15-418 Project Report: Applications of Direction-Optimized Parallel Breadth-First Search

Andrew Gu (andrewg2), Patrick Liu (phliu)

May 2020

1 Summary

We explored direction optimization in parallel breadth-first search applications including low diameter decomposition, strongly-connected components detection, and least-element lists construction. To begin, we implemented a direction-optimizing BFS, which achieved a 4 to 12× speedup. Then, in terms of the applications, we found the approach to be inviable for our low diameter decomposition algorithm and investigated the reasoning behind the incompatibility; we achieved over a 2× speedup for strongly-connected component detection; and we achieved a 4× speedup for least-element lists construction by exploiting an alternative axis of parallelism.

2 Background

2.1 Introduction

Generally, graph algorithms suffer from poor locality and low arithmetic intensity, leading to implementations being memory bound and difficult to speed up. We investigate a specific technique known as direction optimization in breadth-first search applications for both undirected and directed graphs to gain insight into its efficacy for varying graph structures.

2.2 Breadth-First Search

Breadth-first search (BFS) is a form of graph search where vertices are visited from a source in distance-order. It has many applications ranging from testing reachability and finding unweighted shortest paths to partitioning graphs and computing maximum flows. The algorithm employs the notion of a *frontier*, which represents the vertices currently being visited. Starting with a frontier of only the source vertex, it iteratively defines the next frontier to be the unvisited vertices with edges from current frontier vertices until the next frontier is empty.

For a graph $G = (V, E)$ where $n = |V|$ and $m = |E|$, sequential BFS can be implemented to run in $O(n+m)$ using a typical *top-down* approach. At each iteration, the algorithm iterates over the vertices in the frontier, and for each frontier vertex, it iterates over the out-neighbors to construct the next frontier. While there is an inherent sequential dependency between a frontier and the next, the construction of any one frontier can be done in parallel. On the other hand, one could implement BFS with a *bottom-up* approach where at each iteration, each unvisited vertex adds itself to the next frontier by checking (in parallel) if any of its in-neighbors are in the current frontier [3].

Intuitively, we see that the top-down approach benefits when the frontier size is small. More precisely, in the top-down approach, every edge with an endpoint in the frontier is checked for a given iteration; however, many of these checks may fail because an out-neighbor is either already claimed, also in the frontier, or a valid parent (for undirected graphs). Hence, there may be significant wasted work. In contrast, in the bottom-up approach, once a vertex has determined itself to be in the next frontier, it can stop checking any

further in-neighbors; *i.e.* there is opportunity for early stopping. Still, it suffers in the later stages of the BFS since it must equally iterate over all vertices despite there being far fewer unvisited vertices.

Beamer *et. al.* present a hybrid approach that dynamically chooses between processing a BFS step either top-down or bottom-up based on the algorithm state [1]. They call this approach *direction-optimizing*. If we let m_f represent the number of edges to check from the frontier, n_f represent the number of vertices in the frontier, and m_u represent the number of edges to check from unvisited vertices, then the hybrid approach switches from top-down to bottom-up when $m_f > \frac{m_u}{\alpha}$ and switches from bottom-up to top-down when $n_f < \frac{n}{\gamma}$ for empirically determined constants α and γ. This adaptive strategy attempts to take the best of both approaches to reduce the number of redundant vertices and edges checked.

In terms of auxiliary data structures, the top-down BFS requires a frontier set that is explicitly iterated over at each iteration, and the bottom-up BFS requires a bit map that indicates if each vertex has been visited. As such, the hybrid approach requires a conversion from one data structure to the other. The transition from frontier set to bit map can be done implicitly; however, the transition from bit map to frontier set requires an additional iteration over the vertices. However, as we will see in the experiment results in Section 4.2, the cost to transition is amortized well over the algorithm.

2.3 Low Diameter Decomposition

Graph decompositions hope to partition an undirected, unweighted graph $G = (V, E)$ into vertex subsets such that there is high connectivity within each subset and low connectivity across subsets. When the connectedness is measured by diameter, the partition is known as a *low diameter decomposition*. Formally, the partition $S_1, \ldots, S_k$ represents a (β, d) decomposition of G if the diameter of each S_i is at most d and the number of edges connecting different S_i is at most $\beta|E|$. Finding a low diameter decomposition is the basis for several larger graph algorithms such as constructing k-spanners or approximating shortest paths [8].

