Algorithms for Optimal
Dynamic Assignment

J.B. Sinclair
Rice University
Houston, Texas

Tech. Rept. # 7912

August, 1979
ABSTRACT

The problem of determining an optimal dynamic assignment of a modular program in a loosely coupled distributed processing system is considered. An optimal dynamic assignment minimizes the sum of all module execution costs, intermodule communication costs, and module reassignment costs. The two-processor problem is shown to be efficiently solvable through the application of a max-flow min-cut algorithm to a dynamic processor flow graph. The general p-processor problem can be solved using a dynamic programming approach which is equivalent to a simple shortest path algorithm to construct a dynamic assignment tree for the program's execution.
ALGORITHMS FOR OPTIMAL
DYNAMIC ASSIGNMENT IN
DISTRIBUTED PROCESSING SYSTEMS

1. Introduction

Distributed processing systems are often viewed as attractive alternatives to the traditional single large mainframe computer architecture. For any new architecture to be viable, it must offer significant advantages over conventional architectures in terms of cost, performance, and reliability. In the area of reliability, a distributed processing system (DPS) promises to have the necessary potential. Performance will depend on how effectively we are able to manage the resources of the system, and this will in turn help determine whether or not the performance-to-cost ratio will be sufficiently large to justify the DPS.

Distributed processing in simplest terms means that a program has its execution dispersed among the processors of the system. The implicit assumption is that program execution distribution is advantageous in terms of cost to the user. In this paper we will be concerned with the effectively managing the resources of a DPS by maximizing the benefit to the user of allowing his program to visit a number of different processors during its execution.

II. Distributed Systems and Distributed Programs

For our purposes, a DPS must satisfy three requirements. First, it should consist of multiple, possibly nonhomogeneous processors interconnected through a communication subnetwork. Second, the coupling of pro-
cessors by the communication subnetwork is such that interprocessor communication is at less than memory access speeds; the system is not tightly coupled. Third, the individual components of the system are invisible to the average user. Requests for service are made by name of the service only and without regard to the specific resource which will satisfy the request. This implies the existence of a high level operating system which unifies the DPS and integrates all of the resources of the system into the framework of a virtual uniprocessor for each user.

A program to be executed in the DPS will be partitioned into a set of modules named $A, B, \ldots, N$. Each module may be executed many times during a single program execution, but the program is assumed to be strictly sequential. No concurrency in module executions is allowed. An execution of a module is called an instance of that module.

A module instance may be assigned to one of a number of processors. For each processor to which it may be assigned, a given module instance will have an associated cost of execution that is in general different for each processor. Other instances of the same module may have entirely different execution costs. In some cases it may be necessary to every instance of a module to the same processor, due to some unique capability provided by that processor. The cost of executing an instance of that module on any other processor will be assumed to be infinite.

As the program is being executed, modules will need to share or exchange information. Intermodule communications in a DPS must be restricted to a CALL-like mechanism with associated parameter passing (by value). There are a number of reasons for this restriction, all centering around the high
cost of interprocessor communications due to the loose coupling in the communications subnetwork and the absence of global information on the complete state of the system and all processes currently active within the system [1][2]. This restriction effectively limits intermodule communication to communication between successive module instances. For any two consecutive module instances which exchange information we associate a cost of communication. If the instances are assigned to the same processor and can therefore share primary memory address space, the cost is negligible and will be ignored (set to zero). A nonzero communication cost will be assumed to be symmetric with respect to the two processors on which the instances reside, in that the cost will not be a function of the direction of information transfer but only the amount.

If modules are allowed to move from one processor to another during program execution, they will incur a reassignment cost in doing so. Successive instances of the same module which are executed on the same processor have an associated reassignment cost of zero; otherwise, the reassignment cost is nonzero and may be asymmetric. Reassigning a module should not involve the actual transportation of executable code from one machine to another. Rather, a processor should have a copy of each module which might be executed on that processor. Reassignment then becomes a matter of communicating module state information between processors, deactivating the module on the processor where it was previously executed, translating the state information, and activating a copy of the module on the new processor. The latter two operations in particular may have costs which are processor-dependent; hence the total cost of reassignment may be asymmetric. However,
it will be assumed that the reassignment cost is independent of the point in program execution at which the reassignment occurs.

