed number of modules. In each trial, random deviates representing the weights of modules are drawn from a truncated normal distribution. For each set of random deviates, both the optimal static load balance and the binary dissection balance are calculated and the performance is tabulated. Simulations utilizing the predictive dynamic policy are also run using the same set of random deviates. These simulations utilize both static policies and the assignment of a fixed number of modules to each processor as starting conditions. Performance is measured by calculating the average percentage of time processors are occupied advancing modules over the course of the simulations. Performance results are averaged over 50 trials differing only in the values of the random deviates generated.

In Fig. 4 and Fig. 5 the performance obtained through the use of the static and dynamic policies is depicted. In these figures, the performance of the policies is plotted against the variance of the truncated normal distributions from which the module weights were drawn. In the experiments depicted in the above figures, the weights for the modules were drawn from truncated normal distributions with variances of 0.5, 1.0 and 2.0 and mean 1, and the problem was assumed to run for 200 steps. In Fig. 4 during each trial 64 modules were assigned to the system while in Fig. 5 96 modules were assigned to the system. In both of these cases, for all variances tested, the dynamic load balancing method outperformed both static load balancing methods. Note however, that this measure of performance does not take into account the machine dependent cost of shifting modules between processors, a cost that will be studied in more detail below. The binary dissection static method was in all cases noticeably inferior to the optimal static load balance. The use of a static load balancing method initially had a relatively minor positive impact on performance in the experiments with 96 modules, and no discernible impact at all in experiments with 64 modules. The performance impact of the initial use of a static load balancing method is quite dependent on the number of steps required to solve a problem. It will be seen later that for problems that continue for a relatively small number of steps, the initial use of a static load balancing method can markedly improve performance.

In the dynamic load balancing method, the moving of modules from one processor to another will exact a cost that will depend on the details of the machines' interprocessor communication network. In Fig. 6 and Fig. 7 the average number of modules that must be moved from one processor to a neighbor per step of the computation is plotted against performance for a range of values of the dynamic method's safety factor. In each of the two figures, the use of static load balancing does play a notable role in increasing performance and decreasing the frequency with which blocks have to be shifted. On each curve in Fig. 6 and Fig. 7 both the cost and performance were strictly decreasing functions of the safety factor used.

The number of steps advanced are varied and the performance and the overhead in modules moved per step are depicted for the dynamic load balancing method in Fig. 8 and Fig. 9 respectively. In both figures, the effects of using the two static load balancing methods as well as using no load balancing at the beginning of the computation are compared. In all cases, the performance increases with the number of steps advanced.

For problems that do not require a large number of steps, the performance obtained by starting out with a static load balancing method is superior to that arising from the dynamic load balancing method without initial static load balancing. Perhaps somewhat counter-intuitively, initially balancing load with binary dissection leads to better performance than initially performing an optimal balance for problems requiring over 10 steps. The optimal static load balance is not necessarily the initial load distribution that best allows the dynamic load balancing method to move modules so that processor idleness is avoided. As the number of steps increases, the performance differences obtained through the use of different initial load distributions becomes less marked.

The initial use of static load balancing also leads to marked reduction in module transfer overhead as depicted in Fig. 9. In this figure the overhead per step generally increases with the number of steps. For problems with very large numbers of steps, the overheads for the initial load distributions all approach a single value. When no initial static load balancing is used in a problem that is advanced a small number of steps, both low performance and relatively high costs in number of modules transferred are incurred. It is noted that in Figure 9, when of initial static load balancing was not used, the number of modules transferred reaches a local maximum for problems of 10 steps, and then declines briefly before resuming its long term increase. This phenomena has been observed in a number of similar experiments, its cause is unclear.

The performance obtained through the use of binary dissection as a static load balancing method was notably poorer than that produced by the optimal balance. We have observed in these and other experiments that initial static load balancing used along with the predictive dynamic load balancing method improves performance and reduces the frequency with which modules must be moved. The choice of method used to initially balance load does not appear to have a marked impact on performance or cost.

8. Conclusions

The four load balancing methods discussed in this paper each have their own distinct advantages and disadvantages. Finding an optimal static load balancing is in general an NP-Complete problem unless special structure is present to permit a low order polynomial solution. For the test problems that we have considered, the greedy algorithm was an order of magnitude faster than the optimal load balancing algorithm and it provided results as good as the optimal solutions. The binary dissection method and the predictive dynamic load balancing algorithms are both quite useful in situations in which low order polynomial solutions to the optimal static load balancing problem do not appear to be available. The predictive dynamic load balancing method as formulated here however is applicable only to algorithms with considerable regularity in subtask

precedence relations.

The experimental results presented here revealed that the predictive dynamic load balancing method led to processor utilizations that were consistently above those obtained by the optimal static load balancing method. As one would expect, the optimal static load balancing method, in turn, consistently out performed the binary dissection method.

