An Improved Topology Discovery Algorithm for Networks with Wormhole Routing and Directed Links

Ying-Yi Huang*

Department of Computer Science
University of California
Riverside, CA 92521

Mart L. Molle

Department of Computer Science
University of California
Riverside, CA 92521

Abstract

We propose a new parallel topology discovery algorithm for irregular, mesh-connected networks with unidirectional links and wormhole routing. An algorithm of this type was developed for the ATOMIC high speed local area network to avoid the need for manually updating routing tables. Similar needs may arise in wireless networks where channels may be unidirectional because of limited transmission power, multipath, and similar effects. Like the ATOMIC topology discovery algorithm, our algorithm accumulates a map of the network at a distinguished node called the Address Consultant. However, our algorithm is much faster. In addition, our algorithm is more general, because it can correctly resolve topologies that contain multiply connected nodes. We implemented both algorithms in a concurrent simulation environment, and tested them on a variety of topologies.

Keywords: Wormhole Routing, Non-Symmetric Networks, Topology Discovery, Distributed algorithms.

1 Introduction

The ATOMIC network [1] is a novel high-speed LAN that was developed by USC/ISI. ATOMIC differs from traditional LANs (Ethernet, Token Ring, etc.) because it allows arbitrary mesh-connected topologies, and supports parallel transmission of distinct messages over disjoint paths. ATOMIC differs from traditional LANs because it does not support broadcast delivery to all nodes at the physical layer. Conversely, ATOMIC differs from ATM LANs by allowing variable length messages (i.e., normal Ethernet frames, where the lengths may vary by a factor of 20) and in distributing its medium access control functions among the end stations instead of centralizing those functions at an intelligent switch controller. Each node in an ATOMIC network is a Mosaic chip [4], which was originally designed as the processing element for a fine-grain, message-passing, massively-parallel computer system. In effect, ATOMIC takes a massively-parallel Mosaic computer, distributes its processors around a building, and programs its nodes to act as a high-speed LAN that supports variable-length messages.

Because of the design of the Mosaic processor, and the manner in which they can be connected in ATOMIC, an ATOMIC network can exhibit some unusual graph-theoretic properties. First, the communication links are fundamentally *unidirectional*, so that network links will in general form a directed graph. We say that a network is *symmetric* if every link from node A to node B has a matching reverse link from B to A (i.e., the network is an undirected graph); otherwise, it is *non-symmetric*. For example, the upper and lower-right network configurations in Figure 1 are symmetric, but the lower-left network configuration is non-symmetric. In general, non-symmetric ATOMIC networks can arise either from link failures or from intentionally choosing to connect cables in a non-symmetric pattern.

Non-symmetric links may also be a consideration for other applications. For example, small VSAT terminals generally do not have enough power to transmit on the satellite uplink at anywhere close to the same rate as the downlink [5], and in some recent consumer-oriented systems [6] the return link uses a disjoint path through the telephone network. Similar hybrid network topologies have also been proposed for utilizing the cable television system. In addition, mobile computers connected to a narrow band ground-radio system (such as [3]) may be unable to transmit data back to all the stations from which they can receive data from because of power limitations (if one station is a mobile battery powered device and the other is a permanent base station), localized sources of background noise, and multipath interference.

*The project is partially supported by a Computer Science research fellowship from the University of California, Riverside.

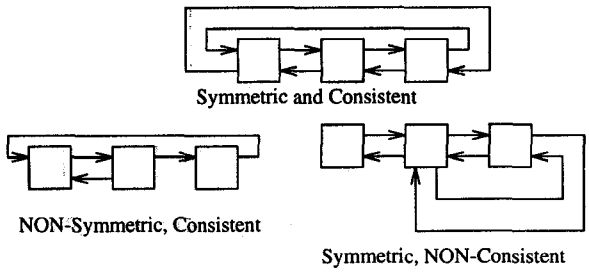


Figure 1: Possible network configurations.

