Echo Algorithms: Depth Parallel Operations on General Graphs

ERNEST J. H. CHANG

Abstract—This paper describes a method for the detection of properties of general graphs in an environment in which each node can be considered an autonomous processor, interacting with its neighbors by passing messages.

These algorithms are decentralized in that they depend on no central controlling process nor on global storage. No node is required to know the configuration or extent of the graph, and no global clock is required. The algorithms are inherently asynchronous, and in general require execution time proportional to the diameter of the graph.

Index Terms—Computer networks, decentralized control, distributed computing, graph algorithms.

I. INTRODUCTION

THE technique of depth-first search is well known in graph algorithms [1]. For example, finding the strong components of a directed graph [9] or the biconnected components of a general graph [1] are based on this method. However, depth-first search is a technique which assumes that only one operation at a time is performed on a graph. We ask whether, given the possibility of parallel graph operations, there is a parallel traversal method which is analogous to depth-first search. In theory, such a traversal method could cover the graph in time linearly proportional to the traversal of the *diameter* of the graph, the longest of the shortest paths between all pairs of nodes. A sequential algorithm can do no better than visit each node in turn. Furthermore, if there is a procedure which can logically be separated into independent parts, it is reasonable to execute these parts simultaneously rather than sequentially.

We will present a class of algorithms for detecting properties of general graphs which are based on the following model.

A distributed computer system is a connected graph in which each node is a processor and each edge is a bidirectional communications link. Let each processor have its own local storage, and assume that it is capable of supporting multiple local processes, so that while some application tasks are executing, it is also capable of sending and receiving messages, and of initiating control algorithms. We assume a message-passing capability which can send in parallel to the immediate neighbors of a node, and has enough memory at each node to store all incoming messages. We do not presume any fixed speed for transmission between nodes, and we do not allow messages to overtake one another on a link. Furthermore, we will assume that message passing between two ends of a link use a protocol requiring positive acknowledgment. Thus, either the message has been sent successfully or the sender knows it has not and retries. Any permanent message loss therefore occurs not in transmission, but in association with node failures. Furthermore, there is no shared memory, global clock, or central controller, nor does any node know the extent or membership of the entire graph. The algorithms we will present are decentralized, and they function through the cooperative behavior of the nodes. They are based on message passing, and they use a parallel graph traversal technique which takes advantage of potential simultaneous activity.

The coordination of a loosely coupled multiprocessor system presents interesting new problems. Coordination implies knowledge of global properties, which is usually not present *a priori* in a loosely coupled system having no global mechanisms. Decentralized algorithms, existing at all processors, must operate asynchronously through message passing to coordinate the system as a whole. Some control functions which are useful in a distributed system are: assigning an identity to a new processor, finding the ordering of all processor identities in the system, finding the configuration of the system, providing a mutual exclusion mechanism which per-

mits only sequential access to a critical set of resources, find-ing the clusters of nodes (subgraphs) which are interconnected by single links, and broadcasting a message quickly to all nodes. Clearly, many of these control functions correspond to the detection of simple graph properties of the multiprocessor system. While we have studied decentralized algorithms for these problems and others not included in this brief list, it is our purpose in this paper only to present the fundamental techniques underlying all such algorithms. We will then use three of them to illustrate specific aspects of decentralized control. A complete description of all the algorithms is outside the scope of this paper, and can be found elsewhere [3]-[5].

The algorithms to be described are called Single-Source Sort, Multisource sort, and Biconnected Component Detection. As we present each algorithm, we will indicate its relevance to the control requirements of a distributed system.

The decentralized nature of our method makes these algo-rithms quite different from other parallel models which have been proposed for finding graph properties. For example, a parallel depth-first method in the literature is based on k-processors sharing common memory [6]. The studies of parallelism for graph algorithms by Arjomandi [2] also assume k-processors and common memory. Rosenstiehl's [8] distrib-uted algorithms based on a network of finite state machines are very close in concept to ours, but assume a synchronous system with simultaneous transitions based on sensing the states of all neighbors at each step.

Fully parallel algorithms on graphs must solve some basic problems. If several edges lead to one node, and the parallel traversal of edges starting from some initial node should arrive at that node simultaneously, how is this to be handled? Does the message from each of the edges get passed on, and if not, what is to be done with the ones which are aborted? How does information get back to the starting node in a coordi-nated fashion? We shall show that the class of parallel graph algorithms which we call *echo algorithms* address these prob-lems in simple and efficient ways.

II. ECHO ALGORITHMS

The basic ideas behind echo algorithms are simple, and will be described informally in this section. Given a general graph with intelligent nodes which can communicate along its edges, the *first* idea is that message passing is the fundamental opera-tion of any echo algorithm. Traversal of the graph therefore means passing messages from one node to another. For any particular node i which starts the execution of an echo algo-rithm, the messages originating from i form a *family*, sharing the identity of i in common.

The *second* idea is that there are two phases in the traversal of a graph: a *forward* phase and an *echo* phase. The forward traversal of a graph from a starting node is accomplished by *explorers*, and the echo phase by *echoes*. Let us confine our-selves at this point to *single-source* echo algorithms, those which are started by one node, so that we can study the be-havior of one family of explorers and echoes.

The *third* idea, then, is that each node which is visited for the *first* time by an explorer will propagate explorers in paral-lel along all the out-edges of that node. For a connected un-directed graph, these would be all edges except the edge at which the first explorer arrived. This edge is called the *first* edge. Explorers coming to an already visited node will turn into echoes, as will explorers coming to a *sink* node, one which has no other edges. In general, echoes travel in a direction opposite to that of explorers.

The *fourth* idea is that of arbitration. We assume a mecha-nism at each node such that if two or more explorers arrive at an unvisited node simultaneously, one and only one of them is chosen as the *first* explorer to the node, and its edge of arrival as the *first* edge of the node. The other explorers are then considered as *subsequent* explorers to a visited node, and turn into echoes.

The *fifth* principle is that of synchronization. A node will echo on its *first* edge after it receives an echo for each explorer sent out. This is called the *echo-merge* mechanism. We as-sume that there is an arbiter mechanism at each node such that if messages should arrive simultaneously, they are given some arbitrary sequential ordering.

Last, but not least, explorers and echoes will carry informa-tion with them about those parts of the graph which they have traversed. A node which synchronizes echoes will process this information, and will send the result with the echo from that node. The starting node will finally receive all its echoes from its out-edges, and after processing this information, will obtain the result of the algorithm.

An echo algorithm, then, is started by some initiating node sending out in parallel as many explorers as there are out-edges, each one carrying the identity of the starting node.

