[Nis90] Naomi Nishimura. Asynchronous shared memory parallel computation. In *Proceedings of 2nd ACM Symposium on Parallel Algorithms and Architectures*, 1990. To appear.

[PF77] Gary L. Peterson and Michael J. Fischer. Economical solutions for the Critical Section Problem in a distributed system. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pages 91–97, May 1977.

[PU87] Christos H. Papadimitriou and Jeffrey D. Ullman. A commmunication-time tradeoff. *SIAM Journal on Computing*, 16(4):639–646, August 1987.

[PY88] Christos H. Papadimitriou and Mihalis Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 510–513, 1988.

[SV82] Yossi Shiloach and Uzi Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[Vis84] Uzi Vishkin. An optimal parallel connectivity algorithm. *Discrete Applied Mathematics*, 9:197–207, 1984.

[Wyl79] James C. Wyllie. *The Complexity of Parallel Computation.* PhD thesis, Cornell University, August 1979. Technical report number TR 79-387, Department of Computer Science.

[CZ89] Richard Cole and Ofer Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proceedings of 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, 1989.

[CZ90] Richard Cole and Ofer Zajicek. The expected advantage of asynchrony. In *Proceedings of 2nd ACM Symposium on Parallel Algorithms and Architectures*, pages 85–94, 1990.

[CZ91] Richard Cole and Ofer Zajicek. The APRAM – The Rounds Complexity Measure and the Explicit Costs of Synchronization. Computer Science Department Technical Report No. 539, Courant Institute, New York University, 1991.

[Gaz86] Hillel Gazit. An optimal randomized algorithm for finding connected components in a graph. In *Proceedings of the 27th Annual Symposium on the Foundations of Computer Science*, pages 492–501, 1986.

[Gib89] Phillip B. Gibbons. Towards better shared memory programming models. In *Proceedings of 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168, 1989.

[HW90] Yijie Han and Robert A. Wagner. An efficient and fast parallel connected component algorithm. *Journal of the ACM*, 37(3):626–642, July 1990.

[HCS79] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, August 1979.

[KRS88a] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. In *Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, pages 333–346. Springer-Verlag, July 1988.

[KRS88b] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. Technical Report RC 13572, International Business Machines, 1988.

[LF81] Nancy A. Lynch and Michael J. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13:17–43, 1981.

[MPS89] Charles Martel, Arvin Park, and Ramesh Subramonian. Optimal asynchronous algorithms for shared memory parallel computers. Computer Science and Engineering CSE-89-8, University of California at Davis, July 1989.

[MSP90] Charles Martel, Ramesh Subramonian, and Arvin Park. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *Proceedings of the 31st Annual Symposium on the Foundations of Computer Science*, 1990.

Theorem 12.1 *Stage 2 completes after* $O(\log n)$ *rounds.*

Proof. Consider any component, C, of the input graph. Immediately after step T_i, the H-depth of C is at most $2|C|$. For any peripheral vertex, v, if v is a root and H-depth$(v) = 2$ or if H-depth$(v) = 1$ then $F(v)$ is a core vertex. It follows from Lemma 12.2 that for some k, $k = O(\log |C|) + T_i$, for any peripheral vertex, u, in C, $F_k(F_k(u))$ is a core vertex ($F_k(u)$ may be a peripheral root). Also, for every core vertex, u, in C, $R_k(u) = l$. In particular, $R_k(l) = l$ and l is promoted by the end of round $k+1$. From this it follows that $F_{k+1}(l) = l$, and the process associated with l terminates during round $k + 1$.

It follows that for some k', $k' \leq k + 4$, all the processes terminate. □

We have shown

Theorem 12.2 (Connectivity) *There is an APRAM algorithm which computes the connected component of an undirected graph in* $O(\log n)$ *rounds using* $n + e$ *processes, where* n *is the number of vertices of the graph and* e *is the number of edges.*

References

[AC88] Alok Aggarwal and Ashok K. Chandra. Communication complexity of PRAMs. In *Proceedings of the 15th International Colloquium on Automata, Languages and Programming*, pages 1–17. Springer-Verlag, July 1988.

[ACS89] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. On communication latency in PRAM computations. In *Proceedings of 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, 1989.

[AFL83] Eshrat Arjomandi, Michael J. Fischer, and Nancy A. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *Journal of the ACM*, 30(3):449–456, July 1983.

[AG87] Baruch Awerbuch and Robert G. Gallager. A new distributed algorithm to find breadth first search trees. *IEEE Transactions on Information Theory*, IT-33(3):315–322, May 1987.

[Awe85] Baruch Awerbuch. Complexity of Network Synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[Awe87] Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 230–240, May 1987.

[CV87] Richard Cole and Uzi Vishkin. Approximate Parallel Scheduling. Part II: Application to Optimal Parallel Graph Algorithms in Logarithmic Time. Technical Report 291, New York University, 1987. to appear, Information and Computation.

v.

Lemma 12.1 *The following holds for any vertex, v, and for any two steps, $T_i < t_1 < t_2$.*

1. *If v is a core vertex then $H_{t_1}(v)$ and $\overline{H_{t_1}(v)}$ are core vertices.*
2. *If $H_{t_1}(v)$ is a core vertex, then $H_{t_2}(v)$ is a core vertex.*
3. *If v is a peripheral vertex then there is an H_{t_1}-path from v to $H_{t_2}(v)$.*

Proof. By Theorem 6.1, once all the edges terminate each component of the input graph corresponds to one component of the p^+-graph, and therefore to one component of the q^+-graph (by Lemma 7.14 (p. 24)). Also, once a leader is promoted during Stage 2 it remains a root (for there are no edges to give hooking suggestions). From this it follows that the only vertex in a q^+-component that can be promoted during Stage 2 is the final leader of the component, which must be a core vertex. In addition, during Stage 2, a vertex v replaces $H(v)$ by $\overline{H(v)}$ and $\overline{H(v)}$ by $H(H(v))$. The lemma follows by induction on the step number. □

Lemma 12.2 *For any vertex, v, and for any round, r, if H-depth$_r(v) \geq 3$ then H-depth$_{r+1}(v) \leq \frac{4}{5} H$-depth$_r(v)$.*

Proof. For any two vertices, x and y, if there is a path in the H-graph from x to y, define the *distance from x to y* to be the difference H-depth$(x) - H$-depth(y).

Let v_1, v_2, v_3 be any three consecutive vertices on some H_{r+1}-path. Then there is a path in the H_r-graph from v_1 to v_3. Consider the H_{r+1}-distance, d, from v_1 to v_3.

$d = 2$**:** Then v_1 is not a root following round $r+1$; so during round $r+1$, v_1 either executed a doubling operation or a hooking operation. Therefore, the distance from v_1 to v_3 in H_r is at least 3.

$d = 3$**:** One of v_1 or v_2 is a root following round $r+1$. If a vertex, u, is a root following round $r+1$, then during the round it must have advanced its next pointer (otherwise it would have hooked). Therefore, the distance from v_1 to v_3 in H_r is at least 4.

$d = 4$**:** Both v_1 and v_2 are roots following round $r+1$. Both v_1 and v_2 advanced their next pointers. Therefore, the distance from v_1 to v_3 in H_r is at least 6.

Thus, taking into account the last vertex on the path, we conclude that if the heaviest H_{r+1}-path has at least two edges, in the worst case, the depth of v was reduced by a at least a factor of $\frac{4}{5}$. On the other hand, if the heaviest H_{r+1}-path has only one edge, H_{r+1}-depth$(v) \leq 2$. The lemma follows. □

Proof. Theorem 11.2 shows that as long as a component of the input graph is in Stage 1 of the algorithm its weight is reduced by at least a constant multiplicative factor, $k = 1 - \frac{1}{34}$.

Consider any component, K, of the input graph. The weight of K is nonnegative. Furthermore, as long as any of the vertices of K are active, the depth of K is at least 2. Initially, the weight of the graph is $(2 + c_s)n < 6n$. Therefore, after $O(\log n)$ rounds all the edges must have terminated and the algorithm is in Stage 2. □

12 Stage II

The second stage of the algorithm begins after all the edges terminate and each component, C, of the input graph comprises exactly one q^+-component. Let T_i denote the step at which the last edge of C terminates. During the second stage the current destination edges of G_{q^+} are replaced by edges of G_q until the vertices of C comprise one q-component. Additional applications of the doubling procedure are performed until all the vertices of C have a common A pointer.

Each component of the q^+-graph has one cycle, possibly a self-loop. We call a vertex a *core vertex* if it is on a cycle of $G^{q+}_{T_i}$; it is called a *peripheral vertex*, otherwise. (Note that whether a vertex is a core vertex or a peripheral vertex is determined by the $q+$-graph at step T_i and does not subsequently change.)

We introduce a function, H, defined as follows: $H(v)$ is equal to $next(v)$ if v is a root; $H(v) = F(v)$ otherwise. Also, let $\overline{H(v)}$ denote the shadow of the corresponding value. Let C be any component of the input graph, and let l be the final leader of the component. Define an auxiliary directed graph, the H-graph, over the vertices of G as follows. For each vertex v, there are two directed edges emanating from v: $\langle v, H(v)\rangle$ and $\langle v, \overline{H(v)}\rangle$.

We assign a weight to each vertex, v; for any step, t, $wt_t(v)$, the *weight of v after step t*, is 2 if v is a root; $wt_t(v) = 1$ otherwise. For a peripheral vertex, v, define an H-path starting at v to be any path in the H-graph starting at v and ending immediately before the first core vertex. For a core vertex, v, an H-path starting at v is defined to be a path in the H-graph starting at v and ending at the first vertex, u, for which $R(u) = l$. The H-weight of an H-path, P, is the sum of the weights of the vertices on the path. The *H-depth of v at round t*, $H\text{-depth}_t(v)$, is the maximum over all H-paths starting at v of the H-weight of the path. For any component, C, define $H\text{-depth}_t(C)$ to be the maximum over all vertices, $u \in C$, of $H\text{-depth}_t(u)$. We show that for any vertex, v, $O(\log |C|)$ rounds after all the edges terminate, if v is a peripheral vertex then $H(v)$ is a core vertex, and if v is a core vertex, $H(v)$ is the final leader, l, where $|C|$ is the number of vertices in the component containing

Theorem 11.2 *For each round, t, and for each component, K, of the input graph, if K is in Stage 1 at round $t+1$ then $weight_{t+1}(K) \leq \frac{33}{34} weight_t(K)$.*

Proof. Recall that the weight of a q-component has two parts: its depth and its auxiliary weight. Let U_t be any natural chain partition of G at the beginning of round $t+1$, let U_{t+1} be any natural chain partition of G at the end of round $t+1$, and let K be any component of the input graph. In Lemma 11.5 we have shown that for each chain, C, of U_t, $tag_{t+1}(C) \geq \frac{1}{10}(F, U_t)\text{-depth}_t(C)$. Therefore,

$$tag_{t+1}(K) \geq \frac{1}{10}(F, U_t)\text{-depth}_t(K).$$

We have also shown that

$$tag_{t+1}(K) \leq (F, U_t)\text{-weight}_t(K) - (F, U_t)\text{-weight}_{t+1}(K).$$

Therefore,

$$(F, U_t)\text{-weight}_t(K) - (F, U_t)\text{-weight}_{t+1}(K) \geq \frac{1}{10}(F, U_t)\text{-depth}_t(K).$$

By Lemma 11.3, the auxiliary weight of each active component of the q-graph is at most $\frac{3}{2}c_s = \frac{24}{5}$. Note that the depth of each such component is at least 2. Therefore,

$$(F, U_t)\text{-depth}_t(K) \geq \frac{10}{34}(F, U_t)\text{-weight}_t(K).$$

Consequently,

$$(F, U_t)\text{-weight}_t(K) - (F, U_t)\text{-weight}_{t+1}(K) \geq \frac{1}{34}(F, U_t)\text{-weight}_t(K),$$

and

$$(F, U_t)\text{-weight}_{t+1}(K) \leq \frac{33}{34}(F, U_t)\text{-weight}_t(K),$$

The auxiliary weight is independent of the chain partition used. It follows from Lemma 10.2 that

$$\text{weight}_{t+1}(K) = (F, U_{t+1})\text{-weight}_{t+1}(K) \leq (F, U_t)\text{-weight}_{t+1}(K).$$

Therefore,

$$\text{weight}_{t+1}(K) \leq \frac{33}{34}\text{weight}_t(K).$$

□

Theorem 11.3 *Stage 1 of the algorithm terminates in* $O(\log n)$ *rounds.*

its component. If the root does not hook and no root hooks to it, all the edges leave the component and it becomes inactive. However, this process may take up to 4 rounds. That is, each root accrual chain may avoid making any progress for three rounds. We account for these chains by artificially reducing the weight of the graph by increasing the debt of the root of the chain.

Lemma 11.6 *An active vertex, v, has nonzero debt only if it is the root of a root accrual chain.*

Proof. By definition, a vertex increases its debt only if it is the root of a root accrual chain. Consider a root chain, C, with root r. If C starts a round, t, as a root accrual chain, the only way it can stop being a root accrual chain is if it either becomes inactive, or, for some vertex u, either r hooks to u or u hooks to r. In the latter two cases the debt of r (and of u) is set to 0 (see Rule 2 in the section describing the spare weight, Section 11.1). □

Definition 11.2 *For any chain, C, and any round, t, let $acc_t(C)$ denote that C is a root accrual chain at round t.*

Lemma 11.7 *For any vertex, v, $db(v) \leq c_d$.*

Proof. Consider any root accrual chain, C, with root r, and let t be the round at which r became a root accrual chain. That is $acc_t(C)\&\neg acc_{t-1}(C)$. Then by Lemma 11.6, $db_t(r) = 0$. Consider what may happen to a root accrual chain which does not hook and is not hooked to.