In addition to non-symmetric links, the ATOMC network is also unusual because wormhole routing is used. Thus, the routing algorithm must identify *which port* (and not just *which node*) is the starting point and ending point for a given link. In particular, wormhole routing is basically combination of source routing and cut-through switching, except that messages are not normally buffered at an intermediate node. Each message begins with a routing tag that might say: "Go n_1 steps in the X direction, then turn left and go another n_2 steps in the Y direction." As the head of a message arrives at some intermediate node, the first element of its routing tag is decremented. If the first element reaches zero, the message may require extra handling such as a change of direction, or perhaps delivery to the node processor. However, if the first element in the routing tag is still non-zero, then the message is immediately forwarded one more hop *in the opposite direction* from which it arrived without any involvement of the nodal processor. In particular, the Mosaic chip has four input and four output ports, labelled North, South, East and West; if a message arrives on the East input port, say, with $n_1 > 1$, it will immediately be sent out the West output port. We say that a link is *consistent* if the input and output ports to which it is connected have opposite labels, i.e., a link that leaves one node from its North output port should arrive at the destination node at its South input port. It is important to note, however, that there is no intrinsic reason why links must be consistent. For example, Figure 2 shows that a link that starts from the outgoing East port of node 1 may in general arrive to any of the input ports at node 2.

Taking advantage of consistency is very important for minimizing the latency and processing overhead associated with the delivery of each message under wormhole routing. This is because continuing some extra steps in the same direction always has a very low cost, whereas changing directions may require the entire message to be buffered at that node, which adds the latency of a store-and-delay and creates a bottleneck to the nodal throughput due to the speed of the Mosaic memory.[1] Thus, for proper routing decisions the routing algorithm must know the input and output labels for each link and not just the identities of its starting and ending nodes.

2 The Topology Discovery Problem

The first step in the routing algorithm in ATOMIC is for one distinguished node called the "Address Consultant" (AC) to invoke an algorithm that allows it to gather topology information about the network, which includes finding all of its nodes, and identifying the source and destination nodes along with the associated input and output direction labels for all links. The topology discovery problem in ATOMIC is further complicated by the fact that the Mosaic processors do not have a built-in unique hardware address, so the AC must assign a unique label to each node as it is found. Once the algorithm terminates, the AC has a complete map of the network topology and can determine the routes from any node to any other node. Moreover, during the execution of the algorithm, all of the other nodes will learn the route to the AC, which they consult as a name server whenever they need to determine a route to another node.

To increase fault tolerance, any host may become an AC if it cannot find one in the network. In a large network, it may make sense for multiple ACs to be running in different parts of the network so that requests from hosts need not travel large distances to get to an AC, and to reduce the computational complexity and storage requirements at each AC. However, in this paper we examine the case where there is only one AC in the network.

2.1 The ATOMIC Algorithm

The topology discovery algorithm currently used by the ATOMIC network is shown in Table 1. In the first phase, all nodes cooperate with the AC to flood all the links in the network with **Probe** messages, travelling one hop at a time away from the AC. Each **Probe** message accumulates the path it followed after leaving the AC, encoded as the sequence of output labels it has traversed so far. Eventually, these Probe messages

[1] In ATOMIC, minimum latency routing is almost equivalent to minimizing the number of elements in the routing tag. However, it is interesting to note that, because the Mosaic chip was designed to support a specific row-column routing algorithm for regular two-dimensional grid networks, there is no cost penalty for changing from the X direction to the Y direction. This feature encourages designers of ATOMIC networks to include $Y \rightarrow X$ inconsistencies to reduce the number of store-and-forward delays in a path.

(a) Address Consultant State Machine:

Initial State — Send a **Probe** message with a null routing tag to all outgoing links. Send a 2-hop **Direction Probe** message to all outgoing links. Go to Mapping State.
Mapping State — For each **Probe** or a **Loop** message, check each node on any new path segments by sending a **Label** message and waiting for its **Respond** message to see if it is a new node. For each **Direction** message, or **Direction Probe** message addressed to the AC, create a Direction Handler for this link.

(b) Ordinary Node State Machine:

Initial State — When first **Probe** message arrives, send a copy on each output link after appending the output direction to the current routing tag. Go to Probed State.
Probed State — For each **Probe** message, store it as a **Loop** message. For each **Direction Probe** message, store it as a **Direction** message. When a **Label** message arrives, then: (a) Accept the node label and store the return path to the AC, (b) Send a positive **Respond** message to the AC, (c) Send a 2-hop **Direction Probe** message to all output links, (d) Send all stored **Loop** and **Direction** messages to the AC, and (e) Go to Connected State.
Connected State — For each **Label** message, send a negative **Respond** message, including your existing node label, to the AC. For each **Probe** message, convert it to a **Loop** message and send immediately to the AC. For each **Direction Probe** message, convert it to a **Direction** message and send immediately to the AC.