Typically, the decomposition parameters are chosen to be $(\beta, O(\frac{\log n}{\beta}))$. The traditional sequential algorithm for finding such a decomposition is known as *ball growing*: While there remains vertices in the graph, choose an arbitrary vertex v to begin growing a ball from; iteratively add neighboring vertices to the ball until the number of edges on the ball boundary is at most a β-fraction of the number of edges within the ball; and remove the ball and its edges from the graph. However, this ball growing algorithm evinces an explicit sequential dependency between one ball and the next, which serves as a barrier against parallelization.

Miller *et. al.* present a novel ball growing algorithm with exponential delays much more amenable to parallelization [8]. At a high level, the algorithm simply attempts to start growing balls from each vertex delayed by a random exponential shift. A more precise description is given by Algorithm 1.

Algorithm 1 Parallel Ball Growing

Parallel Decomposition ($G = (V, E)$, β):
for each $v \in V$, sample $\delta_v \sim \text{Exp}(\beta)$
$\delta_{\max} \leftarrow \max_{v \in V}\{\delta_v\}$
for each $v \in V$, start BFS at time $\delta_{\max} - \delta_v$
for each $v \in V$, assign v to the source of shortest BFS path that reached it

Algorithm 1 yields a $O(\beta, O(\frac{\log n}{\beta}))$ decomposition in expectation for $\beta \leq \frac{1}{2}$ in $O(n+m)$ work. Notably, each step—the exponential sampling, computation of $\delta_{\max}$, BFS, and assignments—can be performed in parallel with respect to the vertices, eliminating much of the sequential dependency exhibited in the traditional sequential algorithm. We explore the application of the direction optimization mentioned in Section 2.2 to parallelizing the BFS in particular.

To make the algorithm more concrete, an example run with toy delays is shown in Figure 1. The balls are indicated by the different colors (*i.e.* red, orange, blue, and green), and the darker-colored vertices are the

sources of the balls. The number next to each vertex represents a made-up $\delta_{\max} - \delta_v$ value, so for example, the red vertex with value 0 was the first to begin a BFS.

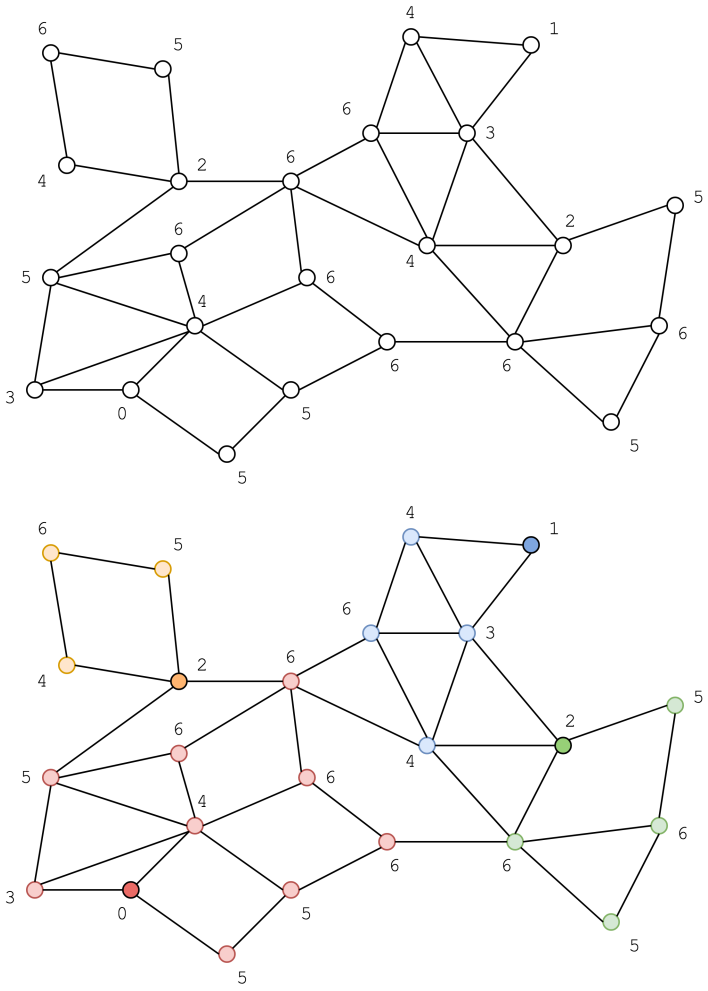

Figure 1: An example run of Algorithm 1 with toy δ's.