1. If, $f_C = 2$, then, by the end of the round every edge with endpoint in C will have its endpoint at r and will "know" that r is a root.

2. If at the beginning of a round all the endpoints are at r and they all know that r is a root, then by the end of round r every suggestion written to r (until r hooks or is hooked to) is outside its component.

3. In at most one additional round, r has a current destination outside its component.

4. If, at the beginning of a round, r has a current destination outside of C, then during the round all the edges migrate out of C and C becomes inactive.

It follows that there are at most three rounds in which the debt of r increases. When the debt increases, it increases by $\frac{1}{3}c_d$ (see the final case in the section describing the reduction in weight, Section 11.3). Any sequence of debt increases of C starts with $db(C) = 0$; therefore, $db(C) \leq c_d$. □

If T (resp. C') is a root accrual chain, set its tag to $\frac{1}{16}c_s = \frac{1}{5}$; otherwise, set its tag to $\frac{1}{4}c_s = \frac{4}{5}$. A root accrual chain does not have any chains associated with it; therefore, T, C', and all the chains associated with them receive a tag of at least $\frac{1}{10}$ their original depth.

Let i be the number of root accrual chains among T and C' (i.e. $i \in \{0,1,2\}$). Then the auxiliary weight is decreased by at least $\Delta = \frac{1}{2}c_s - i \cdot c_d = c_s(\frac{1}{2} - \frac{3i}{16})$. The total of the tags assigned by this case is $c_s(\frac{i}{16} + \frac{1}{4}(2-i)) = \Delta$.

- T is a root and T hooks during the round. Then it is assigned a tag of at least $\frac{1}{10}f_T$ by the preceding paragraph.
- T is not a root accrual chain. As T is minimal and it was not associated with any chain, T is terminal. T is not a link destination (otherwise it is covered by the first case). It follows that T is not a potential leader; hence, $f_T = 1$. During the round all the edges leave T and no new edges enter T. Therefore, all the vertices of T become inactive. Set $tag_{t+1}(T) = 1$. Thus, T and each of the chains associated with it receive a tag equal to at least $\frac{1}{7}$ of their original depth.
- T is a root accrual chain. Its depth is 2. Let r denote its root. If T did not receive a tag by any previous rule, increase the debt of r by $\frac{1}{3}c_d$ and set $tag_{t+1}(T) = \frac{1}{3}c_d (= \frac{1}{5})$. The tag assigned to T is at least $\frac{1}{10}$ its original depth.

This, together with Lemma 11.4, which states that the spare weight of the graph never increases, shows:

Lemma 11.5 *For each round number, t, for any natural chain partition, U_t, for each chain, C, of U_t, and for each component, K, of the input graph,*

$$tag_{t+1}(C) \geq \frac{1}{10}(F, U_t)\text{-}depth_t(C) \tag{2}$$

and

$$tag_{t+1}(K) \leq (F, U_t)\text{-}weight_t(K) - (F, U_t)\text{-}weight_{t+1}(K). \tag{3}$$

11.4 The Debt

Consider a root accrual chain. For every component, C, of the q-graph, which is not a component of the input graph, there is an edge with one endpoint in C and one outside. Eventually, one of these edges gives the root a hooking suggestion. As soon as the root acts on this suggestion and updates its *next* pointer the root has a current destination outside

chain associated with it by Rule 4. The sum of the depths of all the chains associated with a given chain is at most 6.

The only minimal chains not covered by any of the five rules above are leader chains which do not have any children chains. We call such leader chains *root accrual chains*. We show below, in Lemma 11.6, that only the roots of root accrual chains can have debts.

11.3 Weight Reduction

Consider round $t+1$. For any chain C let f_C denote (F, U_t)-depth$_t(C)$, the depth of C relative to U_t at the beginning of the round. We assign a tag, $tag_{t+1}(C)$, to C and show that during the round $tag_{t+1}(C) \geq \frac{1}{10} f_C$. For each component, K, of the input graph, let $tag_{t+1}(K)$ denote the sum of $tag_{t+1}(C)$ summed over all chains, C, contained in K. We show that the sum of the tags assigned to chains of K is bounded by the weight lost by K during the round; that is, $tag_{t+1}(K) \leq (F, U_t)$-weight$_t(K) - (F, U_t)$-weight$_{t+1}(K)$. Note that the auxiliary weight of the graph is independent of the chain partition used.

Consider a chain, C. C may have several chains associated with it. When we assign a tag to C it is understood that C shares this tag with all the chains associated with it, each chain receiving a share of the tag proportional to its depth at the start of the round. Let W_C be the sum of the depths of all the chains associated with C, if any (recall that $W_C \leq 6$).

Consider a chain, T, which is not associated with any other chain. The following paragraphs describe the assignment of tags to chains. Each case preserves the property that the tag assigned is accounted for by a reduction in the weight of the component (this can be in one of two forms: a reduction in the depth of the chain or a reduction in the auxiliary weight of the component.) Set $c_s = \frac{16}{5}$ and $c_d = \frac{3}{16} c_s = \frac{3}{5}$. There are two main cases:

T is not minimal. Then, by Theorem 10.1, during the round the depth of T is reduced by at least $\frac{1}{4} f_T$. Set $tag_{t+1}(T) = \lceil \frac{1}{4} f_T \rceil$. After distributing the tag, the tag associated with T is at least $(\lceil \frac{1}{4} f_T \rceil) f_T / (f_T + W_T)$ which is at least $\frac{1}{10} f_T$. Therefore, each chain, C, associated with T receives a tag equal to at least $\frac{1}{10} f_C$.

T is minimal. There are a number of cases:

- T is a link destination of some root, r. Then during the round r hooks to T; let C' be the chain containing r. By Lemma 11.4, the hooking operation reduces the spare weight of the graph by at least $\frac{1}{2} c_s$. The debt, if any, of C' and T is set to 0.

Proof. The hooking operation is the only operation that affects the spare weight. Consider a hooking operation hooking some root, r, to a vertex, v, at some step, t. If v and r are in the same q_{t-1}-component, $\mathcal{C}$, then $sp_{t-1}(\mathcal{C}) = \frac{3}{2}c_s$ (Theorem 11.1 (p. 36)). After the hooking operation $\mathcal{C}$ is not rooted. It follows that $sp_t(\mathcal{C}) = c_s$ (Lemma 11.3).

If v and r are not in the same q_{t-1}-component, the sum of the spare weights of the two components is at least $2c_s$ (Lemma 11.3). After the hooking v and r are in the same component; that component has spare weight at most $\frac{3}{2}c_s$. □

11.2 Assignment of Short Chains

Let t be any round and let U_t be a natural chain partition of G at the beginning of round t. We examine the chains of U_t.

Chains which are either link destinations or have no children chains are called *terminal chains*. Recall that a chain is *minimal* either if it has depth 1 or it is potential leader with depth 2. Minimal chains are associated with other chains according to the following rules. When a chain, C, is associated with a chain, D, that was associated with another chain, D', by a prior rule, it is understood that C is associated with D'. For each chain we use the first rule that applies, and only that rule.

Rule 1: A link destination chain, C, is not associated with any chain.

Rule 2: Each (minimal) potential leader chain which is not a leader chain is associated with one of the chains in its promoting set. Chains associated by this rule have depth 2.

Rule 3: If a chain, C, has exactly one child, D, either C is a link destination, C is a potential leader or D is a potential leader. If C is a link destination apply Rule 1 and if C is a potential leader, apply Rule 2. If D is a potential leader and C is minimal, C is associated with D. Chains associated by this rule have depth 1.

Rule 4: Each minimal depth 1 chain, C, which has 2 children chains is associated with a distinct ancestor terminal. Each terminal can receive at most one chain due to this rule. (Minimal depth 2 chains are potential leaders – these are taken care of either by Rule 2 or by Rule 5.)

Rule 5: A minimal leader chain which has a child chain is associated with that child. Such a chain has depth 2.

Note that each chain can have at most 4 chains associated with it: one potential leader (Rule 2), one parent of a potential leader (Rule 3), the component leader (Rule 5) and one

Lemma 11.2 *For every vertex, v, if v is a cycle destination then $sp(v) = \frac{1}{2}c_s$.*

Proof. Let v and r be two vertices and assume that, at step t, v is a cycle destination of r. As r is a root, it is not a follower (Lemma 7.2); therefore, r is a p_t-ancestor of v (Lemma 7.12).

Let t_1 be the step in which r started the hooking operation it is performing at step t. Then, at some step t_2, $t_1 < t_2 < t$, $\overline{A(r)}$ became equal to $v \neq r$. As v is a root at step t_2, r is not a q-ancestor of v at step t_2. Therefore, r is not a p-ancestor of v at step t_2 (Lemma 7.13). Consequently, at some step, t', $t_2 < t' < t$, a hooking step hooked a p-ancestor, x, of v to a p-descendant, y, of r. It follows from rule (2-a) for spare weight that $sp_{t'}(v) = \frac{1}{2}c_s$ (v is the vertex w of rule (2-a)). Between steps t' and t, r is a p-ancestor of v and r is a root. Therefore, $sp(v)$ does not change between steps t' and t. □

Theorem 11.1 *For any vertex, v,*

$$sp(v) = \begin{cases} c_s & \text{if } v \text{ is a leader} \\ \frac{1}{2}c_s & \text{if } v \text{ is a cycle destination} \\ 0 & \text{otherwise.} \end{cases}$$

Proof. The first two cases follow from Lemma 11.1 and Lemma 11.2, respectively.

We have to show that if a vertex is neither a leader nor a cycle destination its spare weight is 0. We show this by induction on the step number. Consider such a vertex, v, following some step t. It follows from the definition that v did not receive spare weight at step t. If $sp_{t-1}(v) \neq 0$ then, by the induction hypothesis, following step $t-1$, v is either a root or a cycle destination. In either case, as v is not a leader or a cycle destination at step t, either v creates a hook or is hooked to at step t. It follows that $sp_t(v) = 0$. □

Lemma 11.3 *For any q-component, $\mathcal{C}$, if $\mathcal{C}$ is rooted, $c_s \leq sp(\mathcal{C}) \leq \frac{3}{2}c_s$. If $\mathcal{C}$ is not rooted, $sp(\mathcal{C}) = c_s$.*

Proof. A component, C, has one leader. A vertex has at most one link destination; therefore, C has at most one cycle destination (it has a cycle destination only if it is rooted, the root has a link destination, and that link destination is in C). The lemma follows from Theorem 11.1 (p. 36). □

Lemma 11.4 *The spare weight of the graph never increases and each hooking operation reduces the spare weight of the graph by at least $\frac{1}{2}c_s$ units.*

its weight by at least $\frac{1}{4}f_C$ due to the doubling steps performed on all its internal vertices (Theorem 10.1). If $f_C \leq 2$, C is not so fortunate. To account for its weight we use one of three methods: (1) C receives part of a tag associated with some other vertex, D. (2) C uses the reduction in the spare weight (due to a hooking operation), (3) C increases its debt.

11.1 The Spare Weight

We start by defining the auxiliary weight and showing that each hooking reduces the spare weight of the graph.

Definition 11.1 *A vertex, v, is called a* cycle destination of r *if v is a link destination of r and r is the root of the component containing v.*

For any vertex, v, let $sp_t(v)$ denote the spare weight of v at step t and let $db_t(v)$ denote the debt of v at step t. The auxiliary weight is defined as follows:

1. For each vertex, v, $sp_0(v) = c_s$ and $db_0(v) = 0$.
2. The auxiliary weight is modified at the writing step of a hooking operation. Consider a root, r, and assume r hooks to some vertex, u, at step t. Then $db_t(r) = db_t(u) = 0$. In addition,
   (a) if a q_t-descendant, w, of r is a link-destination of a q_t-ancestor of u then $sp_t(u)$ is set to 0 and $sp_t(w)$ is set to $\frac{1}{2}c_s$.
   (b) If u is a q_t-descendant of r, the hooking forms a p-cycle; let l be the leader of the cycle. $sp_t(l)$ is set to c_s and $sp_t(u)$ is set to 0. If $l \neq r$ then $sp_t(r)$ is set to 0 as well.
   (c) Otherwise, $sp_t(r)$ is set to 0.

The debt is defined shortly. For the purposes of this section we state the following assumption which we later prove (Lemma 11.6 and Lemma 11.7).

Assumption. For any vertex, v, if v is not a root then $db(v) = 0$. If v is a root then $0 \leq db(v) \leq c_d$.

Lemma 11.1 *For any vertex, v, if v is a leader $sp(v) = c_s$.*

Proof. The proof is by induction on the step number. Initially, the lemma holds by definition. A root loses its spare weight only when it ceases being a root. When a hook creates a cycle, the leader of the new cycle has spare weight c_s by definition. □

Now, suppose C_v is a potential leader chain. Recall that C_v comprises one span and, therefore, all the vertices in C_v are followers of l, the potential leader vertex. (Note that l may be a leader, or even a root.) Of the vertices of C_v, l is the only vertex that can be promoted. The remaining vertices execute a doubling operation during round $r+1$. An argument similar to the one given for simple chains shows that during the round the depth of u was reduced from $l(u)+1$ to $\lceil \frac{1}{2} l(u) + 1 \rceil$. □

11 The Potential Function

Finally, using the depth function, F-depth, defined earlier, we define a weight function over the components of the input graph. We then show that at each round each component of the graph loses a constant fraction of its weight. For each component, its first stage ends as soon as its weight is less than 2.

