Distributed algorithms for global structuring*

by RAPHAEL A. FINKEL and MARVIN SOLOMON

University of Wisconsin
Madison, Wisconsin

and

MICHAEL L. HOROWITZ
Carnegie-Mellon University
Pittsburgh, Pennsylvania

INTRODUCTION

In the search for speed and computing power, many researchers in computer science have turned to networks of computers as a possible solution.1,5,8,14,9 These networks consist of minicomputers connected by links across which communication between processors occurs. In homogeneous networks, the computer at each node is identical to the others, with the possible exception of peripherals. Each processor has its own local memory, does not share memory with any other processor, and communicates with other processors via message passing. In order to fully utilize the speed and power inherent in a network, emphasis must be placed on the development of parallel (as opposed to sequential) algorithms.

In this paper, we will investigate concurrent algorithms designed to impose a logical structure on top of the physical computer network, such as a pairing of the processors along communication lines. These algorithms are interesting not only for their relationship to parallel algorithms in general, but also because the resulting structure may be used as a basis for writing other parallel algorithms. We restrict our attention to algorithms that have the following properties:

1. Initially, each processor knows only of its neighboring processors in the network.
2. All processors have the same program to execute.
3. Messages sent between neighbors may take an arbitrarily long amount of time to arrive.
4. Messages between any two connected processors will always arrive in order.
5. No assumptions are made about the physical interconnection pattern except that it is connected.

The algorithms we will discuss form three types of structures on the network. These three problems have been chosen because their results are dependent upon the physical network configuration and because they apply to the solution of other network problems. Pairing algorithms match each node of the network with a direct neighbor. Since a pairing is not always possible, as in the case of an odd number of nodes, a good solution should leave a minimum number of "single" processors at the end of the algorithm. Spanning tree algorithms impose a tree on the network so that a unique path exists between any two processors in the network. The last problem considered is that of forming hierarchies of processors. One step in developing a hierarchy is the formation of processor cliques. Each clique should be of a certain size and one processor in each clique is designated as the "leader." A good solution should try to minimize the average radius of each clique.

The assumptions made earlier about the behavior of messages and about the program each processor runs give rise to several problems. The problem of agreement is that each processor must make consistent decisions with regard to the rest of the network, possibly based on widely differing local information. (For example, in the pairing algorithm, two nodes should not simultaneously believe they are paired to the same third node. In particular, the final state of the computation must not exhibit this behavior.) Each processor makes independent decisions about its future role in the resulting structure. Therefore, any decisions a processor makes that might affect its neighbors' decisions must eventually be communicated to those neighboring processors.

Synchronization is the problem each processor encounters when it is about to finish a phase of the algorithm. Since information about the rest of the network is usually incomplete at each node, the algorithm must be designed so that each processor can make its own decisions based on little information. (For instance, in the pairing algorithm, a processor should not decide to halt its active participation and assume that a neighbor is its mate unless it knows that the neighbor will agree.) In particular, it is hard to decide whether a current local state is final or not. In part, this problem results from the assumption that messages between processors may take a long time to arrive. A processor cannot stop participating unless it knows that no later mes-

* This research was supported in part by United States Army under contract DAAG29-75-C-0024.
message can force a change in its state. Although messages never get lost, they may arrive quite late, and provisions for handling or preventing these messages must be made.

The above two problems just mentioned have an easy solution if we allow ourselves the luxury of a central controlling processor. Such a central control would, however, become a bottleneck as the size of the network grows. We feel that more general solutions can be derived by avoiding this central control. Certainly, any known methods for sequential solution of these problems could be carried out by the central control.

In developing these algorithms, we have totally ignored low-level aspects of message passing. Low-level protocols and the problems of lost or garbled messages are the responsibility of the underlying network implementation. Some research has already been invested into these problems. It is our intent, however, to study the fundamental problems of parallel algorithms themselves.

