An Incremental Distributed Algorithm for Computing Biconnected Components

Bala Swaminathan
Kenneth J. Goldman

WUCS-94-6

March 1994

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130–4899

An Incremental Distributed Algorithm for Computing Biconnected Components*

Bala Swaminathan
bs@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130

Kenneth J. Goldman
kjg@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130

March 29, 1994

Abstract

This paper describes a distributed algorithm for computing the biconnected components of a dynamically changing graph. Our algorithm has a worst case communication complexity of $O(b + c)$ messages for an edge insertion and $O(b' + c)$ messages for an edge removal, and a worst case time complexity of $O(c)$ for both operations, where c is the maximum number of biconnected components in any of the connected components during the operation, b is the number of nodes in the biconnected component containing the new edge, and b' is the number of nodes in the biconnected component in which the update request is being processed.

The algorithm is presented in two stages. First, a serial algorithm is presented in which topology updates occur one at a time. Then, building on the serial algorithm, an algorithm is presented in which concurrent update requests are serialized within each connected component. The problem is motivated by the need to implement causal ordering of messages efficiently in a dynamically changing communication structure.

Keywords: biconnected components, distributed graph algorithms, dynamic configuration.

*This research was supported in part by the National Science Foundation under Grant CCR-91-10029.

1 Introduction

We present a distributed algorithm that incrementally computes the biconnected components of a dynamically changing graph. Although the problem is a general one and is interesting in its own right, our particular motivation to study this problem arose as part of some work we are doing in conjunction with the ATM networking project at Washington University [6].

We are developing an abstraction and supporting software designed to simplify the construction of distributed multimedia applications [7]. In our approach, called *I/O abstraction*, each *module* has a *presentation* that consists of data structures that may be observed and/or manipulated by its external environment. An *application* consists of a collection of independent modules and a *configuration* of *logical connections* among the data structures in the module presentations. Whenever published data structures are updated, communication occurs implicitly according to the logical connections.

Our current run-time support for I/O abstraction provides no ordering guarantees; when one module changes a value in its presentation, communication occurs asynchronously according to the configuration. However, many applications require stronger ordering properties for consistency. Causal ordering (Lamport's "happens before" relation [9]), is one of the most useful ordering restrictions. Informally, it guarantees that if event A directly or indirectly "causes" event B, then no process will receive information about event B before receiving information about event A.

Causal ordering is supported by the ISIS [3] system. ISIS programmers declare causal broadcast *groups*. Within these groups, causal ordering is maintained by an algorithm whose message header (a timestamp vector) has length proportional to the size of the group. In the case of I/O abstraction, the programmer does not declare sets of nodes for which messages must be causally ordered. In fact, the programmer may not know the set of modules with which a given module interacts. Our configuration is under user control, and may be changed dynamically during execution. However, we can exploit the logical configuration information in order to determine the sets of nodes that could potentially violate causal ordering and then run the causal ordering algorithm within each of those sets.

Consider the dynamically changing configuration graph of modules (vertices) and logical connections (edges). By causally ordering the messages within the set of modules in each biconnected

component, we can guarantee that causal ordering is respected over the entire system. This is because there can be no cycle in the graph containing two modules a and b in different biconnected components, implying that information from a to b cannot flow over two node-disjoint paths. In the distributed algorithm presented here, each node maintains the set(s) of nodes corresponding to the biconnected component(s) of which it is a member, precisely the information required for the causal ordering algorithm.

The problem of computing biconnected components has been studied extensively. A number of sequential algorithms exist for dynamic graphs and there are several decentralized algorithms for static graphs, but the algorithms we present in this paper are, to our knowledge, the first distributed algorithms for finding biconnected components in dynamic graphs.

A distributed algorithm for finding biconnected components was given by Chang [4]. This algorithm has a message complexity of $4m-n$, where m and n are the number of edges and vertices in the graph respectively. The distributed algorithm by Ahuja and Zhu [2] has the same message complexity but improves on the message size bound. Hohberg [8] and Park et al. [10] present distributed algorithms for finding biconnected components in a graph with a message complexity of $O(m + n \log n)$. These algorithms require the computation of a depth-first search tree. Hence, they are appropriate for a static graph, but the cost of recomputation of the depth-first search tree (for every change in topology) makes these algorithms impractical for a dynamic setting.

Tarjan and Vishkin proposed an optimal parallel algorithm on CRCW PRAM model [12]. This algorithm is also not incremental, but instead of using depth-first-search, it reduces the biconnectivity problem to the problem of computing connected components. Westbrook and Tarjan [13] proposed a sequential algorithm to compute biconnected components in a dynamically changing graph structure. A block forest of block trees is constructed using the biconnected components and the vertices in the graph. This block forest is used in maintaining the biconnected components of the original graph. Rauch [11] presented a sequential algorithm for maintaining biconnected components. This algorithm involves precomputation and "lazy" updating.

The algorithms presented in this paper are dynamic as well as distributed, and are designed to scale up for large systems. Update requests can be issued at any node in the system and in any order. The nodes have only local knowledge of the system graph and exchange information with other nodes by sending and receiving messages. Only one copy of the topology is maintained and

it is distributed among the nodes in the system.

The remainder of the paper is organized as follows. Section 2 provides essential definitions and graph properties. In Section 3, we define the problem. In Section 4, we present a *serial algorithm* in which the environment is constrained to wait for each graph update request (edge insertion or deletion) to be processed before issuing the next request. Then, in Section 5, we build upon the serial algorithm to construct a *concurrent algorithm* that serializes update requests within each connected component. We serialize the update requests such that each node within a connected component will have the same view of update sequence over that connected component. Because connected components merge and split dynamically, ordering information is maintained with each update in order to achieve this consistent view. Our algorithms have a message complexity of $O(b + c)$ for an insert operation and $O(b' + c)$ for a delete, and a time complexity of $O(c)$ for both operations, where b is the number of nodes in the resulting biconnected component, b' is the number of nodes in the bcc just before the deletion, and c is the maximum number of biconnected components in any of the connected components during the operation.

2 Preliminaries

We use $G = (V, E)$ to denote an *undirected graph*, where V is a totally ordered finite set of vertices and E is a set of unordered pairs of distinct elements of V. We use standard definitions [5] for *path*, *reachability*, *cycle*, *graph union*, etc. Recall that an *articulation point* is a vertex whose removal would disconnect the connected component containing it. A *bridge edge* is an edge whose removal would place its endpoints in different connected components. A *biconnected graph* is a graph with no articulation points. A *biconnected component (bcc)* is a maximal biconnected subgraph. A bridge edge is in a bcc by itself. We define the *size of a bcc* to be the number of vertices in the bcc. In a graph G, for each vertex a, we define $BCC_G(a)$ to be the number of biconnected components in which a is a member. We say that two bcc's are *neighbors* if they have a vertex in common. (Note that a bcc is a neighbor of itself.)

For two vertices a and b, we define $common\_vertex(a, b)$ to be the vertex b if a and b lie in the same bcc, to be the vertex shared by the bcc's of a and b if a and b lie in neighboring bcc's, and to be undefined otherwise. For the sequence of vertices $a_1, \ldots, a_n$, such that $n > 2$, we define

traversed_graph$(a_1, \ldots, a_n)$ to be the union, over all $1 < i < n$, of the bcc containing a_i and *common_vertex*(a_i, a_{i-1}), and the bcc containing a_i and *common_vertex*(a_i, a_{i+1}). For vertices a and b, *traversed_graph*(a, b) is defined as the biconnected component containing both a and b, if $a \neq b$, and the null graph otherwise. Therefore, for a sequence S of vertices such that for every pair of successive vertices a and b in S, a bcc containing a is a neighbor of a bcc containing b, *traversed_graph*(S) is the union of this chain of neighboring bcc's.