A. Definitions

Given a graph $G = <V, E>$ where V is a nonempty set of nodes and E is a set of edges of the form (x, y) where x and y are members of V, let n be the cardinality of V and e the cardinality of E. We distinguish several classes of graphs. All connected undirected graphs are called *C-graphs*. For directed graphs, there are several possible subgraph relationships. A directed graph G is *type I* or *digraph I* if it is a strongly con-nected graph in which every node is reachable from every other node. We call the *reach set* of a node v the set of nodes in G to which there exists a directed path from v. Then a *digraph II* is a directed graph whose reach sets are dissimilar, but one of which is the entire set of nodes of the graph. A *digraph III* is a directed graph which has no reach set contain-ing all the nodes of the graph. Thus, there exists no node from which all other nodes can be reached. A *sink* node in a C-graph is a node with only one edge, while for a digraph, it is a node with no edges leading out from it.

An *initiator* node S is a member of V which produces, in an execution of an echo algorithm, a *family* of explorer and echo messages. Let *explorer*$[a,b]$ represent the explorer going from node a to node b. Then if α represents an arbitrary node, *explorer*$[\alpha,b]$ is any explorer coming to node b, and *explor-er*$[a,\alpha]$ would be any explorer leaving node a. We adopt the same convention for echoes, and use { } to represent a set in the usual manner, with a suffix $-S$ to indicate the initiator.

Thus, {echo[α,b]} – S would refer to all the echoes going to node b belong to the family of messages of initiator S.

By [a,b] we will mean the edge going from a to node b, and by [b,a] we will mean the same edge, but in the sense of b to a. To convey a neutral sense of an edge connecting a and b, we use (a,b).

An explorer which is the *first* to arrive at a node is called a *primary* explorer. An edge carrying a primary explorer is a *P-edge*. A node may have several P-edges, but the P-edge on which it was itself first visited is called its *first* edge. Its other P-edges are *first* edges to their successor nodes. Non-P edges clearly carry explorers to already visited nodes. Every node has a *first* edge except the initiator, which is considered visited *a priori*. A node which has P-edges leading out of it is called a *P-node*, and a node which does not is a *non-P node*.

Echoes arise in two situations: at the termination of an explorer, and from a node which has received an echo for every explorer it has sent out, and then itself echoes on its *first* edge. In the first case, such an echo is called an *initial* echo, and the node at which its corresponding explorers terminated is called the *origin* of the initial echo.

B. General Properties

In order to elicit some properties general to most of the echo algorithms, let us first describe the *pure traversal* algorithm. This will also establish a prototype for the description of other algorithms to follow.

Algorithm 0 is a traversal of a graph from an initiator node, and we cannot traverse nodes which are not reachable. Hence, for a digraph, we can only study the subgraph G' induced from G by the reach set of the initiator node S in G. Call this the *S-reach graph* of the original graph G. Therefore, Algorithm 0 can apply to any C-graph or diagraph I, and to the S-reach graphs of digraph II and digraph III. Fig. 1 illustrates these different types of graphs.

Explorers and echoes represent messages of two types going from node to node. The basic identity of each is thus *type* and family name S. Implicit to the message is the TO and FROM node information, and other protocols which the communications system may require. These are constant, and we include them under the notion of basic identity. Algorithm 0 requires no more than basic identity on a message.

Algorithm 0 – Pure Traversal: First, assume that initiator S sends explorers in parallel on all its out-edges, where an out-edge is a directed edge from S for digraphs and all the edges of S for a C-graph. We must consider the activity at each node for the arrival of an explorer or an echo in a particular edge.

1) If the explorer is the first to arrive at the node, mark the edge as *first*, and send explorers in parallel from the node on the out-edges of the node. For a digraph, these are edges directed from the node. For a C-graph, these are all edges except the *first* edge.

2) If the explorer is not the first or if there are no out-edges, then echo back along the edge on which the explorer arrived.

3) If an echo comes to the node, then mark the edge as having received an echo. If all echoes for the node have ar-
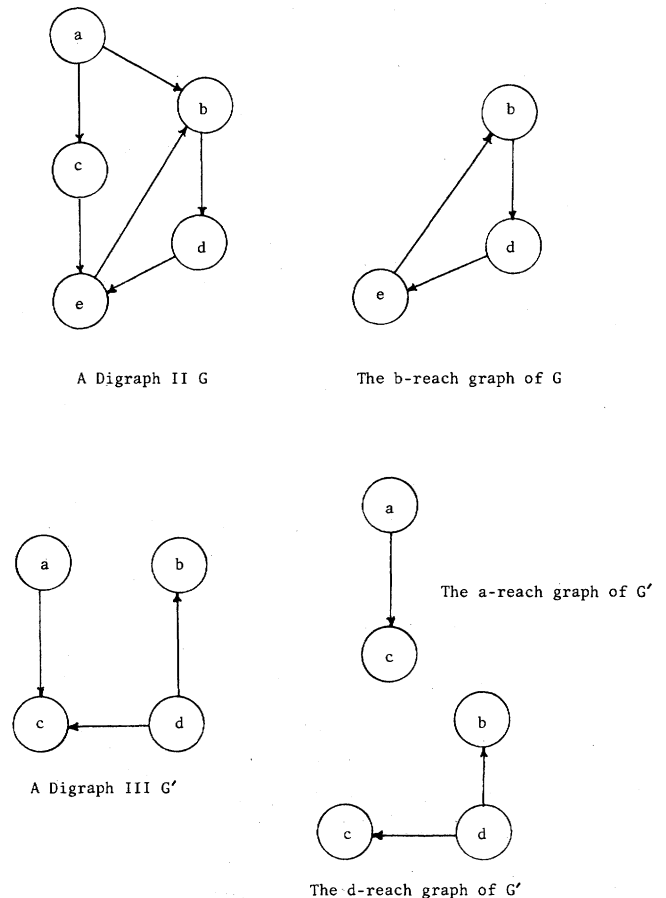

Fig. 1. Some reach graphs.

rived, then send an echo back along the *first* edge of the node, unless the node is the initiator, in which case we are finished.

Let us look at the properties of the pure traversal algorithm. These are based on the three fundamental mechanisms of echo algorithms: explorers are sent in parallel from a node, each node has only one edge on which it is *first* visited by an explorer, and a node waits for all its echoes to come back before it itself echoes on its primary edge.

Property 1: Each node receives at least one explorer.

Argument: By assumption, only those graphs and subgraphs in which all nodes are reachable from S are in question. Therefore, every node has a path from S. If any node did not receive an explorer, then its predecessor on the path from S could not have received one. By induction, either S did not emit any or else the node is not reachable. In any case, presuming no loss of explorers, a contradiction arises. Hence, there is no such node. □

Property 2: Eventually, all explorer activity will terminate.

Argument: We are only concerned with finite graphs. By Property 1, every node will eventually be visited, and any explorers generated thereafter can only come to sink nodes or visited nodes, turning into echoes. □

Property 3: There exist non-P nodes which have no P-edges leading from them.