The analysis proceeds as follows. During each round each chain is tagged. We show that the sum of the tags of all the chains at the end of the round is bounded by the weight lost by the component during the round. The proof is completed by showing that, by the end of the round, the sum of the tags of all the chains is at least a constant fraction of the weight of the graph the beginning of the round.

The weight function is defined relative to a chain partition, U, as follows. We choose two constants, c_s, and c_d. We assign to each vertex, v, auxiliary weight, $ex(v)$, comprised of a spare, $sp(v)$, of at most c_s units, and a debt, $db(v)$, of at most c_d units; both are nonnegative. For each vertex, v, $ex(v) = sp(v) - db(v)$. For each set of vertices, S, the spare weight of S, $sp(S)$, is equal to the sum of the spare weights of all the vertices in S. Define $db(S)$ and $ex(S)$ similarly.

Let C be any component of the input graph. The *weight of C relative to a partition, U, at step t*, (F,U)-weight$_t(C)$ is 0 if all the vertices on C are inactive; otherwise, (F,U)-weight$_t(C)$ = (F,U)-depth$_t(C)$ + $ex(C)$. Recall that for any two natural partitions, U and V, of G at step t, (F,U)-depth$_t(C)$ = (F,V)-depth$_t(C)$. Therefore, (F,U)-weight$_t(C)$ = (F,V)-weight$_t(C)$. Define weight$_t(C)$ = (F,U_t)-weight$_t(C)$ for any natural chain partition, U_t of G at step t.

We constrain the spare weight of each active component of G_q to be between c_s and $\frac{3}{2}c_s$. Therefore, each time a hooking operation merges two components of G_q the spare weight of the graph is reduced by at least $\frac{1}{2}c_s$.

The analysis looks at each chain, C, of the chain partition. If all the vertices of C become inactive at round t, C loses all its weight. If, at the start of the round, $f_C > 2$, C reduces

Lemma 10.2 *For any round, r, (F, U_{r+1})-depth$_{r+1}(G) \leq (F, U_r)$-depth$_{r+1}(G)$.*

Proof. It follows from Part 1 of Theorem 9.1 (p. 30) that U_r is a chain partition of G^q_{r+1}. The lemma follows from Part 2 of Theorem 9.1 by the observation that the F-depth of a chain formed by combining two consecutive chains into one is not larger than the sum of the depths of the two chains. □

If we can show that for some constant, α, $\alpha < 1$, and for any chain, C, of U_r,

$$(F, U_r)\text{-depth}_{r+1}(C) \leq \alpha(F, U_r)\text{-depth}_r(C), \tag{1}$$

then we can conclude that

$$(F, U_r)\text{-depth}_{r+1}(G) \leq \alpha(F, U_r)\text{-depth}_r(G).$$

It would then follow from Lemma 10.2 that, in this case,

$$(F, U_{r+1})\text{-depth}_{r+1}(G) \leq \alpha(F, U_r)\text{-depth}_r(G).$$

However, life is not that simple. In Theorem 10.1 we show that for any chain, C, if C is not U_r-minimal then Equation 1 holds. But, some more work has to be done for U_r-minimal chains.

Theorem 10.1 *Let r be any round, let U_r be any natural partition of G^q_r and let C be any chain of U_r which is not U_r-minimal at round r. Then*

$$(F, U_r)\text{-}depth_{r+1}(C) \leq \alpha(F, U_r)\text{-}depth_r(C),$$

where,

$$\alpha = \begin{cases} 3/4 & \textit{If } C \textit{ is a potential leader chain at round } r \\ 2/3 & \textit{Otherwise} \end{cases}$$

Proof. Let v be any vertex. Assume that the chain, C_v, containing v is not a potential leader. Consider (F, U_r)-dag$_r(v)$. First we note that the hooking and promotion operations have no effect on this dag; the claim is clear for a hooking operation. For a promotion operation note that a vertex is promoted only if it is a leader, and, therefore, the head of a chain. Now we consider the effect of the doubling operation.

During round $r + 1$ each vertex on this dag executes a doubling operation; hence, for every F-edge after round $r+1$ there was an F-path of length at least 2 after round r. Recall that the F-depth of a path is one larger than the number of edges on the path; so $l(v) - 1$ was reduced to at most $\lfloor (l(v) - 1)/2 \rfloor$; Thus, $l(v)$ was reduced to at most $\lceil \frac{1}{2} l(v) \rceil$.

any chain partition, U, of G_q and let v be any active vertex. Define an *$F|_U$-path starting at v* to be a path in the F-graph starting at v and ending at the first vertex, u, for which either

1. $F(u)$ is outside the chain containing v or,
2. u is a potential leader, or
3. $R(u)$ is a potential leader.

Let (F,U)-dag$_t(v)$ denote the collection of $F|_U$-paths starting at v at step t. We show (Lemma 10.1) that (F,U)-dag$_t(v)$ is a directed acyclic graph; therefore, each path in (F,U)-dag$_t(v)$ is finite. Let $l(v)$ denote the number of vertices on the longest such path. The *F-depth of v relative to U at step t*, (F,U)-depth$_t(v)$, is defined to be:

$$(F,U)\text{-depth}_t(v) = \begin{cases} l(v)+1 & \text{If the chain containing } v \text{ is a potential leader} \\ l(v) & \text{Otherwise} \end{cases}$$

The *F-depth of a chain, C, relative to U*, (F,U)-depth$_t(C)$, is the maximum of (F,U)-depth$_t(v)$ over all active vertices, $v \in C$. For any component, K, of the input graph, the *F-depth of K relative to U at step t*, (F,U)-depth$_t(K)$, is the sum over all chains, $C \in U \cap K$, of the F-depth of C. The *F-depth of G relative to U at step t*, (F,U)-depth$_t(G)$, is the sum over all chains, $C \in U$, of the F-depth of C.

Lemma 10.1 *For any vertex, v, (F,U)-dag(v) is a directed acyclic graph.*

Proof. Every q-cycle comprises a single followset (Lemma 7.16). Also, for any vertex, v, $F(v)$ is a q-ancestor of v (Theorem 7.2 (p. 25)). Consider any chain, C. There are two cases: either C is contained in a q-cycle or none of its vertices are on a q-cycle. In the latter case the lemma follows trivially. In the former case, the lemma follows from the observation that if $head(v)$ is on the q-path from v to $F(v)$ then $R(v) = head(v)$ (Theorem 7.2 (p. 25) and Theorem 7.1 (p. 21)). □

Definition 10.1 *For any chain partition, U, a chain, C, of U is called U-minimal at step t if it is either a potential leader of G_t^q and (F,U)-depth$_t(C) \leq 2$ or (F,U)-depth$_t(C) = 1$.*

For any step t, let U_t denote a natural chain partition of U.

When using a round number as a subscript instead of a step number, it refers to the last step in the given round. So, for any round, r, G_r^q refers to the q-graph at the last step of round r.

Proof. The proof is by induction on the step number. Assume the lemma holds for t'' with $t \leq t'' < t'$. We show it holds for t' also.

In order to be a chain partition of $G^q_{t'}$, U must satisfy two conditions (see the definition above). That the first condition holds can be seen as follows. For any vertex, l, if l is a potential leader of $G^q_{t'}$ it is also a potential leader of G^q_t (Lemma 9.3). The first condition follows from Lemma 7.5.

That the second condition holds can be seen as follows. First, suppose that $v \neq head(v)$ at step $t'-1$. Then by Lemma 9.4, $q_{t'}(v) = q_t(v)$ and v and $q_{t'}(v)$ are in the same chain of any natural chain partition at step t'; so they satisfy property 2 of the chain partition.

Second, suppose that $v = head(v)$ at step $t'-1$. If v is a potential leader condition 2 holds trivially. So supppose v is not a potential leader. Then $q(v)$ can change only if $q(v)$ is on a p-cycle which does not contain v. From Lemma 7.13, every p-cycle which contains $q_{t'-1}(v)$ must contain v, since v is the only active $q_{t'-1}$-child of $q_{t'-1}(v)$. In addition, the span containing $q_{t'-1}(v)$ is not a link destination nor is it a potential leader. Consequently, $q_{t'-1}(v)$ cannot acquire any new q-children. It follows that $q_{t'}(v) = q_{t'-1}(v)$ and if v is active at step t', v is the only active $q_{t'}$-child of $q_{t'}(v)$. It remains to show that if v and $q_{t'}(v)$ are in distinct spans at step t' then the span of $q_{t'}(v)$ at step t' is not a link destination. By Lemma 7.9, v and $q_{t'-1}(v) = q_{t'}(v)$ were in distinct spans at step $t'-1$, and so by the inductive hypothesis, the span at step $t'-1$ of $q_{t'}(v)$ was not a link destination. In order to become a link destination, a cycle including $q_{t'}(v)$ would have to be formed at step t'; but then this cycle includes v as it is $q_{t'}(v)$'s only q-child; this puts v and $q'_t(v)$ in the same followset and hence the same span at step t', so the claim follows trivially. This completes the proof of Part 1 of the theorem.

Part 2 follows from the fact that a chain partition, restricted to the active vertices, is a refinement of any natural chain partition. □

10 The Depth Function

In order to measure the progress of the algorithm we assign a depth to each vertex. It is natural to consider the F-distance from the vertex to the leader of the component. However, the underlying q-graph might change between the time a process decides on a new value for $F(v)$ and the time the update actually occurs. Therefore, we define the depth of a vertex as a function of both $F(v)$ and its shadow $\overline{F(v)}$.

The analysis uses a second directed graph, called the F-graph, defined as follows. For any vertex, v, there are two F-edges emanating from v: $\langle v, F(v)\rangle$ and $\langle v, \overline{F(v)}\rangle$. Consider

larger p-ancestor at step s. Note that a vertex can become a link destination at step s only if it is a root at that step and therefore a leader. Also, by definition, a non-root vertex can acquire a new child only if it is a link destination. As v is not a link destination (for then it would be a potential leader for it has no larger p-ancestor), the only way v can acquire new descendants is if a descendant, w, of v acquires new descendants. But then w is a link destination at step s. As v is simple, there is a vertex, x on the path from w to v which is larger then v. x is on the path to v from any descendant acquired by w. Thus v is still simple at step $s+1$. □

Lemma 9.4 *Let v be any vertex, and let t_1 and t_2 be any two steps, $t_1 < t_2$. If v and $q_{t_1}(v)$ are in the same chain of a chain partition at step t_1 and v is active at step t_2 then*

1. *v and $q_{t_1}(v)$ are in the same chain of any natural chain partition at step t_2, and*
2. *If v is not $head_{t_1}(q_{t_1}(v))$ then $q_{t_2}(v) = q_{t_1}(v)$.*

Proof. The proof is by induction on the step number. Assume the lemma holds for step $t' \geq t_1$. We show it holds for step $t'+1$. If v is a follower at step $t'+1$, then v and $q_{t'+1}(v)$ are in the same followset and the lemma follows from Lemma 7.9. Otherwise, v is a head of a followset. By the induction hypothesis, as v and $q_{t'}(v)$ are in the same chain, the chain containing $q_{t'}(v)$ is not a link destination and v is the only active $q_{t'}$-child of $q_{t'}(v)$.

The q-pointer of v may change in one of two ways: if v is a root and v hooks to another vertex, u, or if $q(v)$ is on a p-cycle which does not contain v. If v is a root at step t_1 then $q_{t_1}(v)$ is the tail of the span containing v and the lemma follows from Lemma 7.5.

From Lemma 7.13, every p-cycle which contains $q_{t'}(v)$ must contain v since v is the only active $q_{t'}$-child of $q_{t'}(v)$. In addition, the span containing $q_{t'}(v)$ is not a link destination nor is it a potential leader. Consequently, $q_{t'}(v)$ cannot acquire any new q-children. It follows that $q_{t'+1}(v) = q_{t'}(v)$ and if v is active at step $t'+1$, v is the only active $q_{t'+1}$-child of $q_{t'+1}(v)$. So $q(v)$ is never on a p-cycle which does not contain v. □

Theorem 9.1 (Good Partition) *Let t be any step number. For any natural chain partition, U, of G^q_t and for any $t' > t$, the following two properties hold.*

1. *U is a chain partition of $G^q_{t'}$.*
2. *For any natural chain partition, U', of $G^q_{t'}$, and for every chain, C, of U, the active part of C, at step t', is contained in a chain of U'.*

The remainder of this section justifies the above definitions by showing that inactive vertices can never become active (Lemma 9.2) and that simple vertices can never become potential leaders (Lemma 9.3). Finally, we show some properties of natural chain partitions (Theorem 9.1). These are used in the next section to guarantee that the potential function does not increase.

Lemma 9.1 *The head of a followset is the only vertex in the followset that can be a potential leader.*

Proof. A vertex that has a larger p-ancestor cannot be a potential leader by definition. The lemma follows from Theorem 7.1 (p. 21). □

Lemma 9.2 *If a vertex is inactive at step t, it is inactive at step t', for any $t' > t$.*

Proof. The proof is by induction on the step number. Assume the lemma holds for every step up to and including step, s, for some step s, $s \geq t$. We show it holds for step $s+1$. It suffices to show that for any vertex, v, if v is active at step s then $q_{s+1}(v)$, $next_{s+1}(v)$, $\overline{q_{s+1}(v)}$ and $\overline{next_{s+1}(v)}$ are active vertices at step s.