The initial partitioning of load at the point dynamic load balancing was initiated proved to have a marked effect on the performance of the dynamic load balancing algorithm. All three static load balancing methods used in conjunction with the dynamic load balancing method lead to a substantial improvement in performance. The magnitude of these effects depended on the number of steps the problem is advanced, being most pronounced when a problem is finished after relatively few steps. It is interesting to note that the binary dissection algorithm appeared under some circumstances to consistently lead to results that were superior to optimal load balancing when used in conjunction with dynamic load balancing.

One of the principal costs of the predictive dynamic load balancing method is expected to be the machine dependent cost of transferring the computational modules between processors. The effect of initial load distribution on this cost was examined and it was found that the frequency with which blocks were transferred between processors was markedly reduced when either form of static load balancing was initially employed.

The initial distribution of load in a multiprocessor system is clearly an important determinant of the performance gains achievable by the dynamic load balancing policy; this initial distribution also has a strong influence on the overhead costs of the dynamic policy.

9. References

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

[2] S. Bokhari, "Partitioning Problems in Parallel, Pipelined and Distributed Computing," ICASE Report No. 85-54, November, 1985. (NASA CR-178023.)

[3] M. Berger and S. Bokhari, "A Partitioning Strategy for PDES across Multiprocessors," in *Proc. 1985 International Conference on Parallel Processing,* pp. 166-170, August, 1985.

[4] M. Berger and S. Bokhari, "A Partitioning Strategy for Non-Uniform Problems on Multiprocessors," ICASE Report No. 85-55, November, 1985. (NASA CR-178024.)

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

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

[7] R. Richtmyer and K. Morton, *Difference Methods for Initial-Value Problems,* Interscience Publishers, 1967.

[8] R. Varga, *Matrix Iterative Analysis,* Prentice-Hall, 1962.

[9] L. Adams and H. Jordon, "Is SOR Color Blind?," ICASE Report No. 84-14, May, 1984.

[10] D. Nicol and P. Reynolds, Jr., "The Automated Partitioning of Simulations for Parallel Execution," Tech. Report TR-85-15, Dept. of Computer Science, University of Virginia, 1985.

[11] B. Kernighan, "Optimal Sequential Partitions of Graphs," *Journal of the Association for Computing Machinery,* Vol. 18, No. 1, January 1971, pp. 34-40.

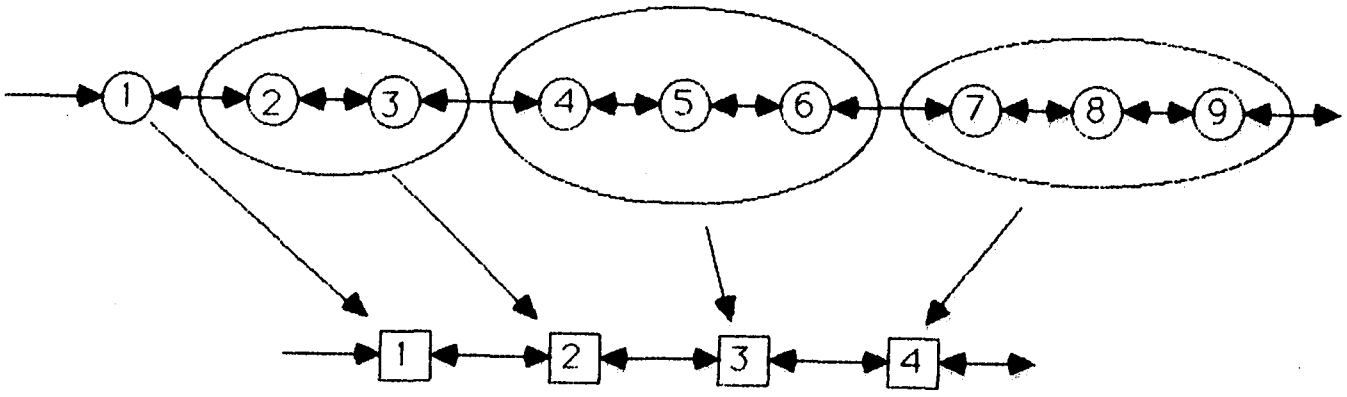

Fig. 1 A 9 module chain mapped onto a 4 processor chain.