Argument: We are referring to edges which are *first* edges to their successor nodes. Trivially, sink nodes have no out-

edges. Furthermore, by Property 1, all nodes eventually get visited for the first time by an explorer. Hence, the last such node can send explorers only to visited nodes. Thus, there are no P-edges leading from it. □

Property 4: A P-node sending a primary explorer to its successor can be said to precede it. Then there can be no cycle of precedence.

Argument: If node a sends a primary explorer to b, then this will cause b to send explorers from b. Hence, the activation of a could not have been from one of these explorers. Thus, if a precedes b, then b could not precede a.

Corollary 1: It follows immediately that an edge (a,b), if it is a P-edge, can only carry an explorer in one direction, either from a to b or vice versa.

Corollary 2: It also follows that a non-P edge must carry explorers in both directions. For if a and b are not activated one by another, they must have both sent out explorers on all their *nonfirst* edges. Clearly, the edge (a,b) is such an edge. Thus, it must carry explorer$[a, b]$ as well as explorer$[b, a]$. □

Property 5: Every explorer on an edge induces a corresponding echo.

Argument: Consider all non-P edges. They carry explorers to visited nodes, and immediately induce an echo. For those P-edges which lead to sink nodes, a corresponding echo is also generated at that node. It follows that a node which only has non-P edges leading from it will get all its echoes, and be able to send an echo on its *first* edge, or else it is a sink node, and also echoes.

A P-node has a primary edge leading out of it, and a non-P node does not. Consider an explorer on a P-edge from a P-node. It either leads to another P-node or to a non-P node. By induction on the finite size of the graph, all P-nodes must eventually lead to non-P nodes. In the previous paragraph, we have shown that non-P nodes will echo on their *first* edges. Hence, the P-edges leading to non-P nodes will receive echos. By induction, all P-edges will eventually receive an echo. □

III. PERFORMANCE OF ALGORITHM 0

The efficiency of this algorithm can be considered using three metrics: total number of message passes in the system, elapsed communication time for the algorithm, and the amount of storage required at each node. We make some important assumptions about elapsed time. First, we assume that processor time is very small compared to communications time. Second, in an asynchronous system, we have no guarantees as to how fast messages move, except that all messages take a bounded time to traverse a link. For purposes of analysis, we will consider the average case where messages take approximately the same time to traverse an edge.

Consider number of message passes first. It is bounded by $4e$ where e is the number of edges in the graph. Since each edge (a,b) can have at most two explorers, one in each direction, and two corresponding echoes, the total number is bounded by $4e$. Note that for a digraph, it is $2e$, since there are no symmetrical pairs of explorers which travel on directed edges.

Assume that each edge takes approximately one unit of time to traverse, so that we can estimate bounds for the communi-

cation cost of a decentralized algorithm. Define the *S-span* of a graph as the longest of the shortest paths from S to any element in the reach set of S. When the metric of weight used is message travel time, we used the term *timed S-span*.

It follows that the timed S-span of a graph represents the time it takes for explorers to reach every node of the graph, called the *forward phase* of the echo algorithm. If we assume that explorers and echoes have the same speeds, then the traversal of the graph from S will take twice the timed S-span of the graph.

For a C-graph or a digraph I, traversals starting from any node i will have the same reach set. The largest of the i-spans is referred to as the *diameter* of the graph. The *timed diameter* is then the maximum of all traversals of the graph for all starting nodes. We extend this notion to include digraph II's, even though the reach sets of different nodes may be different. The largest i-span will be taken to be the diameter of a digraph II. From this point on, unless otherwise specified, we will mean timed path length when we refer to path length, and timed diameter when we refer to diameter.

In the execution of a pure traversal algorithm from a initiator S, the communication time is less than or equal to twice the diameter of the graph. This result follows immediately from the definition of diameter and the parallel activity of the algorithm.

The storage required at each node for a pure traversal algorithm is $O(n)$ bits where n is the number of nodes in the graph. This follows from observing that a node i has at most n edges, each of which needs one bit to mark the arrival of its echo. To mark the primary edge of i requires $\log n$ bits, and to maintain the name of the node only uses $\log n$ bits. Finally, each message carries the basic identity of the initiator and a type, which is $(\log n + 1)$ bits. Finally, one bit is needed to mark a node as visited. Thus, the total is $(n + 3 \log n + 2)$ bits, which is $O(n)$ bits.

IV. TRAVERSAL EXECUTION GRAPH

Since we have not made any particular assumptions as to the exact speed of explorers or echoes, a particular execution of an echo algorithm may cause different sequences of arrivals of explorers and different edges to be primary edges. Each execution of a graph G by a pure traversal algorithm can be represented by an *execution graph EG* drawn as follows.

Draw the node S' in EG to correspond to the node S in G, and for each explorer which goes from S to a successor node i in G, create a new node i' in EG, and draw a directed arc from S' to i'. Do this for each explorer coming from a node i in G, creating a new node in EG to correspond to its successor in G. If an explorer terminates at a visited node or a sink node in G, its corresponding directed edge in EG terminates in a leaf node of EG. Nodes in EG are labeled according to the names of their corresponding nodes in G.

The execution graphs for different types of graphs share the same general characteristics, but differ in some details. We will introduce their salient features by considering the execution graphs of connected undirected graphs first, and then seeing what the differences are in the case of directed graphs. Fig. 2
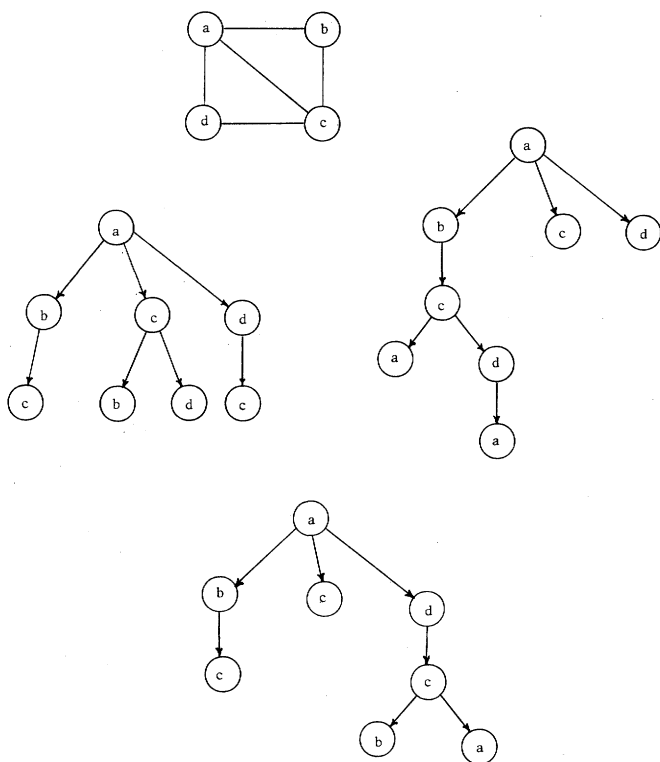
Fig. 2. A C-graph and some of its execution graphs.

shows a C-graph and several of its possible execution graphs. In spite of the differences in the topology of these execution graphs, however, they exhibit some remarkably regular properties.

