Finding Strongly Connected Components in Parallel using $O(\log^2 n)$ Reachability Queries

Warren Schudy
Brown University*

October 6, 2007

Abstract

We give a randomized (Las-Vegas) parallel algorithm for computing strongly connected components of a graph with n vertices and m edges. The runtime is dominated by $O(\log^2 n)$ parallel reachability queries; i.e. $O(\log^2 n)$ calls to a subroutine that computes the descendants of a given vertex in a given digraph. Our algorithm also topologically sorts the strongly connected components.

Using Ullman and Yannakakis's [21] techniques for the reachability subroutine gives our algorithm runtime $\tilde{O}(t)$ using mn/t^2 processors for any $(n^2/m)^{1/3} \leq t \leq n$. On sparse graphs, this improves the number of processors needed to compute strongly connected components and topological sort within time $n^{1/3} \leq t \leq n$ from the previously best known $(n/t)^3$ [19] to $(n/t)^2$.

1 Introduction and main results

Breadth-first and depth-first search have many applications in the analysis of directed graphs. Breadth-first search can be used to compute the vertices that are reachable from a given vertex and directed spanning trees. Depth-first search can: solve these problems, determine if a graph is acyclic, topologically sort an acyclic graph and compute strongly connected components (SCCs) [20]. Efforts to parallelize these algorithms have met with mixed success.

Reif [18] shows that if you insist on a particular ordering of the edges, finding a depth-first-search tree is P-complete and hence unlikely to be in NC. If the search is allowed to explore the children of a vertex in any order, a DFS tree can be found in polylog time with $O(n \cdot n^{2.38})$ processors [1].

Applications of DFS seem easier to solve in parallel than DFS itself. Let *transitive-closure bottleneck problems* [12] refer to breadth-first search and the above applications of BFS and DFS, but not DFS itself. All these transitive-closure bottleneck problems can be solved in polylog time with $n^{2.38}$ processors using matrix-exponentiation techniques to compute the transitive closure of the graph [7, 4]. Spencer [19] gives algorithms for these transitive-closure bottleneck problems that trade-off time and work, with runtime $\tilde{O}(t)$ on $\tilde{O}((n/t)^3 + m/t)$ processors for any $1 \leq t \leq m$.

*Work done at Google Inc., Mountain View CA. Email: ws@cs.brown.edu

Breadth-first search and its applications seem easier to parallelize than applications of DFS such as cycle detection. Ullman and Yannakakis [21] give a clever technique for computing breadth-first search in parallel, with runtime $\tilde{O}(t)$ with $\tilde{O}(mn/t^2)$ processors as long as $(n^2/m)^{1/3} \leq t \leq \sqrt{n}$. In Appendix A, we show that (at least for reachability) the upper-bound on t can be relaxed to $t \leq n$. If $t = n$, this algorithm is work-efficient, so the restriction $t \leq n$ is harmless. We remark that for $1 \leq t \leq n$, if $t \leq n^2/m$, Ullman and Yannakakis's algorithm requires fewer processors than Spencer's, but if $t \geq n^2/m$, Spencer's is more efficient. It follows that Ullman and Yannakakis's algorithm is better for sparse graphs, and Spencer's is better for dense graphs. We also observe in Appendix A that on graphs with longest finite shortest path less than t (like diameter except we ignore disconnected pairs of vertices), we can trivially achieve runtime $\tilde{O}(t)$ with m/t processors. The results in this paragraph are for breadth-first search only, not for cycle detection, SCC or topological sort.

Lemma 1. *[21] With high probability, reachability can be computed in time $\tilde{O}(t)$ using $\tilde{O}(mn/t^2)$ processors as long as $(n^2/m)^{1/3} \leq t \leq n$. If the longest finite shortest path is at most t, m/t processors suffice for runtime $\tilde{O}(t)$.*

If one could compute strongly connected components and topological sort using BFS instead of DFS, one could use Ullman and Yannakakis's better result for that problem on the sparse graphs frequently encountered in applications. Coppersmith, Fleischer, Hendrickson and Pinar [6, 5] give a simple parallel divide and conquer algorithm for computing SCCs using reachability queries. They prove $O(m \log n)$ serial runtime, but do not prove that extra processors reduce the runtime. McLendon, Hendrickson, Plimpton and Rauchwerger [14, 13, 16] successfully apply Coppersmith et. al.'s algorithm to an application in scientific computing (discrete ordinates method for radiation transport).