We define the *local bcc topology* of vertex v in graph G to be the subgraph of G containing exactly the biconnected components containing v. Note that if v is not an articulation point, then its local bcc topology consists of the one biconnected component containing v.

If a and b are vertices in the same connected component of G, then we define the *link vertex set of* (a, b) *in* G, denoted $LVS_G(a, b)$, to be the set of articulation points that appear in every path from a and b in G. (We call this the "link vertex set" because it is the set of vertices that "link" the biconnected components traversed by every path from a to b.) Our algorithms rely on the following properties that follow from the definitions.

Property 1: For any two vertices a and b in graph G, at most one biconnected component of graph G contains both a and b.

Property 2: Vertex a is an articulation point in graph G iff a is a member of at least two bcc's in G.

Property 3: If nodes a and b are in the same biconnected component G', then every node on a simple path between a and b is also in G'.

Proof: Let a and b be in the same bcc and let x be a node in a simple path p between a and b. Since G' is a biconnected component, there must be path p' from a to b such that p and p' have only vertices a and b in common. Paths p and p' form a cycle containing x, a, and b; hence x must belong to the same biconnected component as a and b. ■

Corollary 4: Let each biconnected component be represented as a super node. Consider the set of articulation points and the set of super nodes. Create an edge between a super node and an articulation point whenever the articulation point belongs to the corresponding bcc. The resulting graph is a tree. (That is, a connected component induces a tree of biconnected components [13].)

Lemma 5: Let a be a vertex in graph $G = (V, E)$, let $n = BCC_G(a)$, and let b and c be vertices such that b is reachable from c. If $G' = (V, E \cup \{(b, c)\})$, then $n \geq BCC_{G'}(a) \geq n - 1$.

Proof: If b and c are in the same bcc in G, then $BCC_{G'}(a) = BCC_G(a)$. If b and c are in different bcc's, then the value of $BCC_{G'}(a)$ depends on whether or not a is a link vertex. If $a \in LVS_G(b, c)$, then $BCC_{G'}(a) = BCC_G(a) - 1$; otherwise $BCC_{G'}(a) = BCC_G(a)$. ■

Lemma 6: Let a and b be vertices in different biconnected components of a graph $G = (V, E)$. Let $G' = (V, E \cup \{(a, b)\})$. Then a node $c \in LVS_G(a, b) \setminus \{a, b\}$ is an articulation point in G' iff c is a member of at least three biconnected components in G.

Proof: Since c is a link vertex, there is a path p from a to b in G, and p traverses two bcc's B_1 and B_2 both containing c. Suppose c is an articulation point in G'. Since p and (a, b), form a cycle in G', we know that a, b, and c are in the same bcc in G' and that this bcc contains both B_1 and B_2. Since c is an articulation point in G', there must be a bcc B_3 (different from B_1 and B_2) in which c is a member. Therefore, c is a member of bcc's B_1, B_2, and B_3 in G, so we have $BCC_G(c) \geq 3$. To show the reverse implication, suppose $BCC_G(c) \geq 3$. From Lemma 5, $BCC_{G'}(c) \geq BCC_G(c) - 1$. Therefore, $BCC_{G'}(c) \geq 2$. From Property 2, it follows that c is an articulation point in G'. ■

3 Problem Statement

We assume a failure-free distributed system in which each node in the system may send a message to any other node, messages between each pair of nodes are delivered in the order sent, and eventual delivery is guaranteed. We are interested in computing the biconnected components of a dynamically changing *logical configuration graph* that defines the communication pattern of the application. Initially, the logical configuration graph has no edges. The environment may cause changes to the graph by requesting the insertion or deletion of an edge at the node corresponding to either endpoint of the edge. The node at which the request is made is called the *requesting node*, and the node at the other endpoint is called the *peer*.

We say that an algorithm solves the biconnected component problem if after any sequence of requests (edge insertions and deletions), each node n in the system knows the set of vertices in each of the biconnected components containing n. It is important to note that the algorithm is not

constrained to send messages only to neighbors in the logical configuration graph. Instead it is free to send messages between arbitrary nodes in order to compute the biconnected components of the logical configuration graph. This is because the logical configuration graph corresponds only to the communication pattern of the application, and not to the physical topology of the network.

Lower bound: In any solution, an update request must conclude with each node knowing the sets of nodes in the biconnected components of which it is a member. Since the nodes must be informed of these new sets, the lower bound on the number of messages for an insert edge operation is $\Omega(b)$, where b is the size of the resulting biconnected component. Similarly, the lower bound for a remove edge operation is $\Omega(b')$, where b' is the size of the biconnected component that contained the deleted edge. This lower bound on message complexity can be achieved by a centralized solution that maintains all topology information at a leader node and broadcasts changes to affected nodes in constant time. However, we are interested in distributed solutions that scale by distributing topology information among the nodes.

4 The Serial Algorithm

In the serial algorithm, we assume that the environment makes requests (edge insertions and deletions) one at a time, and that the processing of each request completes before the next request is made. The node v maintains information about its local bcc topology. If v is the minimum vertex in the bcc, then it keeps its local bcc topology graph. Otherwise, the node v keeps a set, each element of which is the set of vertices in a biconnected component containing v. In this way, each node maintains the set(s) of vertices in the biconnected component(s) in which it is a member.

The *coordinator for node p* is defined to be the minimum node in the local bcc topology of p. The *coordinator for a bcc B* is the minimum node in that bcc. In addition to its local topology, the coordinator node for a bcc B keeps track of the neighboring coordinator nodes. That is, the coordinator maintains a mapping $\mathcal{C}$ from the set of nodes V in the system, to the set of all multisets over the elements of V. The mapping $\mathcal{C}$ relates each articulation point q in the bcc B to the set of coordinators for the bcc's containing q, and for all nodes x which are not articulation points $\mathcal{C}(x)$ is defined to be the empty set. In this way, each coordinator knows the set of nodes for which it is the coordinator and the set of coordinators for neighboring bcc's.

Figure 1: Cases for edge insertion.

Overview: There are two incremental operations: insert an edge and delete an edge. When inserting edge (r, p), we must determine if r and p belong to the same biconnected component. In this case we have an "easy link" (see Figure 1 a, b), and we simply update the local bcc topology information maintained in the coordinator of this biconnected component. Otherwise, we search for a path between r and p. If there is no path, then they are in different connected components and we have a "component link." In this case, we create a new biconnected component containing only this new edge (see Figure 1 c, d). If there is a simple path between r and p, then the topology of the new biconnected component, formed by adding this edge, is constructed by taking the union of the biconnected subgraphs along the path (see Figure 1 e, f). It should be noted that any such path from r to p will have the same subsequence of link vertices. We call this "path condensation" after [13].

Messages: Nodes participate in the algorithm by message-passing. Each message has a name (its type) and includes the source and destination nodes, and the requesting node r and peer node p for the request. Some messages contain additional information. We assume that each request has a unique identifier carried in each of its messages. A summary of message types is presented in Figure 2 to aid the reader in understanding the algorithm description. We use c to denote the coordinator handling the request.