We have examined the performance of these algorithms by simulating a network in order to obtain sample results. Different network configurations were used, and comparative performance between different configurations varied, but the relative performance of each algorithm remained the same for each configuration. The configuration used most often was a square grid of varying sizes. We schedule the time of the receipt of a message on a time line, and assume that the computation at each node takes negligible time. The time for a message to be delivered is selected from an exponential distribution with mean \(\lambda\). Truncated to lie between 0.50 and 1.50 time units. In this way, many actions may occur "simultaneously" in simulation. The programs were written in Pascal and executed on a PDP 11/40 and a PDP 11/45.

Each of the following algorithms operates in the following manner: Upon receipt of a message, a processor executes a program segment that depends upon the processor's state and the message received. In this program segment, the processor may save whatever information it wants from the message, send out new messages, and change its internal state.

PAIRING ALGORITHMS

To achieve a pairing, each processor must agree either to become paired with a neighbor or to become single. A correct solution is one in which a paired processor and its mate agree that they are paired, and no neighboring processors are both single. An optimal solution is one in which there are a minimum number of single processors, which may require some analysis of the network configuration to derive.

We will start with a simple algorithm, called Algorithm A, and then suggest improvements. The basic pairing algorithm has four states. A processor in the idle state has not yet started the algorithm. When it is waiting, a processor expects a reply from its chosen mate. Paired means the processor considers itself paired, and single means it considers itself single. There are five messages: awake, which starts an idle processor; query, which indicates that the sender wishes to pair with the recipient; agree, which tells a waiting processor that the sender agrees to become paired with it; disagree, which indicates that a processor's chosen mate is itself awaiting an answer from its own intended mate; and refuse, which indicates that a processor's intended mate is already paired with another processor.

At the start, each processor is idle, and one awake message has been sent to it. (It is not important to this discussion how awake messages are generated.) Initially, all direct neighbors are potential mates. If the processor receives the awake message in its idle state, it chooses a random neighbor from its list of potential mates as its intended mate, sends it a query and changes state to waiting. This query may reach the neighbor before any awake message. In this case, the recipient chooses to become paired with the sender and returns an agree message. This case, when the recipient of a query is in its idle state, is the only one in which an agree message is sent. Since agree, disagree and refuse messages are sent out only in response to a query, and queries are not sent out by idle processors, we only have to consider these two cases when the processor is idle.

If a processor is in its waiting state, many different actions may occur. Any processor in the waiting state must already have seen the one awake message directed to it. If a query is received, a disagree is sent back if the query is not from the intended mate; otherwise, the processor becomes paired. No agree need be sent in the latter case; since the sender of the message is this processor's intended mate, a query was sent to it and the mate will take the same action. If an agree is received from the intended mate, then the processor also becomes paired. If a disagree is received from the mate, then the processor chooses a new intended mate, sends it a query and remains in its waiting state. The new intended mate may be the same as the first one. If the processor receives a refuse from its chosen mate, it removes the sender from its list of potential mates, chooses a new intended mate and sends it a query. If there are no more potential mates, the processor becomes single. A single processor should not receive any messages at all, since no query is outstanding and all neighbors have refused, implying that they are paired. Finally, if the processor is paired, it can receive either a query, in which case it sends back a refuse, or an awake, which it ignores.

If we examine Algorithm A, we see that once all of the processors are started, a processor and its intended mate must choose to query each other before they can become paired. Many false starts may happen before this pairing actually occurs. In an effort to reduce contention, Algorithm B begins by sending only one processor an initial awake message. If an idle processor receives an awake or query message, it sends out awake messages to all of its neighbors except the sender of the received message. Simulations show that the number of singles left remains about the same for Algorithm B as for A regardless of the network structure, and the elapsed time (simulated time, not running time) becomes progressively worse as the size of the network increases. This behavior implies that contention is not really time-consuming. Also, the time needed to activate the processors across the network from the originally started pro-
Algorithm C is a modification to Algorithm A designed to provide each processor with earlier information concerning the state of its neighbors so that it can make a better choice of intended mate. Since the refuse message indicates that the sending neighbor is already paired, queries can be forestalled by broadcasting refuse messages to all direct neighbors (except the mate and those neighbors known to be already paired) as soon as a processor becomes paired. This modification has three parts. First, whenever a processor becomes paired, it must send out refuse messages to the appropriate neighbors. Second, if a processor receives a refuse from a processor other than its intended mate, it removes the sender from its potential mate list. Finally, a paired processor that receives a query need not respond, since it has already sent out a refuse. The observed saving in elapsed time is quite dramatic, and the number of singles left also seems to decrease. The amount of information available, then, appears to make a considerable difference in performance.