2.4 Strongly-Connected Component Detection

Strongly-connected components (SCCs) are defined as sets of vertices that are pairwise connected in a graph, where connectivity between vertices is defined by forward and backwards reachability (*i.e.* connectivity in both directions). SCCs help characterize the connectivity of graphs, which is important in modeling real world phenomena such as networks of various types, and their detection is used for diverse applications ranging from model checking [10] to data flow analysis [9].

A well-known algorithm for SCC detection, Tarjan's algorithm, involves a single-pass depth-first search on the graph, requiring $O(n + m)$ time [2]. However, depth-first search is difficult to parallelize. Algorithm 2 describes a different approach that allows us to exploit parallelism using a "pivot" vertex v_i that partitions the remaining vertices into four subsets: vertices that are forward-reachable from v_i, (R^+), vertices that are backward-reachable from v_i, (R^-), vertices that are both forward and backward-reachable from v_i, $(R^+ \cap R^-)$, and vertices that are neither forward nor backward-reachable from v_i, $(S - (R^+ \cap R^-))$ [2]. The algorithm iteratively processes each vertex v_i, calling a "black box" subroutine for `forward_reachability` and `backward_reachability`. It is in these subroutines in particular that we later aim to uncover potential parallelism.

Algorithm 2 Sequential SCC-Detection

SCC-Detection ($G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$):
$\mathcal{V} = \{V\}$
$S_{scc} = \{\}$
for $i \leftarrow 1$ to n **do**
  Let $S \in V$ be the subgraph that contains v_i
  if $S = \emptyset$ **then** `continue`
  $R^+ =$ `forward_reachability`(S, v_i)
  $R^- =$ `backward_reachability`(S, v_i)
  $V_{scc} = R^+ \cap R^-$
  $\mathcal{V} = (\mathcal{V} - \{S\}) \cup \{R^+ - V_{scc}, R^- - V_{scc}, S - (R^+ \cup R^-)\}$
  $S_{scc} = S_{scc} \cup \{V_{scc}\}$
end for
return S_{scc}

It can be shown that if the vertices are uniform randomly permuted, then Algorithm 2 runs in $O(m \log n)$ [2]. The primary source of parallelism comes from partitioning the graph into the four subsets that were previously described. Namely, we can adapt the BFS subroutine with a few modifications, constraining the search space to the subgraph S (iff $S \neq \emptyset$), to perform the partitioning.

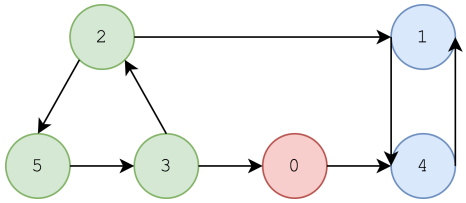

Figure 2: Example run of SCC detection.

2.5 Least-Elements List Construction

For our final application of BFS, we consider constructing least-element lists (LE-lists) for unweighted graphs. The data structure has applications in estimating neighborhood sizes of vertices, such as quantifying a vertex's influence in a social network [4] or producing probabilistic tree embeddings [5]. Formally, for a fixed ordering on the vertices $\{v_1, \ldots, v_n\}$ where $n = |V|$, the LE-lists are given by $L(v_i) = \{v_j \in V \mid d(v_i, v_j) < \min_{k=1}^{j-1} d(v_i, v_k)\}$ sorted by the distance $d(v_i, v_j)$ [2]; *i.e.* a vertex v_j is in v_i's LE-list exactly when there are no earlier vertices (with respect to the fixed ordering) closer to v_i. It is common to also store $d(v_j, v_i)$ with the vertex v_j in v_i's LE-list.