Our main result:

Theorem 2. *There is an algorithm for the topological sort and strongly connected components problems with runtime $O(\tau \log^2 n)$, where τ is the runtime for a reachability query (executed in parallel).*

Put another way, if there is a p-processor NC algorithm for reachability, then there is a p-processor NC algorithm for cycle detection, topological sort and strongly connected components.

Corollary 3. *There is an algorithm for SCC and topological sort with runtime and number of processors given in Lemma 1 (the $\tilde{O}$ notation hides the extra factor of $\log^2 n$).*

On sparse graphs, Corollary 3 improves the number of processors needed to achieve runtime $n^{1/3} \leq t \leq n$ from the previously best known [19] $(n/t)^3$ to $(n/t)^2$.

Many special cases of these problems admit efficient parallel algorithms. There are linear-processor, polylog-time algorithms for finding connected components of *undirected* graphs (see [9] for a survey) and strongly connected components and topological sort of *planar* directed graphs [10, 2, 11]. Kao and Klein [11] reduce finding an ear decomposition of a strongly connected graph to finding directed spanning trees. They also reduce planar reachability to planar topological sort and planar SCC.

Algorithm 1 A quicksort-like SCC algorithm in the spirit of [19, 6, 5, 3, 8]

SinglePivot(V):

- Choose a random pivot vertex v
- Use reachability queries to compute:
  - B = vertices reached from vertex v
  - C = vertices that reach and are reached from v. /* $C \subseteq B$ *is an SCC.* */
- In parallel, compute *SinglePivot*($V \setminus B$) and *SinglePivot*($B \setminus C$).
- Return the SCCs from the recursive calls in the following topological order: *SinglePivot*($V \setminus B$), then C, and finally *SinglePivot*($B \setminus C$).

Most of this paper is devoted to our SCC/topological sort algorithm. The SCC problem is sufficient for motivating our algorithm and analysis, so we henceforth only occasionally mention that our algorithm returns the SCCs in topologically sorted order.

Notation: let (V,E) be a directed graph with n vertices and m edges. V, n and m refer either to the entire graph or to a subgraph resulting from a recursive step of our algorithm; the meaning should be clear from context. Without loss of generality, we will assume that every vertex has a self-loop; note this implies $m \geq n$.

Definition 4. Vertex u *reaches* vertex v, denoted $u \rightsquigarrow v$, if there is a directed path from u to v. If u reaches v we say u is an *ancestor* of v. If in addition $u \neq v$, we say u is a *strict ancestor* of v. For a vertex set S and vertices $u, v \in S$, let $u \overset{S}{\rightsquigarrow} v$ denote that there is a path from u to v in the subgraph induced by S.

Several existing SCC algorithms [19, 6, 5, 3, 8] are variants of quicksort. All vertices in an SCC have identical ancestor and descendant sets. This implies that for a pivot vertex v you can divide and conquer, recursing on the vertices reachable from v and the vertices not reachable from v. The SCC containing v is precisely those vertices that are both reachable from v and reach v, so this SCC can be output and removed (this is the base case). See Algorithm 1.

Unfortunately the SinglePivot Algorithm and the algorithms of [19, 6, 5, 3, 8] have recursion depth $O(n)$ on a graph with no edges. This is the deficiency remedied by our MultiPivot Algorithm.

In our solution we sample several vertices instead of just one and compute which vertices are reachable from any of the sampled vertices (i.e. we compute the union of the descendants of the sampled vertices). We do not know how to choose the sample size a priori, so we simply do a binary search for a sample size that divides the problem evenly. See Algorithm 2 for the definition of the MultiPivot Algorithm and Figure 1 for an illustration. Figure 2 shows the relationship between the sets of the MultiPivot Algorithm. The subtle part of the analysis is showing that $B \setminus (A \cup C)$ (as defined in the MultiPivot Algorithm) is not too big.

Algorithm 2 MultiPivot Algorithm. See Appendix B for heuristic variations.