Since concurrency and early information both improve the speed of the algorithm, we tried Algorithm C, activating all of the processors at the same time. The previous algorithm sent the broadcast awake message to all processors at the same time, but that message arrived according to our message-delay distribution. Again, the elapsed time performance improves, but no real conclusions can be made concerning the number of singles left. In fact, the number of singles may increase for larger networks.

Of the two, more information makes a greater improvement in the performance of the algorithm than concurrency. In order to follow this idea further, Algorithm D provides an individual processor with still more information about its neighbors. A new message, neighbor-list, is introduced. This message contains the size of the potential mate list of the sender. These neighbor-list messages are sent out to all potential mates upon receipt of a refuse message, which changes this number. When a processor receives a neighbor-list message, it enters the new data in a table. When it comes time to choose a mate, each processor picks a random neighbor from among those with the fewest potential mates. One expects that the number of singles will decrease, because better choices can be made. For the smaller networks, the new approach seems to make no difference, but for the larger networks, significantly fewer singles result. Algorithm D also performs faster than Algorithm C for all network sizes.

Perhaps we could do better with a different choice of a neighbor to query. Instead of choosing a neighbor with the fewest potential mates, Algorithm E chooses a neighbor with the most. The results justify the intuition that choosing the neighbor with the fewest potential mates leaves fewer singles. Surprisingly, however, Algorithm E is still an improvement over Algorithm C, in which an intended mate is randomly chosen, both in elapsed time and number of singles. Again, increased information creates an algorithm that performs better.

Since none of these algorithms guarantees an optimal solution (that is, one in which the number of singles is minimal), we introduce a second phase to the basic algorithm during which singles migrate, eventually to meet and pair. All the previous algorithms have the property that each processor knows when its active participation in the algorithm is over. In the subsequent pairing algorithms, termination is not locally discernable.

Algorithm F introduces a break message that is sent out by a single to a randomly chosen neighbor. This neighbor, if still paired, will then send back an agree, send an eloped message to its old mate to indicate that they are no longer paired, and become paired with the single that sent the break message. The recipient of a break message may be waiting or single, though, since processors can now become paired and unpaired an arbitrary number of times. If the recipient is waiting or single, and its intended mate is not the sender of the break message, it then sends back a disagree. Otherwise, the recipient becomes paired, much as in the previous case when two processors send queries to each other. If a disagree message is received by a single processor, then the sender of the disagree is added to the potential mate list, the processor enters the waiting state and a new query is sent out. When a single processor receives an agree, query or break message from its intended mate, it becomes paired. Eloped messages are ignored if they do not come from one’s mate, since the message can be late, but must be handled otherwise. They are treated as a refuse in response to a query, which means that the receiving processor must now become single or waiting, depending upon the state of its potential mate list. An appropriate message (break or query, respectively) is also sent. The algorithm has halted in simulation when all of the processors have become paired, or the minimum number of singles is left for the given network.

The simulated time to execute phase two is much higher than that for phase one, and varies widely in different test runs. Of course, this phase allows little concurrency, since only those processors near singles are involved. Moreover, networks that cannot be completely paired fare better during phase two than networks of similar size that can be paired, since more singles are migrating throughout the network. We tried Algorithm F in conjunction with both Algorithms A and C, that is, with refusals both broadcast and not broadcast. The results were a bit puzzling at first—Phase Two time increases when refusals are broadcast. The cause is probably that the potential mate list becomes more of a hindrance than an aid in the second phase, since it restricts the choices a processor can query before it must become single. In addition, the average path length, or number of break messages per single at the end of Phase One, also increases when refusals are broadcast.