It turns out that if the vertices are uniform randomly permuted, then each LE-list has length $O(\log n)$ with high probability [4]. Hence, for an unweighted graph, the LE-lists can be constructed in $O(\log n(n+m))$ time using Algorithm 3. Notably, finding the set S can be done using BFS for an unweighted graph, which yields a major source of parallelism. However, one important distinction is that, by the definition of S, actually only the vertices in S and their out-edges need to be explored, so the search space is likely to be more limited compared to that of a full BFS. We are interested in seeing how direction optimization performs under this modified regime.

As an example, for the graph shown in Figure 3, one possible set of LE-lists would be $L(0) = [(2, 1)], L(1) = [(0, 2), (2, 1)], L(2) = [(0, 1)], L(3) = [(0, 1)], L(4) = [(0, 1)]$, and $L(5) = [(0, 2), (3, 1)]$ where each tuple element is of the form $(v_j, d(v_i, v_j))$.

Algorithm 3 Incremental LE-Lists Construction

LE-Lists ($G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$):
for each $v \in V$, $\delta(v) \leftarrow \infty$, $L(v) = \varnothing$
for $i \leftarrow 1$ to n **do**
  Let $S = \{u \in V \mid d(v_i, u) < \delta(u)\}$
  for $u \in S$ **do**
    $\delta(u) \leftarrow d(v_i, u)$
    Append $(v_i, d(v_i, u))$ to $L(v)$
  end for
end for
return $L(v)$ for all $v \in V$

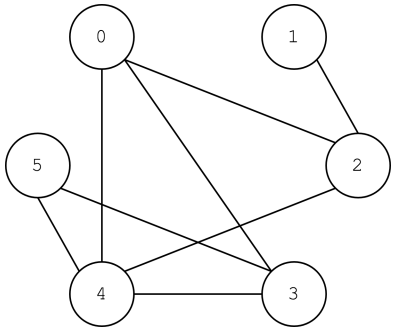

Figure 3: Example graph for LE-lists construction.

3 Approach

3.1 Graph Representation

Because we examine applications for both undirected and directed graphs, we require our graph data structure to accommodate both. As such, we decided to store an out-edge list $\mathcal{E}_{out}$ with offsets $\mathcal{O}_{out}$ and an in-edge list $\mathcal{E}_{in}$ with offsets $\mathcal{O}_{in}$—all represented as C `int` arrays.

Specifically, $\mathcal{E}_{out}[\mathcal{O}_{out}[i]], \mathcal{E}_{out}[\mathcal{O}_{out}[i]+1], \ldots, \mathcal{E}_{out}[\mathcal{O}_{out}[i+1]-1]$ are the out-edges of vertex i for $i \in [n]$ (and analogously for the in-edges). For undirected graphs, $|\mathcal{E}_{out}| = |\mathcal{E}_{in}| = 2m$, while for directed graphs, $|\mathcal{E}_{out}| = |\mathcal{E}_{in}| = m$. For indexing convenience, we define $\mathcal{O}_{out}[n] = \mathcal{O}_{in}[n] = |\mathcal{E}_{out}| = |\mathcal{E}_{in}|$. This representation has the advantage of linear time iteration over a vertex's out-neighbors and in-neighbors with good locality.

3.2 Breadth-First Search

3.2.1 Top-Down BFS

Recall that for top-down BFS, we need to explicitly store the frontier. In our implementation, we represent the frontier using a `struct` called a `vertex_set`, which consists of `num_vertices`—an `int` giving the number of vertices in the set—and `vertices`—a C `int` array containing the vertices in the set. Further, we store both the current frontier, `frontier`, and the next frontier, `next_frontier`, using the former to construct the latter at each iteration.

To begin, the initial frontier consists of only the source vertex. In each step, the algorithm iterates over the vertices in the frontier and the out-neighbors of each of those vertices. If an out-neighbor is unvisited, then it is added to the next frontier. For the parallel implementation, we distribute the iteration over the vertices in the frontier across multiple OpenMP threads. Each thread stores a private `local_frontier` to accumulate the vertices it determines to be in the next frontier, and at the end, it atomically copies its contribution to

the shared `next_frontier`. To prevent a vertex from being added multiple times to the next frontier, we use a compare-and-swap to atomically check if the vertex is unvisited before claiming it to a `local_frontier`.