First of all, it is easy to see that each is a *directed tree* in which the root is the initiator S and directed edges represent the movement of explorers from a node which is the root of a subtree to its successors. The movement of echoes is up the tree EG, with the echo-merge mechanism operating at each root of a subtree, to produce a new echo. A leaf node in EG which has a corresponding internal node in EG represents the stopping of an explorer at a visited node in G. In fact, if a is the leaf node and b is its immediate predecessor in EG, then not only do a and b exist as internal nodes in EG, but there also exists, by Property 4, the edge $[a,b]$ in EG, with b being a leaf. A leaf in EG with no corresponding internal node represents a sink node in G.

A. The Leaves of EG

The number of leaves of EG is equal to the number of distinct explorers in EG, which is also the number of distinct paths taken in the traversal of G. The graph EG has a number of internal nodes $EG.int$ and a number of leaf nodes $EG.leaf$. The original graph G has n nodes, one of which is S, the initiator of the algorithm. The number of edges of S is called the *degree* of S, written $S.d$. Let V' be the set of nodes of G with S removed. Then the sink nodes of G among V' can be designated $V'.sink$. Each of these corresponds to a leaf node in EG which has no matching internal node. Note that although the initiator node S may be a sink node in G, nevertheless, it will

always be an internal node of EG, being the root of the tree EG. The internal nodes in EG are exactly the nodes of G which are not sinks in V', i.e.,

$$EG.int = n - V'.sink.$$

The number of leaves of EG can be found by the following calculation for C-graphs.

Let $S.d$ be the degree of S, let n be the number of nodes of G, let $\{R\}$ be the set of nodes in G which are not sinks or S, with cardinality r, and let θ be the sum of the degrees of the nodes in $\{R\}$.

Proposition 1:

$$EG.leaf = S.d + [\theta - 2r].$$

Argument: Start with the degree of S. There are at least that many explorers in the execution of G. Consider the remaining nodes. A node with two edges has one primary in-edge and one out-edge. Thus, an explorer coming to such a node does not create an additional path, but merely extends an existing one.

Therefore, the number of edges in excess of two at each of the remaining nodes represents the number of additional paths created by explorers starting from S. However, if a node is a sink in G, clearly it does not add any more paths since it only has one edge. Thus, we exclude all sink nodes and the initiator node from consideration. Call this remaining set of nodes R, with cardinality r. For each of these nodes, the number of additional paths is the number of edges at the node, i.e., the degree of the node, in excess of two. For all nodes in $\{R\}$, then, the additional paths are the sum of the degrees of these nodes less twice their number $(2r)$. □

B. The Edges of EG

Each edge in EG represents the movement of an explorer, and therefore the total number of edges in EG is a measure of the total work done in one execution of a pure traversal algorithm. It turns out that this number is dependent only on the original graph G, and not at all on the manner of traversal. Furthermore, the number of edges in EG can be computed from a C-graph G as follows.

If $EG.e$ is the number of distinct edges of EG we wish to count, and $EG.leaf$ is the number of leaves of EG which we can compute from G as above, then given that we also know $G.e$, the total number of edges in G, and $V'.sink$, the number of sink nodes in G not including S, we have the following.

Proposition 2:

$$EG.e = G.e + (EG.leaf - V'.sink)/2.$$

Argument: Note first that every edge of G is in EG, either as an edge leading to an internal node or an edge leading to a leaf node which is a sink. Thus, the number of edges in EG is at least $G.e$, the number of edges in G.

Now consider those leaf nodes in EG which represent the stopping of an explorer at a visited node. If such a path is $[a,b]$, from a to b, then by Property 5, there must be a symmetric path $[b,a]$ which holds an explorer going the opposite

way which stops at node a. Each edge in G carrying an explorer to a visited node therefore contributes an additional edge to EG.

The number of such additional edges is easily found. The leaf nodes of EG represent either explorers stopping at sinks or at visited nodes, in which case such explorers occur in pairs. Therefore, if we know the number of sink nodes in G which contribute to the leaves of EG, the remaining leaves of EG are those from explorers stopping at visited nodes. The number of sinks in G can be counted by simply examining G, with the proviso that if the detector node S is a sink node (has only one edge), then it is not included, for it cannot contribute to a leaf of EG, being by definition the root of EG.

The number of leaf nodes in EG from visited nodes in G is then $EG.leaf - V'.sink$, and the number of additional edges of EG in excess of $G.e$ is half this number. But $EG.leaf$ can be found, by Proposition 1, from knowing some parameters of the original graph G. Therefore, the number of distinct edges in EG can also be determined from the graph G alone.   □

For directed graphs, a similar execution graph can be drawn. The number of leaf nodes is found by taking the out-degree of S (the number of out-edges of S), and adding the number of out-edges in excess of one at each of the remaining nodes. This follows from the simple observation that additional paths are created only at nodes which have more than one out-edge.

The EG for a directed graph has the nice property that the number of edges in EG is exactly the number of edges in G. This is so because each node can only send one explorer on an out-edge, and each edge, being directed, can get an explorer only from its source node.

C. The Traversal Spanning Tree

Observe that if we remove from EG all the leaf nodes representing the termination of explorers at visited nodes and the edges directed into them, we are left with a tree in which each node of G is represented only once, and each edge is a *first* edge to its successor node. This is exactly a *spanning tree* of G. We call it a *traversed spanning tree* and for brevity, a *P-tree*, since each edge is a *P*-edge. Thus, we see that the parallel traversal method guarantees the construction of a spanning tree in which every node is visited once. The traversal execution graph not only includes a spanning tree, but also an edge-spanning tree in which each edge is traversed. Note that in a *P*-tree, all internal nodes are *P*-nodes and all leaf nodes are non-*P* nodes, since none of their out-edges in EG is a *P*-edge.

This is the main reason why echo algorithms will be seen to be a basic technique for distributed systems. It uses a method of constructing a spanning tree in parallel, with communication time just twice the diameter of the graph. It is, furthermore, a method in which, regardless of the exact sequencing of the messages, the total number of message passes (a measure of overall work), is constant for a given graph, and can be precomputed.

In a computer network, it may be argued that once a minimum spanning tree has been found, it is the fastest way to broadcast a message to all nodes. However, because of the variability of communication delays, any predetermined span-

ning tree may not, in fact, represent the fastest current set of paths which reach all nodes. On the other hand, a pure traversal echo algorithm always takes the minimum amount of time to span the entire graph, and thus, in general, may be expected to perform slightly better than any given minimum spanning tree. This does not presume that the echoes for the pure traversal also take the minimum time to return. However, we note that an echo algorithm requires $2e$ messages, while a minimum spanning tree traversal only needs n messages once it has been established.