In an effort to reduce the time spent by singles that migrate randomly throughout the network, Algorithm G includes a homing signal for the singles so that they might find each other more easily and directly. Whenever a processor becomes single, instead of immediately breaking a random neighbor’s pairing, it broadcasts a homing signal containing some random number. This message is passed on by every processor until some other single receives the signal. This
single will then break the pairing of the neighbor who relayed the signal. In this way, one hopes that single processors will meet and pair much sooner, since they can move toward each other directly. One also hopes that the signal broadcast messages will not seriously affect performance.

Unfortunately, the phase two performance of Algorithm G turns out to be seriously worse than random migration. First, broadcast messages take up some time, since every processor must receive each signal twice before the message is squelched. More importantly, breaking the pairing of a neighbor in the direction of another single does not necessarily place the new single any closer to the originator of the signal. In a square matrix configuration, for example, there is a two-out-of-three chance that the new single would be just as far away, as shown in Figure 1. Only a better understanding of the configuration of the network at each processor will allow a more intelligent break message to be sent after a processor becomes single.

The pairing algorithm is an excellent medium for studying parallel structure-producing algorithms. We discovered that concurrency, even if it does increase conflict among the processors, decreases the elapsed time of the algorithm without seriously affecting other aspects of performance. Also, gathering greater amounts of information leads to faster and more accurate results. Finally, the network configuration cannot be totally ignored in the development of an algorithm: an algorithm that ignores configuration information will do worse than one which incorporates the data into its "solution."

As a passing note, we also tried our algorithms on different connection patterns for the network. The algorithms perform consistently better on square matrix configurations than on any of the "flake" configurations of similar sizes with respect to the number of singles. (See Reference 3 for a definition of "flake" network configurations.)

![Diagram of network configurations](image)

**SPANNING TREE ALGORITHMS**

A spanning tree is a subset of the physical links of the network such that there exists a unique path along the selected links from any given processor to any other processor in the network. An algorithm to produce a spanning tree should select appropriate links in such a way that each node agrees with its direct neighbors as to which of the connecting links are in the tree. A useful algorithm by-product is a routing table at each node associating destinations with locally-selected links. We will examine two algorithms for this problem.

In Algorithm H, each processor can be in one of three states—idle, working or done. Awake and known-nodes are the only messages. Awake starts an idle receiving processor. Each known-nodes message contains a list of those nodes the sender knows it can reach through selected links other than the one on which the message is sent. Each processor associates each direct neighbor with those nodes it thinks that neighbor can reach, according to the most recent information passed to it via known-nodes messages. A list of selected links is also kept.

Upon starting, each processor tells its direct neighbors that it can reach itself. When a processor receives a known-nodes message, if the connecting link is currently selected, and nodes newly reachable through it could be reached through other links in the spanning tree, then the link to the sending neighbor is de-selected. If the newly reachable nodes do not conflict with any current information, then the link to the sending processor remains in the tree. On the other hand, if the connecting link is not currently selected, and if the reachability set the neighbor has sent does not conflict with the current set of reachable nodes, then the connecting link is selected. Whenever a known-nodes message arrives, it may show that some links have been removed from the spanning tree, so the receiving processor also checks if any of the other connecting links can now be selected. Finally, the new information about its status of reachable nodes is sent to all of the receiving processor's direct neighbors, unless no new information is to be reported (the last message sent to each neighbor is saved to facilitate this decision). A processor can determine when its role in the algorithm is finished when it chooses not to send any messages to its neighbors, and all nodes in the network are reachable through selected links. In order to perform this test, each processor must know the names (but not the locations) of all nodes in the network, which can be supplied in the awake message.