3.2.2 Bottom-Up BFS

Unlike top-down BFS, the bottom-up counterpart does not need to explicitly store the frontier. Rather, it can exploit the output `distances` array to determine which vertices were visited in the last iteration (*i.e.* by checking if `distances[i] == iter-1`) and to determine which vertices are unvisited (*i.e.* by checking if `distances[i] == UNVISITED` where `UNVISITED` is some negative constant for example).

At the start, `distances[source]` is set to zero. Then, in each step, the algorithm iterates over all vertices, and for each unvisited vertex, checks if any in-neighbors are in the frontier—upon finding a parent vertex, the algorithm can break from the loop. Moreover, in contrast to the top-down approach, bottom-up BFS does not require any additional synchronization. Hence, for the parallel implementation, we simply distribute the iteration over the vertices across OpenMP threads and use an OpenMP `reduction` to determine the frontier size. The algorithm terminates when the frontier size reaches zero.

3.2.3 Direction-Optimizing BFS

To implement the direction-optimizing BFS, we must keep track of two additional quantities: the number of edges to check from the frontier and the number of edges to check from unvisited vertices. The former quantity can be computed while constructing the next frontier, and the latter quantity can be computed by maintaining the total number of edges left to check and subtracting the number of edges checked during an iteration. Hence, both of these quantities require constant additional time to compute. At each step, the algorithm decides whether it should switch from top-down to bottom-up or *vice versa*, reconstructing the appropriate data structure if needed, and proceeds. The top-down and bottom-up logic are precisely as described in Sections 3.2.1 and 3.2.2, and we determined suitable values for the switching constants α and γ empirically.

3.3 Low Diameter Decomposition

We utilize the principles discussed in Section 3.2 and apply them to parallel ball growing with exponential delays. As such, the top-down and bottom-up implementations for Algorithm 1 directly mirror those for top-down and bottom-up BFS respectively. The main complication involves discretizing the continuous notion of time used by the algorithm (to determine when a vertex should start its own BFS). To handle this, we create an implicit super source vertex with an edge to each vertex v in the original graph, where that edge has weight $\delta_{\max} - \delta_v$, and we start a single parallel BFS from that super source vertex. We say that this is done implicitly because we do not actually have to modify the inputted graph data structure; rather, by keeping track of the iteration count `iter`, we simply add all unvisited vertices with delay less than `iter` to the next iteration's frontier to emulate starting a BFS from those vertices. Moreover, we break ties between potential parents by comparing their delays and propagating a parent's delay to the child once it is claimed.

The issue with this need for tie breaking is that an advantage of the bottom-up approach is nullified. A vertex can no longer break immediately upon finding a parent vertex since parents are no longer equal (instead, they are distinguished by their owner's δ values). However, the absence of additional synchronization constructs imply that there is still reason to expect performance gains from the bottom-up approach. On the other hand, the top-down approach also suffers from a complication. Because on each iteration, any unvisited vertex could be starting its own BFS, there must be some kind of iteration over unvisited vertices. This is at odds with the top-down approach's advantage of only needing to iterate over the frontier. In particular, at the beginning of the algorithm, the number of unvisited vertices is a large fraction of the full vertex set, so the workload of the top-down approach becomes roughly equal to that of the bottom-up approach, but the former approach still requires additional synchronization.

Nonetheless, we attempt a hybrid direction-optimizing version of Algorithm 1 using the same dynamic switching criterion described in Section 2.2. Specifically, we follow the original BFS implementation except with

the following modifications: We store a set `std::unordered_set<int> unvisited` so that the top-down steps can determine which unvisited vertices should be starting their own BFS, and we no longer break immediately once a valid parent has been found in the bottom-up steps to accommodate the non-integral start times. We also store additional data structures to keep track of ball owners, ball IDs, and δ's.

The question of how to represent the unvisited vertices for the top-down steps was nontrivial. We wanted the data structure to allow for constant-time lookup, constant-time deletion, and efficient iteration over the unvisited vertices. We considered using C++ data structures versus native C data structures. On one hand, C++ data structures like the `std::unordered_set` provide the desired access and modification speeds, but OpenMP does not support parallel iteration over them. On the other hand, in C, we could either use a bit map representation implemented by a length-$|V|$ `unsigned char` array or a queue representation implemented by an underlying length-$|V|$ `int` array with an `int` to give the number of elements in the queue—however, neither of these satisfy the full criteria. We experimented with both the C++ and C methods and found the former to be faster across all inputs tested; hence, that was our choice.