(c) Direction Handler State Machine:

Waiting State — Wait for the return of **Direction** message for the given link, the **Respond** message that returned after sending a **Probe** message across the given link, and the **Respond** message that returned after sending a **Probe** message over the link from the probed node to the node that returned the **Direction** message for this link. Apply the direction finding method shown in Figure 2.

Table 1: State Machine Description of the ATOMIC Topology Discovery Algorithm. (Because of wormhole routing, we ignore all messages passing through the current node on their way to another destination.)

intersect a previously-probed part of the network (initially just the AC itself) where they are held as **Loop** messages until they can be returned to the AC.

In the second phase, the AC examines the stored routes in the incoming **Loop** messages to discover new path fragments, which are used to expand its map of the network. Initially, the AC labels itself as node 0, and thereafter, each time a new path fragment is found, the AC queries each node in sequence along the new path fragment to establish its identity. More precisely, the AC sends a **Label** message to each node along the new path fragment in sequence, which offers to assign the next unused node label to that node if it is currently unlabelled, and gives it a return path to the AC. The destination node sends back a **Respond** message to the AC to indicate that the node accepts the new node label or to tell the AC what is its existing node label. Using this information, the AC updates its map to include the starting node label, ending node label and output direction for the each link in the newly discovered path fragment.

The third phase of the algorithm is used to determine the input direction by which each link arrives at its destination node. First each newly-labelled node, N, sends a message to its one-hop neighbors in all directions, giving its own node label and the outgoing direction taken by this one-hop message. For example, with reference to Figure 2, such a message would inform node 4 that it is the West neighbor of node 2. (Note that these messages are not returned to the AC, which already learned this information through a Label/Respond transaction in phase 2.) Thereafter, node N uses wormhole routing to deliver a two-hop **Direction Probe** message to each of its two-hop neighbors in a fixed direction (i.e., no "turns" in the route). Each of the recipient nodes, R, holds its message until it has been labelled by the AC, at which point R sends a **Direction** message to the AC that contains the source node label, N, the original outgoing direction, and its own node label, R. Using this information, the AC is now able to determine the incoming direction of the link from N to R. Given the starting node, N, and the original outgoing direction, the AC can use its map from phase 2 to determine the intermediate node I through which the Direction Probe message must have travelled to reach R. Thus, since messages passing through an intermediate node come and go from opposite ports under wormhole routing, the AC concludes that the input direction from node N to node I must be the output to the output direction from node I to node R, which is given in the phase 2 map. Figure 2 shows how this

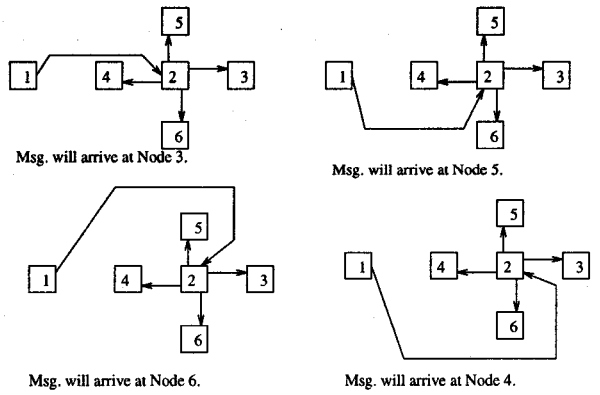


Figure 2: Finding the input direction from node 1 to node 2.

direction handler works. For example, if the two-hop Direction Probe message sent from node 1 through node 2 ends up at node 3, and node 3 is the East neighbor of node 2, then the link from node 1 must have arrived at node 2 from the West.

2.2 Some Weaknesses with the ATOMIC Algorithm

The ATOMIC topology discovery algorithm has two major weaknesses. First, it is very slow because much of the algorithm is sequentially executed by the AC. Indeed, only the initial distribution of **Probe** messages involves any significant parallelism: a node can only advance from the Probed state to the Connected state through a Label/Respond transaction, and these transactions are executed *sequentially* as the AC checks each link in a newly discovered path fragment. That is, the AC sends a **Label** message containing a unique node label to a specific node on the new path fragment, and then waits for the node to return a **Respond** message before issuing the next **Label** message. This is done so the nodes can all be assigned unique labels during the topology discovery process, since the AC does not know whether or not the target node of a particular **Label** message will accept the new node label until it receives its **Respond** message. Worse still, these sequential labelling transactions must actually cover every edge in the graph once, and not just every node once, so the running time of the algorithm is at least $O(E)$.