If $next(v)$ is modified at step $s+1$, then $next_{s+1}(v) = \overline{next_s(v)}$ by definition. $q(v)$ is modified at step $s+1$ in one of two cases: When v is executing a hooking step and when $p_s(v)$ becomes part of a cycle which does not include v (note that $q(v)$ is not modified by a promotion operation). In the former case, $q_{s+1}(v) = \overline{q_s(v)}$. In the latter case, v is not a follower and $q_{s+1}(v) = tail_{s+1}(p_{s+1}(v))$ (see Definition 7.3 (p. 23)). In this case, $p_{s+1}(v) = p_s(v)$. As $p_s(v)$ is a q_s-ancestor of v (Lemma 7.13) $p_s(v)$ is active at step s and it follows that $tail_{s+1}(p_s(v))$ is active at step s as well.

The shadow variable $\overline{q(v)}$ is modified only when v is executing a step hooking it to some vertex, u. In this case, $\overline{q_{s+1}(v)} = F_s(next_s(v))$ which is a q_s-ancestor of $next_s(v)$ (by Theorem 7.2 (p. 25)).

The shadow variable $\overline{next(v)}$ is modified in three cases: 1) when $next(v)$ is being updated to $F(next(v))$. Then the lemma follows from Theorem 7.2 (p. 25)). 2) v just read a suggestion from edge e. Then $\overline{next_{s+1}(v)} = V_s(e)$, and the lemma follows. 3) v read the vertex, $w = F_s(\overline{next_s(v)})$, which it will assign to $next(v)$. Then w is a q_s-ancestor of $\overline{next_s(v)}$ by Theorem 7.2. □

Lemma 9.3 *If a vertex, v, is simple at step, $t' < t$, it is simple at step t.*

Proof. The proof is by induction on the step number. If v has a larger ancestor at step s then the lemma follows from Lemma 7.1. So suppose v is simple and does not have a

$\overline{next(v)}$, and $\overline{q(v)}$ are active.

A vertex is called a *potential leader* if it is either a leader or for some link destination, u, v is the largest vertex on the p-path from u to the leader of the component; if v is not a leader, the link destination, u, is said to be in the *promoting set of* v. Intuitively, if r is the root of the p-component containing v and u is in the promoting set of v, then hooking r to u would make v a leader. Vertices which are not potential leaders are called *simple*. We show that a simple vertex can never become a potential leader (Lemma 9.3).

A span that contains a potential leader is called a *potential leader span* or simply a *potential leader*. Lemma 9.1 shows that the head of the span is the only vertex of the span that can be a potential leader. For any span, s, we call the span containing $q(head(s))$ the *parent of* s and denote it by $q(s)$; s is called a *child of* $q(s)$. We say that a span is *active* if its vertices are active. A span which is neither a potential leader nor a link destination and has at most one active child is called *chained*. By Lemma 7.11, a span which is neither a potential leader nor a link destination is chained if and only if its tail has at most one active q-child.

Define a *chain partition* of G_q to be a partition of the q-graph into disjoint q-paths, called *chains*. Each chain, C, satisfies:

1. For any potential leader, l, if $C \cap span(l) \neq \emptyset$ then $C \subseteq span(l)$.

2. If v is active but not a potential leader, and if $q(v)$ is in the chain containing v, then v is the only active q-child of $q(v)$. In addition, if v and $q(v)$ are in distinct spans then $q(v)$'s span is not a link destination.

A chain partition, U, is a *natural chain partition* if, in addition:

1. Each span of G_q is contained in a chain of U. (This implies that a chain which contains a potential leader comprises exactly one span of G_q.)

2. All the active spans contained in a chain form a sequence, $s_1, s_2, \ldots s_l$, satisfying:
   a) $s_{i+1} = q(s_i)$, for $1 \leq i < l$,
   b) for any i, $2 \leq i \leq l$, s_i is chained, and
   c) $q(s_l)$ is not chained.

A natural chain partition is unique on the active vertices of the graph; chain partitions need not be unique, but on the active vertices they are refinements of a natural chain partition.

Consider any chain partition, U. For each chain, τ, of U, $q(\tau)$, $next(\tau)$ are defined in the obvious way. Likewise, we say a chain is a *link destination*, a *current destination*, a *root*, a *leader*, a *potential leader*, or an *endpoint* if it contains a vertex of the corresponding type.

much simpler.

In Section 9 we divide the vertices into active and inactive vertices. We then define a chain partition of the q-graph. In Section 10 we use this partition and define the depth function. Finally, in Section 11 we define the potential function and analyze the first stage of the algorithm. In Section 12 we analyze the second stage of the algorithm.

9 Active Vertices

Recall that before a root vertex, r, can create a new link it must find the root of the component to which it is attempting to hook. If r is the root of a star, it may not make any "real" progress while waiting. The component, C, r is trying to hook to may be shrinking quite fast; however, C may have many stars waiting to hook onto it. Therefore, the progress C makes may not be enough to offset the lack of progress made by all the waiting roots.

We overcome this difficulty as follows: We divide the vertices of the graph into two disjoint sets: the *active* vertices and the *inactive* vertices. The analysis uses a potential argument, assigning each component of G_q^+ a potential; the potential of the graph is the sum, over all the components, of the potential of the component. A portion of the potential is due to weights assigned to the vertices; the inactive vertices have no weight and thus do not contribute to the potential.

Recall that the edge processes migrate out of roots along the current destination edges. This guarantees that if r is a root of a star waiting to hook to some component, C, after a small constant number of steps either r hooks to C or some other root hooks to r, in which case progress is made, or r and its component become inactive, decreasing the potential. We then show that at each round, the potential of the graph decreases by a constant multiplicative factor. This reduction is aided by the fact that some of the vertices become inactive during the round.

We view the q-graph as divided into spans; for each vertex, v, the q-path from $tail(v)$ to $head(v)$ is the *span* containing v (there is such a path by Lemma 7.10). v is called an *endpoint* if for some edge, e, of the input graph either $v = V(e)$ or $v = \overline{V(e)}$. If v is executing a hooking operation and $\overline{A(v)} = w \neq v$ we say that v is in the process of hooking to w; w is called the *link destination of* v. A span of G^{q+} is called an *endpoint* if it contains an endpoint of G_q; it is called a *link destination* if it contains a link destination.

The vertices are divided into *active* and *inactive* vertices as follows. For any edge, e, the endpoint $V(e)$ and its shadow $\overline{V(e)}$ are active. In addition, for each vertex, v, if a vertex in the span containing v is active then v is active. Also, if v is active then $next(v)$, $q(v)$,

A vertex can be promoted only if it is the largest vertex on a p-cycle. Also, followers cannot be promoted (Lemma 7.2). It follows that $R(v)$ (resp. $\overline{R(v)}$) is the only vertex on the q-path from $q(v)$ to $R(v)$ (resp. from $q(v)$ to $\overline{R(v)}$) that can be promoted (for by Lemma 7.15 any other vertex on this path has a larger p-ancestor). If $R(v)$ (resp. $\overline{R(v)}$) is promoted at step t, $F_t(v) = R(v)$ (resp. $\overline{F_t(v)} = \overline{R(v)}$) and the theorem follows.

If a vertex, $x \neq R(v)$, on the q-path from $R(v)$ to $F(v)$ is promoted, then $head(R(v))$ is not on the p-cycle promoting x (for $head(R(v)) \geq R(v) > x$). In this case, after the promotion, there is a q-path from $R(v)$ to $F(v)$ that does not go through x. All the vertices on the new q-path, which were not on the old q-path, are followers, by definition. Part 1 of the theorem follows. An identical argument replacing $R(v)$ with $\overline{R(v)}$ and $F(v)$ with $\overline{F(v)}$ shows that Part 2 of the theorem holds as well. □

8 Analysis

The analysis proceeds in two stages, corresponding to the two stages of the algorithm. The input graph, G, may have several components. The goal of the algorithm is to select a representative for each component, and set all the other vertices of the component to point to the representative.

During Stage 1, the algorithm links components of the underlying graph until each such component contains all the vertices of exactly one component of the input graph. During this stage, we divide the vertices into *active* and *inactive* vertices. Roughly speaking, the inactive vertices do not affect the performance of Stage 1.

The analysis of Stage 1 uses a potential function argument. Recall that the promotion operation disconnects the underlying p-graph and a hooking operation disconnects the q-graph. We partition the graph into *chains* and define a depth function on the vertices relative to the partition. The partition we chose guarantees that chains are never broken; were a chain to break, the sum of the potential of its two parts might be larger than the potential of the original chain. The potential of the graph is a function of the sum of the depths of all the chains. The chains fall into two classes, *long chains* and *short chains*. We show that in each round the long chains lose a constant fraction of their depth. The short chains require a more delicate accounting scheme. We show that a constant fraction of the short chains lose a constant fraction of their weight.

During Stage 2 of the algorithm each component of the underlying graph corresponds to a component of the input graph. In addition, the only vertices that can be promoted during Stage 2 are the final leaders of the graph. Consequently, the analysis of Stage 2 is

Lemma 7.16 *All the vertices of a q-cycle comprise a single followset.*

Proof. Consider a q-cycle, C. If C has two non-follower vertices, u and v, then by Lemma 7.12, u and v are on a p-cycle. But, each p-cycle can have at most one non-follower vertex. It follows that each q-cycle has at most one non-follower vertex, l. Let w be a vertex of C. By Lemma 7.10, C includes $head(w)$, which is a non-follower vertex. As $x = head(w)$ is the only non-follower vertex, $q(x)$ must be in $followset(w)$ (for otherwise $head(q(x))$ would be a different non-follower vertex on C). So $q(head(w)) = tail(w)$. The lemma now follows from Lemma 7.10. □

Lemma 7.17 *For any two vertices, u and v, if there is a path, P_1, from u to v in G^q_{t-1} and none of the vertices on this path are promoted at step t then there is a path, P_2, from u to v in G^q_t; also*

1. *$P_1 \subseteq P_2$,*
2. *all the vertices in $P_2 - P_1$ are followers.*

Proof. As none of the vertices on P_1 is promoted, the only way a vertex, x, on P_1 can change its q pointer is if x is not a follower and at step t, $p_{t-1}(x)$ becomes part of a p-cycle which does not go through x. In this case, $p_t(x) = p_{t-1}(x)$ and $q_t(x) = tail_t(p_t(x))$. The q_t-path going from $tail_t(p_t(x))$ to the leader of the new cycle goes through $p_t(x)$ and includes all the vertices on the q_{t-1}-path from $q_{t-1}(x) = tail_{t-1}(p_{t-1}(x))$ to $p_{t-1}(x) = p_t(x)$. In addition, all the vertices added are followers by definition. □

Definition 7.4 *For any pair of vertices, u and v, we say that u* dominates *v if there is a q-path from v to u and u is larger than all the non-follower vertices on this path. Similarly, we say that v* forward dominates *u if there is a q-path from v to u and v is larger than all the non-follower vertices on this path.*

Theorem 7.2 *There are simple q-paths from $q(v)$ to $R(v)$, from $R(v)$ to $F(v)$, from $q(v)$ to $\overline{R(v)}$ and from $\overline{R(v)}$ to $\overline{F(v)}$; also:*

1. *$R(v)$ dominates $q(v)$ and forward dominates $F(v)$.*
2. *$\overline{R(v)}$ dominates $q(v)$ and forward dominates $\overline{F(v)}$.*

Proof. The proof is by induction on the step number. The theorem is maintained by any step of a doubling operation and any step of a hooking operation, by Lemma 7.17. We must consider the effect of a promotion.

by definition. If $p(w)$ is a follower, then the first non-follower on the q-path starting at $p(w)$ is $head(p(w))$ and there is a p-path from $p(w)$ to $head(p(w))$ (by Theorem 7.1 (p. 21)). So, by induction, the non-followers on the q-path starting at u all lie on a p-path in the same order as on the q-path. □

Lemma 7.13 *For any vertex, v, $p(v)$ is a q-ancestor of v.*

Proof. Let v be any vertex, and consider some step, t. If v is not a follower at step t, then $q_t(v) = tail_t(p_t(v))$ (by definition). By Lemma 7.10, there is a q_t-path from $tail_t(p_t(v))$ that goes through all the vertices of the followset containing $p_t(v)$; in particular, this path goes through $p_t(v)$.

So suppose s is the step in which v becomes a follower. By the definition of the q pointer, $q_s(v) = tail_{s-1}(p_s(v))$. Let $u = p_s(v)$. There is a q_{s-1}-path from $tail_{s-1}(u)$ to u (Lemma 7.10). All the vertices on this path, except possibly u, are followers and, therefore, do not change their q-pointers during step s or subsequently (Lemma 7.9). Therefore, for $t \geq s$, there is a q_t-path from $tail_{s-1}(u) = q_s(v) = q_t(v)$ to $u = p_s(v) = p_t(v)$.

□

Corollary 7.1 *For any vertex, v, $p^+(v)$ is a q^+-ancestor of v.*

Proof. Note that if $q^+(v) \neq q(v)$ then $q^+(v) = tail(p^+(v))$. The corollary follows from Lemma 7.10. □

Lemma 7.14 *Each p^+-component is a q^+-component.*

Proof. By Corollary 7.1, each p^+-component is contained in a q^+-component. It remains to show they are equal. It suffices to show that v and $w = q^+(v)$ are in the same p^+-component. If v and w are in the same followset, $head(v) = head(w)$, which by Theorem 7.1, part 3 is a p-ancestor of v and w; so v and w are in the same p-component and hence the same p^+-component. If v and w are in distinct followsets, then $w = tail(p^+(v))$; the argument of the previous sentence show that $p^+(v)$ and w are in the same p^+-component, and hence so are v and w. □

Lemma 7.15 *For any vertex, v, if v has a larger q-ancestor, it has a larger p-ancestor.*

Proof. By Theorem 7.1, the head of a followset is the largest vertex in the followset; also it is not a follower. Let w be the largest q-ancestor of v and let $x = head(w)$. By Lemma 7.12, x is a p-ancestor of v; further it is at least as large as w and hence is larger than v. □

Lemma 7.9 *For any vertex, v, and any step, s, if v becomes a follower at step s then, for any t, $t > s$*

1. *$q_t(v) = q_s(v)$, and*
2. *v and $q_t(v)$ are in $followset_t(v)$.*

Proof. Follows from the definition. □

Lemma 7.10 *For any vertex, v, there is a simple q-path from $tail(v)$ to $head(v)$. The path goes through all the vertices of $followset(v)$.*

Proof. The proof is by induction on the step number. Assume the lemma holds for some step, t, and consider step $t+1$. If a new p-cycle is created at step $t+1$ then the only non-follower vertices that become followers at step $t+1$ are vertices on the new p-cycle. The lemma for step $t+1$ follows by induction on the number of such non-followers.