Message type	Purpose
Req_Update	informs c that an update was requested
Check_For_Easy_Link	asks p if it lies in a bcc with r
Easy_Link	informs c that p and r share a bcc
Not_Easy_Link	informs c that p and r do not share a bcc
Find_Peer	asks receiving coordinator if p lies in its bcc subtree
Return_Peer$(G,\mathcal{C})$	indicates that p lies in the connected component, where G is a subgraph of the resulting bcc, and $\mathcal{C}$ is the accumulated coordinator mapping
No_Peer	indicates that p is not in the sender's bcc subtree
Add_Edge	informs the receiver of an easy link between r and p
Coord_Update$(G,\mathcal{C})$	informs the receiver that it is the coordinator of a new bcc with topology G and coordinator mapping $\mathcal{C}$
Bcc_Update(V)	instructs the receiver to update its local bcc topology with the set V
New_Bcc	informs the receiver about a new bridge edge (r,p)
Adjust_Mapping(V)	informs the receiving coordinator to adjust its mapping because a new bcc with nodes V has been formed
Note_Coord(a,c',X)	informs the receiving coordinator of a new coordinator c' for node a in the neighboring bcc (to replace the coordinator from set X)
Remove_Coord(a,c')	informs the receiving coordinator that c' is not the coordinator for some bcc containing a because of a bridge deletion
Coord_Set(S)	informs the receiver about the set of coordinators for the sender in the bcc's containing the sender
Check_For_Bridge$(\{r,p\})$	requests the peer to check if $\{r,p\}$ is a bridge
Is_Bridge$(\{r,p\})$	informs the requesting coordinator that $\{r,p\}$ is a bridge
Not_Bridge$(\{r,p\})$	informs the requesting coordinator that $\{r,p\}$ is not a bridge
Delete_Edge(r,p)	informs the receiver to delete an edge that is not a bridge
Delete_Bcc$(\{r,p\})$	informs the receiver that an incident bridge $\{r,p\}$ is deleted

Figure 2: Message types.

Stages: For edge insertion, the algorithm proceeds in three stages: In the *coordinator notification stage*, the requesting node notifies the coordinator node that will coordinate the operation. In the *classification stage*, the graph is searched for the peer node in a broadcast over a tree of coordinators rooted at the coordinator for the requesting node. Relevant topology and coordinator mapping information are collected in a convergecast. The classification stage determines whether the insertion is an easy link, component link, or path condensation. In the *update stage*, the local bcc information is updated at each affected node, according to the information collected in the search stage and the mapping of the (possibly new) coordinator is updated and neighboring coordinators are notified. Edge deletion is similar, but the classification stage determines whether the edge to be deleted is a bridge edge or not. The next two subsections explain the stages of the algorithm for insert and delete.

Remarks: The algorithm we describe takes advantage of the structure of the logical configuration graph by conducting the search along the coordinators of the biconnected components. An alternative approach would be to assign fixed coordinators to arbitrary sets of nodes regardless of the logical configuration (for example, in a centralized algorithm all nodes would be assigned to the same coordinator). This avoids the problem of maintaining the coordinator mapping, but does not take advantage of the structure of the problem. The search in that algorithm would require communication among all coordinators in the worst case, even if only one biconnected component (containing one node assigned to each coordinator) is affected by the operation.

4.1 Edge Insert

4.1.1 Coordinator notification stage

```
procedure Insert(r, p)
    begin
        (link_type, graph, mapping) = Insert_Classify(r, p)
        Insert_Update(r, p, link_type, graph, mapping)
        send done message to the requesting node
    end Insert
```

Figure 3: Insert procedure.

In the coordinator notification stage, the requesting node simply sends a *Req_Update* message to its coordinator node. This instructs the coordinator to process the update on behalf of the requesting node (see Figure 3). In the concurrent algorithm, the coordinator is used to help serialize the requests.

4.1.2 Classification stage

The coordinator first checks if both the requesting node and the peer node lie in the same bcc as itself. If so, it then proceeds to the update stage for a "very easy link." Otherwise, if r is an articulation point, edge (r, p) may be an easy link in another of r's bcc's. Therefore, it sends a *Check_For_Easy_Link* message to the peer[1]. If the coordinator receives an *Easy_Link* message from the peer, it proceeds to the update stage. On the other hand, if r is not an articulation point or if the coordinator receives a *Not_Easy_Link* message from the peer, the coordinator sends a *Find_Peer* message to itself, initiating a broadcast over the bcc's in its connected component. This broadcast continues until either the *peer* node is found or all bcc's in the connected component have been searched (see Figures 4 and 5).

When a node n receives a *Find_Peer* message, it remembers the node from which it received the message as its *parent* and checks if the peer and n lie in the same bcc. If so, it sends a *Find_Peer* message to the peer and the peer responds with a *Return_Peer*$(\emptyset, \emptyset)$. If n and the peer do not lie in the same bcc, n continues the broadcast by sending a *Find_Peer* message to each of the coordinators in the neighboring bcc's except those nodes that are also neighbors of *parent*. It remembers this (possibly empty) set of nodes as its *children*.

The convergecast of the classification stage begins at a node when it has received a *No_Peer* response from all of its children or a *Return_Peer* response from a child or the peer. If the node n has received a *Return_Peer*$(G_1, \mathcal{C}_1)$ message from another node b, then it computes the graph G as the union of G_1 and *traversed_graph*$(parent, n, b)$, defined in Section 2. This traversed graph is the union of those bcc's that are on the path from r to b for which n is the coordinator. The graph

[1]In the serial algorithm, we could have the requesting node make the easy link determination before sending the request to the coordinator. However, in the concurrent algorithm, the topology may change between the time the request is made and the time it is processed. Therefore, for ease of explanation, we have the coordinator ask the peer for this information. The peer, rather than the requesting node, is chosen because the concurrent algorithm must communicate with the peer anyway for serialization purposes.

```
procedure Insert_Classify(r, p) returns (link_type, graph, coordinator mapping)

    begin
        if r and p lie in a bcc for which c is the coordinator then
            return ("very easy link", ∅, ∅)
        else if r is an articulation point
            send Check_For_Easy_Link(r, p) to the peer p, and wait for a response
            if an Easy_Link message is received from the peer
                then return ("easy link", ∅, ∅)
        start the broadcast by sending a Find_Peer to self, and wait for a response
        if response from self is Return_Peer(G,C) then
            return ("path condensation", G, C)
        else (response is No_Peer)
            return ("component link", ∅, ∅)
    end Insert_Classify
```

Figure 4: Classification procedure for coordinator c.

G, computed in this manner, is a subgraph of the resulting bcc. It then sends a *Return_Peer*(G,$\mathcal{C}$) message to its parent, where the accumulated mapping $\mathcal{C}$ is the union of $\mathcal{C}_1$ (the mapping received), and the current mapping of the coordinator. Thus, we accumulate the topology and the coordinator mapping for each bcc along a simple path from the peer to the requesting node in order to build the topology and the mapping of the new bcc. Otherwise (if the node has not received a *Return_Peer* message), it sends a *No_Peer* message to its parent.

When the coordinator receives a *Return_Peer*(G,$\mathcal{C}$) message from itself, the Insert_Classify procedure returns "path condensation" along with the graph G and mapping $\mathcal{C}$. If the coordinator receives a *No_Peer* message from itself, the classification procedure returns "component link," and the algorithm proceeds to the update stage.

4.1.3 Update stage

The update stage begins when the coordinator has received responses from all of its children (see Figures 6 and 7). There are three cases:

- On receiving *Check_For_Easy_Link*(r, n):

  if n and r are in the same bcc
  then send *Easy_Link* message to the sender
  else send *Not_Easy_Link* message to the sender

- On receiving *Find_Peer* from node b:

  parent $\leftarrow b$
  if $n = p$ **then** send *Return_Peer*$(\emptyset, mapping)$ to *parent*
  else if peer is in the local bcc **then**
  send *Find_Peer* to p
  children $\leftarrow \{p\}$
  else continue the broadcast:
  children $\leftarrow$ set of all neighboring coordinators a such that $a \notin \mathcal{C}(common\_vertex(n, parent))$
  if *children* $\neq \emptyset$ **then**
  send *Find_Peer* to each node in the set *children*
  else send *No_Peer* to *parent*
  parent $\leftarrow$ *nil*

- On receiving *Return_Peer*$(G_1 = (V_1, E_1), \mathcal{C}_1)$ from another node b:

  remove the sender from *children*
  send *Return_Peer*$(G, \mathcal{C})$ to *parent*, where
  $G = G_1 \cup traversed\_graph(parent, n, b)$
  $\mathcal{C}(a) = \mathcal{C}'(a) \cup \mathcal{C}_1(a)$ such that $\mathcal{C}'$ is the current mapping, for each node a in G
  parent $\leftarrow$ *nil*