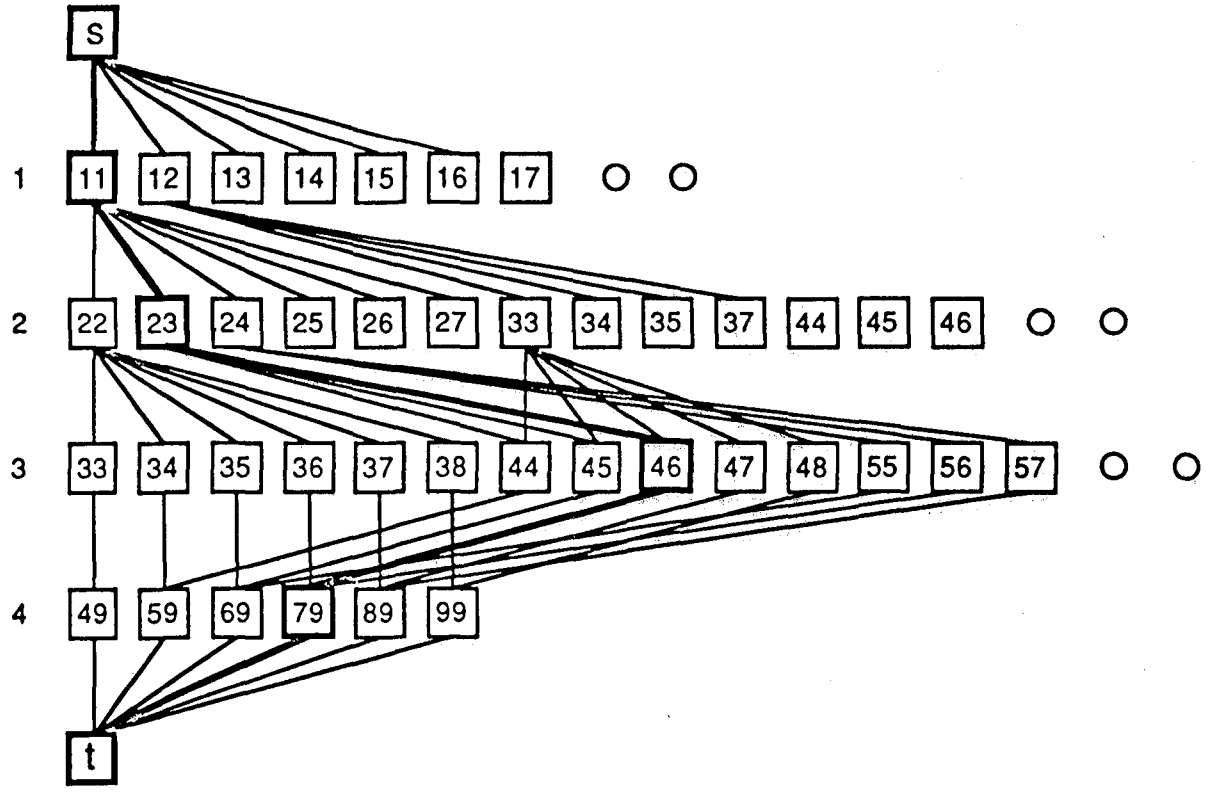

Fig. 2 The layered graph for a problem with 9 modules and 4 processors.

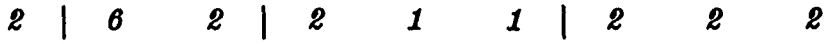

Fig. 3(a) A 9 module chain, with each module represented by its execution cost, mapped onto a 4 processor chain using the Binary Dissection method. The load on the most heavily loaded processor is 8 units.

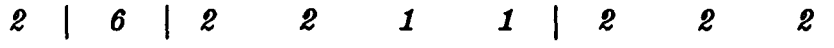

Fig. 3(b) Under an optimal mapping of the 9 module chain on the processor chain, the load on the most heavily loaded processor would only be 6.