If a new p-cycle is not created at step $t+1$, the lemma for $t+1$ follows from the induction hypothesis and Lemma 7.9. □

Definition 7.3 *We extend the definition of the q mapping to non-followers by defining $q_t(v) = tail_t(p_t(v))$, for every non-follower, v.*

Lemma 7.11 *For any vertex, v, if $q(v)$ is not in $followset(v)$, then $q(v)$ is a tail.*

Proof. The lemma follows from Lemma 7.10. □

Define the q-graph, G_q, to be the (pointer) graph induced by the q-mapping. We now examine the relation between the p-graph and the q-graph. The main result of this section is stated in Theorem 7.2 which shows that $F(v)$ is a q-ancestor of v. Let G_t^q denote the q-graph at step t. Define the mapping q_t^+ as follows: For any vertex, r, if r is a root then $q_t^+(r) = tail_t(next_t(r))$; $q_t^+(r) = q_t(r)$ otherwise. (Recall that a vertex is a root if and only if $R(v) = v$.) Define the q^+-graph, G_t^{q+}, to be the graph induced by the mapping q_t^+.

Lemma 7.12 *For any two vertices, u and v, if there is a q-path from u to v then all the non-follower vertices on this path are p-ancestors of u.*

Proof. The proof is by induction on the length of the path. $head(u)$ is the first non-follower vertex on any q-path starting at u ($head(u) = u$ if u is not a follower). By Theorem 7.1 (p. 21), there is a p-path from u to $head(u)$.

Let w be any non-follower vertex on the q-path from u to v. Then $q(w)$ is $tail(p(w))$ and all the vertices on the q-path from $tail(p(w))$ to $p(w)$, except possibly $p(w)$, are followers,

3. There is a p-path from v to $head(v)$.

Proof. Lemma 7.4 implies that each followset has at least one non-follower vertex and Lemma 7.6 shows it has at most one; Part 1 of the theorem follows. This implies that if $v \neq head(v)$, v is a follower of $head(v)$. Therefore, there is a sequence of vertices, v_1, v_2, ..., v_l, such that $v_1 = v$, $v_l = head(v)$ and for any i, $1 \leq i < l$, v_i is a supporter of v_{i+1}. The last two parts of the theorem follow from Lemma 7.3 by induction on the length of the sequence. □

7.1 Finite Counters

In this section we digress a bit and show that a node can be promoted at most $n-1$ times. From this it follows that counters of logarithmic size suffice.

Lemma 7.7 *A vertex, v, cannot hook to one of its followers.*

Proof. The proof is by induction. Initially there are no followers. We show that if a vertex u becomes a follower of v at some step s then v cannot hook to u at any step $s' > s$.

Let u and v be any two vertices. If u becomes a follower of v at step s (for some s), then, at step s, v is a non-root leader. Furthermore, by Lemma 7.2 applied to u, u is not a leader at any step $s' \geq s$. If v links to some vertex, w, at some step, t, $t > s$, then, at some step, t', $s < t' < t$, w is a root, and hence a leader. It follows that v cannot hook to u at any step $s' > s$. □

Lemma 7.8 *A vertex, v, is promoted at most $n-1$ times.*

Proof. By Lemma 7.7, each time a p-cycle is created, at least one non-follower vertex becomes a follower. □

7.2 The q pointers

We define a linear ordering among the vertices of a followset by defining a mapping, q, recursively, as follows. Each followset has two designated vertices called the *head* (which was defined above) and the *tail* (which is defined here). Let v be any vertex. Initially, v is in a singleton followset, $\{v\}$, and $tail_0(v) = head_0(v) = v$. Whenever a p-cycle with leader l is created at step t, $followset_t(l)$ includes all the vertices on the cycle and all their followers. For each vertex, u, on the cycle, except l, if u is a follower before step t, define $q_t(u) = q_{t-1}(u)$; otherwise, define $q_t(u) = tail_{t-1}(p_t(u))$. Also, $tail_t(l) = tail_{t-1}(p_t(l))$. This defines the q mapping for followers.

Proof. Let t be any step and assume the v is a supporter of u at step t. Then at some step, s, $s < t$, v became a supporter of u. By definition, u and v are on a p_s-cycle and $v < u$; thus, there is a p_s-path from v to u. Furthermore, all the vertices on the p_s-path from v to u are supporters of u. By Lemma 7.2, a supporter cannot become a leader, therefore none of these vertices change their p-pointers at any step, t', $t' > s$. It follows that there is a p_t-path from v to u and $v < u$. □

Lemma 7.4 *The transitive closure of the supporter relation is irreflexive.*

Proof. It follows by induction from Lemma 7.3 that if u is a transitive supporter of v then $u < v$. □

Definition 7.1 *Define the relation* follower *to be the transitive closure of the supporter relation: For any two vertices, u and v, u is a follower of v if u is either a supporter of v or a supporter of a follower of v.*

Definition 7.2 *Partition the vertices into sets, called* followsets*, as follows: For any two vertices, u and v, u and v are in the same followset if either u is a follower of v or if both u and v are followers of some vertex, w.*

For any vertex, v, let $followset(v)$ denote the followset containing v. If the followset contains exactly one non-follower vertex, that vertex is called the head of the followset *and is denoted by $head(v)$.*

Lemma 7.5 *For any step, t, and for any two vertices, u and v, if u and v are in the same followset at step t they are in the same followset at step t', for any t', $t' > t$.*

Lemma 7.6 *For any vertex, u, if u is a supporter of some vertex, v, at step t, and u becomes a supporter of some vertex, w, $w \neq v$, at step $t+1$, then v becomes a supporter of w at step $t+1$.*

Proof. As u becomes a supporter of w at step $t+1$, u is on a p_{t+1}-cycle with leader w. By Lemma 7.3, there is a p_{t+1}-path from u to v; therefore, v is on the same p_{t+1}-cycle. It follows that v becomes a supporter of w at step $t+1$. □

Theorem 7.1 *Let v be any vertex. Then*

1. *$followset(v)$ has exactly one head, $head(v)$.*
2. *$head(v)$ is the largest vertex in the followset.*

vertex r and therefore does not have to be read at the current step). r receives the values associated with $next(r)$, namely $Ar = A(next(r))$ and $Rr = R(next(r))$. Then, at step s_3, it reads the record associated with Rr to determine the value of $F(next(r))$. Suppose that at step s_4 it writes Ar into $A(r)$. Then at step s_2, $\overline{A(r)} = Ar$ and at step s_4, $A(r) = Ar$.

Recall the function $F(v)$. We define the shadow of $F(v)$, $\overline{F(v)}$, as follows. If $ctr(\overline{R(v)}) > \overline{Rctr(v)}$ then $\overline{F(v)} = \overline{R(v)}$; otherwise, $\overline{F(v)} = \overline{A(v)}$.

The shadow variable of $next(v)$ is a special case. Suppose v is a root and it is executing a step which assigns $next(v)$ a new value, w. Let t' be the step in which v reads w. If the operation does not require a suggestion from an edge process, $\overline{next(v)}$ becomes equal to w at step t'. Otherwise, $\overline{next(v)}$ is updated in two steps. Suppose that v reads a suggestion from $new(v)$ at step t_1; this suggestion is a pointer to an edge, e. It then reads the value of $u = V(e)$ at some step t_2, $t_2 > t_1$. Define $\overline{next(v)}$ to be equal to u at step t_2. It then becomes equal to $w = F(u)$ at step t'.

Followers and Followsets. We define a relation, *supporter*, among the vertices. Initially, none of the vertices are supporters. If a p-cycle with leader l is created at step t, then all the vertices on the cycle, except l, become supporters of l. In the next lemmas we prove some properties of the supporter relation, including that it is a partial order; we then define the *follower* relation to be the transitive closure of the supporter relation.

Lemma 7.1 *For any vertex, v, and for any step, t, if v has a larger p-ancestor at step t, it has a larger p-ancestor at step t', for any $t' > t$.*

Proof. The proof is by induction on the step number. Let v be a vertex and assume that v has a larger p-ancestor, y, at step s. We show it has a larger p-ancestor at step $s+1$. If, at step $s+1$, y ceases to be a p-ancestor of v, then at that step a vertex, x, on the p-path from v to y, is promoted. In that case, x is a p-ancestor of v at step $s+1$ and, by Theorem 5.1 (p. 12), $x > y > v$. □

Lemma 7.2 *If v is a supporter at step t, it is not a leader at step t', for any $t' \geq t$.*

Proof. A vertex that has a larger p-ancestor is not a leader. A vertex, u, that becomes a supporter at step s has a larger p-ancestor at that step and, therefore, by Lemma 7.1, is not a leader at any step, $t' \geq s$. □

Lemma 7.3 *If v is a supporter of u then $v < u$ and there is a p-path from v to u.*

of the input graph then, by Theorem 6.1, u and v are in the same p^+-component and thus $A(u) = A(v)$. On the other hand, if $A(u) = A(v)$ then, by Lemma 6.8, u and v are in the same p^+-component. By Theorem 6.1, u and v are in the same component of the input graph. □

7 The q Graph

The analysis of the rounds complexity uses a potential function. In order to simplify the analysis, we define the potential of the graph in a way that guarantees the potential cannot increase. Consider two vertices, u and v, on an A-path (a path in the pointer graph induced by the A pointers). We may view the A-distance between u and v to be the number of edges on this path. Unfortunately, we cannot guarantee that for every vertex, v, $A(v)$ and $F(v)$ are p-ancestors of v. Without this property, it was hard to define a good potential function.

In this section we define a second underlying graph, called the q-graph, in which $F(v)$ is always a q-ancestor of v ($A(v)$ need not be an q-ancestor of v but we show that this has no adverse effects). We then define the potential of the graph as a function of the F-distances. Replacing the p-graph by the q-graph simplifies the analysis considerably.

We start by introducing some notation.

Shadow Variables. The analysis examines high level operations such as doubling, promotion and hooking, each comprising a number of events. However, the operations are not executed atomically; an operation executed by a process might use information it read at some previous step; at the time this information is used it might be out of date.

The operations performed by the algorithm modify pointers and counters; the new value assigned to a pointer is a value read during the operation. Suppose that a process, P, performs an operation on vertex r which writes to some pointer, for example, $A(r)$, a new value, v. Further, suppose that the actual change to $A(r)$ occurred at some step, t. Then, at some step, s, $s < t$, of the operation, the process read the value v. This value is imagined to be assigned to a *shadow* of variable $A(r)$, denoted $\overline{A(r)}$, at step s. Between step s and step t other processes may have executed events, and possibly modified other variables, including the variable from which $\overline{A(r)}$ was read. Initially, for any vertex, u, and any pointer, A, $\overline{A(u)} = A(u)$. It follows by induction that immediately before a process starts executing an operation on vertex r, $\overline{A(r)} = A(r)$.

For example, suppose that a hooking operation performed at a root, r, begins at step s_1. It reads the record associated with $next(r)$ at step s_2 ($next(r)$ is a pointer associated with

Theorem 6.1 (Stage-I) *If there are no active edge processes then for any pair of vertices, u and v, u and v are in the same p^+-component if and only if they are in the same component of the input graph, G.*

Proof. Let u and v be two vertices. If $C(u) = C(v)$ and $C^+(u) \neq C^+(v)$ then by Lemma 6.7 there is an edge, e, with one endpoint in $C^+(v)$ and one outside. By Lemmas 6.4 and 6.5, the endpoints of e were never in the same p^+-component and thus e could not have terminated. Conversely, if $C^+(u) = C^+(v)$, u and v are in the same component of the input graph by Lemma 6.6. □

6.3 Vertex Termination

The main theorem of this section is Theorem 6.2 which shows that if the algorithm terminates, it terminates correctly.

Lemma 6.8 *For any vertex v, if the process associated with v terminated then $A(v)$ is the root of the p^+-component containing v.*

Proof. Assume that all the edge processes have terminated. There are two ways in which a vertex v can terminate. First, if v is a root of G_p and $next(v) = v$; then, v is also a root of G_{p^+}. Second, if $A(v)$ is a terminated root (and hence $A(v) \neq v$); then, v is not a root of G_p. So a root of G_p terminates only if it is also a root of G_{p^+}. After a process associated with a vertex, v, starts its termination procedure, $R(v)$ and $A(v)$ are not subsequently modified. Therefore, every terminated root remains a root in G_{p^+}; likewise, every other terminated node remains a non-root in G_p and hence in G_{p^+}.