V. SPECIFIC ECHO ALGORITHMS

The basic traversal algorithm can be modified to yield a large number of decentralized parallel graph algorithms. In this section, we will present three echo algorithms. The first one is a simple Single-Source algorithm in which a particular node initiates the algorithm, which executes in parallel, the required answer finally being obtained by the initiator node. The task to be performed is to get the identities of all the processors in the system in sorted form (assuming a total ordering on identities).

The second is a simple Multiple-Source echo algorithm. Multiple-Source algorithms are designed to take into account the situation in which several nodes may initiate an activity which has a common global end. In this case, all nodes which initiate this algorithm within a certain functional time bound participates in the algorithm, which produces a distributed ordering by identity (or priority) of the participants. By this, we mean that each node becomes aware of the identities of its predecessor and successor in the ordering, with the highest and lowest nodes being aware of their distinctive rank.

The third is a nontrivial Single-Source algorithm for finding the biconnected components of an undirected and connected graph. The answer will be found in a member of each biconnected component, and is thus in distributed form.

A. Algorithm 1—Single-Source Sort

In order for a new node to be added to a distributed system, it is necessary for it to obtain a unique identity. Assume that the set of all possible identities has a total ordering. Then a node can simply take an identity currently unused or larger than the current largest one. Although the new processor cannot itself participate in the system, we can postulate a mechanism by which it asks its nearest processor in the network to obtain an identity for it. To avoid duplicates, this mechanism should only be used sequentially in a global sense. We have previously [3], [4] indicated how a decentralized mutual exclusion mechanism might be implemented and assume its existence for the present. Once a node is allowed to proceed, it need only obtain a sorted list of all nodes presently in the network, find an unassigned number, and initiate the new node into the network. The algorithm which follows describes how such a sorted list is easily obtained. It applies to C-graphs and S-reach graphs of directed graphs. Each echo needs to be able to carrying the names of all the nodes. Every initial echo carries the name of its origin. Explorers only need basic identification, i.e., the name of the initiator, and a type.

Each node holds a *current list* initially containing only its own name.

1) Let the initiator S start the forward phase of a pure traversal by sending explorers in parallel on its edges.

2) An explorer coming to a node for the *first* time marks the edge as *first*, and sends explorers in parallel on the other edges of the node. If the node is a sink, the explorer terminates, and an *initial* echo is sent instead on the *first* edge of the node.

3) A *subsequent* explorer at a node terminates, and an *initial* echo is sent on the edge on which the explorer arrived.

4) Each *initial* echo carries the name of its origin.

5) As an echo arrives at a node, the list of names carried by that echo is merged into the current list at the node, deleting duplicates.

6) After all echoes have arrived at a node, it sends off its echo, on its *first* edge, containing its current list.

7) If the node in 6) is the initiator, then the current list is the sorted list of all the nodes of the graph.

In considering the execution graph, clearly the algorithm is collecting a merged list of the nodes in each subtree, progressively towards the root. We are not proposing this as an improved sorting algorithm, but rather pointing out that a simple echo algorithm can perform a basic function effectively.

In terms of communication time and message passes, this is the same as a basic traversal. In terms of storage, each echo may have to carry n names, and hence each node needs at least $2n \log n$ bits to accommodate its own current list and the list carried by an arriving echo.

We can improve the algorithm by a simple modification. Let only P-nodes include their names in the sublists being constructed. Since P-nodes form a spanning tree, each node is included once and only once in any list. There is no redundancy, and the number of operations in the merges is the same as in a conventional merge-sort. This change is accomplished by having echoes which arise from explorers at visited nodes carry an empty list.

B. Algorithm 2—Multisource Sort

If a subset K of processors simultaneously requires access to a set of resources which exist in distributed form, but only one processor is to proceed at a time, a serializing mechanism will be needed. Such a mechanism cannot be placed at any specific resource because the set of required resources is distributed. Instead, the contending processors must agree among themselves as to who proceeds and in what order. One class of problems in which this situation arises is the multiple-copy update of distributed databases. A rational and simple way in which such an agreement might be reached is for the processors to enter into an ordering cycle, following which an update cycle can occur, each processor executing in order of priority. For simplicity, consider the case in which priority is equivalent to the identity of the processor. The algorithm below produces a distributed ordering of those nodes which wish to update, and are included in this cycle of updates. There are two mechanisms, one to include candidates for a particular cycle, and the other to produce the distributed ordering.

The inclusion mechanism works as follows. Each node has a *status* which is either *asleep*, *awake*, or *shutoff*. Initially, all nodes are *asleep*. Some node that spontaneously wishes to start turns itself *awake*. An explorer coming to an *awake* node considers that node included. An explorer coming to an *asleep* node turns it to *shutoff*, and clearly an explorer at a *shutoff* node considers it excluded. Note that a *shutoff* node does not participate in the node ordering, but must act to receive echos and send explorers, since it may be intermediate in the path between two *awake* nodes.

To do the ordering of nodes, each echo keeps two fields: *larger* and *smaller*. Initially, *larger* contains some largest implementation number TOP, and *smaller* some smallest implementation number BOT. Each included node performs a basic traversal, and among the included nodes it encounters, it finds the smallest node larger than itself and the largest node smaller than itself. When the algorithm is finished, the largest of the included nodes will know that fact because its *larger* remains as TOP, while the smallest of the nodes has its *smaller* value unchanged from BOT.

1) Let an *asleep* node wishing to participate turn itself *awake*, and send its explorers on all out-edges.

2) An explorer coming to an *asleep* node turns it to *shutoff*.

Note: From this point, consider only the activities for a particular family of messages, originating from a particular initiator. Thus, an explorer coming to a node on a P-edge means an explorer of *that family* visiting a node for the first time.

3) An explorer on a P-edge to a node marks the node as having been visited by that family, the *first* edge for that family, then sends out more explorers. If the node is a sink, then if it is an *awake* node, 5) is done; or else if it is a *shutoff* node, 4) is done.

4) An explorer coming to a visited *shutoff* node creates an initial echo, carrying the initiator name, with *larger* = TOP and *smaller* = BOT. The node sends the echo back.

5) An explorer coming to a visited *awake* node creates the same echo, but if the node is smaller than the initiator, it places the node name in *smaller* and if it is larger, it places it in *larger*. The node echoes.

6) As a node receives echos for a family, it constructs an echo which will contain the largest of the {*smaller*} and the smallest of the {*larger*} fields in the echos. After all the echos have arrived, an *awake* node tries to place its own name in either the *larger* or the *smaller* field of the constructed echo. Then it echoes along its *first* edge. A *shutoff* node simply echoes on its *first* edge.

7) If the node in 6) is the initiator of the echos in question, then the algorithm terminates for that initiator. The initiator that is largest among the *awake* nodes identifies itself by finding that its *larger* field still contains TOP.