The cost of executing a program is the sum of the modules instance execution costs, intermodule communication costs, and module reassignment costs for the entire program execution, given the processors to which each instance in the sequence of module executions is to be assigned. Optimizing the program's distribution means finding a dynamic assignment for the modules of the program which minimizes this sum. An optimal dynamic assignment is a result of the interaction of three factors which in generally may have conflicting influences. Module instances tend to be assigned to processors on which their execution costs are lower. Successive instances of the same module tend to be assigned to the same processor to eliminate reassignment costs. Similarly, successive module instances tend to be assigned to the same processor to eliminate communication costs.

III. Dynamic Assignment Using Network Flow Algorithms

The problem of determining an optimal static assignment can be efficiently solved using a commodity flow network approach [3]. A static assignment is a special case of a dynamic assignment in which every instance of any given module is executed on the same processor, thus eliminating reassignment costs. This approach can be extended in an obvious and for the most part straightforward manner to the dynamic problem. In this section we present one such extension of Stone's method, based on the assumptions made earlier. For a somewhat different formulation of the dynamic problem and its solution, see the paper by Bokhari [4].
A commodity flow network may be represented as a directed or undirected graph having a set of nodes \( N \), a set of edges \( E \), and a set of edge capacities. \( N \) consists of three mutually disjoint subsets \( S, T, \) and \( V \), where \( S \) is the set of all source nodes, \( T \) is the set of all sink nodes, and \( V \) is the set of all interior nodes. Source nodes represent supply centers which are capable of producing infinite quantities of the commodity. Sink nodes correspond to demand centers which have infinite capacities for absorbing the commodity. All of the remaining nodes are interior nodes and represent transshipment points along the commodity transportation routes, which are themselves represented by the edges in \( E \). The capacity of any edge \((A, B)\), written \( c(A, B) \), is the maximum amount of commodity that can be shipped along the associated link from \( A \) to \( B \). For an undirected edge, \( c(A, B) = c(B, A) \).

A commodity flow is represented as a set of weighted arrows associated with the edges of the graph. The direction of an arrow indicates the direction of flow along its associated edge, while the weight of the arrow is the amount of commodity which is being transported along that edge. A feasible commodity flow is one in which

(i) the flow along any edge is less than or equal to the edge capacity,

(ii) the net flow into (out of) any interior node is zero, and

(iii) all source nodes have nonnegative flow out and all sink nodes have nonnegative flow in.

The value of a commodity flow is the net flow into \( T \) (for a feasible flow, this is also the net flow out of \( S \)). A commodity flow whose value is
maximum among all possible feasible flows is called a **maximum flow**.

A *cutset* of a commodity flow graph is a set of edges from $E$ such that

(i) if all edges in the cutset are removed from the graph, all source nodes are separated from all sink nodes, and

(ii) no proper subset of the cutset is a cutset.

The **capacity** of a cutset in an undirected commodity flow graph is the sum of the capacities of the edges in the cutset. Ford and Fulkerson were able to show the following relationship between commodity flows and cutset capacities [5].

The value of the maximum flow from the source nodes to the sink nodes in a commodity flow network is equal to the minimum capacity of a cutset of the network.

This is known as the Max-Flow Min-Cut Theorem. Ford and Fulkerson developed an algorithm for determining the maximum flow by finding a minimum capacity cutset. Subsequent improvements have reduced the time complexity for solving this problem [6][7]. Most recently an algorithm has been given for finding a maximum flow in $O(|V|^{5/3}|E|^{2/3})$ time [8].

These results are important to the solution of the optimal dynamic assignment problem because it can be reformulated as a maximum flow problem. Let $n$ be the number of modules in the program, $m$ be the number of module instances during an execution of the program, and $p$ be the number of processors. Initially we will restrict our attention to the case $p=2$. If $A_i$ is the name of a module in the program, then $A_{ij}$ is defined to be the $i^{th}$
instance of A. The last instance of A will be denoted \( A_{\text{max}(A)} \) or, unambiguously, \( A_{\text{max}} \).