- On receiving *No_Peer* from another node:

  remove the sender from *children*
  if *children* $= \emptyset$ and a *Return_Peer* has not been sent to *parent* **then**
  send *No_Peer* to *parent*
  parent $\leftarrow$ *nil*

Figure 5: Message handlers for classification procedure for the node n.

procedure Insert_Update(r, p, link_type, $G = (V, E), \mathcal{C}$)

begin

if link_type = "very easy link" **then**

add edge (r, p) to the local bcc topology

else if link_type = "easy link" **then**

send *Add_Edge*$((r, p))$ to p, and wait for acknowledgement

else if type = "path condensation" **then**

let $X = \bigcup_{a \in V}(\mathcal{C}(a) \cap V)$ (i.e. X is the multiset of minimum vertices from each bcc in G)

send *Adjust_Mapping*(V) message to each element in X once

compute G_{new} to be the same as G with the addition of the new edge (r, p)

for each $a \in V$, $\mathcal{C}_{\text{new}}(a) \leftarrow \mathcal{C}(a) \setminus X \cup \{\min(X)\}$

if $|\mathcal{C}_{\text{new}}(a)| = 1$ **then** $\mathcal{C}_{\text{new}}(a) \leftarrow \emptyset$

send *Coord_Update*$(G_{\text{new}}, \mathcal{C}_{\text{new}}, X)$ to the coordinator (minimum vertex in V)

wait for acknowledgement

else if type = "component link" **then**

send *New_Bcc*$(\{r, p\})$ to both r and p, and wait for acknowledgements

end Insert_Update

Figure 6: Update procedure for edge insert.

- On receiving *Add_Edge*$((r,p))$:

  Let x be the coordinator of the bcc containing both r and p
  if $(n = x)$
  then update local bcc topology with the edge (r,p)
  else send *Add_Edge*$((r,p))$ to x, and wait for an acknowledgement
  send acknowledgement to the sender

- On receiving *Bcc_Update*$(G = (V,E))$:

  if there is a set W in the local bcc topology such that $W \subseteq V$ **or** $V \subseteq W$ **then**
  replace W by V (or the graph corresponding to W by G)
  else
  add G as a new bcc in the local bcc topology
  send acknowledgement to the sender

- On receiving *Coord_Update*$(G = (V,E), \mathcal{C}, X)$:

  adopt G and $\mathcal{C}$ as the new local bcc topology and the coordinator mapping
  send *Bcc_Update*(G) to all other nodes v in V, and wait for acknowledgements
  for each articulation point a in G
  send *Note_Coord*(a,n,X) to the coordinators for other bcc's containing a
  wait for an acknowledgement
  send acknowledgement to the sender

- On receiving *New_Bcc*$(\{r,p\})$:

  add $\{r,p\}$ to the local bcc topology
  let $i = \min(r,p)$ and $j = \max(r,p)$
  if $n = j$ **then**
  send *Coord_Set*(S_j) to i, where S_j = set of minimum vertices of bcc's containing j
  if j is the coordinator for some bcc **then** $\mathcal{C}(j) \leftarrow S_j$
  else $(n = i)$
  $\mathcal{C}(n) \leftarrow S_i$, where S_i = set of minimum vertices of bcc's containing i
  wait for *Coord_Set*(S_j) message from j
  $\mathcal{C}(j) \leftarrow S_j$
  for each bcc B containing n
  send *Note_Coord*$(n,i,\emptyset)$ to the coordinator of B, and wait for acknowledgement
  send acknowledgement to the sender

- On receiving *Adjust_Mapping*(V):

  if n is coordinator for some bcc $B = (V_B, E_B)$ such that $V_B \not\subseteq V$ **then**
  $\mathcal{C}(a) \leftarrow \emptyset$ for all $a \in V \setminus \{n\}$
  else
  $\mathcal{C}(a) \leftarrow \emptyset$ for all $a \in V$

- On receiving *Note_Coord*(a,c,W):

  Let $\mathcal{C}$ be the coordinator mapping of n
  $\mathcal{C}(a) \leftarrow (\mathcal{C}(a) \setminus W) \cup \{c\}$
  if $n \notin \mathcal{C}(a)$ **then** $\mathcal{C}(a) \leftarrow \mathcal{C}(a) \cup \{n\}$

Figure 7: Message handlers for the Insert_Update procedure for the node n.

1. In the case of a very easy link, the coordinator updates its local bcc topology and sends a *done* message to the requesting node informing it that the update is complete. In case of an *Easy_Link* message from the peer, the coordinator sends an *Add_Edge* message to the peer. The peer forwards this message to the coordinator node of the bcc that contains both the requesting node and the peer node. The local bcc topology of that coordinator is updated and the message is acknowledged. When the coordinator of the requesting node receives an acknowledgement from the peer, it sends a *done* message to the requesting node.

2. When the Insert_Classify procedure returns "component link," the coordinator sends a *New_Bcc*(G) message to both the requesting node and the peer, where G is the new bcc containing just the requesting node and the peer node. Both nodes update their local bcc topology to include G.

   Let i and j be the minimum and maximum nodes, respectively, in the new 2-node bcc. The minimum node i becomes the coordinator for the new bcc. Node j sends a *Coord_Set*(S) message to i, where S is the set of minimum vertices of the bcc's containing j. Further, the node j sets the mapping for j to S_j, if j is the coordinator for some bcc. Node i modifies its coordinator mapping by setting $\mathcal{C}(i)$ to the set of minimum vertices of the bcc's containing i, and letting $\mathcal{C}(j)$ be the set S received from node j. Both nodes then inform their neighboring coordinators that i has become the coordinator of the new bcc G (the *Note_Coord* message). All messages are acknowledged. When the coordinator of the requesting node receives an acknowledgement from the peer and the requesting node, it sends a *done* message to the requesting node.

3. In case of path condensation, the coordinator receives the graph G_1 and the accumulated mapping $\mathcal{C}_1$ from the classification procedure. The coordinator sends an *Adjust_Mapping*(V) message to the coordinators of each bcc that are being "merged", where V is the set of nodes in the new bcc. The node receiving this message nullifies the coordinator mapping if it is no longer the coordinator for any bcc. On the other hand, if the node receiving *Adjust_Mapping*(V) is a coordinator for some other bcc, then it nullifies the mapping for all articulation points in V except itself, because the mapping for this node would contain minimum vertices from other bcc's containing this node. The coordinator then sends a *Coord_Update*($G = (V, E)$,$\mathcal{C}$) to the coordinator c' of the newly formed bcc (c' is the minimum vertex in V), where G is the

local bcc topology of the new bcc computed from G_1 by adding the new edge (r, p), and $\mathcal{C}$ is the new coordinator mapping computed as follows. For an articulation point a in G, $\mathcal{C}(a)$ is obtained by adding c' to the set given by $\mathcal{C}_1(a)$ after removal of minimum vertices from each of the bcc's in G'. The coordinator c', on receiving the *Coord_Update* message, replaces its local bcc topology and the coordinator mapping with G and $\mathcal{C}$ respectively. It also sends a *Bcc_Update*(G) message to all other nodes in the newly formed bcc and sends a *Note_Coord* message to all the neighboring coordinators informing those nodes that it has become the coordinator of the new bcc G. Each message is acknowledged by the receiver, after which the coordinator for G sends an acknowledgement to the requesting node's original coordinator which informs the requesting node that the update is complete.