3.4 Strongly-Connected Components Detection

The major change to the original sequential algorithm described in Algorithm 2 was to parallelize the computation of the forward and backward reachable vertex sets. In particular, because the iteration over the vertices themselves (which could be randomized) exhibit a sequential dependency (since the algorithm iteratively splits sub-graphs based on previously pivoted vertices), we focused on the parallelism exposed by the queries of reachable vertices and the computation required to partition the graph into the desired sub-graphs.

We took an iterative approach when tackling this problem. In the beginning, we focused on achieving a correct implementation that was reasonably fast. This was done by leveraging C++ constructs as much as possible: specifically, we used a `std::unordered_set` to naturally model the partitioned sub-graphs at each pivot vertex. Although this approach gave us consistent and significant speedups relative to the sequential implementation, when we profiled and investigated the performance, we discovered that the computation of the sets was extremely expensive. This was due to an iteration over all the vertices at the end of each search to generate the partitions for the next. This was exceedingly costly since some reachability queries were performed on small sub-graphs, where the overhead of any parallel computation dominated the performance gained. As a result, we rearranged the computation and synchronization used to distribute this computation across the algorithm.

In addition to reducing the expensive cost of computing sub-graphs at the end of each query, we also noticed from our profiling that the actual computation of each strongly-connected component was expensive (which was done at each pivot vertex). We were initially unable to parallelize this part due to the `unordered_set` representation (with OpenMP making it unwieldy to parallelize over C++ data structures). After switching to a bit map representation, we were able to achieve significant parallelism by clearing the sequential SCC set computing bottleneck. After profiling our code and experimenting with various OpenMP constructs (many of which were not applicable to this algorithm), we were able to achieve a 11× speedup on each of the parallel implementations over our initial attempts at parallelizing the algorithm.

3.5 Least-Element Lists Construction

The application of parallel BFS to Algorithm 3 is relatively direct. The BFS is used to find the set S, so we parallelize that computation using top-down, bottom-up, and direction-optimizing techniques—the only question is the representation of the set S. Originally, we hypothesized that, in the spirit of each approach, a queue representation would work well for the top-down approach, and a bit map representation would work well for the bottom-up approach. This is because in the former case, the computation of S is perfectly bundled into the computation of the next frontier (*e.g.* without need for extra synchronization or iteration), and in the latter case, the independence of the vertices is maintained.

We experimented extensively with the hybrid direction-optimizing implementation. We tried using the two different representations of S mentioned above as well as tried unifying the representations to reorganize the computational burden (since the algorithm must iterate over both the part of S found by top-down steps and the part of S found by bottom-up steps). We played with reordering the two loops and including a `#pragma omp nowait` on the earlier loop, and we also tuned the chunk size. However, all of these attempts did not yield any successful speedup. We discuss this in detail in Section 4.5.

Unsatisfied, we looked into alternative approaches to parallelizing this algorithm. Under an idea mentioned by Blelloch *et. al.* [2], we opted to try a different axis of parallelism. Instead of limiting ourselves to parallelizing the BFS used to compute the set S, we focused on parallelizing over the $|V|$ LE-lists. The key insight was that if the LE-lists were computed in parallel with arbitrary interleaving, then the only deviation from correctness would be that some vertices' δ values would not be as small as they should be, resulting in extraneous elements being added to the LE-lists. However, with a clever post-processing step involving sorting each LE-list by the vertex ordering and filtering elements that violated the decreasing distance invariant, the correct solution could be recovered. Notably, each LE-list could be post-processed independently, so again, we could parallelize over this step.

As we will make concrete in Section 4.5, we found that the size of $|S|$ to be very small on average. Therefore, even though we had to use locks to ensure the atomicity of the updates to the δ's and the appends to the LE-lists, the overall cost of synchronization was dwarfed by the new parallelism to be exploited. Furthermore, we saw speedup (of a few percent) in employing a `dynamic` scheduling policy rather than a `static` one to accommodate the load imbalance: As the algorithm progresses, the δ values will converge to their minimums so most more and more vertices will be immediately rejected in the BFS, yielding shorter and shorter searches.