We begin by constructing a dynamic module interconnection graph. We create \( m \) nodes, labeled with the \( m \) module instance names. There will be an undirected edge between two nodes labeled \( A_i \) and \( B_j \), \( A \neq B \), if and only if \( A_i \) and \( B_j \) are executed consecutively. The capacity of \( (A_i, B_j) \) will be the cost of communication between \( A_i \) and \( B_j \) if they are assigned to different processors. In addition, there will be an edge between each pair of nodes \( A_i \) and \( A_{i+1} \). Let \( r_{j,k}(A) \) be the cost of reassigning module A from processor \( j \) to processor \( k \). Then we define

\[
c(A_i, A_{i+1}) = \frac{1}{2} \left[ r_{1,2}(A) + r_{2,1}(A) \right], \quad 1 \leq i < \text{max}(A) \tag{1}
\]

Note that for symmetric reassignment costs, \( c(A_i, A_{i+1}) = r_{1,2}(A) = r_{2,1}(A) \).

Using the dynamic module interconnection graph as a basis, we then construct a dynamic processor flow graph. We add two nodes labeled \( P_1 \) and \( P_2 \) for processors 1 and 2, respectively, and 2\( m \) new edges. There will be an undirected edge from each module instance node to \( P_1 \), and another to \( P_2 \). Let \( e_j(A_i) \) be the cost of executing \( A_i \) on processor \( j \). Then the capacities of the new edges are computed as follows:

\[
c(A_i, P_2) = e_1(A_i), \quad 1 \leq i \leq \text{max}(A) \tag{2}
\]

\[
c(A_i, P_1) = e_2(A_i) + \frac{1}{2} \left[ r_{2,1}(A) - r_{1,2}(A) \right], \quad \text{max}(A) \neq 1 \tag{3}
\]

\[
c(A_{\text{max}}, P_1) = e_2(A_{\text{max}}) + \frac{1}{2} \left[ r_{1,2}(A) - r_{2,1}(A) \right], \quad \text{max}(A) \neq 1 \tag{4}
\]

\[
c(A_i, P_1) = e_2(A_i), \quad 1 < i < \text{max}(A) \text{ or } 1 = i = \text{max}(A) \tag{5}
\]

The asymmetry of these equations with respect to \( P_1 \) and \( P_2 \) is explained by
the fact that Eqs. (2)-(5) do not represent a unique valid assignment of edge capacities. The final graph thus consists of \(m+2\) nodes and \(4m-(n+1)\) edges.

The dynamic processor flow graph is then interpreted as a commodity flow graph with single source \(P_1\) and single sink \(P_2\). Using a max-flow min-cut algorithm, we find the minimum capacity cutset.

**Claim:** The capacity of a minimum capacity cutset of the dynamic processor flow graph is equal to the cost of the minimum cost dynamic assignment of the program.

Clearly there is a one-to-one correspondence between the set of all possible cutsets of the graph and the set of all possible dynamic assignments of the program. Then we claim that the capacity of any cutset of the graph is equal to the cost of the corresponding assignment, in which a module instance \(A_i\) is assigned to processor \(j\) if and only if \((A_i, P_j)\) is not in the cutset. To prove this assertion, we examine each of the three cost categories separately.

**Module instance execution costs:**

If \(A_i\) is assigned to processor 1 by the cutset, then \(c(A_i, P_2)\) is added to the capacity of the cutset. \(c(A_i, P_2)\) is the cost of executing \(A_i\) on processor 1. If \(A_i\) is assigned to processor 2, then \(c(A_i, P_1)\) is added to the cutset capacity. \(c(A_i, P_1)\) includes the cost of executing \(A_i\) on processor 2.

**Intermodule communication costs:**

If \(A_i\) and \(B_j\), \(A \neq B\), are assigned to different processors by the cutset, then any edge \((A_i, B_j)\) if it exists must be in the cutset. Otherwise, there
would exist a path from $P_1$ to $P_2$ through nodes $A_i$ and $B_j$ over which the flow could be increased. If $A_i$ and $B_j$ communicate, the dynamic processor flow graph will contain an edge $(A_i, B_j)$. Consequently, the cutset capacity will include the cost of communication between $A_i$ and $B_j$ if the cutset assigns them to different processors.

Module reassignment costs:

There are four cases to consider for each module $A$ in the program.

(i) $A_1$ assigned to $P_1$, $A_{\text{max}}$ assigned to $P_1$ -

For each time that $A$ is reassigned to $P_2$, it is subsequently reassigned to $P_1$. If $A$ is reassigned to $P_2$ a total of $x$ times, then there will be $2x$ edges in the cutset between successive instances of $A$. The total contribution to cutset capacity from these edges is $2x \cdot \frac{1}{2} [r_{1,2}(A) + r_{2,1}(A)] = x \cdot r_{1,2}(A) + x \cdot r_{2,1}(A)$, as required.