Handling topology update messages: While inserting an edge in the path condensation case, a *Bcc_Update*$(G = (V, E))$ message is sent to nodes in the new (larger) bcc to update their bcc set, that is a subset of V, with the (larger) set V. Similarly, we will see that when such an edge is deleted resulting in many smaller bcc's, the nodes in these smaller bcc's receive *Bcc_Update*$(G = (V, E))$ message and they replace a superset of V with V. Thus, whenever a node receives a *Bcc_Update*(V) message, it removes any subsets or supersets of V from its local bcc topology and adds the set V to its local bcc topology.

Remark: After the update stage, every node knows the set of vertices that are in the same bcc as its vertex and coordinator nodes know their local bcc topology and the set of coordinators for neighboring bcc's (through the mapping).

4.2 Edge Delete

4.2.1 Coordinator notification stage

The requesting node sends a *Req_Update* message to its coordinator, which calls the delete procedure (see Figure 8).

4.2.2 Classification stage

The purpose of the classification stage is to determine whether the edge to be deleted is a bridge edge or not. The classification procedure is invoked from the delete procedure by the requesting

```
procedure Delete(r, p), where r is the requesting node and p is the peer
    begin
        link_type = Delete_Classify( r, p )
        Delete_Update( r, p, link_type )
        send done message to the requesting node
    end Delete
```

Figure 8: Procedure for edge delete.

node's coordinator. In the classification procedure (see Figure 9), the coordinator checks if $\{r, p\}$ is a bcc in its local bcc topology. In this case, it returns "bridge edge." If r and p are in some bcc of the coordinator, since $\{r, p\}$ is not a bcc by itself, the classification procedure returns "internal edge." Otherwise, the edge is outside of the requesting coordinator's local bcc topology. So the coordinator sends a *Check_For_Bridge*($\{r, p\}$) message to p. The procedure returns "bridge edge" if the coordinator receives *Is_Bridge* from p, and returns "external edge" if it receives *Not_Bridge* from p.

```
procedure Delete_Classify(r, p) returns link_type
    begin
        if {r, p} is a bcc in the coordinator's local bcc topology then return "bridge edge"
        if r and p are in a same bcc in the coordinator's local bcc topology then return "internal edge"
        send Check_For_Bridge({r, p}) message to p, and wait for acknowledgement
        if an Is_Bridge message is received then return "bridge edge"
        if a Not_Bridge message is received then return "external edge"
    end Delete_Classify

• On receiving Check_For_Bridge(G) from b
        if G is a member of the local bcc topology
            then send Is_Bridge(G) to b
            else send Not_Bridge(G) to b
```

Figure 9: Classification procedure for delete and its message handler.

procedure Delete_Update(r, p, link_type), where r is the requesting node and p is the peer

begin

if link_type = "bridge edge" **then**

send *Delete_Bcc*($\{r, p\}$) message to the requesting node and the peer

if link_type = "internal edge" **then** send *Delete_Edge*(r, p) to self

if link_type = "external edge" **then** send *Delete_Edge*(r, p) to p

wait for acknowledgements

end Delete_Update

- On receiving *Delete_Bcc*($\{r, p\}$):

  remove $\{r, p\}$ from the local bcc topology

  let $i = \min(r, p)$ and $j = max(r, p)$

  if $(n = i)$ **then** $\mathcal{C}(j) \leftarrow \emptyset$

  if $(n = j)$ **then** $\mathcal{C}(i) \leftarrow \emptyset$

  for each bcc B containing n

  send *Remove_Coord*(n, i) to coordinator of B

  wait for acknowledgements

  send acknowledgement to the sender

- On receiving *Delete_Edge*(r, p):

  let x be the coordinator of the bcc containing both r and p

  if $(n = x)$ **then**

  remove edge (r, p) from the local bcc topology

  locally compute the bcc's in the modified graph

  if new bcc's are formed **then**

  for each new bcc $G = (V, E)$, send *Coord_Update*$(G, \mathcal{C}, V)$ to coordinator of G,

  where $\mathcal{C}$, the mapping for G is computed by

  (1) projecting the old mapping on V

  (2) adding the coordinators for the newly formed neighboring bcc's, and

  (3) removing the old coordinator where it is no longer a neighbor

  wait for acknowledgements

  else send *Delete_Edge*(r, p) to x, and wait for an acknowledgement

  send acknowledgement to the sender

- On receiving *Remove_Coord*(a, x):

  Let $\mathcal{C}$ be the coordinator mapping of n

  $\mathcal{C}(a) \leftarrow \mathcal{C}(a) \setminus \{x\}$

  if $|\mathcal{C}(a)| = 1$ **then** $\mathcal{C}(a) \leftarrow \emptyset$

Figure 10: Update procedure for edge delete and its message handlers for node n.

4.2.3 Update stage

The procedure for the update stage of edge deletion and associated message handlers are shown in Figure 10. If the edge to be deleted is a bridge, then the coordinator sends a *Delete_Bcc*(G) message to both the requesting node (which may be itself) and the peer node, where G is the bcc to be deleted. On receipt of the *Delete_Bcc* message, a node removes the bcc from its local bcc topology information, removes the entry for the other node from its mapping, and sends a *Remove_Coord* to the coordinators in each bcc containing itself. The node receiving this message adjusts its coordinator mapping by removing the minimum node from the set of coordinators for the sender and if its mapping becomes a singleton set, then it is no longer an articulation point and hence sets its mapping to empty, and sends an acknowledgement to the sender. The sender then acknowledges the requesting coordinator. When the requesting coordinator receives acknowledgements from both the requesting node and the peer node, it informs the requesting node that the update is complete.

If the edge being deleted is not a bridge, the coordinator sends a *Delete_Edge*(r, p) message to itself in case of an "internal edge," and to the peer node in case of an "external edge." The peer forwards this message to the coordinator of the bcc that contains both the requesting node and the peer node. The coordinator node (that receives the *Delete_Edge*(r, p) message) removes the edge from the local bcc topology, locally runs a sequential bcc algorithm over this subgraph and computes the local bcc topologies corresponding to this bcc (containing the requesting node and the peer node), and sends the *Coord_Update* messages to the coordinators of each new bcc that is formed. On receipt, each *Coord_Update* of these messages is handled as described for edge insertion. After receiving acknowledgements of these messages, the coordinator informs the requesting node that the update is complete.

4.3 Correctness

Let S be a sequence of graph update requests for a graph containing vertex set V such that for each pair of vertices (u, v), the subsequent requests involving (u, v) in S is a (possibly empty) prefix of an alternating sequence of *insert* and *delete* beginning with *insert*. Let G be the result of applying S to an initial graph containing vertex set V and no edges. Consider a distributed system of nodes, one for each vertex in V, executing the serial algorithm. Suppose that the environment issues the sequence S of requests, and that each request is made at one of the two endpoints of the affected

edge. We say that the system is in a *consistent global state* after S if for each node n in the system (1) the coordinator c of n knows c's local bcc topology in G, (2) the coordinator of n knows the set of neighboring coordinators v in G, and (3) n knows the set of vertices in each of the biconnected components containing n in G.

The proof that the system is always in a consistent global state proceeds by induction on the length of S. The base case (no requests) is trivial since each node is in a connected component by itself. Initially each node has as its local topology the graph consisting of just itself, and each node in the coordinator mapping is mapped to the empty set.

The inductive step is by case analysis, showing that for each possible insert or delete request, the algorithm leaves the system in a new consistent global state. For insert requests, the cases are easy link, component link, and path condensation. For delete requests, the cases are a bridge edge and a non-bridge edge. For each case, we argue directly from the algorithm that if the request is processed starting from a global consistent state, then it will result in a global consistent state. The argument for termination is straightforward and is based on the acknowledgements of messages. We now present the arguments in more detail.

Lemma 7: The algorithm starts in a consistent global state for the empty sequence of requests.

Proof: Initially, all the nodes are in a component by themselves, their local bcc topologies contain a graph with just the vertex corresponding to that node, and each node in the coordinator mapping is mapped to an empty set. ■

Lemma 8: If the system is in a consistent global state for a sequence S, then the coordinator mappings in each connected component define a tree spanning all bcc's in that connected component.