The second problem involves the inability of the input direction finding algorithm to handle multiple links connecting the same pair of nodes. To see this, consider the example shown in Figure 3, where messages sent through the North output port from node A reach node B after one hop, and reach node C after

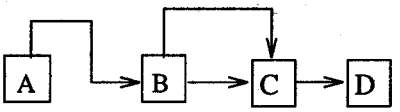


Figure 3: In order to find the input direction of the A, B link, node A must send a 3-hop probe message through B to node D.

two hops. Since nodes B and C are multiply connected, i.e., node C is both the North and East neighbor to node B, we cannot determine from the given information if the input direction to node B from node A was from the West or South. Fortunately, the ambiguity in this case can easily be resolved if we notice that a *three-hop* message sent by node A through its North output port reaches node D. Since D is the East neighbor of C, the continuation of the path from node A to node B must include the link from node B that reaches C from the West.

2.3 A New Parallel Algorithm

The new parallel topology discovery algorithm is shown in Table 2. This algorithm is dramatically faster, and uses significantly fewer messages, than the ATOMIC algorithm. These performance improvements come about because of the following observations.

First, *the nodes can label themselves* during the link flooding procedure in phase I. That is, since the output ports on each node are distinguishable (as North, South, East and West, or perhaps as First, Second, Third, etc.), each routing tag relative to the given AC uniquely identifies the destination node. In other words, a node can choose the routing tag it finds in any incoming phase I message as its node label, and still be assured that no other node in the network can choose the same label. Thus, in our algorithm each node labels itself with the routing tag of the first message to arrive in phase I, so we "promote" the phase I message type to become a **Label** message.

The second observation is that *the nodes don't need to inform the AC of their choice of node labels*, since the AC can deduce this information at no cost by examining the incoming **Label** messages from phase I and returned **Loop** messages from phase II. This is true because the outbound wave of **Label** messages in phase I stops as soon as it intersects a previously-labelled node, at which point the messages are held until they can be returned to the AC as **Loop** messages in phase II. Thus, during phase I every **Label** message that gets forwarded by a given node must contain its own node label as a prefix of the outgoing

(a) Address Consultant State Machine:

Initial State — Send a **Label** message with a null routing tag followed by a a 2-hop **Direction Probe** message to all outgoing links. Go to Mapping State.
Mapping State — For each **Label** or a **Loop** message, identify the new path segment as L new links separated by $L-1$ new nodes, and add them all to the network map. If $L > 1$, send a **Return Path** message with node count $L-1$ to the first new node. For each **Direction** message, or a **Direction Probe** message addressed to the AC, create a Direction Handler for this link.

(b) Ordinary Node State Machine:

Initial State — When the first **Label** message arrives, accept its routing tag as the node label. For each output link, extend the routing tag by one hop in the corresponding direction and send the revised **Label** message followed by a two-hop **Direction Probe** message out that link. Go to Labelled State.
Labelled State — For each **Label** message, store it as a **Loop** message. For each **Direction Probe** message, store it as a **Direction** message. For each **Notice** message, increment the hop count for that link and send another **Direction Probe**. When a **Return Path** message arrives, then: (a) Store the return path to the AC, (b) Decrement the hop count and either throw the message away if it reaches zero or remove the first step from the return path, and send it one hop in that direction, (c) Send all stored **Loop** and **Direction** messages to the AC, and (d) Go to Connected State.
Connected State — For each **Notice** message, increment the hop count for that link and send another **Direction Probe** message. For each **Direction Probe** message, convert it to a **Direction** message and send immediately to the AC.

(c) Direction Handler State Machine:

Waiting State — Wait for the return of **Direction** message for the given link, and enough **Label** and **Loop** messages to determine the neighboring nodes and apply the direction finding method shown in Figure 2. If the input direction at the last hop cannot be determined because nodes are multiply connected, increment the number of hops required and send a **Notice** message to the source node. Otherwise stop.

Table 2: State Machine Description of the Parallel Topology Discovery Algorithm. (Messages in transit to other nodes are ignored, due to wormhole routing.)

routing tag, and hence that *every prefix of the routing tag* for a **Label** message that either:

- returns on its own to the AC during phase I; or
- is being held at an intermediate node as a **Loop** message until phase II

must be the chosen node label of the corresponding node. In other words, the set of routing tags generated in this way is *consistent* in the sense that the set of links where the source node label is a prefix of the destination node label forms an outbound spanning tree rooted at the AC. Moreover, each of the remaining links in the network appears as the "final hop" in some loop message.