(ii) $A_1$ assigned to $P_2$, $A_2$ assigned to $P_2$ -

This is similar to (i). The contributions to cutset capacity from $(A_1, P_1)$ and $(A_{\text{max}}, P_1)$ due to reassignment costs cancel each other.

(iii) $A_1$ assigned to $P_1$, $A_2$ assigned to $P_2$ -

If $A$ is reassigned to $P_2$ $x$ times, it is reassigned $x-1$ times to $P_1$.

The contribution to cutset capacity due to edges between successive instances of $A$ will be $[x+(x-1)] \cdot \frac{1}{2} [r_{1,2}(A) + r_{2,1}(A)]$. In addition, there will be a contribution of $\frac{1}{2} [r_{1,2}(A) - r_{2,1}(A)]$ from $c(A_{\text{max}}, P_1)$, resulting in a total of $x \cdot r_{1,2}(A) + (x-1) \cdot r_{2,1}(A)$ in the cutset capacity due to reassignment of $A$.

(iv) $A_1$ assigned to $P_2$, $A_2$ assigned to $P_1$ -

This case is similar to (iii), but $(A_1, P_1)$ is in the cutset, not
\((A_{\max'}, P_1)\). \(c(A_1, P_1)\) includes \(\frac{1}{2} [r_{2,1}(A) - r_{1,2}(A)]\), resulting in a total contribution to cutset capacity due to \(x\) reassignments of \(A\) to \(P_1\) and \(x-1\) reassignments to \(P_2\) of \((x-1) \cdot r_{1,2}(A) + x \cdot r_{2,1}(A)\).

As a result of using this approach to the solution of the optimal dynamic assignment problem, the time complexity of the algorithm outlined above is the time complexity of the max-flow min-cut algorithm used in implementing it. Since there are \(4m-(n+1)\) edges \((m > n)\), the dynamic processor flow graph can be constructed in \(O(m)\) time. Using Galil's max-flow min-cut algorithm on the graph requires \(O(|V|^{5/3}|E|^{2/3}) = O(m^{7/3})\) time to find a minimum capacity cutset.

To illustrate the application of this algorithm, consider the program whose costs are defined in Tables I-III. Figure 1 shows the construction of the dynamic processor flow graph for this example and indicates the minimum capacity cutset of the graph. The minimum cost assignment of the example is unique; however, this will not be the case in general. If there exist more than one minimum cost assignment, additional criteria might be used to select a "best" minimum cost assignment. For instance, we might wish to choose the optimal assignment which minimizes interprocessor communications to relieve congestion within the communication subnetwork.

(A simple modification of the construction of the processor flow graph allows us to find such an assignment directly.)

Thus far we have ignored one potential source of difficulty. According to Eqns. (3) and (4), it is possible for either \(c(A_1, P_1)\) or \(c(A_{\max'}, P_1)\) to have a negative value. Max-flow min-cut algorithms work only with non-negative edge capacities. However, this problem can be easily corrected.
Suppose \( c(A_1, P_j) = -x \), where \( x > 0 \). Then add \( x \) to \( c(A_1, P_j) \) and \( c(A_1, P_2) \). Exactly one of these two edges is in any cutset of the graph, and hence every cutset will have its capacity increased by \( x \). A minimum capacity cutset in the original graph will be a minimum capacity cutset of the modified graph. A similar procedure can be used if \( c(A_{\max}, P_j) < 0 \).

Extending the algorithm to problems involving more than two processors is unfortunately not so readily accomplished. There are at least two major difficulties. It does not appear possible to construct a dynamic processor flow graph for more than two processors which incorporates asymmetric reassignment (or communication) costs. Furthermore, if \( p \geq 3 \), reassignment and communication costs may be processor dependent, in the sense that the costs may depend on the particular pair of processors involved. The dynamic processor flow graph does not allow inclusion of nonuniform costs of this nature, either.