MultiPivot(V):

- Permute the vertices in V randomly and assign corresponding indices $1, 2 \cdots n$.
- Do a binary search for the smallest index s such that the first s vertices together reach vertices that induce a subgraph with at least $m/2$ edges. */* Uses $O(\log n)$ reachability queries. */*
- Use reachability queries to compute:
  - A = vertices reached from vertex set $\{1, 2, \cdots s-1\}$
  - B = vertices reached from vertex s
  - C = vertices that reach and are reached from s. */* $C \subseteq B$ is an SCC. */*
- In parallel, recursively compute the strongly connected components of $V \setminus (A \cup B)$, $A \setminus B$, $B \setminus (A \cup C)$, and $(A \cap B) \setminus C$.
- Return the SCCs in the following topological order: the SCCs from the recursive call $MultiPivot(V \setminus (A \cup B))$, then $MultiPivot(A \setminus B)$, then the SCC C, then $MultiPivot(B \setminus (A \cup C))$, and finally $MultiPivot((A \cap B) \setminus C)$.

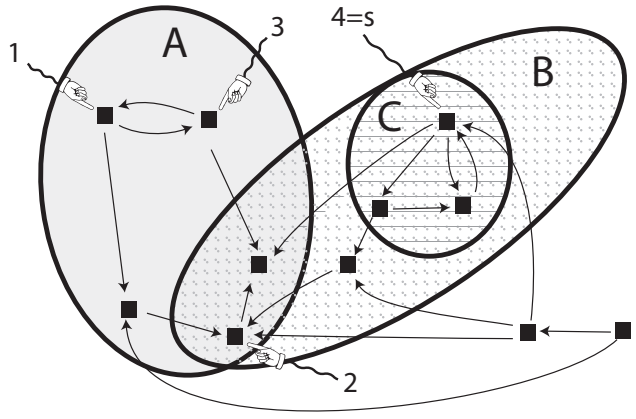

Figure 1: Illustration of MultiPivot Algorithm. The first 3 vertices reach only 6 edges out of 18, but the first four vertices reach 13 out of 18 edges and hence $s = 4$. The self-loops are omitted to reduce clutter.

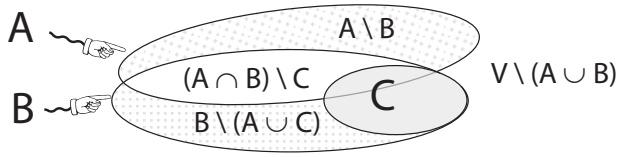

Figure 2: Venn Diagram for MultiPivot Algorithm.

We prove Theorem 2 via two lemmas. The straightforward proof of the following Lemma is in Section 2.

Lemma 5 (Correctness)**.** *The MultiPivot Algorithm correctly computes the SCCs and a topological sort thereof.*

The interesting proof of the following Lemma is in Section 3.

Lemma 6 (Runtime)**.** *For any $\gamma > 1$, with probability at least $1 - n^{1-\gamma}$, the MultiPivot Algorithm takes time $O(\gamma\tau(\log n)^2)$ on a CRCW PRAM with p processors, where $\tau \geq m/p + \log p$ is the time required for a reachability query.*

2 Analysis: correctness (Lemma 5)

Note: this section is straightforward.

Since $C \subseteq B$ by definition of C, we have (see also Figure 2):

Claim 7. *The sets $\{V \setminus (A \cup B), A \setminus B, C, B \setminus (A \cup C), (A \cap B) \setminus C\}$ form a partition of V.*

Lemma 8. *Every set the algorithm outputs (claimed to be an SCC) satisfies the property that every vertex in it is reachable from every other.*

Proof. Every vertex in C can reach and be reached from s by definition, so therefore any vertex in C can reach any other vertex, via s. □

Lemma 9. *The topological sort has no edges going from a set to another set before it in the order.*

Proof. Recall that the sets are output in the order $V \setminus (A \cup B)$, $A \setminus B$, C, $B \setminus (A \cup C)$, $(A \cap B) \setminus C$. We show in turn that each of these sets has no edges to it from sets to the right of it in the order.