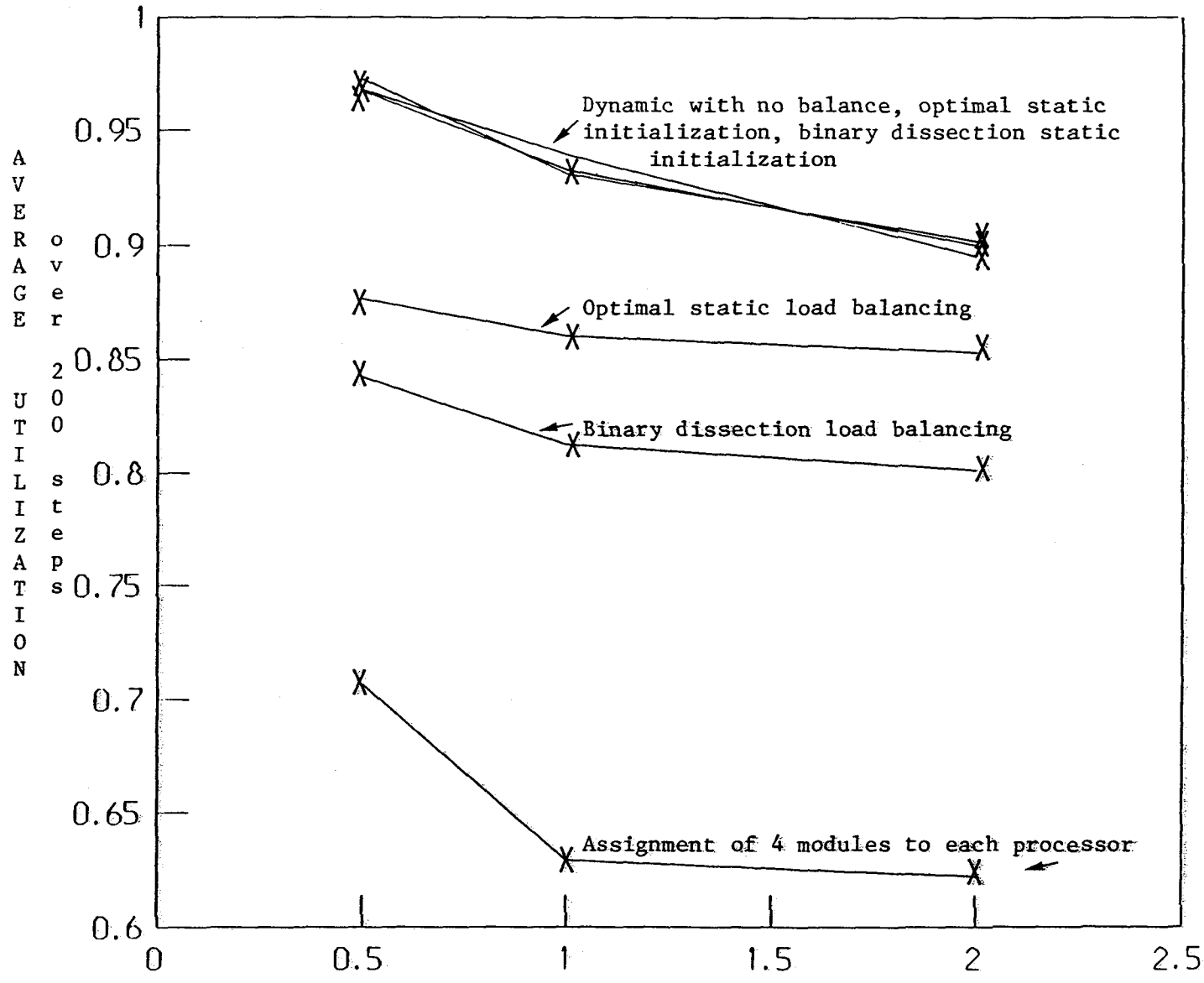

Figure 4. 16 processors, 64 modules, each trial run for 200 steps. Module weights drawn from truncated normal distribution with unit mean and standard deviations of either 0.5, 1.0, or 2.0, prior to truncation.