Even if the reassignment and communication costs are symmetric and uniform, however, the most serious difficulty in extending this approach still remains. There does not exist a general algorithm for the efficient computation of a minimum-capacity multi-partition cutset. For \( p = 3 \) an algorithm has been developed which involves the use of a minimum of two and a maximum of three applications of a max-flow min-cut algorithm to find the minimum capacity tricutset [9]. However, the algorithm may find a nonoptimal solution, in which case it gives an assignment together with a bound on how far from optimal the solution might be. Although nonoptimal solutions are rare, the method does not seem to generalize for \( p > 3 \). For \( p \geq 4 \), the optimal static assignment problem can be shown to be NP-com-
plete [10]. It seems likely then that an efficient algorithm for the solution of the general dynamic problem does not exist.

IV. Optimal Assignments by Dynamic Programming

In this section we present an entirely different approach to solving the optimal dynamic assignment problem, an approach which may be applied to problems with any number of processors and with asymmetric as well as nonuniform costs. The price we must pay for this is increased time complexity, although the algorithm will be shown to be far better than exhaustive enumeration of all possible dynamic assignments.

We begin by noting that Belady's Principle of Optimality certainly is applicable to our problem. If we have determined the optimal assignments for the first \( i \) module instances in the program execution, then the assignments of the remaining \( m-i \) instances must represent an optimal decision sequence, given the \( i \) assignments already made. Hence, we can use dynamic programming methods to solve the problem. A problem suitable for solution by a dynamic programming algorithm has a number of characteristics [11]. It can be divided into a series of stages, with a policy decision required at each stage. Each stage has a number of states associated with it, and the effect of a policy decision at a given stage is to transform a state associated with that stage into one of the states associated with the next stage. The solution procedure begins at the first (or last) stage by identifying the optimal decision for each of the possible states, and then proceeds forward (or backward) stage by stage, finding optimal policies for all of the states in each stage. The procedure terminates
upon reaching the final (or initial) stage, at which point we can identify the entire sequence of decisions which optimizes the solution.

In the optimal dynamic assignment problem, each module instance corresponds to a single stage, while the decision to be made at each stage is the choice of assignment for the module instance corresponding to that stage. The efficiency of a dynamic programming solution will depend on the number of states in each stage as well as the complexity of the decision-making process. Assuming that we are at stage \( s_i \), the stage corresponding to the \( i \)th module instance, the states of \( s_i \) might be all possible assignments for the first \( i \) module instances, since this would certainly be sufficient knowledge to compute optimal assignments for the remaining \( m-i \) modules, given a state in \( s_i \). However, this would in effect mean creating a \( p \)-ary decision tree of depth \( m \), leading to an overall complexity of at least \( O(p^m) \) as \( m \) gets large.

Fortunately, we can do better than this. Suppose that we have just finished \( s_{i-1} \) and are now ready to begin \( s_i \). Let \( N(i) \) be the name of the module whose instance corresponds to \( s_i \), and let \( M(i) \) be the \( i \)th module instance. Then the cost of assignment for \( M(i) \) depends only on two assignments for instances which precede \( M(i) \). These are the assignment of \( M(i-1) \) and the assignment of the most recent instance of \( N(i) \) among the first \( i-1 \) instances. Consequently, the information necessary to define a state of any stage is the set of most recent assignments for all the modules encountered prior to that stage. This reduces the maximum number of states in a stage from \( p^m \) to \( p^n \). In going from \( s_{i-1} \) to \( s_i \), all states are transformed by the identity of the most recent instance of \( N(i) \).
The dynamic programming algorithm can most easily be explained as a shortest path algorithm, albeit a very simple one in which all paths between two points have the same number of edges. We will also make use of the fact that a state in stage $s_i$ will consist of most recent assignments for all modules which occurred among the first $i$ instances, not all $n$ modules, and hence there will be a reduction in the number of states which must be considered in each stage until we encounter the first instance of the $n$th distinct module.

Define $s_0$ to be the initial stage before any module instances have been assigned. We will construct a dynamic assignment tree by creating nodes for each stage of the algorithm and edges to them from nodes in the previous stage. Nodes created during $s_i$ are said to be contained in $s_i$. At any stage $s_i$, $1 \leq i \leq m$, we will create $p^\lambda(i)$ nodes, where $\lambda(i)$ is the number of distinct modules among the first $i$ instances. Each node will uniquely correspond to one of the states associated with $s_i$. $s_0$ will consist of a single node $S_0$ called the start node which will be the root of the tree. At the conclusion of $s_i$, there will be a path of $i$ edges from $S_0$ to each node of $s_i$. The length of the path will be the sum of the edge weights in the path. The edge weights will be defined later.