For a root, r, in G_p, $A(r) = r$. Thus, on termination, a root, r, in G_{p^+}, has $A(r) = r$. As $A(r)$ is not modified subsequently, the lemma holds for terminated roots. For a non-root, v, with $A(v)$ a terminated root, $A(v)$ must be a root of G_{p^+}. v and $A(v)$ are in the same p-component (by Lemma 6.3) and, therefore, in the same p^+-component. Thus, $A(v)$ is the (unique) root of the p^+-component containing v. So the lemma holds for terminated non-roots as well. □

Theorem 6.2 (Correct Termination) *Let u and v be any two vertices whose associated processes have terminated. Then u and v are in the same component of the input graph if and only if $A(u) = A(v)$.*

Proof. Let u and v be any two vertices whose processes have terminated. By Lemma 6.8, $A(v)$ is the root of the G_p^+ component containing v. If u and v are in the same component

Proof. Let $t \leq t'' < t'$. The proof is by induction on the step number. So we suppose inductively that $V_t(e)$ and $V_{t''}(e)$ are in the same p^+-component at steps $\geq t''$. Suppose that at step $t''+1$, an endpoint, $v = v_{t''}(e)$, migrates either to $F_s(v)$ or to $F_{s'}(next_s(v))$, for some $s \leq s' \leq t''$. $F_s(v)$ and v are in the same component of G_s; and thus, by Lemma 6.1, of $G_{t'''}$, for $t''' > t''$. Likewise, $w = next_s(v)$, and $F_{s'}(w)$ are in the same component of $G_{t'''}$, for $t''' > t''$. Finally, w and v are in the same component of G_s^+ by definition, and hence of $G_{t'''}^+$, for $t''' > t''$, by Lemma 6.4. Clearly the inductive hypothesis holds for $t' = t'' + 1$; so the lemma follows by induction. □

Lemma 6.6 *For any pair of vertices, u and v, if u and v are in the same p^+-component, then they are in the same component of the input graph, G.*

Proof. The proof is by induction on the step number. Initially each vertex is a p^+-component and the lemma holds vacuously. Assume the lemma holds immediately before step t, for some t. The only new intercomponent edges added to G_{p^+} at step t are current destination edges. A new current destination edge, $\langle x, y\rangle$, is added to G_{p^+} only if x is a root and y is a possible destination of x. In order for y to become a possible destination of x there must be an input edge, $e = \langle x', z'\rangle$, and some step numbers, $t_1 \leq t_2 < t_3 < t$, for which $V_{t_1}(e) = x$ and $V_{t_2}(\overline{e}) = z$, where $F_{t_3}(z) = y$. By Lemma 6.5, x and x' are in the same p^+-component at step t_1, and, thus, by the induction hypothesis, x and x' are in the same component of the input graph, G. Similarly, z and z' are in the same component of the input graph G.

As there is an edge in G from x' to z', x' and z' are in the same component of G. Finally, by Lemma 6.3, $F_{t_2}(z) = y$ and z are in the same component of G_{t_2}. Thus, by the inductive hypothesis, y and z are in the same component of the input graph. Therefore, all the vertices in the new p^+-component created by adding the current destination edge $\langle x, y\rangle$ are in the same component of G, as required. □

Lemma 6.7 *Let v be any vertex, and let t be any step number. If $C_t^+(v) \neq C(v)$ then there is an edge, e, with $V(e) \in C_t^+(v)$ and $V(\overline{e}) \notin C_t^+(v)$.*

Proof. Assume that $C_t^+(v) \neq C(v)$. By Lemma 6.6, $C_t^+(v) \subset C(v)$; therefore, there must be a vertex, w in $C(v) - C_t^+(v)$.

Since v and w are in the same component of G there is a path in G from v to w. As $v \in C_t^+(v)$ and $w \notin C_t^+(v)$, there must be an edge, $e = (x, y)$, in the input graph, G, on a G-path from v to w for which $x \in C_t^+(v)$ and $y \notin C_t^+(v)$. By Lemma 6.5, at step t, $V(\langle x, y\rangle)$ is in $C_t^+(v)$ and $V(\langle y, x\rangle) \in C_t^+(y) \neq C_t^+(v)$. □

Lemma 6.3 *For any vertex, v, v, $A(v)$, and $F(v)$ are in the same component of G_p.*

Proof. The lemma follows from Lemma 5.1 and Lemma 6.2. □

Lemma 6.4 *For any pair of vertices u and v, and for any step number, t_1, if u and v are in the same component of $G^+_{t_1}$ then they are in the same component of $G^+_{t_2}$ for any $t_2 > t_1$.*

Proof. The proof is by induction on the step number. Initially the lemma holds. Assume the lemma holds at step t' for all $t' < t$. We show it holds at step t.

The p^+-edges are of two types: p-edges and current destination edges. p-components are never broken (by Lemma 6.1). A current destination edge, $\langle x, y\rangle$, with $x \neq y$, is removed in one of three cases: when a new link is created from x to y, when $y = x$, or when $\langle x, y\rangle$ is replaced by $\langle x, F_s(y)\rangle$, for some $s < t$.

When a link is created, linking x to y, the p^+-edge $\langle x, y\rangle$ is replaced by a p-edge $\langle x, y\rangle$; this operation does not modify the p^+-components. If $y = x$, clearly removing $\langle x, y\rangle$ from G_{p^+} does not alter the p^+-components. y and $F_s(y)$ are in the same component of G_s (by Lemma 6.3) and, therefore, in the same component of G_t (Lemma 6.1). Thus, replacing $\langle x, y\rangle$ by $\langle x, F_s(y)\rangle$ does not alter the p^+-components.

□

6.2 Edge Termination

Next, we show the correctness of the edge processes by showing that once all the edge processes terminate the p^+-components are exactly the components of the input graph G (Theorem 6.1). This proceeds as follows: Lemma 6.5 shows that despite the edge migration, at any time, the current endpoint, $V(e)$, of an edge, e, is in the same p^+-component as its original endpoint. This, together with Lemma 6.6, which shows that p^+-components are contained in components of G, demonstrates that the graph resulting from adding all the current (non-terminated) edges to G^+ has the same components as the input graph. (Recall that the current edge refers to the (virtual) edge connecting the current endpoints of an input edge.)

For any vertex v, let $C(v)$ (resp. $C^+_t(v)$) denote the component of the input graph (resp. of G^+_t) containing v. Lemma 6.7 shows that for any vertex, v, as long as $C^+_t(v) \neq C(v)$, there is a non-terminated edge, e, whose endpoint is in $C^+_t(v)$.

Lemma 6.5 *For each edge, e, and for any two steps, $t < t'$, $V_t(e)$ and $V_{t'}(e)$ are in the same p^+-component at steps $\geq t'$.*

The following vertex termination code is added to the vertex processes:

Termination condition for vertex v:
    if All the edges have terminated **then**
        if v is a root and $next(v) = v$ **then** Terminate
        if $A(v)$ is a terminated root **then** Terminate

Comment. A prefix sum computation over the edge processes can be used to determine when all the edge processes have terminated.

The following definitions are helpful. Recall that for a root r, $next(r)$ is called a *current destination*; u is called a *possible destination* if there is a root vertex r, for which $next(r) = r$ and $F(V(new(r))) = u$; that is, r is not committed and u is the endpoint of the most recent suggestion given to r. The edge $\langle r, next(r)\rangle$ is called a *current destination* edge. For an edge, e, of the input graph, the current edge corresponding to e is the (virtual) edge $\langle V(e), V(\bar{e})\rangle$.

The analysis uses an auxiliary graph, G_{p^+}, which is obtained from G_p by replacing each self loop, $\langle r, r\rangle$, in G_p, by the edge $\langle r, next(r)\rangle$ (r must be a root of G_p). The algorithm is viewed as having two stages; Stage 1 ends when all the edge processes terminate. We show that at the end of Stage 1 the p^+-components are exactly the components of the input graph G (where the components are considered to form a partitioning of the vertices). We let G_t^+ denote the graph G_{p^+} at step t.

We start by showing that components of G_p^+ grow monotonically.

6.1 Components of the Underlying Graph

Lemma 6.1 *For any two vertices, u and v and any step t, if u and v are in the same component of G_t then u and v are in the same component of $G_{t'}$ for any $t' > t$.*

Proof. Only two operations modify the p pointers: a hooking and a promotion. A hooking operation replaces a self loop by a new edge and cannot break a component of G_p. By Theorem 5.1, a vertex is promoted only if it is the largest vertex in a cycle of G_p and thus a promotion cannot break components of G_p. □

Lemma 6.2 *For any two vertices, u and v, and for any step number, t, if u and v are in the same component of G_t^*, they are in the same component of G_t.*

Proof. The edge $\langle u, v\rangle \in E_t^*$ if and only if $p_s(u) = v$ for some $s \leq t$. Then, u and v are in the same component of G_s and, therefore they are in the same component of G_t (by Lemma 6.1). □

Now consider any vertex $x \neq v$ and let s be any step number, $s < t-1$ such that $p_s(x) \neq p_{t-1}(x)$. To show that the theorem holds for x it suffices to show that v is not reachable from $p_s(x)$ without going through x. This follows from Part 2a of the induction hypothesis at $t-1$, for as $p_{t-1}(v) = v$, there is no path from v to x.

Doubling. At step t, v performs a doubling step. The doubling operation does not affect the p pointers; therefore, $G_t^* = G_{t-1}^*$ and $p_t(v) = p_{t-1}(v)$. Part 2 of the theorem follows from Part 2 of the induction hypothesis. To complete the proof we have to show that if $R_t(v) = v$ then v is the largest element on a cycle of G_t.

Assume that $R_t(v) = v$. Then, by Lemma 5.1, there is a path in G_{t-1}^* from $p_{t-1}(v)$ to v, and v is the largest vertex on that path; let P be the shortest such path. We show that P is a path in $G_{t-1} = G_t$.

Consider two consecutive vertices, u and w, on P. Since P is a shortest path, there is a path in G_{t-1}^* from w to v that does not go through u. As v is larger than u, it follows from Part 2 of the induction hypothesis that $p_{t-1}(u) = w$. Since u and w were arbitrary vertices of P, we conclude that P is a path in G_{t-1} from $p_{t-1}(v)$ to v. Thus, v is the largest element on a cycle of G_{t-1}, and also of G_t.

This concludes the proof. □

Lemma 5.2 *For any vertex, v, $R(v)$ is a p-ancestor of v.*

Proof. By Lemma 5.1, for any step, t, there is a path from $p_t(v)$ to $R_t(v)$ in G_t^* for which $R_t(v)$ is the largest vertex; let P be the shortest such path. We show that P is a path in G_t. (An identical proof was used as part of the proof for Theorem 5.1).

Consider two consecutive vertices, u and w, on P. Since P is a shortest path, there is a path in G_t^* from w to $R_t(v)$ that does not go through u. As $R_t(v)$ is larger than u, it follows from Theorem 5.1 that $p_t(u) = w$. Since u and w were arbitrary vertices of P, we conclude that P is a path in G_t from $p_t(v)$ to $R_t(v)$. □

6 Correct Termination

The purpose of the edges is to merge components of the underlying graph. As soon as both enpoints of an edge point to the same vertex the edge is no longer needed. In fact it does no useful work at any later time. We add the following line to the process for edge e:

if $V(e) = V(\overline{e})$ or $\overline{e}$ terminated **then** Terminate

(a) There is a path in G_t from w to v.

(b) $v > w$.

Proof. The proof is by induction on the step number. Initially, the first part of the theorem holds by definition and the second part holds vacuously. Assume the theorem holds immediately before step t. We show it holds after step t.

Edge operations do not affect the correctness of the theorem. There are three types of vertex operations: promotion, doubling, and hooking. The only steps that affect the theorem are those that write to $A(v)$ and $R(v)$. The proof has three cases depending on the operation completed at step t.

Promotion. At step t, v was promoted. Then, $R_t(v) = p_t(v) = v$ and the first part of the theorem follows. Now, $R_{t-1}(v) = v$ and by Part 1 of the induction hypothesis v is the largest vertex on a cycle of G_{t-1}. It follows from Part 2 of the induction hypothesis that for each vertex, x, on a cycle of G^*_{t-1}, no vertex larger than x can be reached from x without using the edge to $p_{t-1}(x)$. As v is the largest vertex on a cycle of G_{t-1}, v is the largest vertex reachable from $p_{t-1}(v)$ in G^*_{t-1}. The only change made during step t is setting $A_t(v)$ and $p_t(v)$ to v. Clearly the second part of the theorem follows for v.

Next, we show Part 2 of the theorem for vertices $x \neq v$. Part 2b of the theorem follows for x by applying Part 2 of the induction hypothesis for $t-1$ and from the observation that $G^*_t = G^*_{t-1} \cup \{< v, v >\}$; i.e., G^*_t is obtained from G^*_{t-1} by adding a self loop (the G_{t-1} edge removed by the promotion appears in G^*_t).

To show Part 2a of the theorem for x it suffices to show that if for any step, s, $s < t-1$, $p_s(x) \neq p_{t-1}(x)$ then v is not reachable in G^*_{t-1} from $p_s(x)$ without going through x. (For by Part 2a of the induction hypothesis, if w is reachable from $p_s(x)$ in G^*_{t-1} without going through x, then there is a path in G_{t-1} from w to x; this path exists in G_t also unless it includes v; but the assertion then shows this path includes x contrary to assumption.) So suppose, for the sake of contradition, that v is reachable from $p_s(x)$ without going through x. It follows from the induction hypothesis (Part 2a) that there is a path in G_{t-1} from v to x and from Part 2b that $x > v$. However, this contradicts Part 1 which states that v is the largest vertex of a cycle of G_{t-1}.

Hooking. At step t, v is hooked to u. So, $u \neq v$, $A_{t-1}(v) = R_{t-1}(v) = v$, and $p_t(v) = R_t(v) = u$; the first part of the theorem follows. Part 2 of the theorem follows for v from Part 2 of the induction hypothesis at $t-1$ and the observation that $p_{t-1}(v) = v$.