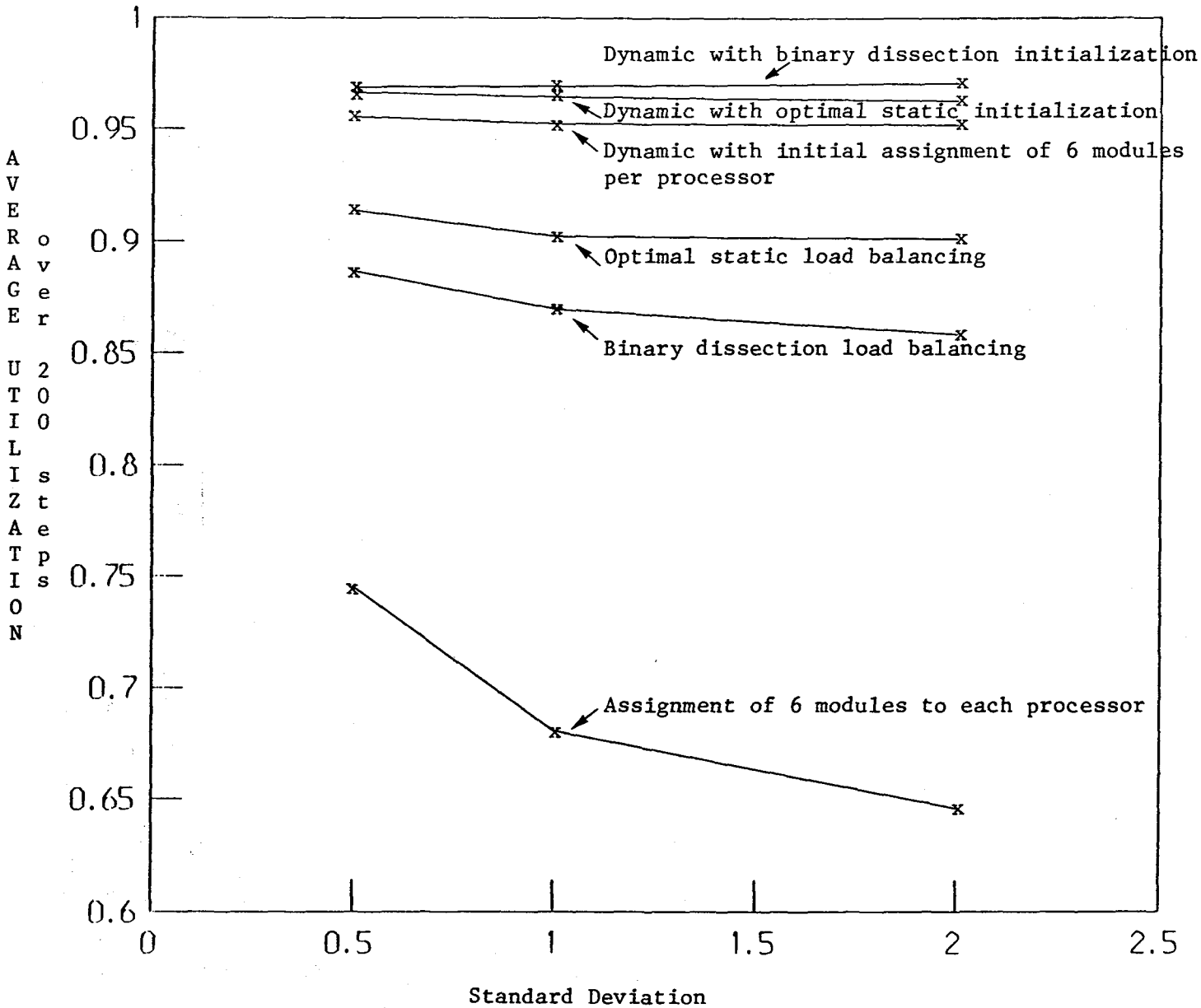

Figure 5. 16 processors, 96 modules, each trial run for 200 steps. Module weights drawn from truncated normal distribution with unit mean and standard deviations of either 0.5, 1.0, or 2.0, prior to truncation.