We begin $s_1$ by creating $p$ nodes, one for each of the possible assignments of $M(1)$. We also create an edge from $S_0$ to each node $V$ of $s_1$, and give $(S_0,V)$ a weight equal to the cost of assigning $M(1)$ to the processor indicated by the state associated with $V$.

Assume now that the algorithm has just completed stage $s_{i-1}$, and that the length of the path from $S_0$ to a node $V$ in $s_{i-1}$ is the cost of the opti-
mal dynamic assignment of the first \( i-1 \) instances, given the state associated with \( V \). In \( s_i \), we create \( p^\lambda(i) \) new nodes and extend the dynamic assignment tree to all nodes in \( s_i \) so that the length of the (unique) path from \( S_0 \) to a node \( W \) in \( s_i \) is the cost of an optimal dynamic assignment of the first \( i \) instances, given the state associated with \( W \).

We must consider two cases. If \( \lambda(i) > \lambda(i-1) \), then \( M(i) \) is the first instance of \( N(i) \). Each state in \( s_{i-1} \) is transformed into \( p \) unique states in \( s_i \) by augmenting the state by each of the choices of assignment for \( M(i) \). This is represented in the dynamic assignment tree by creating \( p \) edges from each node \( V \) in \( s_{i-1} \) to \( p \) nodes in \( s_i \). If \( W \) is a node in \( s_i \) with an edge to it from \( V \), then the weight of \((V,W)\) is the cost of assigning \( M(i) \) to the processor indicated by the state associated with \( W \), given the state associated with \( V \). Obviously, if the length of the path from \( S_0 \) to \( V \) is the cost of an optimal dynamic assignment for the first \( i-1 \) instances, given the state associated with \( V \), the length of the path from \( S_0 \) to \( W \) will be the cost of an optimal assignment for the first \( i \) instances, given the state associated with \( W \).

The situation is more complex if \( \lambda(i)=\lambda(i-1) \). The algorithm has already encountered an instance of \( N(i) \), and so the number of states in \( s_i \) and \( s_{i-1} \) are equal. Once again, a state of \( s_{i-1} \) may be transformed into \( p \) states of \( s_i \), one for each of the choices of assignment for \( M(i) \). Now, however, each state of \( s_i \) may be the result of the transformation of any one of \( p \) states in \( s_{i-1} \), those states which differ only in the most recent assignment of \( N(i) \). That is, there are potentially \( p \) different paths by which we may reach a node in \( s_i \) from nodes in \( s_{i-1} \) (and hence from \( S_0 \)).
There are p-1 other nodes in s_i which may be reached from the same p nodes in s_i-1. The states associated with p nodes in s_i which are each reachable from the same p nodes of s_i-1 differ only in the assignment of M(i); furthermore, their states are identical with those of the related p nodes in s_i-1 except in the most recent assignments for N(i). Fig. 2 illustrates all of the possible edges which can be used to extend the dynamic assignment tree from s_i-1 to s_i for one such set of p nodes in s_i.

Our task is to select exactly one edge to each node in s_i. The weight of an edge from a node V in s_i-1 to a node W in s_i is the cost of making the transformation from the state associated with V to the state associated with W; that is, it is the cost of making the assignment for M(i) indicated by the state associated with W, given the state associated with V. Hence, the length of the path from S_0 to W through V is the cost of the optimal assignment of the first i instances, given the state associated with W and given that the previous assignment of N(i) was as indicated by the state associated with V. Since there are paths from S_0 to W through p different nodes of s_i-1 whose associated states cover all possible assignments of N(i) immediately prior to s_i, one of these paths must represent the optimal assignments for the first i module instances, given the state of W. We simply choose the path of shortest length and add the edge from s_i-1 to s_i in that path to the dynamic assignment tree. Note that choosing the shortest length path to W is equivalent to choosing the optimal assignment of the previous instance of N(i), given the state associated with W.