- There are no edges into $V \setminus (A \cup B)$ from the other sets because A and B have no edges leaving them.
- There are no edges into $A \setminus B$ from C, $B \setminus (A \cup C)$, $(A \cap B) \setminus C \subseteq B$ because B has no edges leaving it.
- There are no edges into C from $B \setminus (A \cup C)$, $(A \cap B) \setminus C \subseteq B \setminus C$ because any vertex in B that can reach a vertex in C can also reach s and is therefore in C.

- There are no edges into $B \setminus (A \cup C)$ from $(A \cap B) \setminus C$ because there are no edges leaving A.

□

Proof of Lemma 5. Claim 7 implies that our output includes every vertex exactly once.

Since vertex s is always in C, each recursive call has at least 1 fewer vertices than its parent so the algorithm terminates.

Lemma 8 shows that we never output an SCC that is too big. Lemma 9 implies that we never output an SCC that is too small and that our topological sort is correct. □

3 Analysis: runtime (Lemma 6)

It is easy to see that all of the steps of the algorithm except for the reachability queries can be completed in time $O(m/p + \log p) \leq \tau$, so the reachability queries dominate the runtime. (See [17] for an algorithm that computes random permutations in $o(\log n)$ time using $o(n)$ processors).)

We can perform reachability queries on as many disjoint subgraphs as we want at once easily: simply remove edges between the subgraphs, add a new source vertex, and add edges from the source vertex to the source sets of each subgraph. Each SCC call uses at most $O(1) + \lg n$ reachability queries, so it remains to show that the depth of the recursion tree is $O(\log n)$ with high probability.

Definition 10. Let $\mu(S)$ denote the number of edges in the subgraph induced by vertex set S. Note that $\mu(V) = m$.

As in the analysis of quicksort, we must show that we divide more or less evenly to get a good runtime. Some progress is easy to show:

Lemma 11. *All vertex sets S in $\{V \setminus (A \cup B),\ A \setminus B,\ (A \cap B) \setminus C\}$ satisfy $\mu(S) \leq \mu(V)/2$.*

Proof. By the choice of s we know $\mu(A) < m/2$ and $\mu(A \cup B) \geq m/2$, which implies $\mu((A \cap B) \setminus C),\ \mu(A \setminus B) \leq \mu(A) < m/2$ and $\mu(V \setminus (A \cup B)) \leq m/2$. □

Definition 12. Define $T(S)$ for vertex set S as follows:

$$T(S) \equiv \{\, (u, v) \in S \times S \mid u \neq v \text{ and } u \overset{S}{\rightsquigarrow} v \,\}$$

The following Lemma, inspired by Spencer's Lemma 5.4 [19], is the core of our analysis:

Lemma 13. *With probability at least 1/3, $|T(B \setminus (A \cup C))| \leq \frac{3}{4}|T(V)|$.*

We first prove Lemma 13. Then we use Lemmas 11 and 13 and the potential function $\max(|T(V)|, 3/4) \cdot \mu(V)$ to prove Lemma 6.

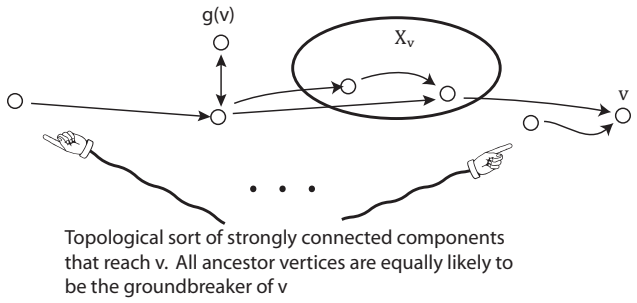

Figure 3: Illustration of Lemma 16. The self-loops are omitted to reduce clutter.

3.1 Proof of Lemma 13

Definition 14. The *groundbreaker* of a vertex v, denoted $g(v)$, is the ancestor of v with minimum index.

The groundbreaker $g(v)$ of a fixed vertex v is a random variable because the permutation of the vertices is picked randomly.

Definition 15. Let X_v be the set of strict ancestors of v that have the same groundbreaker as v but are not in the same SCC as $g(v)$. Precisely:

$$X_v = \{\, w \neq v \mid w \rightsquigarrow v \text{ and } g(w) = g(v) \text{ and } w,\, g(v) \text{ are in different SCCs} \,\}$$

Let the number of strict ancestors of v be α_v.