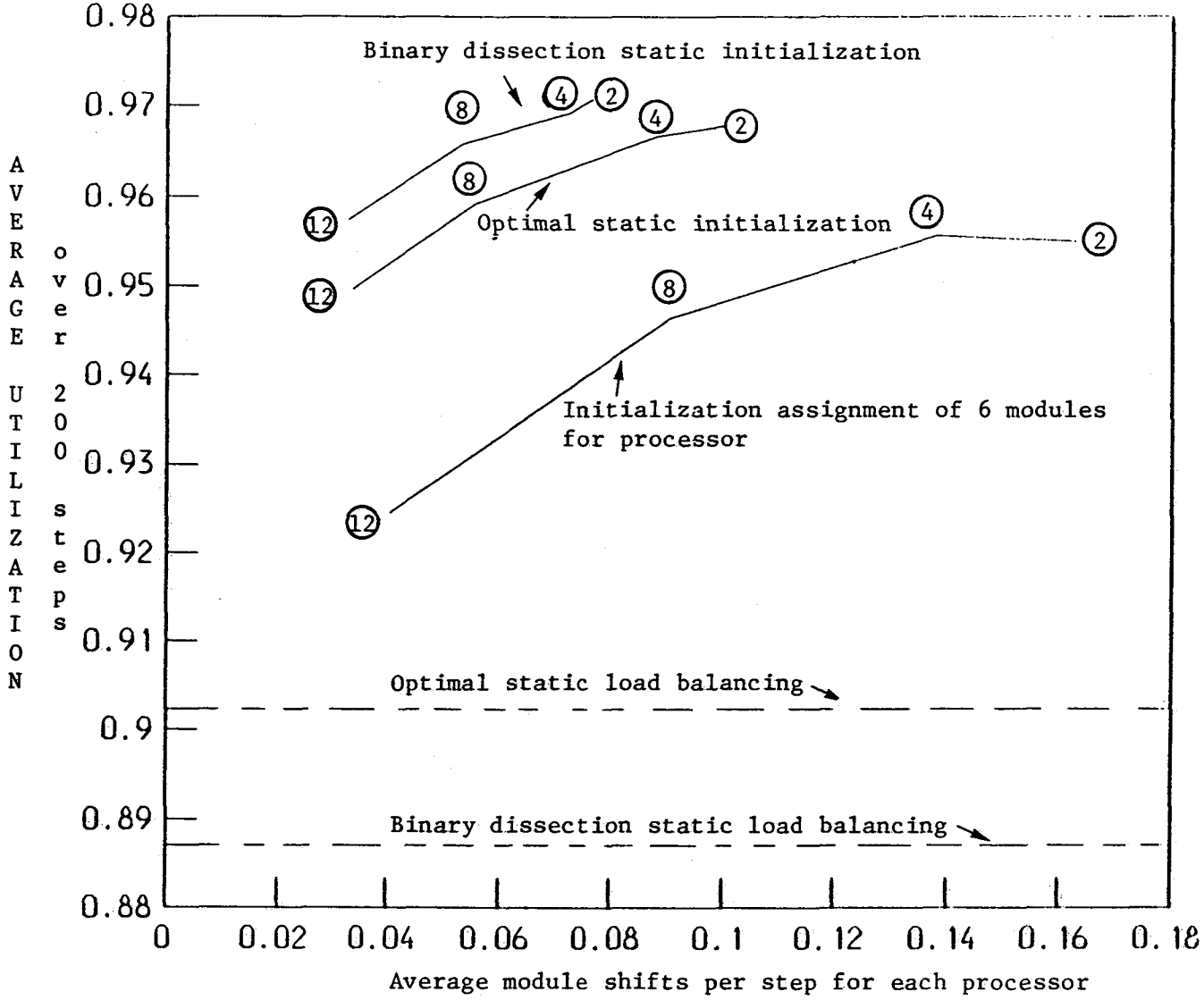

Figure 6. 16 processors, 96 modules, each trial run for 200 steps. Module weights drawn from truncated normal with unit mean, standard deviation of 0.5. Circled figures represent safety factors. Optimal static load balancing and binary dissection static load balancing performance for the same input data included for purposes of comparison.

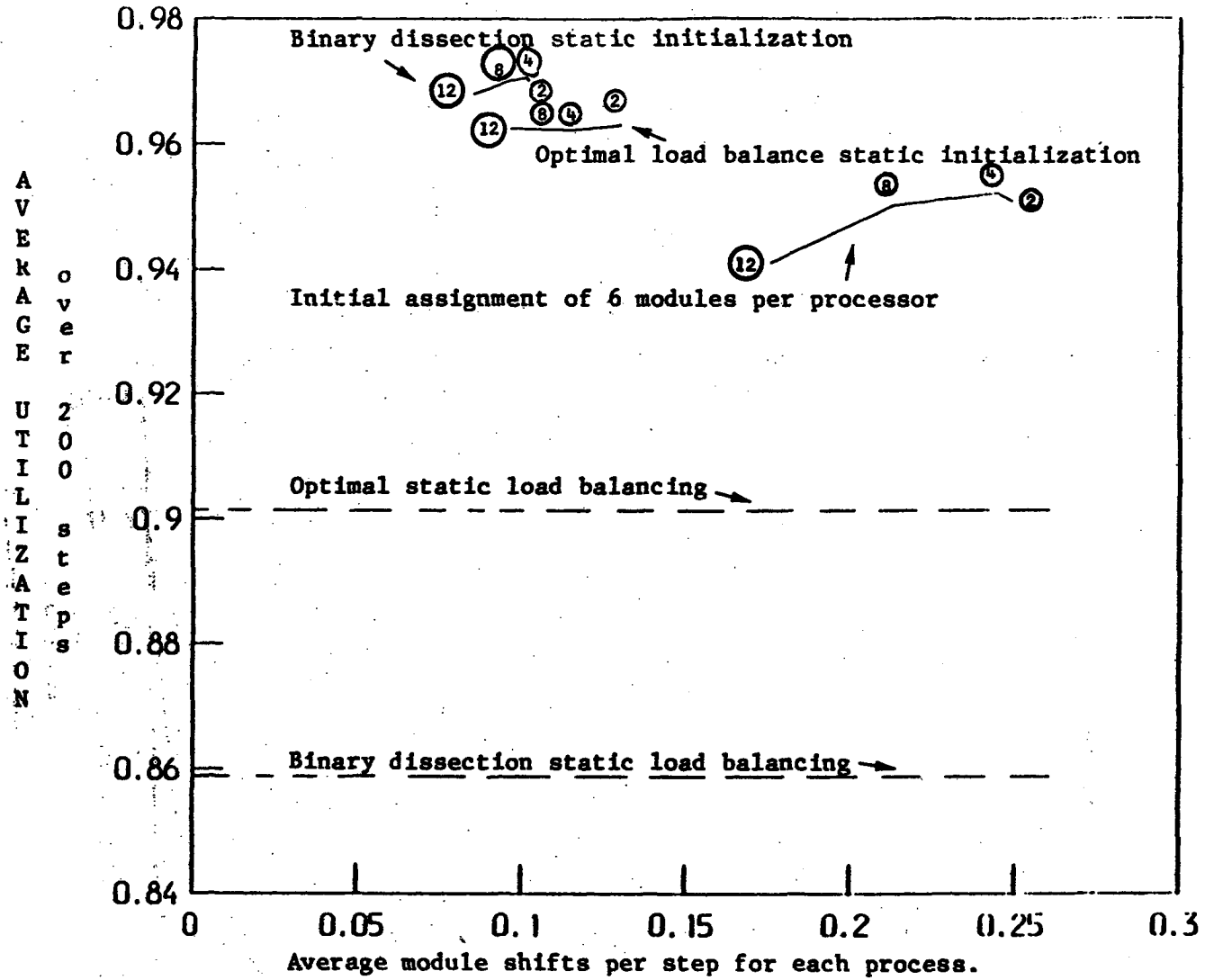

Figure 7. 16 processors, 96 modules, each trial run for 200 steps. Module weights drawn from truncated normal with unit mean; standard deviation of normal distribution is 2.0. Circled figures represent safety factors. Optimal static load balancing and binary dissection static load balancing performance for the same input data included for purposes of comparison.