The p Pointers. The p pointers are updated as follows.

1. Initially, for each vertex, v, $p(v) := v$.
2. When a root, r, is hooked to a vertex, u, $p(r) := u$.
3. When a vertex, l, is promoted, $p(l) := l$.
4. The p pointers are not modified by the doubling operation.

5 Correctness

The key to the correctness of the algorithm is that for any vertex v, $R(v) = v$ only if v is the largest element in a cycle. We start by proving this property.

Consider a computation of the algorithm and let v be any vertex. Recall that a computation comprises a sequence of steps. When referring to the graph *at step* i we mean the graph after the event of step i was completed and before step $i+1$ has begun. Let $p_t(v)$, $A_t(v)$, $R_t(v)$, $F_t(v)$, and $V_t(v)$ denote the corresponding values at step t and let G_t denote the graph G_p at t (i.e. G_t is shorthand for G_{p_t}).

Define the edge set E_t^* to be the set of edges induced by the collection of hooking steps that took place on or before step t; i.e. $E_t^* = \{\, \langle u, w\rangle \mid p_s(u) = w \ \text{ for some } s \le t \,\}$. Define the *history graph*, G_t^*, to be $G_t^* = (V, E_t^*)$.

The next lemma follows directly from the above definitions by induction, and from the fact that, for any t, $E_t^* \subseteq E_{t+1}^*$. It is used in Theorem 5.1, the main theorem of this section.

Lemma 5.1 *For any vertex, v, and for any step, t, there is a path in G_t^* from $p_t(v)$ to $R_t(v)$ to $A_t(v)$ and $R_t(v)$ is the largest vertex on one such path.*

Proof. Since E_t^* grows with t, the only concern are operations that change A and R. By inspection, the lemma holds for hooking operations. For the doubling operation by vertex v, we note that for some number, $s < t$, v sets $A_t(v)$ to either $A_s(A_{t-1}(v))$, $R_s(A_{t-1}(v))$, $A_s(R_{t-1}(v))$, or to $R_s(R_{t-1}(v))$. If $R_t(v)$ is modified, it is either set to $R_s(A_{t-1}(v))$ or to $R_s(R_{t-1}(v))$. □

Theorem 5.1 *The following two properties hold for any vertex, v, and for any step, t.*

1. *If $R_t(v) = v$ then v is the largest vertex on a cycle of G_t.*
2. *If $p_t(v) \neq p_s(v)$, for some step $s < t$, then for any vertex, w, reachable from $p_s(v)$ in G_t^* without going through v:*

second counter called $Rctr(v)$, also initially zero. For each vertex, v, we need to distinguish the time intervals delimited by its promotions. This is the role of $ctr(v)$: Each time v is promoted, $ctr(v)$ is incremented.

The algorithm makes the following atomicity assumption: When a vertex is accessed its record is retrieved atomically; that is, the four values $A(v)$, $R(v)$, $ctr(v)$ and $Rctr(v)$ are read simultaneously.

For each R pointer, $R(v)$, we store a corresponding counter, $Rctr(v)$; it records the time interval of the vertex $w = R(v)$ being indicated by this pointer. Thus we update $Rctr$ as follows. $Promotion(v)$ is implemented by incrementing $ctr(v)$ and simultaneously assigning $Rctr(v)$ the new value of $ctr(v)$. When $R(v)$ is updated in a doubling operation, $Rctr(v)$ is simultaneously updated: it is given the $Rctr$ value associated with the new value of $R(v)$ (note that two $Rctr$ values are at hand; one for each possible update to $R(v)$). Finally, when a hooking is performed by $root(r)$, $Rctr(r)$ is updated to be equal to $ctr(R(r))$.

Now, it is straightforward to implement F. By definition, vertex $R(v)$ has been promoted with respect to v if and only if $Rctr(v) \neq ctr(R(v))$. So $F(v)$ is assigned $A(v)$ if $Rctr(v) = ctr(R(v))$ and is assigned $R(v)$ otherwise.

We show later, in Section 7.1, that each variable $ctr(v)$ is incremented at most n times; so the counters are readily stored and accessed.

Finally, we describe the implementation of the doubling procedure, the procedure for nonroot vertices, v. v starts by accessing $R(v)$'s record at step t_R. Let ARv and RRv denote the values of $A(R(v))$ and $R(R(v))$, respectively, at step t_R.

If $R(v)$ had been promoted with respect to v at step t_R, v accesses RRv's record, at step t_{FR}, to determine if RRv has been promoted; if not, it sets FRv to ARv and otherwise it sets FRv to RRv. Then v computes $newR(v) := \max\{R(v), RRv\}$ and assigns $newA(v) := FRv$.

While if $R(v)$ had not been promoted with respect to v at step t_R, v accesses $A(v)$'s record, at step t_A. Let AAv and RAv denote the values of $A(A(v))$ and $R(A(v))$, respectively, at step t_A. Next, v accesses RAv's record, at step t_{FA}, to determine if RAv has been promoted; if not, it sets FAv to AAv and otherwise it sets FAv to RAv. Now, v computes $newR(v) := \max\{R(v), RA(v)\}$ and assigns $newA(v) := FAv$.

If $v = newR(v)$, v modifies $newA(v)$ to be v and sets $newctr(v) := ctr(v)+1$ (otherwise, $newctr(v) := ctr(v)$). v concludes with the following simultaneous assignment to its record:

$$R(v), A(v), ctr(v) := newR(v), newA(v), newctr(v)$$

(There is also an associated updated to $Rctr(v)$ which we have not detailed.)

Each root, r, performs the following procedure:

Procedure RootProc(r):

```
1   next(r) := F(next(r))
2   if next(r) = r then next(r) := F(V(new(r)))
    if IsRoot(next(r)) and next(r) ≠ r then
        R(r) := A(r) := next(r)
        next(r) := r
    end if
```

Remark. It seems counterproductive for a vertex to create a new link to one of its descendants. For this reason, roots are restricted to create new links only to other roots. When a vertex, r, becomes a root, it reads a suggestion from the variable $new(r)$, a pointer to an edge e. It then commits to connect to the component containing the vertex, $V(e)$, by writing $F(V(e))$ into $next(r)$, the *current destination of* r (recall $V(e)$ is the current endpoint of edge e). r then repeatedly advances its current destination by replacing $next(r)$ with $F(next(r))$ until $next(r)$ is a root. If the root, $s = next(r)$, is not r, r proceeds to create a link to s, otherwise, it reads a new suggestion from $new(r)$ and repeats the process. Note that between the time r checked whether $next(r)$ is a root, s, and the time the new link is created, s might have created a link and no longer be a root. For the purpose of the analysis, it was easier to advance the current destinations of roots as described in the code (Steps 1 and 2), rather than, for instance, waiting until $F(next(r))$ is a root.

Each nonroot performs the following procedure. (The max function applied to two pointers to vertices returns the pointer to the larger vertex.)

Procedure NonRootProc(v):

Advance the ahead and reference pointers

```
    R(v) := max{R(v), R(F(v))}
    A(v) := F(F(v))
```

Check if v became a leader at this step and if so promote it:

```
    if R(v) = v then
        A(v) := new(v) := next(v) := v
        promote(v)
    end if
```

4.1 Implementation Details

The *Promotion* operation and the computation of F are implemented using counters, as follows. Each vertex v maintains a counter, $ctr(v)$, initially zero. In addition, v stores a

- Function *Forward of* v, $F(v)$, v is a vertex:
  If $IsPromoted(v, R(v))$, return $R(v)$, otherwise return $A(v)$.

Initializations:

For each vertex v*:*

$A(v) := v$
$R(v) := v$
$next(v) := v$
$new(v) := null$

For each directed edge $e = \langle u, v \rangle$*:*

$V(e) := u$
$\bar{e} := \langle v, u \rangle$

Edge Procedure:

The procedure for edge e*:*

1 $V(e) := F(V(e))$
2 **if** $V(e) \neq V(\bar{e})$ **then** $new(V(e)) := \bar{e}$
3 **if** $IsRoot(V(e))$ **then** $V(e) := F(next(V(e)))$

Remark: In the Shiloach/Vishkin algorithm, the edge processes write new A values to the appropriate vertices. Due to the asynchronous nature of our algorithm, the edge processes do not modify the A values but simply write suggestions into a shared variable called new. Thus, an edge, e, connecting $u = V(e)$ to $v = V(\bar{e})$ would suggest to u to link to v by writing $\bar{e}$ into $new(u)$; all the edge processes with endpoint u write to $new(u)$ independently. If u is a root, the vertex process associated with u reads $new(u)$ and uses the value there to create a new link. For reasons that have to do with the analysis, an edge advances its endpoints (Step 1), writes an edge-id into new and not a vertex-id (Step 2) and migrates from roots (Step 3). (The procedure RootProc executed at r defines the update to $next(r)$, also called the current destination of r.)

Vertex Procedures: There are two procedures for the vertices: A procedure for roots and a procedure for nonroots.

The procedure for vertex v

if $R(v) = v$ **then** Execute procedure *RootProc(v)*
else Execute procedure *NonRootProc(v)*
end if

around the (old) cycle. Note that if $R(x) = l$, $R(x)$ remains equal to l as long as $A(x)$ is on the cycle. However, due to the asynchronous nature of the algorithm, testing whether $R(x)$ is a root does not suffice. l can be promoted and create a new link before x tests whether l is a root. Therefore, we tag each vertex in a manner described later. When a vertex is promoted its tag is modified; using the tags a vertex can easily tell whether another vertex has been promoted since the last time it checked. Thus, if x realizes that $R(x)$ had been promoted, even if $R(x)$ is no longer a root, x treats $R(x)$ as if it were $A(x)$. To facilitate the exposition a function F is introduced: For each v, $F(v) = R(v)$ if $R(v)$ had been promoted and is equal to $A(v)$ otherwise. The actual doubling procedure then becomes:

$$R(v) := \max\{R(v), R(F(v))\}$$
$$A(v) := F(F(v))$$

In Section 4 we describe the graph connectivity algorithm and in Section 5 we prove that the algorithm is correct. That is, if the algorithm terminates, upon termination any two vertices, u and v, are in the same component of the input graph if and only if $A(u) = A(v)$. In Section 6 we specify the termination conditions for both the edge processes and the vertex processes (these are details of the algorithm that are not given in Section 4, below). We conclude, in Section 8, by proving an $O(\log n)$ bound on the rounds complexity of the algorithm.

4 Pseudo Code

In addition to $A(v)$ and $R(v)$, a vertex, v, is assigned two variables, $next(v)$ and $new(v)$, whose purpose will become clear below. For each undirected edge, (u, v), in the input graph there are two associated directed edges, $\langle u, v\rangle$ and $\langle v, u\rangle$; each directed edge has an associated process. For each directed edge, $e = \langle u, v\rangle$, $\bar{e}$ points to the directed edge $\langle v, u\rangle$ and $V(e)$ points to a vertex called the *current endpoint* of e.

Auxiliary Functions:

- Function *IsRoot(v)* for vertex v:
  Return true if v is a root, false otherwise; v is a root iff $R(v) = v$.
- Function *Promote(v)* for vertex v:
  Change the tag of v.
- Function *IsPromoted*$(v, R(v))$, $R(v)$ a pointer to a vertex w:
  Return true if the vertex w has been promoted with respect to vertex v.

Our key departure from the Shiloach and Vishkin algorithm is that we allow non-trivial cycles in the underlying graph. So we are forced to modify the doubling procedure, which we do as follows. Given a non-trivial cycle, our goal is to break the cycle at the leader and then make the leader the root of the component. The R pointers help carry out this computation. The following explanation, while not completely correct, provides the intuition behind the algorithm. Suppose the cycle is defined by p pointers, the initial values of the A pointers. Also suppose the doubling is performed by the A pointers, using the assignment $A(v) := A(A(v))$. (Note incidentally, that for an even length cycle if these assignments are performed simultaneously the cycle partitions into two new cycles, each having half the length of the original cycle; the new cycles need not be connected in the A-graph. Clearly we need to retain this connectivity information; this is achieved, implicitly, by the R pointers.) $R(v)$ is defined to be the largest vertex on the path from $p(v)$ to $A(v)$. Thus, initially $R(v) = p(v)$. When $A(v)$ is updated by doubling, $R(v)$ is simultaneously assigned $\max\{R(v), R(A(v))\}$. Clearly, if $v = R(v)$, v is the largest vertex on the cycle (the leader of the cycle).

We say that a vertex, v, *can see a p-leader*, l, if l is a p-leader and $R(v) = l$. The doubling procedure (and the rest of the algorithm) maintains the invariant that for any vertex, v, v, $R(v)$ and $A(v)$ are in the same p-component and there is an A-path from v to a vertex that can see the p-leader of that component. In this sense the doubling procedure does not break components.

The hooking step operates on roots and links them to other vertices. In order to make progress, components with non-trivial cycles eliminate these cycles by promoting the p-leader to a root. This is done in the following manner. When doubling proceeds in a cycle with p-leader l, eventually $R(l)$ becomes equal to l; also, $R(l) = l$ only if l is a leader. Therefore, when $R(l) = l$, the process associated with l promotes l to a root by setting $A(l) = l$; $p(l)$ becomes equal to l as well. At that point the component is rooted and the root is ready to make new links. At any time, at most one vertex in any p-cycle can be promoted and thus the promotion operation does not disconnect p-components.

Note that when a vertex is promoted, some p-paths might be broken. As a consequence, for a vertex, v, $A(v)$ is not guaranteed to be a p-ancestor of v. The heart of the correctness proof of the algorithm is proving that, nonetheless, v is promoted only if it is the largest element in a p-cycle.