At the moment when each **Label** message returns to the AC during phase I, we can identify a new cycle in the graph, using its routing tag, in which each node knows its own node label, and the AC now knows all of their node labels. However, none of these nodes yet knows the return path to the AC, and the AC knows nothing about the subordinate loops for which the routing tags are being held as **Loop** messages at one of these nodes. Thus, our new parallel algorithm also requires second phase in which the AC tells each node about a return path to the AC. However, our phase II is done in parallel, using piggybacked messages, based on a third observation about the problem dynamics, namely that *the new information contained in each* **Label** *or* **Loop** *message that returns to the AC is a single path fragment of known length.* In other words:

- in the first-to-return **Label** message, the entire path is a new path fragment; and
- in each subsequent **Label** or **Loop** message, the remainder of the path, starting from the point where it diverges from previously mapped paths and ending at the point where it either returns to the AC or is held as a **Loop** message, is a new path fragment.

Thus, unlike the ATOMIC algorithm, which uses a series of individual Label/Respond transactions to check each newly discovered link, our new parallel algorithm sends a single piggybacked **Return Path** message over the new path fragment. The message is initialized to contain the return path to the AC relative to the first node on the new path fragment, together with a count of the number new nodes in the path fragment, and is then sent directly to the first node on the new path fragment via wormhole routing. Thereafter, as the **Return Path** message reaches each of

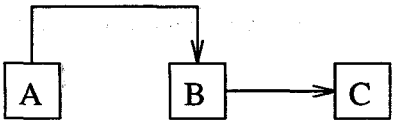


Figure 4: The input direction of AB cannot be found.

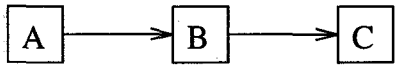


Figure 5: The input direction of AB can be found.

the new nodes along this new fragment, it saves the complete return path for its own use, decrements the node count and throws the message away if it reaches zero, and finally deletes the first step from the return path and uses it to select the outgoing link on which to forward the message to the next node. Once the **Return Path** message has been taken care of, the node then sends any saved **Loop** messages directly to the AC via wormhole routing.

2.4 Handling Multiply Connected Links

The remainder of the algorithm involves finding input directions. Input directions are important under wormhole routing, since messages can be sent "directly" to a destination n hops away *in the same direction* without store-and-forward packet switching delays at the intermediate nodes. Thus, it is important to know if the path from node A to node C looks like Figure 4, where wormhole routing cannot be used, or like Figure 5, where it can. Fortunately, both the ATOMIC algorithm and our new parallel algorithm can find the input direction in Figure 5, where it is needed by the AC to decide that a wormhole path exists from node A to node C.

As described above, the normal case is handled by sending two-hop **Direction Probe** messages out each port, which eventually get forwarded to the AC by the recipient as **Direction** messages. However, unlike the ATOMIC algorithm, in our case the nodes can send the **Direction Probe** messages right after they send the **Label** messages in phase I, since they already have their node labels. In addition, our algorithm handles multiply connected nodes using the technique described in section 2.2, where the AC resolves the ambiguity by sending a **Notice** message to the source node, requesting it to send another **Direction Probe** message with the target set one more hop away. Thus, our algorithm can resolve the input direction at node B in Figure 6, whereas the ATOMIC algorithm cannot. (Moreover, neither algorithm can

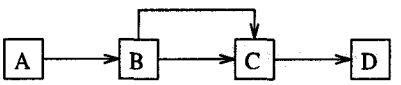


Figure 6: The input direction of AB can be distinguished by our algorithm, but not the ATOMIC algorithm.

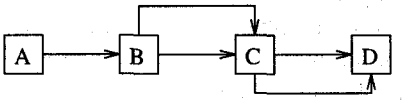


Figure 7: The input direction of AB cannot be distinguished by any algorithm.

handle the case in Figure 7 — although the answer is unimportant since there is no way to use wormhole routing any further than node D anyway.) In general, our algorithm can resolve the input direction if there exists an n-hop path, $n \geq 2$, in which the *last hop* is singly connected. In this case, the source node will eventually receive a **Notice** message that triggers an n-hop **Direction Probe** message, which allows the AC to resolve the input direction at the last hop, from which the other input directions are found by backtracking.