It is easy to prove that this method will produce a correct solution. First, if a link is selected by a given processor when it has terminated, then the set of nodes reachable through that link is the complement of the nodes reachable through the other selected links, by the termination condition. If the neighbor on the other side of the link has not selected the link, and has also finished, then it must think it can reach the first processor through another selected link. If this is the case, it would have told the first processor so. The first processor could not then have selected that
link, because it causes a conflict (a processor can always reach itself without going through any links). So the neighbor must also have selected that link. Second, the sets of nodes reachable through each of the selected links will be disjoint, so any path along the tree will be unique. Finally, all of the processors are in the tree when the algorithm halts. A major drawback to Algorithm H, however, is that the computing time required for each action and the amount of space required to maintain the appropriate information at each node are both large.

A second method, Algorithm I, sacrifices some concurrency for ease of computation. The idea is based on the pairing algorithm. During the algorithm, every processor belongs to a partial spanning tree, and each tree is controlled by one of the member processors. Partial trees merge using a version of the pairing algorithm, with the controlling processors representing the trees. At the start, each processor starts as its own spanning tree. Partial trees are built by repeatedly asking other processors to join. Each controlling processor keeps track of which nodes are in its tree, and asks only those nodes not in the tree. If the controlling processors of both trees agree (i.e., the "trees" have queried each other as in the pairing algorithm), a combined tree is formed, and one of the two controlling processors is chosen as the new controller. Each controlling processor has an associated random number; whichever has the higher one becomes the controller for the combined tree. Processors that no longer control a tree relay messages to their controlling processor. As soon as one transaction is complete, the surviving controller initiates the next one, until the set of neighbors not in the current tree is empty. This algorithm does not require any advance knowledge of which processors are in the network, since the set of tree neighbors can be calculated from the set of nodes in the tree and the set of neighbors of nodes in the tree. When the set of nodes neighboring those that are in the tree is a subset of the nodes in the tree, the entire tree has been formed.

Algorithm I can be sped up in two ways, resulting in Algorithm J. First, when one controlling processor becomes chosen by another, it informs all of the nodes in its tree which processor is now controlling the tree, so that messages can be relayed directly. The second idea provides a major improvement, because it increases the concurrency of transactions. When a controlling processor, \( \alpha \), receives a query from another controller, \( \beta \), that has a smaller associated random number, and \( \alpha \) is also waiting for an answer from a third tree, \( \gamma \), then \( \alpha \) sends an acceptance, not a disagreement, to \( \beta \). Processor \( \beta \) is destined to lose its position as controller in any case. Update messages are now required, however, since the controller \( \gamma \) may be the surviving controller of the transaction between \( \alpha \) and \( \gamma \). That controller will not be aware immediately that other nodes have joined \( \alpha \)'s tree in the interim, since such information is normally passed along with the queries and \( \gamma \) may therefore generate an indirect query to itself, delayed by some relays. Update messages enable this processor to rectify such a situation.

These two improvements together decrease the elapsed time of Algorithm J until it is comparable to the time of Algorithm H. In addition, it seems that the simulated time increases more slowly with the size of the network for Algorithm J than for Algorithm H.

It is obvious, however, that Algorithm H is much cleaner and that it derives its speed from its inherent concurrency. In Algorithm J, all chosen processors are idle except when relaying messages, and the speed is derived primarily from the lack of conflict. If Algorithm J could be changed to distribute the decision process, then perhaps it could become a more powerful solution. On the other hand, the number of messages sent in Algorithm H grows significantly faster with the size of the network that in Algorithm J. Therefore, for larger networks, Algorithm H may prove impractical since message passing may become a bottleneck. In general, though, it is much better because it does not rely on random numbers or shortcuts to derive its speed, although a good way to terminate the algorithm may be harder to develop.

The previously-mentioned algorithms were developed solely under the assumption that more than one processor can begin within a short amount of time (much shorter than that required to complete the algorithm). If only one processor is started, then Algorithm K, a much simpler method, may be used: When a processor receives an awake or query message and it is idle, it selects the link over which the message came and responds with agreement. It then sends out its own queries along the other links. It only selects those links across which agreements are received. In order to allow each node to determine termination, disagreements can be sent to all subsequent requestors. When all neighbors are accounted for, the node is finished. This procedure is clean and simple, but requires that exactly one processor be started. Running time is directly proportional to the diameter of the network and to the maximum degree of the nodes.