Proof: Using the coordinator mapping of a bcc we can determine the set of coordinators of the neighboring bcc's. Thus, the Lemma follows immediately from Corollary 4. ■

Lemma 9: If the system is in a consistent global state for S', then processing an insert request ρ for edge (r, p) leaves the system in a consistent global state for $S = S'\rho$.

Proof: Let the node r be the requesting node. There are three cases. In each case, we identify how the consistent global state must change as a result of the operation, and then identify the messages that are used to achieve those changes.

1. Easy link (and very easy link): When r and p lie in the same bcc, then the coordinator of this bcc updates its local bcc topology and other state information does not change. So the system is in a consistent global state for S.

2. Path condensation: When the search for the peer, in the classification stage, returns a *Return_Peer* message to the requesting coordinator, we have the path condensation case. In this case, after inserting the edge (r, p), the nodes r and p will be in the same bcc as that of all the nodes in the path between them (Property 3). If $r, a_1, \ldots, a_n, p$ was the search path from r to p, then the resulting bcc is computed by *traversed_graph*$(r, a_1, \ldots, a_n, p)$ and adding in the new edge (r, p). This traversed graph is the union of all the bcc's along the search path. The mapping and local bcc topology information of the coordinators of these bcc's change due to the update, and the new coordinator of the new bcc gets a new mapping and new local bcc topology. Further, the neighboring coordinators must be notified about the new neighbor and each node in the new bcc must also update its local bcc topology.

   In the algorithm, the requesting coordinator sends a *Coord_Update*$(G, \mathcal{C}, X)$ message to the new coordinator, where G is the new local topology, $\mathcal{C}$ is the new coordinator mapping, and X is the multiset of coordinators of the old bcc's. From *Return_Peer* messages, we see that if $\Psi = r, a_1, \ldots, a_n, p$ was the search path from r to p, then the graph G is the result of adding the edge (r, p) to the union of the graphs *traversed_graph*(x, y, z), for every three consecutive vertices x, y and z in Ψ. Thus, G is same as the graph *traversed_graph*$(r, a_1, \ldots, a_n, p)$ with the addition of the new edge (r, p), the resulting bcc, by definition (see Section 2). The mapping $\mathcal{C}$ is computed for each node in the new bcc by removing the old coordinators and adding the new coordinator (the minimum node in the bcc). If a link vertex ceases to be an articulation point (see Lemma 6), its mapping reduces to a singleton set. The coordinator mapping is adjusted so that the non articulation points are mapped to empty sets in the mapping. When a *Coord_Update*$(G, \mathcal{C}, X)$ message arrives at the new coordinator, it informs the neighboring coordinators of the possible change in the coordinator (*Note_Coord* message), and sends *Bcc_Update* messages to all the nodes in the new bcc. Since all nodes receiving *Bcc_Update* messages update their local bcc topologies, the insert operation leaves the system in a consistent global state.

3. Component link: When the search returns a *No_Peer* to the coordinator for r, Lemma 8 ensures that the peer does not lie in the connected component of r. Hence, the local topologies of r and p, the data structures of the coordinator for the new bcc being created, and the mappings of the neighboring coordinators of the new bcc are modified.

   In the update stage, when a component link is detected, the requesting coordinator sends a *New_Bcc* message to both r and p. Nodes r and p add the new bcc to their local bcc topology, and the maximum of these nodes sends the set of minimum vertices of the bcc's containing itself to the other node and updates its mapping if it is the coordinator for some bcc . The minimum node, the coordinator of the new bcc, simply updates its mapping for the maximum node using the set received and the mapping for itself as the set of minimum vertices of the bcc's containing itself. Then the maximum and the minimum nodes send *Note_Coord* messages to the neighboring coordinators, informing them about the new bcc and its coordinator. Hence, the system is in a consistent global state after the operation.

■

Lemma 10: If the system is in a consistent global state for S', then processing a delete request ρ for edge (r, p) leaves the system in a consistent global state for $S = S'\rho$.

Proof: Let the node r be the requesting node. There are three cases. In each case, we identify how the consistent global state must change as a result of the operation, and then identify the messages that are used to achieve those changes.

1. Edge (r, p) is not a bridge and new bcc's are not formed as a result of deleting this edge. The coordinator of the requesting and the peer nodes updates its local bcc topology and other state information does not change. So the system is in a consistent global state for S.

2. Edge (r, p) is not a bridge and new bcc's are formed as a result of deleting this edge. This has the reverse effect of path condensation for an insert operation. The mappings of the neighboring coordinators and both the mappings and the local bcc topologies of each new bcc formed are to be recomputed to maintain the global consistent state.

   In the algorithm, the coordinator node for both r and p runs a standard sequential algorithm [1] to compute the local bcc topologies of all nodes within this bcc, and sends a

Coord_Update message to all coordinators of the new bcc's formed as a result of the deletion. Since the *Coord_Update* message to the coordinator for a bcc updates the local bcc topologies of all nodes in its bcc and since the neighboring coordinator information is also updated as a result of receiving this message, the system is left in a consistent global state after the operation.

3. Edge (r, p) is a bridge. The local bcc topologies and the mappings (if any) of both r and p and the mappings of the neighboring coordinators of r and p change as a result of the operation.

   In the update stage, the requesting coordinator sends a *Delete_Bcc* message to r and p. A node receiving a *Delete_Bcc* message removes the bcc corresponding to this bcc from its local topology and sends a *Remove_Coord* message to the coordinator nodes of other bcc's containing itself. Further, the mapping for the other node is set to empty, as the other node is no longer in the same bcc. Since the coordinator mappings are also updated, the system is left in a consistent global state after the operation.

■

Theorem 11: The serial algorithm solves the biconnected component problem.

Proof: Consider a sequence of update requests, beginning in the initial state. We show by induction on the length of the sequence of requests that the system is left in a consistent global state. The base case of an empty sequence of requests is given by Lemma 7. From Lemma 9 and Lemma 10, we have the inductive step. Since after any sequence of requests, each node knows the set(s) of nodes in its bcc(s), the serial algorithm solves the biconnected component problem. ■

4.4 Complexity

Messages: While inserting an edge, the connected component of the requesting node is searched for the peer in order to classify the new edge. Hence, the classification stage takes $O(c)$ messages, where c is number of bcc's in the connected component. Also, during the update stage, each node in the new bcc is sent a *Bcc_Update* message informing it about its new local bcc topology, and the neighboring coordinators are informed about any changes in coordinator information (*Note_Coord* message) as a result of an operation. Since the number of neighboring bcc's is bounded by $O(c)$,

the update stage takes $O(b+c)$ messages, where b the number of nodes in the resulting biconnected component. Thus, insert edge operation takes $O(b+c)$ messages.

While deleting an edge, the classification stage takes $O(1)$ messages. During the update stage, if new bcc's are formed, then every node in the new bcc is sent a *Coord_Update* message and the neighbors of each bcc are informed about the new coordinators formed. Hence, the update stage takes $O(b'+c)$ messages, where b' is the number of nodes in the biconnected component in which the update request is being processed. Thus, an edge removal takes $O(b'+c)$ messages.

Time: We measure the time complexity in terms of units of message transmission time. Since the new coordinator notification messages to the neighboring coordinators take $O(1)$ time and since the search is carried out through out the connected component, an insert operation takes $O(c)$ time, where c is the number of biconnected components in the connected component. Since the classification stage of the delete need not search for the peer, a delete operation takes $O(1)$ time. Note that the update stage of an operation takes constant time because the messages can be processed concurrently.