This algorithm contains k parallel executions of a modified basic traversal. In thinking of echoes coming back up the subtrees of the execution graph, all nodes are involved for echoing purposes, but only *awake* nodes compete for being either the

largest of the nodes smaller than the initiator or the smallest of the larger nodes. The result of this algorithm is that a doubly linked list of the sorted nodes is found and stored in distributed form among the *larger* and *smaller* fields of the k involved nodes.

Communication time for executing n simultaneous executions of a traversal algorithm can be considered to be the same as executing one algorithm if the processors are very fast compared to communications. Thus, as messages arrive, they are handled with a negligible loss of time. The messages are sequential on any particular link, and as long as the transmission rate is larger than the message generation rate, messages will not back up at a node. Under these assumptions, the elapsed time for this algorithm is just $O(D)$, where D is the diameter of the graph, since there are k traversals, but each traversal occurs in parallel. Furthermore, the number of message passes in total is bounded by $4ke$ since a single traversal needs at most $4e$ message passes.

Finally, each node needs at most n bits to mark echo arrivals for each family in the worst case of a fully connected graph. In addition, for each family, each node needs to maintain an echo while it is being constructed. This is the name of three nodes and only needs $(3 \log n)$ bits. Therefore, storage is bounded by $n + 3 \log n$ bits at a node for each family. For the entire algorithm, then, we need $nk + 3k \log n$ bits. Since k is, at most, n, the storage needs are of $O(n^2)$.

If we were to use this technique for the multiple copy update problem, the largest contender would start by sending an update using parallel traversal to all nodes. We would adopt Ellis' technique [7] of using two types of update messages, an *update* and an *update final*. When all echoes for an update traversal have returned to the initiator of the update, it transfers control to the node described in its *smaller* field. If it is the last node to update, then its *smaller* contains the implementation number BOT, and the node sends out *update final* messages. All other nodes which initiate an update use *update* messages. Nodes which echo following the execution of *update final* messages will reset their states to *asleep*, so that another set of updates can be performed.

C. Biconnected Components of a Graph

For a connected undirected graph G, its biconnected components [1] are those subsets of nodes and edges which share a common cycle. There may be edges in G which belong to no cycles, and these are considered to be a biconnected component of two members. All sink nodes, therefore, are in biconnected components of two members.

Equivalently, biconnected components contain no internal articulation points where "a vertex a is said to be an *articulation point* of G if there exist vertices v and w such that v, w, and a are distinct, and every path between v and w contains the vertex a" [1]. However, the biconnected components themselves must be connected through articulation points of the graph G; or else they would form a single biconnected component. Thus, although a biconnected component does not contain any internal articulation points, one or more of its nodes may be articulation points of the whole graph G. Articulation points are critical in the reliable functioning of a

network in that a node failure of an articulation point produces a disjoint graph. Thus, it is important from time to time for a network to discover its articulation points and the members of its biconnected components.

This Single-Source algorithm is based on the following observation: if G contains articulation points, then for any biconnected component of G with nodes $\{a\}$, there exists a node v in $\{a\}$ which is an articulation point of G (or is the initiator node) such that v is the root of a subtree in EG in which all the nodes of $\{a\}$ are internal nodes. Call this subtree $<v>$. Then the explorers in $<v>$ terminate only at the nodes of $\{a\}$. At node v, therefore, all the leaf nodes of $<v>$ are accounted for by its internal nodes. By using echoes to bring information concerning the internal and termination nodes of lower subtrees in $<v>$, it is possible at v to find that all leaf nodes are accounted for by the internal nodes of $<v>$. At this point, all of $\{a\}$ has been identified as members of one biconnected component. For this iterative process to function properly, biconnected components, as detected, reduce themselves to single nodes. Thus, the sink nodes in V' are the first biconnected components to be identified and reduced. The algorithm finishes when the initiator node receives all its echoes and does a final echo-merge. At termination, each articulation point of the graph G knows the biconnected components of which it is a member. If necessary, the initiator node can gather this information by sending appropriate requests which trigger responses only at articulation nodes.

1) Termination and Intermediate Sets: Echoes are used to describe their subgraphs as they move up the tree EG. An echo will consist of basic identity, *status*, and two sets, *INT* and *TER*. The set *TER* consists of the terminal nodes in the subtree described by an echo still unaccounted for by the nodes of *INT*, the set of internal nodes in the same subtree. These will be referred to collectively as the *pair* of the echo (INT,TER). *Status* describes whether the termination of the explorer for an *initial* echo was at a sink node or a visited node. An internal node v of EG which receives an initial echo from node w with status *sink* will identify itself as an articulation point for the biconnected component $\{v,w\}$, and replace the biconnected component by the equivalent node $\{v\}$. Thus, the description of subtrees by echoes is simplified as echoes move up the tree if biconnected components are present.

A node v receiving more than one echo with termination status *visit* looks for a common cycle by trying to account for all the terminal nodes of any subset of its echoes by the internal nodes of the same subset. If this condition can be found, then a biconnected component has been identified.

Proposition 3: If an echo has a status of *visit*, then there is another echo that also has a status of *visit*, which shares at least one common ancestor node in the execution graph. Moreover, for the two echoes, there is a last common ancestor at which the unaccounted termination nodes of each echo will be in the set of internal nodes of the other.

Argument: We know for a C-graph, by Property 4), that explorers stop in pairs on an edge if the termination condition is *visit*. By definition, all explorers for a Single-Source algorithm originate from the initiator node, and trivially, they have a common ancestor.

If two explorers stop on the same edge mutually, then if the edge is (v,w), and the explorer stopping at w took some path p from the initiator, then the explorer stopping at v took some path q. The path p has as its last two nodes vw, and q has as its last two nodes wv. Thus, the termination of path p is an internal node of path q and vice versa.

Two explorers stopping mutually must have taken different paths. We already know they must have at least one common ancestor, the initiator node. However, if they have more than one common node on their paths, there must be a last such common node before their paths diverge. This last node is the root of a subtree in EG, and the echos which it receives will describe the divergent subpaths and their terminations. □

Corollary: By Proposition 3, explorers which share a common path from the initiator but diverge at a last common node v will produce echoes which converge at node v to describe the subtrees traversed by the divergent explorers. Node v, the last common node, is therefore the *first* node which can detect whether the explorers have traversed the nodes of a biconnected component of which v is an articulation point of G. This justifies looking for biconnected components as echoes move *up* the subtree.

Now we can describe how echoes are merged at a node, which we will call Echo-Merge rules.

1) An initial echo whose origin is a sink node w is created with $INT = \{\phi\}$ and $TER = \{w\}$ and status *sink*. Then if v receives an echo from w with status *sink*, $\{v,w\}$ is a biconnected component, and v marks the edge (v,w) *inactive*.