Fig. 3 shows the graph which consists of all edges considered by the dynamic programming algorithm in constructing the dynamic assignment
tree for the example used in the previous section. Fig. 4 is the dynamic assignment tree for this problem. The final stage of a dynamic assignment tree will have $p^n$ nodes. The length of the path from $s_0$ to a node $V$ in the last stage is the cost of the optimal assignments for all $m$ instances, given that the final assignments of the $n$ modules are as indicated by the state associated with $V$. The length of the shortest path from $S_0$ to any of the nodes in $s_m$ is therefore the cost of an optimal assignment, and the sequence of states associated with the nodes along this path define an optimal dynamic assignment for the program's execution. The optimal dynamic assignment is shown in Fig. 4 as the path in bold lines.

The dynamic programming algorithm will solve the optimal dynamic assignment problem for any number of processors and for asymmetric as well as nonuniform reassignment and communication costs. The penalty for its wider applicability is greatly increased time and space complexity bounds. If we use as the measure of time complexity the number of comparisons which the algorithm makes, then for every stage $i$ such that $\lambda(i)=n$, there are $p^n p = p^{n+1}$ comparisons needed to extend the dynamic assignment tree to the next stage. The overall time complexity is thus bounded by $O(mp^{n+1})$. Since there is an edge from $S_{i-1}$ to each node in $S_i$, the space complexity of the dynamic assignment tree is bounded by $O(mp^n)$. Both of these bounds suggest that the algorithm is best suited to those problems in which $n$ is relatively small, and in particular where $m \gg n$. 
V. Conclusions

The commodity flow algorithm for the solution of the optimal dynamic assignment problem is extremely efficient for a problem involving two processors, but cannot be easily extended to accommodate a larger number of processors. On the other hand, a dynamic programming approach allows the solution of a much wider range of problems with a concomitant increase in space and time complexity bounds.

The increased space complexity of the dynamic programming algorithm can be partially mitigated by pruning from the dynamic assignment tree all paths which terminate before the final stage, thus reducing the size of the tree. We may also be able to construct the tree with an expected time complexity that is far less than the worst case bound. It has been speculated that a program may exhibit a kind of "locality" in the sequence of its module executions. The dynamic programming algorithm may be modified to take advantage of the predictability that this implies to significantly reduce the number of comparisons that it must make [12].

There are a number of topics of interest related to the optimal dynamic assignment problem that need to be investigated. Principal among these is the development of heuristic algorithms for the predictive or adaptive dynamic assignment of modules in real time. Related to this area is the problem of monitoring program behavior and accumulating information for the estimation of cost parameters to be used in making a predictive assignment. It remains to be seen if dynamic program execution can be successfully (that is, cost-effectively) implemented in a distributed
processing environment, but the optimal assignment algorithms in this paper clearly indicate the potential for improved resource utilization that distributed program execution has to offer.
REFERENCES


TABLE 1
Module Instance Execution Costs

<table>
<thead>
<tr>
<th>Module instance</th>
<th>Cost on processor 1</th>
<th>Cost on processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>B₁</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>A₂</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>C₁</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>B₂</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>A₃</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>
# TABLE II
Intermodule Communication Costs

<table>
<thead>
<tr>
<th>Communication between</th>
<th>Cost of communication *</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁ - B₁</td>
<td>2</td>
</tr>
<tr>
<td>B₁ - A₂</td>
<td>7</td>
</tr>
<tr>
<td>A₂ - C₁</td>
<td>12</td>
</tr>
<tr>
<td>C₁ - B₂</td>
<td>1</td>
</tr>
<tr>
<td>B₂ - A₃</td>
<td>7</td>
</tr>
</tbody>
</table>

* if the communicating instances are assigned to different processors
TABLE III
Module Reassignment Costs

<table>
<thead>
<tr>
<th>Module</th>
<th>Cost of reassignment</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P₁ to P₂</td>
<td>P₂ to P₁</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Fig. 1. Dynamic processor flow graph for the module execution sequence and costs defined in Tables I - III. The optimal dynamic assignment assigns $A_1$, $B_2$, and $A_3$ to processor 1 and $B_1$, $A_2$, and $C_1$ to processor 2 at a cost of 52.
Fig. 2. All possible edges from $p$ related nodes of $s_i$ to nodes in $s_{i+1}$ when $\lambda(i) = \lambda(i+1)$. 
Fig. 3. All edges considered in constructing the dynamic assignment tree. The upper and lower edges from a node to the next stage represent assignment to processor 1 and processor 2, respectively.
Fig. 4. Dynamic assignment tree, with the optimal assignment shown in bold lines.