5 Concurrent Algorithm

In the concurrent algorithm, we allow the environment to issue multiple simultaneous requests. The algorithm serializes these requests within each connected component using a timestamping technique that uses logical clocks [9]. In order to allow at most one update request to proceed within a connected component at any one time, we maintain a queue of requests (along with their timestamps) at all coordinator nodes. As connected components merge and split, these queues are updated to achieve a consistent view of the order at all coordinators in a connected component. For each request, we collect timestamps from all the nodes in its connected component and use the maximum value as the priority value of the request. This final timestamp is then communicated to all the other nodes in the connected component. This ensures that all nodes see the same sequence of requests.

Building on the serial algorithm, we add a *timestamp collection* stage, that immediately follows the coordinator notification stage and serves to serialize the requests within a connected component. We use logical clocks [9] to assign a time to each request, and requests are processed in timestamp

Message type	Purpose
Request_Timestamp	requests maximum timestamp from the subtree of the receiver
Return_Timestamp	returns to the parent, the maximum of all timestamps collected
Final_Timestamp	informs coordinators of the final timestamp for a request
Request_Peer_Timestamp	a request to timestamp the current update request at the peer's component
Peer_Ready	the peer coordinator informs the requesting coordinator to start the search
New_Queue	the receiver is informed of the modified Q_{req} as a result of the previous operation

Figure 11: Serialization message summary.

order. As in the serial algorithm, each request is initially handed over to the coordinator by the requesting node. The coordinator collects timestamps from all the coordinators in its connected component and uses the maximum as the timestamp for this request. The coordinator then sends messages to other coordinators about the new timestamp for its request. Once a request becomes the oldest unprocessed request in a connected component, the corresponding coordinator runs the serial algorithm to complete the operation. However, if the request is for an insert operation and the peer is in a different biconnected component from the requesting node during timestamp collection, then timestamps are collected in both components and the maximum is used. The summary of message types is given in Figure 11.

Each coordinator c maintains the following additional state variables: *clock* is an integer variable with the semantics of Lamport's logical clock, Q_{req} is a timestamp ordered priority queue of requests. For each request ρ, coordinator c maintains $parent\_ts(\rho)$ as the parent of c during the timestamp collection stage for request ρ, $children\_ts(\rho)$ as the set of children of c during the timestamp collection stage for ρ, and a timestamp $max\_time(\rho)$. The priority queue Q_{req} is used to order the concurrent requests made by the environment, *parent_ts* and *children_ts* are used in constructing broadcast trees during timestamp collection, and $max\_time(\rho)$ contains the timestamp value to be assigned for the request ρ.

Note that we use the value of the *clock* variable at each node to determine the final timestamp of a request. Since the timestamps are collected from all coordinators in a connected component, it is guaranteed that a later request will have a larger timestamp, and since the final timestamps

are broadcast to all coordinators, all the coordinators will have the same sets of requests pending in the connected component.

As connected components merge and split, the queues of the coordinators must be updated so that they have, at all times, the same set of requests pending in the connected component. (In fact, it is possible that the coordinator that was originally notified about the request is not ultimately the coordinator that processes the request, due to intervening topology updates.) This bookkeeping requires an extension to the update stage of the algorithm. We begin by describing the timestamp collection stage and then describe the extension to the update stage.

5.1 Timestamp collection stage

procedure Timestamp(R), where R is a request

begin
insert $(R, clock)$ in Q_{req}
send *Request_Timestamp*(R) to all neighboring coordinators
receive *Return_Timestamp*(R, t) from all neighbors
change the key for R in Q_{req} to *max_time*(R), the maximum of all t values received
send *Final_Timestamp*(R, *maxtime*(R)) message to all neighboring coordinators
end Timestamp

Figure 12: Timestamp procedure.

In describing the timestamp collection stage, we assume that no update is in progress in the connected component during timestamp collection, and so the topology of the connected component is stable (does not change). Later we explain how we ensure that this is indeed the case. Timestamp collection is carried out in three phases: the request phase, the return phase, and the report phase. The timestamp collection procedure and the associated message handlers are given in Figures 12 and 13 respectively.

1. In the **request phase**, the requesting coordinator for request r sends a *Request_Timestamp*(r) message to all the coordinators in its set of neighboring coordinators. A coordinator receiving its first *Request_Timestamp*(r) message for r enqueues the request r along with a timestamp greater than its current logical clock. It remembers the sender of the message as its parent

for that request. It then forwards this message away from its parent, creating a broadcast tree of all the coordinators in the connected component.

2. When a coordinator node receives *Return_Timestamp*(r, t) messages from all of its children, it enters the **return phase** of the algorithm and sends a *Return_Timestamp*(r, t') message to its parent, where t' is the maximum of all the timestamps received from its children and the timestamp assigned by itself to request r.

3. The **report phase** begins when the requesting coordinator receives a *Return_Timestamp*(r, t) from each of its children. It computes the maximum t' of all the timestamps and its own logical clock. It then enqueues (r, t') in its priority queue, sets its logical clock to t', and broadcasts the timestamp value by sending the message *Final_Timestamp*(r, t') to the neighboring coordinators. Any coordinator receiving *Final_Timestamp*(r, t') adjusts its clock forward to a value greater than t', replaces the corresponding entry for r in its priority queue, and continues the broadcast by forwarding the message to all neighboring coordinators away from its *parent*.

When an entry (r, t) is at the front of the queue, the coordinator for r sends a *Request_Timestamp*(r, t) to the peer. The peer forwards the request to its coordinator. If the coordinator for the peer does not already have an entry for this request operation in its queue, it must run the above three phase algorithm to assign a timestamp for this request in its connected component using, as its final value t', the maximum of t and the timestamps collected. It then sends a message *Return_Timestamp*(r, t') to the requesting node's coordinator. The requesting node's coordinator then updates this entry in its queue. If request r is no longer at the front of the queue, the requesting node's coordinator repeats the "Report phase" of the timestamping step of the serialization stage with the new final time so that other nodes in the connected component can update their priority queues.

When the entry for a request reaches the front of the queue at the peer coordinator, the coordinator sends a *Peer_Ready* message to the requesting node's coordinator. When the entry for a request reaches the front of the requesting coordinator's queue and the requesting coordinator has received a *Peer_Ready* message for that request, then the current coordinator for the requesting node can begin processing the actual request.

- On receiving *Request_Timestamp*(R) from b:

  parent_ts$(R) \leftarrow b$
  max_time$(R) \leftarrow$ *clock*
  insert $(R,$ *max_time*$(R))$ in Q_{req}
  children_ts$(R) \leftarrow$ set of neighboring coordinators that are not neighbors of *parent*
  if *children_ts*$(R) \neq \emptyset$
    then send *Request_Timestamp*(R) to each node in *children_ts*(R)
    else send *Return_Timestamp*$(R,$ *max_time*$(R))$ to *parent_ts*(R)

- On receiving *Return_Timestamp*(R,t) from node b:

  children_ts$(R) \leftarrow$ *children_ts*$(R) \setminus \{b\}$
  max_time$(R) \leftarrow \max($*max_time*$(R),t)$
  if *children_ts*$(R) = \emptyset$ **then**
    send *Return_Timestamp*$(R,$ *max_time*$(R))$ to *parent_ts*(R)

- On receiving *Final_Timestamp*(R,t) from node b:

  change key for request R in Q_{req} to t
  send *Final_Timestamp*(R,t) to all neighboring coordinators except sender

- On receiving *Request_Peer_Timestamp*(R,c) from b:

  if self is the coordinator for the peer **then**
    if entry for R is in the front of Q_{req} **then**
      send *Peer_Ready* to c
    else
      Timestamp(R)
      wait for R to reach the front of Q_{req}, and then send *Peer_Ready* to c
  else
    send *Request_Peer_Timestamp*(R,c) to the coordinator for self

- On receiving *New_Queue*(q) from b:

  $Q_{\text{req}} \leftarrow q$
  send *New_Queue*(q) to all neighboring coordinators except b, and wait for acknowledgements
  send acknowledgement to the sender

Figure 13: Message handlers for timestamp collection.