2) Now consider the set of echos with status *visit* which come to v. Each echo carries the pair (INT, TER). We successively perform set intersection of each TER with INT from the other echoes. If all these interesections are *NULL*, then we produce a new pair (INT, TER) from the union of all the INT's and the union of all TER's. A new echo is then formed, with the new pair and a status of *visit*, and sent along the *first* edge of v. It represents all intermediate and terminal nodes seen in the subtree of v, and furthermore, it describes the property that no cycles were found, to date. Each intersection represents cycle detection.

3) If any $TER \cap INT \neq \phi$, then form a new pair (INT, TER) from two echoes by doing unions on TER and INT. Replace the two pairs by this new pair, and continue 3) until the condition in 2) is found: the intersections of all *pairs* are *NULL*. Rule 3) is called *maximal cycle detection*.

4) Now take each such pair (INT, TER) from 3), and if any element of TER is a member of its corresponding INT set, remove it from TER. At the end, if TER is empty, then we have found a biconnected component whose members are $\{v\} \cup INT$. By Proposition 3, if v is the articulation point of a biconnected component, it is also the first node at which it is possible to identify the member of the biconnected components. We now mark *inactive* all edges whose echoes have gone to make up this pair, save $\{v\} \cup INT$ as a biconnected component at v, and remove this pair from the pairs of (INT, TER) being considered. The process of eliminating members of TER is called *bicon identification*.

5) After 4) has been applied to all pairs, either there are no pairs left, in which case v is considered a sink, and an echo

with status *sink* and INT of $\{\phi\}$ and TER of $\{v\}$ is sent along the *first* edge of v. If there are pairs left, then a new echo is formed, as described in 2). The combined process of maximal cycle detection, and then bicon identification, is called *bicon composition*.

Proposition 4: The Echo-Merge rules allow a biconnected component to be identified at one of its member nodes which corresponds to the root of a subtree in an execution graph EG.

Argument: By Rule 1, biconnected components of two nodes are found as soon as an echo with status sink is received at a node v, which is a member of the biconnected component $\{v, sink \text{ node}\}$. Furthermore, the *sink* node is a leaf node in EG and v is an internal node.

Now consider a biconnected component M of at least three nodes, one of which is v, with v being an articulation point of G. Then all explorers going into M have come through v, and v is the common ancestor of all the echoes in M. Whatever paths are taken from v into M, they must end in nodes which are in M. Therefore, by Proposition 3, in the echoes which come back to v, the set of terminal nodes must all find matching internal nodes.

Consider a node at which bicon composition yields the pair (INT, TER) with TER being empty, called the *zero pair*. We show that the nodes in INT must be a biconnected component. Maximal cycle detection assures us that terminal nodes of the paths taken from v end in intermediate nodes of other paths, and therefore that these nodes are linked. Furthermore, there is no path which terminates at a node outside of the nodes in INT.

The only way in which the nodes and edges of INT do not form a biconnected component is if there is some path from one of the nodes in INT which leads to a node outside of INT, thus making the INT part of a larger biconnected component. This cannot happen without this edge terminating at some node x outside of INT. If so, then bicon identification would have produced TER containing x, not empty. Thus, there is no such edge leading out of the nodes of INT.

Note that for v to be the articulation point of a biconnected component in the graph G, there must be at least two echoes returning to v. Otherwise, it would not be the root of divergent paths leading into the biconnected component, and some other node, either further up or down the tree, would identify the biconnected component. □

Algorithm 3–Biconnected Component Detection

The algorithm for finding the biconnected components of a C-graph has been largely described above by the rules for bicon composition. Given an initiator node S, let it execute the forward phase of a pure traversal by sending explorers in parallel on the *nonfirst* edges of nodes as they are first visited.

1) Let the initiator send explorers in parallel on its edges. The *first* explorer at each node causes more explorers to be sent in parallel from that node. An explorer coming to a visited node v sends an echo back with the pair (INT, TER) of $(\{\phi\}, \{v\})$ and status of *visit*. If the node is a sink, the same (INT, TER) pair will be sent, but the status of the echo will be *sink*.

2) If a node v receives an echo with status *sink* having (INT, TER) of $(\{\phi\}, \{w\})$, then $\{v,w\}$ is a biconnected component,

and the edge on which the echo arrived is marked *inactive*. Now try 3).

3) If all the echoes for a node v have arrived and all edges have been marked *inactive*, then create a new echo with status *sink* and (*INT,TER*) of ({ϕ}, {v}), and send the echo on the *first* edge of the node.

4) Otherwise do bicon composition according to Echo-Merge rules 2)–4), identify all biconnected components, and mark appropriate edges as *inactive*. If all edges are now *inactive*, then do 3); or else send the echo built by Echo-Merge rule 2) along the *first* edge of the node.

5) If the node in 3) and 4) is the initiator, then the algorithm is done. A check for correctness here is that all edges should be *inactive*, and there should be no terminal nodes still unaccounted.

We note that Algorithm 3 detects biconnected components at nodes which are articulation points of G and at the initiator node (which may not be an articulation point). Even so, the rules for Echo-Merge which identify biconnected components apply equally. Essentially, the roots of the smallest subtrees which contain all the nodes of a biconnected component as internal nodes are the nodes at which biconnected components can be identified.

Behavior: We must now consider the behavior of the algorithm. Given that there is a single initiator node, the algorithm is very similar to the pure traversal algorithm in that explorers make one forward sweep and echos make one sweep back towards the initiator. Thus, elapsed time, mainly considering communications, is approximately $2D$ where D is the diameter of the graph.

The number of messages is, as with a pure traversal algorithm, between $2e$ and $4e$. The major component of effort in this algorithm comes at each Echo-Merge and in the amount of storage required at each node.

Each echo carries the sets *INT* and *TER*. At the worst, the set *INT* can contain n nodes, and the same can be said of the set *TER*. The fact that at any time, the sum of the nodes held in *TER* and *INT* can be no more than $2n$ helps in considering overall work, but not in the amount of space that must be allocated for the worst case. If each set of *INT,TER* must contain n nodes, then $n \log n$ bits are needed for each set. For a node to hold n echoes, then at least $2n^2 \log n$ bits are needed for all echoes to describe the pairs (*INT,TER*). Of course, if we know the value of n *a priori*, then a bit map would suffice for storing the sets, which brings the requirement down to n^2.

Now consider the number of operations at each bicon composition. Maximal cycle detection involves the intersection of each *TER* against a set *INT* from a different echo. There can be at most n echoes at each node. If each *TER* is intersected against $n - 1$ *INT* sets, then there are n^2 intersections. At best, each set operation can be considered as one operation. Thus, there are at least n^2 operations at each node for cycle detection.

Bicon detection, given the set of disjoint pairs found by maximal cycle detection, consists of removing elements of *TER* which are in its corresponding *INT* set. There are at most

$n/2$ new pairs, and each such removal can be done in at most n operations. The operations at each node are dominated by maximal cycle detection, which is basically n^2.