It is worth mentioning at this point that the only difference in the final result of executing our new parallel topology discovery algorithm instead of the ATOMIC algorithm is in the node labels, which are fixed-length consecutive integers in the ATOMIC algorithm, and variable-length routing tags in our algorithm. For example, since the ATOMIC network has 4 outputs per node, we could encode the addresses as bit strings, using two bits per hop. However, ATOMIC-style consecutive integer node labels are easy to put into our algorithm without using any additional messages. Recall that the AC already knows the exact number of new nodes and their respective self-assigned node labels on each new path fragment as soon as it receives the corresponding **Label** or **Loop** message. Thus, the AC could reserve the required number of new node labels for that path fragment and inform each node of its new label via an additional field in the **Return Path** message. The AC simply initializes the field to the new node label for the first node on the path fragment, and thereafter, each new node increments the field before passing it one more hop along the path.

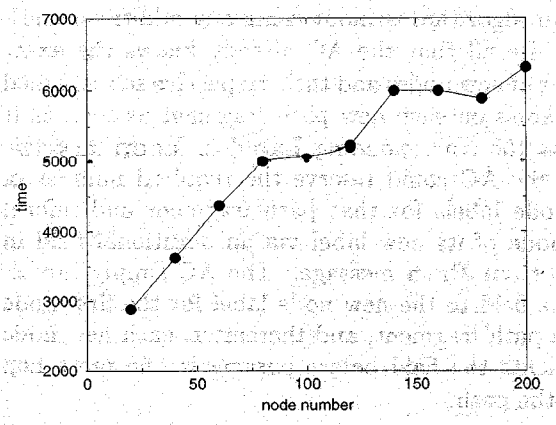
3 Experimental Results

Both topology discovery algorithms were tested using a detailed simulation model, which was constructed using the SMURPH network simulation environment [2]. The SMURPH environment is optimized for simulating network protocols by emulating the physical transmission of data over various links between independently executing hosts. Thus, our SMURPH model involves defining the network topology and programming each host to follow the protocols given in Tables 1 and 2. The correctness of the simulation was ensured by adding various sanity-check assertions about the global state of the system into the code. In addition, a separate program was developed to verify the output to make sure that network map produced by the AC, including the nodes, edges and input/output directions, matches the actual topology of the network.

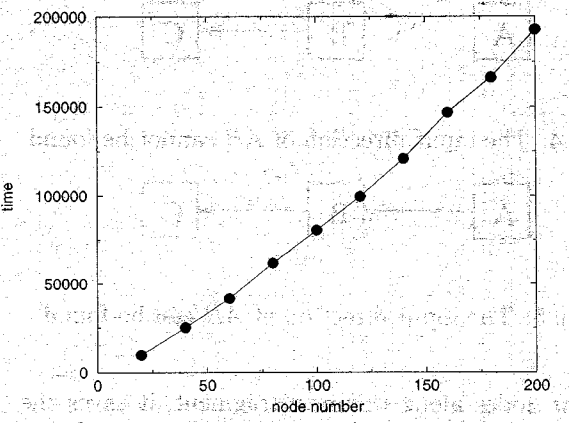
Using the simulation model, various experiments were conducted for three kinds of graph:

- Grid graphs. This is a regular 4-connected graph similar to the original Mosaic topology. Graphs from twenty nodes to two hundred nodes were tested.
- Random sparse graphs. We make one pass through the set of all ports, arbitrarily selecting pairs of ports (consisting of one input port and one output port) under the restriction that both ports cannot belong to the same node. Then we connect them to form a link with probability 0.5. Graphs from twenty nodes to two hundred nodes were tested. For each node size, six graphs were generated, and the mean execution time is shown.
- Random dense graphs. Same as above but each link is connected with probability 0.75.

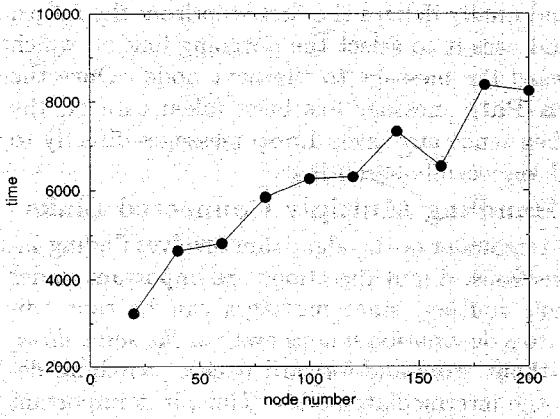
Mean time of dense graph using new alg.