Lemma 16. *For any $v \in V$, we have $\mathbf{E}\left[|X_v|\right] \leq \alpha_v/2$ and $0 \leq |X_v| \leq \alpha_v$.*

Proof. See Figure 3. Consider a topological sort of the ancestors of v (with vertices in same SCC adjacent in the sort). The groundbreaker of v is the first ancestor chosen, so all the $\alpha_v + 1$ ancestors (including v itself) have equal probability of groundbreaking. If an ancestor u of v has the same groundbreaker as v but is in a different SCC from $g(v)$, then u must be after $g(v)$ in the topological sort. The expected number of vertices strictly between $g(v)$ and v in the topological sort is:

$$\frac{1}{\alpha_v + 1}\left(0 + \sum_{i=1}^{\alpha_v}(i-1)\right) = \frac{1}{\alpha_v + 1} \cdot \frac{(\alpha_v - 1)(\alpha_v - 1 + 1)}{2} \leq \alpha_v/2.$$

□

Definition 17.

$$Y = \{\, (u, v) \in V \times V \mid u \in X_v \,\}.$$

Corollary 18. *We have* $\Pr\left(|Y| \leq \frac{3}{4}|T(V)|\right) \geq 1/3$.

Proof. Since $\sum_{v \in V} \alpha_v = |T(V)|$, $\mathbf{E}[|Y|] = \mathbf{E}[\sum_v |X_v|] \leq |T(V)|/2$ and $0 \leq |Y| \leq |T(V)|$. We use a Markovian argument: the probability of Y being less than $\frac{3}{4}|T(V)|$ subject to these constraints is minimized when $|Y| = \frac{3}{4}|T(V)| + 1$ with probability 2/3 and $|Y| = 0$ with probability 1/3. □

To finish the proof of Lemma 13 we argue that $T(B \setminus (A \cup C)) \subseteq Y$, and hence $|T(B \setminus (A \cup C))| \leq \frac{3}{4}|T(V)|$ with probability at least 1/3 by Corollary 18. To see this, consider some $(w, v) \in T(B \setminus (A \cup C)$. Clearly both w and v are reachable from s but not from $1 \cdots s-1$, so $g(v) = g(w) = s$. The SCC of s is C, so u and v are not in the groundbreaker's SCC by definition. Therefore $w \in X_v$ so $(w, v) \in Y$.

3.2 Putting the pieces together

In this section we use Lemmas 11 and 13 to prove that the recursion tree has depth $O(\log n)$ with high probability, finishing the proof of Lemma 6. It is easy to show that the expected work is small, but for a parallel algorithm we need to show that the maximum depth, not just the average depth, is small. We show every vertex has probability at most $n^{-\gamma}$ of exceeding $O(\gamma \log n)$ depth for any $\gamma > 0$, implying by a union bound that all vertices finish within that depth with probability at least $1 - n^{1-\gamma}$.

We use $\max(T(V), 3/4) \cdot \mu(V)$ as our potential function. Declare a call to SCC(V) to be *successful* if all of the non-empty sets V' it recurses on satisfy $\max(T(V'), 3/4) \cdot \mu(V') \leq \frac{3}{4} \max(T(V), 3/4) \cdot \mu(V)$. If $T(V) \geq 1$, we have the probability of success is at least 3/4 by Lemmas 11 and 13. If $T(V) = 0$, each vertex is isolated so $B \setminus (A \cup C) = \{\}$, so by Lemma 11 the probability of success in this case is 1. Clearly we have $\min(3/4, T(V)) \cdot m \leq n(n-1)n(n+1) \leq n^4$ (assuming $|V| \geq 2$). We assume that every vertex has a self-loop, so $3/4 \cdot \mu(V') \geq 3/4$ for any non-empty V'. It therefore suffices to show that with probability $1 - n^{-\gamma}$ we have at least $4 \log(4n/3)/\log(4/3) \leq 14 \log n$ sucessses (for n sufficiently large) before the depth of a fixed vertex exceeds $63\gamma \log n$, where log denotes the natural, base-e logarithm. This probability is bounded by the probability that $63\gamma \log n$ independent coin flips, each with probability of success 1/3, has less than $14 \log n$ successes.