Consider a vertex, x, in a cycle with leader l. As soon as l is promoted, l is the most "advanced" vertex in the component; in order to make progress, x must (eventually) recognize that l was promoted and then set $A(x)$ to l; otherwise the pointer $A(x)$ simply chases

Iterate $\log_{3/2} 2n$ *times*
For each vertex v in parallel **do**:
1 $A(v) := A(A(v))$ (* *doubling* *)
Assume each (undirected) edge is represented as two directed edges.
For each directed edge $\langle u, v \rangle$ in parallel **do**:
2 **if** $A(u)$ is a root and $A(u) < A(v)$ **then**
$A(A(u)) := A(v)$ (* *hooking* *)
3 **if** $A(u)$ is a root and $A(u)$ did not get new children in Steps 1 or 2 **then**
$A(A(u)) := A(v)$ (* *stagnant hooking* *)
For each vertex v in parallel **do**:
4 $A(v) := A(A(v))$ (* *doubling* *)

To show that the algorithm terminates in $O(\log n)$ steps, each component of the input graph is assigned a potential with respect to the A-graph: it is equal to the number of A-components plus the sum of the edge heights of all the A-components (the edge height of a one vertex component is defined to be one). The potential of a component of the input graph is reduced to two when, for each pair, u, v, of vertices in this component, $A(u) = A(v)$. Prior to this point, on each iteration of the above algorithm, the potential of the component is reduced by at least one third. For, in each doubling, A-components of height greater than one have their height reduced by at least one third, while A-components of height one are hooked to each other in Steps 2 and 3.

Unfortunately, it is not clear how to implement this algorithm directly in the APRAM model without synchronizing all the processes after each step. The progress obtained in the doubling step is guaranteed only as long as the graph remains acyclic; to ensure this, the hooking of Step 2 must be a controlled hooking and the stagnant hooking, performed in Step 3, is key to achieving $O(\log n)$ running time; however, the stagnant hooking can be executed by the process associated with $\langle u, v \rangle$ only after it synchronizes with all the processes that are creating new links to $A(u)$. This can be done in constant time in a synchronous model but it is not clear how to achieve this asynchronously in $o(\log n)$ rounds.

The description of our algorithm uses an auxiliary mapping p. Informally, p records all the links created by the algorithm and is unaffected by the doubling step. To facilitate the transition to our algorithm we define the p pointer with respect to the Shiloach/Vishkin algorithm although it is not needed there. p is initialized to the identity mapping on V. Whenever a root, r, is hooked to a vertex, v, (in either Step 2 or Step 3) $p(r)$ is set to v. p is not modified during the doubling steps (Steps 1 and 4). The same definition for p is used in the informal description of our algorithm, below. The graph G_p is called the *underlying graph*.

3 The Algorithm

The input is a graph $G = (V, E)$. Let $n = |V|$, $e = |E|$. The goal of the algorithm is to find a mapping, $f : V \mapsto V$, such that for each pair of vertices u, v, $f(u) = f(v)$ if and only if u and v are connected in G. We describe a connectivity algorithm which uses $O(n + e)$ processes and $O(\log n)$ rounds.

The algorithm computes a series of refinements. It manipulates two mappings, A and R, on the vertices of G. We call $A(v)$ the vertex *ahead* of v and $R(v)$ the *reference* of v. The algorithm starts by initializing A and R to the identity mapping on V; A and R are then modified while maintaining that at any point in time, for any vertex v, $A(v)$, $R(v)$ and v are connected in G. We show that upon termination we have the desired property: u and v are connected in G if and only if $A(u) = A(v)$. As an aid to the analysis the algorithm uses an additional mapping: $p : V \mapsto V$, the *parent* mapping; however, the algorithm does not explicitly manipulate p; in fact, it is unaware of p.

A few definitions are helpful. A *pointer graph* is a directed graph in which each vertex has out degree one; an edge may be a self-loop. A pointer graph is a *tree graph* if each cycle is a self-loop. A pointer graph is *weighted* if there is a weight associated with each vertex. We will use v to denote both the vertex and its weight, where no ambiguity results. By definition, each component of a pointer graph has exactly one cycle, possibly trivial (a self-loop). In a weighted pointer graph with unique weights, the largest element in a cycle is called a *leader*; each component has exactly one leader. If the component has a trivial cycle, the vertex in the trivial cycle is called a *root* and the component is said to be *rooted*. For any mapping, f, from V to V, define, E_f, the edge set *induced* by f, by $E_f = \{ \langle v, f(v) \rangle \mid v \in V \}$; likewise, define G_f, the *pointer graph induced by* f, by $G_f = (V, E_f)$. An f-property refers to the corresponding property with respect to graph G_f. For instance, when we say u is an f-ancestor of v we mean u is an ancestor of v in G_f (i.e., u is reachable from v in G_f).

Our point of departure is the PRAM algorithm of Shiloach and Vishkin [SV82]. Our algorithm is substantially different to the Shiloach/Vishkin algorithm. However, in order to understand our approach it is helpful to review the Shiloach/Vishkin algorithm and explain why it is difficult to implement it directly on the APRAM model.

The Shiloach/Vishkin algorithm manipulates a mapping, A, and constructs a tree graph G_A on the vertex set of the input graph. Initially, $A(v) = v$ for each vertex v. On termination, $A(u) = A(v)$ if and only if u and v are in the same component of the input graph. A vertex, v, is called a *root* if $A(v) = v$.

Their algorithm follows.

2 The APRAM Model

We provide here a short description of the APRAM model, which, though incomplete, suffices for understanding the algorithm presented in this paper. For the full details the reader is referred to [CZ91]. The APRAM is an asynchronous shared memory parallel computation model. It comprises a collection of *processes*, each executing operations called *events*. An APRAM computation is a serialization of all the events of all the processes; there may be more than one such serialization.

In order to assess the complexity of asynchronous algorithms, the APRAM model provides a complexity measure called the *rounds complexity*: the global clock used by the PRAM model is replaced by a *virtual clock*. This approach was introduced in [PF77] and used in [AFL83,LF81,KRS88b] and is common in the area of distributed computing (see [Awe87], [Awe85,AG87]). Consider a computation, C. A *virtual clock of* C is an assignment of unique virtual times to the events of C; the times assigned are a non-decreasing function of the event number.

The virtual clock is meant to correspond to the "real" time at which the operations occurred in one possible execution of the algorithm, called a computation. The time difference between two consecutive events of a process is called the *duration* of the later event. The length of a computation is the time assigned to the last event in the computation.

Under the rounds complexity, a virtual clock is valid if the duration of each event is at most one. In effect, a computation is divided into contiguous segments, called *rounds*, where each segment contains at least one event from each process. The complexity of a computation is the number of rounds in the computation maximized over all possible subdivisions. The rounds complexity of an algorithm on a given input is the rounds complexity of the computation with largest complexity. The rounds complexity of an algorithm is then the rounds complexity on a given input maximized over all inputs of a given size. [CZ90] uses probabilistic variants of the virtual clock in order to obtain complexity measures which measure the implicit costs of synchronization.

It is often easier to analyze algorithms in terms of higher level constructs, each built of a constant number of low level building blocks. [CZ91] shows that the resulting complexity will be off by only a constant multiplicative factor, which we ignore, as is often done in asymptotic analysis. This allows one to ignore low level details of the algorithm. In the discussion of the connectivity algorithm we refer to operations rather than events, where each operation comprises a constant number of events.

of synchronization. We do not define the notion of explicit and implicit costs formally, for we do not think we have enough experience to justify definitive definitions.

The basic measure used in [CZ91] is the rounds complexity measure of distributed computing; this allows the explicit costs of synchronization to be measured. [CZ90] introduces two other measures, the *bounded delays* measure and the *unbounded delays* measure, which allow the implicit costs of synchronization to be measured. This paper will only be concerned with the rounds complexity measure.

There are a considerable number of PRAM algorithms for computing the connected component of an undirected graph, [HW90,HCS79,Wyl79,SV82,Vis84,Gaz86,CV87]. They all adopt the following basic strategy: select a subset of the edges forming a forest and then compress each tree to a star (a tree of depth 1); iterate. Shiloach and Vishkin gave the first $O(\log n)$ algorithm by interleaving these steps; it uses $n + e$ processes under the CRCW PRAM (n is the number of vertices and e is the number of edges). Cole and Vishkin gave an algorithm with time complexity $T(n) = O(\log n)$ using $(n + e)\alpha(e, n)/T(n)$ processes, where $\alpha(e, n)$ is the inverse Ackerman function. Their algorithm is optimal for $e \geq n \log^* n$; however, the constants hidden behind the big-O notation are quite large due to the use of expander graphs.

In this paper we present an asynchronous APRAM algorithm for graph connectivity; an algorithm which is significantly different from all known PRAM algorithms. At a very high level, our approach is similar to the approach of Shiloach and Vishkin; however, their algorithm depends substantially on the implicit synchronization of the PRAM model and we have not been able to obtain an APRAM algorithm simply by slightly modifying their algorithm.

We relaxed some of the constraints (on building forests) present in the known PRAM algorithms; in fact, we may build structures having a single cycle. In order to cope with the cycles in the compression phase, we need to introduce further pointers. This results in a substantially different algorithm. It also has the effect of reducing the required synchronization, which makes the algorithm's progress appear rather chaotic; nonetheless, we can prove that the algorithm terminates correctly in $O(\log n)$ rounds (rounds are defined in Section 2). (The correctness of the algorithm relies on the ability to read and write records of 4 fields atomically.) The asynchronicity of the algorithm makes the proof of correctness and the complexity analysis substantially more challenging than would be the case for a PRAM algorithm.

Before describing our algorithm we briefly review the APRAM model and the rounds complexity measure.

costs from the user. In many cases, an algorithm may have to be redesigned in order to allow it to run efficiently in an asynchronous environment. In this paper, we are concerned with the design of a connectivity algorithm which performs well in the presence of the non-uniformity introduced by asynchrony. Another approach to the problem of asynchrony is to seek to efficiently compile PRAM algorithms to operate in asynchronous environments; this approach is followed by Martel, Park and Subramonian [MPS89,MSP90]. They give efficient randomized simulations of arbitrary PRAM algorithms on an asynchronous model, given certain architectural assumptions (e.g., the availability of a compare&swap instruction). It is not clear whether similar deterministic compilers exist; in the absence of such compilers, to obtain deterministic algorithms it appears necessary to design them in an asynchronous environment; this is the focus of our work.

In a companion paper, [CZ91], the APRAM, a model for asynchronous parallel computation, is defined (this model was first suggested by [KRS88a]). By contrast with the PRAM model, there is no global clock governing the computation; all synchronization required by a particular application has to be provided explicitly. Consequently, we expect the complexity of asynchronous algorithms to be at least as large as that of their synchronous counterparts (for now the costs of synchronization have to be explicitly paid for).

[CZ91] discusses two algorithms, summation along an implicit binary tree and recursive doubling. They show that efficient APRAM algorithms can be obtained from known PRAM algorithms by replacing global synchronization by local synchronization. The resulting algorithms have a more flexible computation structure and, thus, perform better in an asynchronous environment.

In this paper we consider a third problem: computing the connected components of an undirected graph. Our algorithm is substantially different from all known PRAM algorithms. The key to designing the algorithm was to identify a structure for the computation that would be flexible enough to allow the computation to proceed efficiently, while at the same time be rigid enough to guarantee that the algorithm terminates correctly. The result is an apparently chaotic algorithm, whose description is simple, but whose analysis is not.

The presence of asynchrony calls for new complexity measures. Before discussing these, we remark that there are two types of synchronization costs: *explicit* and *implicit*. By explicit costs we mean the overhead for achieving synchronization. This could be, for instance, the cost of executing extra code that must be added to the algorithm in order to synchronize. When processes proceed at different speeds, if the algorithm is required to proceed in lock step, the time required to execute a step is dictated by the slowest process. By implicit costs we refer to the cost associated with lock step execution apart from the explicit costs

An Asynchronous Parallel Algorithm for Undirected Graph Connectivity*

Richard Cole Ofer Zajicek

New York University

Abstract

An algorithm for computing the components of an undirected graph in the (asynchronous) APRAM model is given; the algorithm uses $O(n+e)$ processes and $O(\log n)$ rounds.

1 Introduction

This paper is part of an investigation of the effect of process asynchrony on parallel algorithm design. As is well known, the main effort in parallel algorithm design has employed the PRAM model. This model hides many of the implementation issues, allowing the algorithm designer to focus first and foremost on the structure of the computational problem at hand – synchronization is one of these hidden issues.

In turn, the work on sychronization is part of a broader research effort which has sought to take into account some of the implementation issues hidden by the PRAM model. Broadly speaking, two major approaches have been followed. One body of research is concerned with asynchrony and the resulting non-uniform environment in which processes operate[1] [CZ89,CZ91,CZ90,Nis90,MPS89,MSP90]. The other body of research has considered the effect of issues such as latency to memory, but assumes a uniform environment for the processes [PU87,PY88,AC88,ACS89,Gib89].

The PRAM is a synchronous model and thus it strips away problems of synchronization. However, the implicit synchronization provided by the model hides the synchronization

*The work was supported in part by NSF grants CCR-8902221 and CCR-8906949, and by a John Simon Guggenheim Memorial Foundation Fellowship.

[1] We distinguish between processes and processors in order to emphasize that the APRAM is not a machine model but rather a programming model; it is the task of a compiler to implement the programming model on actual machines. The term processor will be used to refer to this component of a machine.