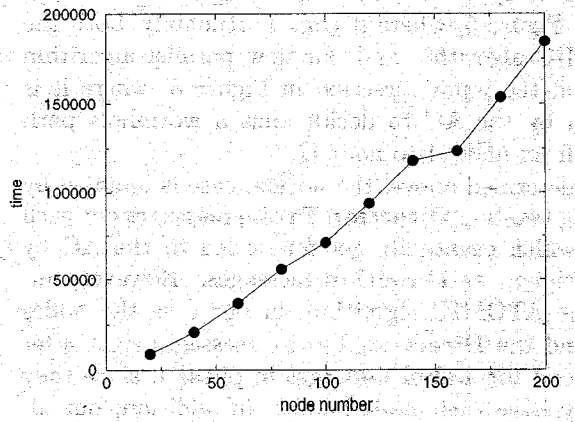
Mean time of dense graph using old alg.



Mean time of sparse graph using new alg.



Mean time of sparse graph using old alg.



In each test, both algorithms were run on exactly the same set of graphs, and for each graph the elapsed simulation time (assuming it takes one time unit for a packet to travel one hop), the total number of messages generated by the protocol, and the total number of hops travelled by all messages were recorded. We found that the major difference between the two algorithms was speed, with our new parallel algorithm

running at least ten times faster than the ATOMIC algorithm. This improvement was expected, because the new algorithm eliminates the serialization bottleneck in the ATOMIC algorithm due to the Label/Respond transactions (see Sections 2.2 and 2.3). This speed advantage is even more remarkable when you consider that the timing for our algorithm also included the relatively-expensive additional steps for handling multiply connected nodes.

4 Conclusion

We have presented a new parallel topology discovery algorithm for directed networks with wormhole routing. Our algorithm is much faster than the one developed for the ATOMIC project, mostly because our improvements eliminate an obvious serialization bottleneck that is present in their algorithm. In addition, our algorithm includes a number of more subtle refinements, such as piggybacked delivery of information from the AC to a sequence of nodes, early transmission of direction probe messages, and the generalization of the direction finding algorithm to properly handle nodes with multiple connections.

It is interesting to note the significance of the seemingly minor decision to use routing tags as node labels. Even if we ignore the serialization bottleneck, this change reduces the time until a node is labelled by an entire round-trip delay (i.e., the time for the **Probe** or **Loop** message to return to the AC, followed by the time for the AC to send a **Return Path** message back to the node). Similarly, because of the change the AC does not need the Label/Respond transaction to identify (and possibly assign a label to) the nodes on a newly discovered path fragment. Of course, sequential integer node labels are more convenient than variable length routing tags, but we can easily add that type of node label to the algorithm without any additional messages.

Although we believe that our parallel topology discovery algorithm is quite efficient, there are still some interesting extensions possible that we plan to explore. As the size of the network becomes very large, centralizing the routing functions in a single AC may become unmanageable. Moreover, if more than one AC is used, having each one map the entire network is very inefficient. Thus, we plan to investigate methods for partitioning the topology discovery problem amongst multiple ACs. In addition, since network topology changes may occur from time to time, we plan to study efficient techniques for incrementally remapping the network in response to topology changes.

References

[1] R. Felderman, A. DeSchon, D. Cohen and G. Finn, "ATOMIC: A High-Speed Local Communication Architecture", *Journal of High Speed Networks*, Vol 3(1), pp.1-30 (1994).

[2] P. Gburzyński, *Protocol Design for Local and Metropolitan Area Networks*, Prentice-Hall, Englewood Cliffs, New Jersey, 1996.

[3] Metricom, Inc., *The Ricochet Wireless Network Overview*, URL: *http://www.ricochet.net/ricochet/netoverview.html.*

[4] C. I. Seitz, N. Boden, J. Seizovic, and W. Su, "The Design of the Caltech Mosaic C Multicomputer", *Proceedings of the Washington Symposium on Integrated Systems*, Seattle, WA (1993).

[5] W. Stallings, *Data and Computer Communications*, Fifth Edition, Prentice-Hall, Englewood Cliffs, New Jersey 1997.

[6] WebTV Home Page, URL: *http://www.webtv.net/wtvnet.html.*