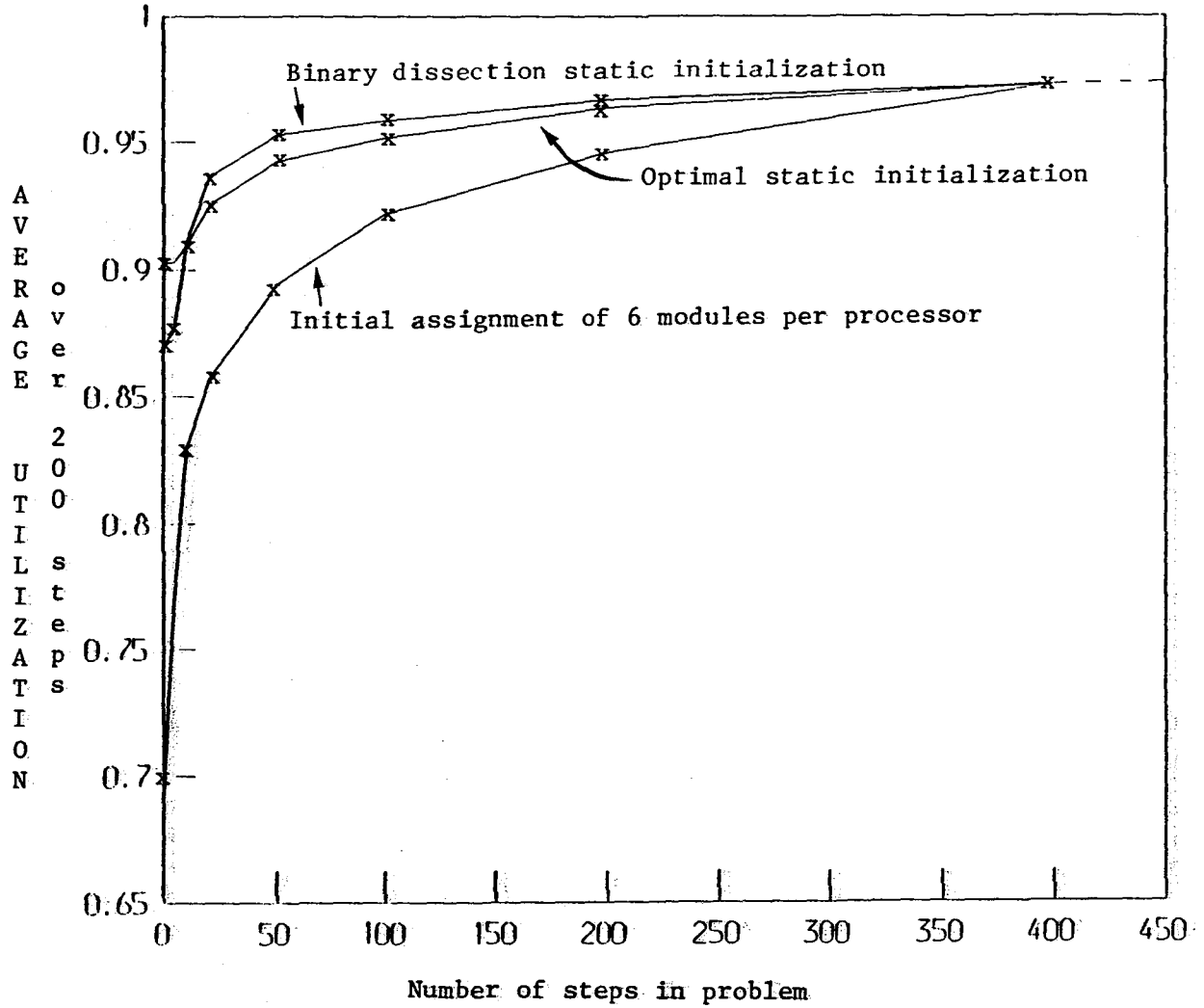

Figure 8. 16 processors, 96 modules, each trial run for 5, 10, 20, 50, 100, 200, 400, 800 steps. Module weights drawn from truncated normal with unit mean, standard deviation of 1.0.