5.2 Extended Update

A coordinator receiving an update message forwards any "new" requests to the new coordinator if that has changed and waits for an acknowledgement before acknowledging the update. Note that a coordinator c' receiving an update can determine the new coordinator of a node for which it was the coordinator. In case of a path condensation insert case or a delete edge where new bcc's are created, c' the new coordinator for a node is determined from the graph in the message. In case of a component link insert case or a delete bridge, the new coordinator is determined from the coordinator mapping.

After the classification and update stages of the serial algorithm are completed, the entries in the queues must be updated if two connected components have merged (due to a component link insertion) or if a connected component has split (due to a bridge delete).

In the case of a component link, the coordinator for the requesting node gets the priority queue from its old coordinator of the peer, merges it with its own priority queue, and broadcasts this combined queue q in a *New_Queue*(q) message to all the coordinators of the connected component. Acknowledgements are collected with the usual convergecast mechanism.

In the case of a bridge delete, the requesting node and the peer node inform their respective (new) coordinators that a bridge edge has been deleted. Each coordinator then collects, in a broadcast/convergecast over the tree of coordinators, the set of pending requests in the connected component (with their timestamps). Each coordinator reports the requests for which it is either the requesting coordinator or the peer coordinator. These are collected in the convergecast into a new queue that is broadcast in a *New_Queue* message to all the coordinators in each respective connected component.

5.3 Stable Intervals

In order to ensure that a timestamp was collected from every coordinator in the connected component, we made the assumption earlier that timestamp collection is performed only while no update is in progress. In order to achieve this, we add to the end of the update stage a *stable* message that is broadcast to all the coordinators after all the other messages from the update stage have been acknowledged.

When receiving a *stable* message, a coordinator can initiate timestamp collection, knowing that

no further updates will proceed in the component until it has completed its timestamp collection and acknowledged the *stable* message. However, when the queue is empty, a coordinator cannot wait for a *stable* message before collecting timestamps, because there is no update in progress that can result in such a message. Therefore, in case of an empty queue, a coordinator may immediately begin timestamp collection without receiving a *stable* message. If more than one coordinator does this, then the one with the lowest timestamp "wins" and it may start its update, while the others discard the results from that collection and wait for the *stable* message at the end of the winning coordinator's update.

If a coordinator p is waiting for a *stable* message, it may send a *stable_request* message to the requesting node for the first request in the queue, and the requesting node will forward that message to its coordinator. If the coordinator has not yet begun the update stage, then it will send a *stable* message to p, and wait for p to acknowledge the message before proceeding to the update stage. During that stable interval, p may carry out timestamp collection. (This provision is needed to prevent a deadlock situation that could otherwise occur due to a cyclic waiting chain for timestamp collection by peer coordinators. For example, two bridge edges may be added between a pair of components with their requesting nodes in different components. Without this provision, if both requests reach the tops of the queues in the two components, then the peer nodes would never receive a *stable* message and so would be unable to carry out timestamp collection.)

When a coordinator receives acknowledgements for all of the *stable* messages (through a convergecast), it then broadcasts to all the coordinators to inform them that the update is complete, and they remove the request from their queues.

5.4 Correctness

Since each request has one final timestamp and that timestamp is known to all coordinators, at most one update request is in progress at a time in each connected component. Therefore, the correctness of the serial algorithm implies the correctness of the concurrent algorithm.

Liveness (no starvation) is argued by showing that (1) every request is eventually assigned a timestamp, (2) once a request has been assigned a time in both the requesting and peer components, no later request will be assigned a smaller timestamp in either component, and (3) once a request reaches the top of the queue in the components of both the peer and the requesting node, then it

cannot be blocked.

5.5 Complexity

Messages: The serialization of a request involves a broadcast and convergecast over the connected components in which the endpoints are members. The actual search for the peer may take place in a modified connected component, so the message complexity is proportional to the maximum number of bcc's in any of these instances. Thus, the algorithm takes $O(b + c)$ messages for an insert and $O(b' + c)$ for a delete, where c is the maximum number of bcc's in any of the connected components during the operation, b is the number of nodes in the resulting bcc, and b' is the number of nodes in the bcc before the deletion.

Time: The serialization stage of the algorithm takes $O(c)$ time, where c is the maximum of the number of bcc's in either the requesting node's connected component or the peer's connected component. Since the coordinator that timestamped the request need not be the coordinator that actually performs the request, the algorithm takes $O(c)$ time for both an insert and a delete operation, where c is the maximum number of bcc's in any of the connected components during the operation.

5.6 Optimizations

Some simple optimizations on the concurrent algorithm are possible. In particular, there are some kinds of update requests for which timestamp collection can be avoided. Since the coordinator contains the local bcc topology, it can immediately detect the easy link case. Since the easy link insert case does not change the set of articulations points or coordinators, the coordinator can update the topology during the stable interval safely. Similarly, the coordinator can detect when deleting an edge will not change the set of articulation points or coordinators. In practice, for the causal ordering application, we could be lazy about the delete requests, processing them only when bcc's grow large.

6 Conclusion and Future Work

We have presented two incremental distributed algorithms for computing biconnected components in a dynamically changing graph. The serial algorithm requires that the environment issue only one update request at a time. The concurrent algorithm allows the environment to make multiple concurrent update requests. The algorithm serializes the requests within each connected component with each node in a connected component having the same view of the update sequence, while allowing requests in different connected components to proceed in parallel. The algorithm uses logical clocks and collects timestamps from nodes in a connected component in order to achieve identical view of the update sequence across nodes. As the graph changes dynamically, ordering information is propagated to ensure consistency.

We are working on a new concurrent algorithm in which multiple updates may proceed concurrently within a connected component. The requesting nodes unilaterally assign timestamps to the requests and the nodes prioritize the requests according to these times. Whenever messages belonging to two requests "collide" at a node, either the younger request may be deferred or the requests may be combined into one update. For an example of the latter, if two or more insert requests find their peers within a connected component and collide, then these requests may merge to form one large biconnected component.

Acknowledgements

We thank George Varghese for his careful reading of an earlier draft.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffery D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[2] Mohan Ahuja and Yahui Zhu. An efficient distributed algorithm for finding articulation points, bridges, and biconnected components in asynchronous networks. In *Proceedings of the 9th Conference on Foundations of Software Technology and Theoretical Computer Science, Bangalore, India. LNCS 405*, pages 99–108. Springer–Verlag, December 1989.

[3] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[4] E. J. H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, 8(4):391–401, 1982.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[6] Jerome R. Cox, Jr., Mike Gaddis, and Jonathan S. Turner. Project Zeus: Design of a broadband network and its application on a university campus. *IEEE Network*, pages 20–30, March 1993.

[7] Kenneth J. Goldman, Michael D. Anderson, and Bala Swaminathan. The Programmers' Playground: I/O abstraction for heterogeneous distributed systems. In *27th Hawaii International Conference on System Sciences (HICSS)*, pages 363–372, January 1994.

[8] Walter Hohberg. How to find biconnected components in distributed networks. *Journal of Parallel and Distributed Computing*, 9(4):374–386, August 1990.

[9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 27(7):558–565, July 1978.

[10] Jungho Park, Nobuki Tokura, Toshimitsu Masuzawa, and Kenichi Hagihara. Efficient distributed algorithms solving problems about the connectivity of network. *Systems and Computers in Japan*, 22(8):1–16, May 1991.

[11] Monika Rauch. Fully dynamic biconnectivity in graphs. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 50–59, October 1992.

[12] Robert E. Tarjan and Uzi Vishikin. An efficient parallel biconnectivity algorithm. *SIAM Journal of Computing*, 14(4):862–874, 1985.

[13] Jeffery Westbrook and Robert E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7:433–464, 1992.