HIERARCHY ALGORITHMS

The final class of algorithms investigated are those designed to form a hierarchy of processors. The major step is to form cliques of processors, with one processor chosen as the head of the clique. The hierarchy can then be built by repeating the clique-forming algorithm on the heads of the cliques formed in the previous step. Therefore, we shall only look at algorithms that form cliques. A correct solution is one in which each clique is well formed—there is exactly one head, and all of the processors in the clique agree that they are in the clique. An optimal solution is one in which the average radius, as measured from the head of the clique in message links, is minimal. A constraint is the acceptable range of sizes (number of nodes) per clique.

We will discuss two algorithms. The first assumes that all the processors are started within a short time of each other, whereas the second assumes that a central controller directs the algorithm. In Algorithm L, each processor is in one of two stages—head and chosen. Each processor begins as a
head and tries to form its own clique. It keeps asking processors not in the clique to join. Each head also keeps track of how close it is to its goal. Chosen processors are those that think that they are members of some clique. When a chosen processor receives a query from another clique's head, it relays the request to its own head. The head either denies the request or tells the processor to switch allegiance to the other head. The chosen processor then sends back a disagree or an agree, respectively, to the requesting head. If a head is itself queried, it decides either to abdicate or to disagree. If it abdicates, the head tells its chosen processors to start all over and sends an agree to the requester; otherwise it sends a disagree. The head stops asking new processors to join its clique when it feels that the clique is of the correct size. In order to minimize the average radius of each clique, the head prefers to query those processors that are closer. This algorithm requires decision algorithms for selecting potential members, freeing current members and abdication. We could not produce very good hierarchies with Algorithm I, with the various decision algorithms tried. Algorithm M is a variant of one developed by Larry Witte. In this method, one clique at a time is formed, and a central control is needed to ensure that the algorithm terminates. The algorithm operates in two stages, the first of which informs each processor of all the other nodes in the network and their distances in message links. This preparation can be done by a straightforward modification of Algorithm H.

The second stage of Algorithm M forms the cliques. The central control sends a message to an arbitrary processor in the network ordering it to form a clique. This processor chooses and queries the right number of nodes, sending along the calculated radius of its proposed clique. If one or more of these chosen processors can form its own clique with a smaller radius, it responds to that effect. Otherwise, the chosen processor sends back an agree. If, when all of the chosen processors have replied, at least one has sent back a better radius for its proposed clique, the querying processor chooses one from among those with the best radius for a proposed clique, and tells it to try and make a clique. If all processors respond with agreements, then the querying processor becomes the head of the clique and sends messages to the chosen processors indicating that they are members of the clique. It then chooses some processor not in any clique and tells it to form its own clique from among the other processors that are not in any clique. When as many cliques as possible are formed, the algorithm halts.

Algorithm M is not very concurrent, but does produce fairly good hierarchies. It is also possible to devise a network for which this algorithm cannot produce an optimal solution, since the algorithm uses local hill climbing. If concurrency could be added, this algorithm would be very powerful indeed.

It appears to be more difficult to design good distributed algorithms for forming processor cliques. Either some heuristics are needed so that the different cliques forming simultaneously do not interfere with each other, or concurrency needs to be sacrificed. Some further study into the basic nature of the problem appears to be required in order to find a good, concurrent clique-forming algorithm.

CONCLUSION

We set out to develop parallel algorithms for globally structuring networks of processors. We discovered that although concurrency is an important aspect of the performance of these algorithms, the amount of information about the problem available at each processor plays an essential role. We also found that the nature of the particular problem has to be examined for sources of concurrency.

In future research, we hope to examine not only other problems and algorithms for concurrent solutions, but also problems for which the associated solutions will use the imposed network structures produced by the algorithms in this paper. Since processor networks appear to be emerging as an important computing resource, it is essential that research into the design of distributed algorithms be continued and expanded.

REFERENCES