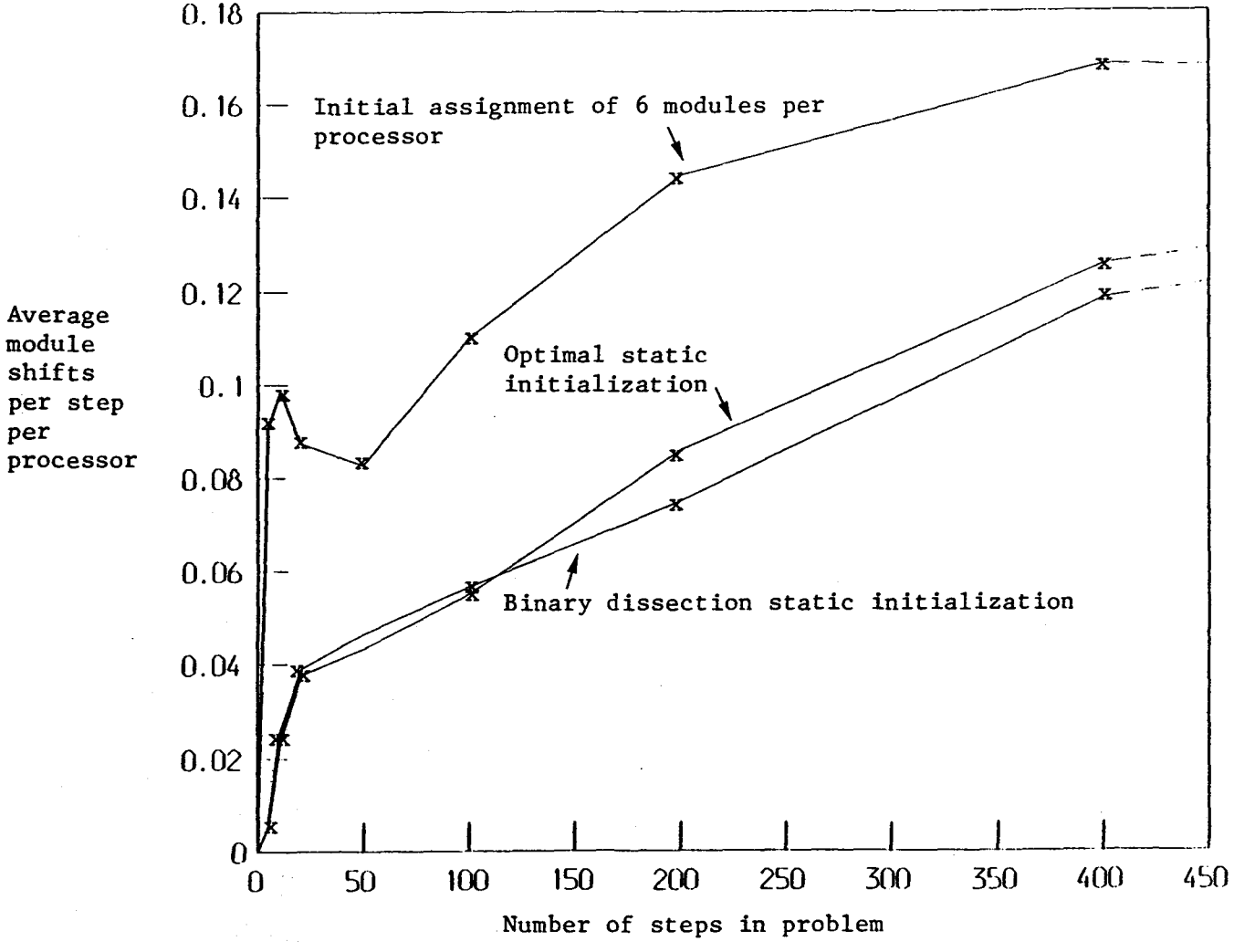

Figure 9. 16 processors, 96 modules, each trial run for 5, 10, 20, 50, 100, 200, 400, 800 steps. Module weights drawn from truncated normal with unit mean, standard deviation of 1.0.

Standard Bibliographic Page

1. Report No. NASA CR-178073 ICASE Report No. 86-13	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle PERFORMANCE TRADEOFFS IN STATIC AND DYNAMIC LOAD BALANCING STRATEGIES		5. Report Date March 1986 6. Performing Organization Code
7. Author(s) M. Ashraf Iqbal, Joel H. Saltz, and Shahid H. Bokhari		8. Performing Organization Report No. 86-13
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. Comput.

Final Report

16. Abstract

We consider the problem of uniformly distributing the load of a parallel program over a multiprocessor system. We analyze a program whose structure permits the computation of the optimal static solution. We then describe four strategies for load balancing and compare their performance.

The four strategies are: (1) the optimal static assignment algorithm which is guaranteed to yield the best static solution, (2) the static binary dissection method which is very fast but sub-optimal, (3) the greedy algorithm, a static fully polynomial time approximation scheme, which estimates the optimal solution to arbitrary accuracy, and (4) the predictive dynamic load balancing heuristic which uses information on the precedence relationships within the program and outperforms any of the static methods.

It is also shown that the overhead incurred by the dynamic heuristic (4) is reduced considerably if it is started off with a static assignment provided by either (1), (2), or (3).

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
load balancing multiprocessor greedy algorithm binary dissection	66 - Systems Analysis 61 - Computer Programming & Software Unclassified - unlimited

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	24	A02

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

NASA Langley Form 63 (June 1985)