These bounds by no means reflect an *average* situation. For one thing, if there are only sinks, then maximal cycle detection is never needed. In the case of a fully connected graph, each path is, on the average, two long. The expected number of set operations at each noninitiator node, which receives $n - 2$ echoes, is just one since each echo has an *INT* of {ϕ}, and hence does not have to participate in any set intersection. At the initiator node, $n - 1$ echoes arrive, but each intersection of an *INT* with a *TER* from a different echo will find a matching element. Thus, only $n - 2$ set operations are needed. In comparison, if all *INT* and *TER* are disjoint, then $(n - 1)^2$ intersections would have been needed. We leave the study of this aspect to future research.

VI. Discussion

In this paper, we have presented a model of a distributed computer system in which autonomous processors can cooperate without using centralized mechanisms. The need to obtain global information motivated the creation of algorithms which, although decentralized in control, nevertheless achieve a system-wide goal. These algorithms are based on the technique of parallel depth-traversal of a general graph such that a minimum edge covering is obtained in an asynchronous fashion. As a class, we have called them *echo* algorithms.
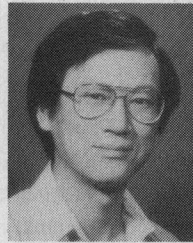
The general technique which we have introduced has been shown to be simple, versatile, and efficient. Echo algorithms function in parallel, asynchronously, and take execution time for communications of the order of the diameter of the graph modeling the network of machines. Although echo algorithms operate strictly on message passing, the number of messages, in general, in bounded by $4e$ where e is the number of edges in the graph. The storage required at a node, for most algorithms, is small and bounded by n^2 bits where n is the number of nodes in the graph. Echo algorithms can be single-source, originating and terminating at a specific node, or multiple-source, in which several nodes in the network coordinate their activity in a consistent way, although they independently initiate their participation in the algorithm.

Further research which leads naturally from the material presented includes the effect of failures on echo algorithms, the relationship of heterogeneous processors of mixed capabilities on their execution, the formal verification of decentralized algorithms such as echo algorithms, and the better quantification of performance by including the effects of queueing and congestion in the network. This paper represents only a first step in the application of a new class of algorithms to the better understanding of decentralized systems.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.
[2] E. R. Arjomandi and D. G. Corneil, "Parallel computations in graph theory," *SIAM J. Comput.*, vol. 7, May 1978.

[3] E.J.H. Chang and R. Roberts, "An improved algorithm for decentralized extrema finding in circular configurations of processes," *Commun. Ass. Comput. Mach.*, vol. 22, May 1979.

[4] E.J.H. Chang, "Decentralized algorithms in distributed systems," Ph.D. dissertation, Univ. Toronto, Toronto, Ont., Canada, 1979.

[5] E.J.H. Chang and K. Sevcik, "Decentralized algorithms for the multiple-copy update problem," Univ. Victoria, Victoria, B.C., Canada, Tech. Rep. 1979.

[6] D. M. Eckstein and D. A. Alton, "Parallel graph processing using depth-first search," in *Proc. Conf. Theoretical Comput. Sci.*, Univ. Waterloo, Waterloo, Ont., Canada, Aug. 1977.

[7] C. A. Ellis, "A robust algorithm for updating duplicate databases," in *Proc. 2nd Berkeley Workshop on Distributed Data Management and Computer Networks*, Univ. California, Berkeley, May 1977.

[8] P. Rosenstiehl *et al.*, "Intelligent graphs: Networks of finite automata capable of solving graph problems," in *Graph Theory and Computing*, R. Read, Ed. New York: Academic, 1972.

[9] R. E. Tarjan, "Depth first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, 1972.

Ernest J. H. Chang was born in Shanghai, China. He received the B.Sc. degree from the University of Manitoba, Winnipeg, Man., Canada, in 1964, the M.D. degree from the University of British Columbia, Vancouver, B.C., Canada, in 1970, the M.Math. degree in computer science from the University of Waterloo, Waterloo, Ont., Canada, in 1974, and the Ph.D. degree in computer science from the University of Toronto, Toronto, Ont., Canada, in 1979.

He interned in Kaiser Foundation Hospitals, San Francisco, CA, and worked with Medical Methods Research, Kaiser Foundation, Oakland, CA, as a Research Fellow in 1970. He was a Research Assistant Professor in Computer Science and a National Health Scholar from 1974–1976. Since 1979, he has been an Assistant Professor at the Department of Computer Science, University of Victoria, Victoria, B.C., Canada. His research interests include decentralized algorithms, computer graphics in videotex systems, computer-assisted instruction, and medical informatics.

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

Load Balancing in Distributed Systems

TIMOTHY C. K. CHOU AND JACOB A. ABRAHAM, MEMBER, IEEE

Abstract—In a distributed computing system made up of different types of processors each processor in the system may have different performance and reliability characteristics. In order to take advantage of this diversity of processing power, a modular distributed program should have its modules assigned in such a way that the applicable system performance index, such as execution time or cost, is optimized. This paper describes an algorithm for making an optimal module to processor assignment for a given performance criteria. We first propose a computational model to characterize distributed programs, consisting of tasks and an operational precedence relationship. This model allows us to describe probabilistic branching as well as concurrent execution in a distributed program. The computational model along with a set of seven program descriptors completely specifies a model for dynamic execution of a program on a distributed system. The optimal task to processor assignment is found by an algorithm based on results in Markov decision theory. The algorithm given in this paper is completely general and applicable to *N*-processor systems.

Index Terms—Computer networks, distributed processing, optimal scheduling, performance analysis.

I. INTRODUCTION

CURRENTLY, the field of distributed processing is the focus of a great deal of research interest [15], [16], [14], [4]. This is primarily due to the availability of inexpen-

sive computer hardware, growing sophistication in distributed software, and an increased interest in improving system performance and reliability. In this paper we define a *distributed computer system* to be any computer system with two or more arbitrarily interconnected processors. In particular, we are interested in loosely coupled systems [7], where the communication time between processors is an important system parameter. Systems such as CM* [17], ICOPS [19], DCS [5], and those based on IBM's SNA [12] and DEC's DECNET [3] are all good examples of the type of system we are considering. The results of this paper are also applicable to avionics system design and distributed process control where dedicated processors of various types are used.

Furthermore, we define a *distributed program* as a program that consists of several program modules or tasks that are free to reside on any processor in a distributed system. In general, we are considering a heterogeneous processor system in which each processor may have different performance and reliability characteristics. In order to fully utilize this diversity of processing power it is advantageous to assign the program modules of a distributed program to the processors in such a way that the execution time of the entire program is minimized. This assignment of tasks to processors to maximize performance is commonly called *load balancing*.

We are basically studying a problem closely related to optimal deterministic task scheduling. Therefore, we should be aware that results from deterministic scheduling are valid in basically two cases: first, if we are interested in only a rough approximation of system performance such as a lower bound