Chernoff bound for any $0 < \delta < 1$ [15]:

$$\Pr(Z < (1-\delta)\mu) \leq e^{-\mu\delta^2/2}$$

Letting Z be the number of successes, $\mu = 21\gamma \log(n)$, and $\delta = 1/3$, yields

$$\Pr(Z < 14\gamma \log(n)) \leq n^{-\frac{21}{18}\gamma} \leq n^{-\gamma}$$

Clearly for $\gamma > 1$ we have $14\gamma \log(n) > 14 \log n$, so this is sufficient.

4 Future Work

The most imporant related open problem is how to answer reachability queries efficiently. For example, consider a constant-degree (sparse) directed graph with longest finite shortest path $\Theta(n)$ and one processor per edge ($m = p = \Theta(n)$). Is it possible to answer a reachability query on this graph in $\tilde{o}(\sqrt{n})$ time?

5 Acknowledgements

I'd like to thank D. Sivakumar for suggesting the parallel strongly connected components problem, Glencora Borradaile, Maurice Herlihy and Claire Mathieu for advice that dramatically improved the presentation, and Google Inc. for free food.

References

[1] Alok Aggarwal, Richard J. Anderson, and Ming-Yang Kao. Parallel depth-first search in general directed graphs. *SIAM J. Comput.*, 19(2):397–409, 1990.

[2] D. Bader. A practical parallel algorithm for cycle detection in partitioned digraphs. Technical Report AHPCC-TR-99-013, Electrical & Computer Eng. Dept., Univ. New Mexico, Albuquerque, NM, 1999.

[3] Roderick Bloem, Harold N. Gabow, and Fabio Somenzi. An algorithm for strongly connected component analysis in n log n symbolic steps. *Form. Methods Syst. Des.*, 28(1):37–56, 2006.

[4] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM Symposium on Theory of Computing*, pages 1–6, New York, NY, USA, 1987. ACM Press.

[5] Don Coppersmith, Lisa Fleischer, Bruce Hendrickson, and Ali Pinar. A divide-and-conquer algorithm for identifying strongly connected components. Technical Report RC23744, IBM Research, 2005.

[6] Lisa Fleischer, Bruce Hendrickson, and Ali Pinar. On identifying strongly connected components in parallel. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 505–511, London, UK, 2000. Springer-Verlag.

[7] Hillel Gazit and Gary L. Miller. An improved parallel algorithm that computes the bfs numbering of a directed graph. *Inf. Process. Lett.*, 28(2):61–65, 1988.

[8] Raffaella Gentilini, Carla Piazza, and Alberto Policriti. Computing strongly connected components in a linear number of symbolic steps. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[9] John Greiner. A comparison of parallel algorithms for connected components. In *SPAA '94: Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, pages 16–25, New York, NY, USA, 1994. ACM Press.

[10] M.-Y. Kao. Linear-processor nc algorithms for planar directed graphs i: Strongly connected components. *SIAM J. Comput.*, 22:431–459, 1993.

[11] M.-Y. Kao and P. N. Klein. Towards overcoming the transitive-closure bottleneck: efficient parallel algorithms for planar digraphs. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 181–192, New York, NY, USA, 1990. ACM Press.

[12] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. *Handbook of theoretical computer science (vol. A): algorithms and complexity*, pages 869–941, 1990.

[13] William McLendon, III, Bruce Hendrickson, Steve Plimpton, and Lawrence Rauchwerger. Finding strongly connected components in parallel in particle transport sweeps. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 328–329, 2001.

[14] William McLendon, III, Bruce Hendrickson, Steven J. Plimpton, and Lawrence Rauchwerger. Finding strongly connected components in distributed graphs. *J. Parallel Distrib. Comput.*, 65(8):901–910, 2005.

[15] M Mitzenmacher and E Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*, chapter 4.2, page 66. Cambridge University Press, 2005.

[16] Steve Plimpton, Bruce Hendrickson, Shawn Burns, and William McLendon, III. Parallel algorithms for radiation transport on unstructured grids. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 25, Washington, DC, USA, 2000. IEEE Computer Society.

[17] Sanguthevar Rajasekaran and John H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, 1989.

[18] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[19] Thomas H. Spencer. Time-work tradeoffs for parallel algorithms. *J. ACM*, 44(5):742–778, 1997.

[20] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

[21] Jeffrey D. Ullman and Mihalis Yannakakis. High probability parallel transitive-closure algorithms. *SIAM J. Comput.*, 20(1):100–125, 1991.

A Reachability

The key ideas are in Ullman and Yannakakis [21], so we only sketch our variant of their work.

On graphs with longest finite shortest path that's less than t, breadth-first-search can be parallelized completely naively, using only m/t processors to process the edges in time. Ullman and Yannakakis [21] proved that $\tilde{O}(t)$ time is achievable with $p = nm/t^2$ processors as long as

Algorithm 3 Our variant of the reachability algorithm of Ullman and Yannakakis [21].

Reachability(u):

- Let t be the time budget and $p = nm/t^2$ be the number of processors
- Take a random sample of *distinguished nodes* of size $\tilde{O}(n/t)$. Let D be the distinguished nodes.
- Run BasicBFS(G, D, t), and create a new graph G' with extra "shortcut" edges between distinguished nodes that have paths of length at most t in G.
- Run BasicBFS($G', \{u\}, |D| + 2t$) to determine the nodes reachable from u.
- Check if there are no edges from reachable vertices to unreachable ones. If there are, we were unlucky, so try again with a new set of distinguished nodes.

$(n^2/m)^{1/3} \leq t \leq \sqrt{n}$, so to prove Lemma 1 it suffices to analyze the case $\sqrt{n} \leq t \leq n$. For simplicity I assume that there is only one source vertex u.

Let *BasicBFS(G, S, d)* refer to the naive parallel breadth-first-search algorithm on graph G with depth limit d, $|S|$ independant source vertices. This is Ullman and Yannakakis's Basis Rule B2.

Their analysis of the work required in Basis Rule B2 (page 109 of [21]) conservatively assumes that the number of processors is m and hence the work required is $\tilde{O}(m|S| + md)$. We use the tighter bound $\tilde{O}(m|S| + pd)$. A nice consequence of this change is that the algorithm is efficient when $t = n$.

If a vertex v is reachable from u, there must be a path P from u to v. Consider taking a random sample of vertices S by samping each vertex with probability $\tilde{O}(1/t)$. One can show that there probably won't be a string of unsampled vertices in the path of length more than t. The key observation of Ullman and Yannakakis [21] is you can then decompose the path into at most $\tilde{O}(n/t)$ hops between sampled vertices, with each hop being a path of length at most t. Hence we have Algorithm 3.

The time spent looking for shortcuts amoung the distinguished nodes is $\tilde{O}((n/t)(m/p) + t) = \tilde{O}(t^2/t + t) = \tilde{O}(t)$. The number of shortcuts added is at most $(n/t)^2 \leq n^2/(\sqrt{n})^2 = \tilde{O}(m)$ (using $t \geq \sqrt{n}$ by assumption), so G' is not much bigger than G. The time spent searching for paths in G' is $\tilde{O}(m/p + n/t + t) = \tilde{O}(t)$ (using $t \leq n$).

B Practicalities and heuristics

This section describes modifications to the algorithm that may improve practical performance and do not hurt the asymptotic bounds.

- If in the course of the binary search for s you find a value of s such that the first s vertices reach $\Omega(m)$ edges and do not reach $\Omega(m)$ edges, you can abort the binary search and divide

into reached and not reached vertices. If the graph is reasonably uniform, doubling s will at most double the number of edges reachable from vertices $1 \cdots s$. Therefore, if the graph is reasonably uniform and you do the binary search for the log of s instead of s, early aborts will improve the number of reachability queries per SCC call to $O(\log \log n)$.

- When the number of edges in a subgraph under consideration falls to less than the overall number of edges per processor, use a serial SCC algorithm such as DFS [20] instead. If the graph is uniform enough that $B \setminus (A \cup C)$ has few edges (no need for the transitive closure argument), this will improve the recursion depth to $O(\log p)$.