3.6 Addendum

We wrote all algorithm code from scratch in C++, leveraging OpenMP for parallelism and targeting the GHC machine CPUs for execution. Additionally, we wrote Python scripts to help generate the input graphs for testing; specifically, we used several functions from the NetworkX library.

4 Results

4.1 Benchmark Graphs

We test against several different graphs with varying structures:

- Powerlaw cluster graphs: These are undirected graphs with vertex degrees sampled from a powerlaw distribution, *i.e.* $f(d) = \frac{1}{d^\alpha}$ for some $\alpha > 0$, with additional clustering due to a triangle formation step in their generation [6]. In other words, they represent scale-free graphs with high clustering. They can be characterized by specifying the number of vertices, the average vertex degree, and a probability p of forming a triangle with each random edge added. We try two parameter settings: (1) $|V| = 20000$, $\mu_{deg} = 5$, and $p = 0.25$; and (2) $|V| = 20000$, $\mu_{deg} = 10$ and $p = 0.25$. We will refer to the former as Powerlaw Cluster Type 1 and the latter as Powerlaw Cluster Type 2.

- Internet Autonomous System (AS) graphs: These are undirected graphs resembling subgraphs of the overall Internet routing graph. In other words, the vertices represent routers, and the edges indicate communication between two routers. It turns out that the Internet AS graphs similarly exhibit approximate powerlaw degree distribution [7]. We randomly generate Internet AS graphs on 10,000 vertices; these graphs have approximately 26,000 edges.

- Watts-Strogatz graphs: These graphs model small-world properties well. For example, a social network graph exhibits small-world properties if two people who have seemingly no relation to each other are actually indirectly connected *via* a short sequence of mutual friends. The graphs can be parameterized by $|V|$, k, and p, where each vertex is joined with its k nearest neighbors and an edge is rewired with

probability p. We randomly generate Watts-Strogratz graphs on 20,000 vertices and approximately 200,000 edges (using $k = 20$ and $p = 0.25$).

The powerlaw cluster graphs and the Internet AS graphs were chosen due to their powerlaw degree distributions. Those characteristics induce high load imbalance because some vertices have much higher degree than others. In a similar vein, the small-world properties of the Watts-Strogatz graphs makes partitioning the graph nontrivial due to the complex connectivity between seemingly distant vertices (which makes it particularly interesting to test strongly-connected component detection against). Hence, we thought it would be meaningful to benchmark against these adversarial graphs.

4.2 Breadth-First Search

We wrote sequential implementations for top-down and bottom-up BFS to verify the correctness of and benchmark against our parallel implementations. We made sure to run a correctness check for every graph tested to ensure the fidelity of the parallel implementations.

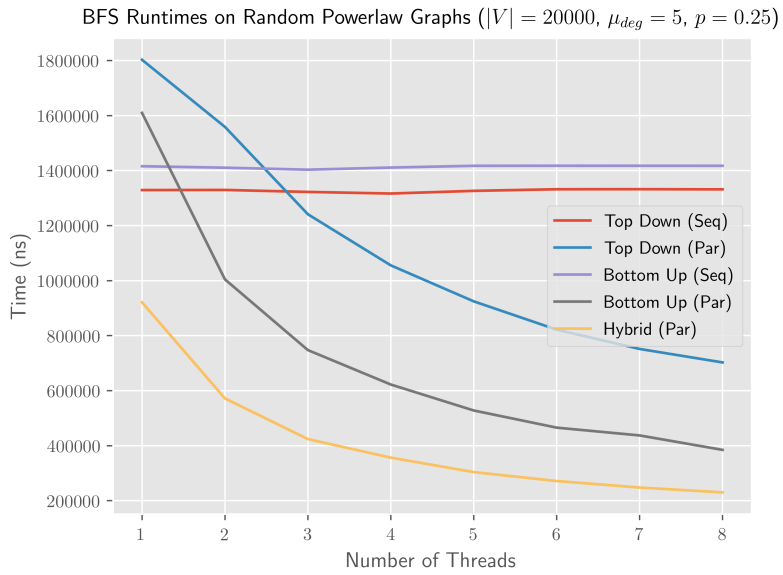

Figure 4: BFS runtimes averaged over 50 random powerlaw cluster graphs ($|V| = 20000$, $\mu_{deg} = 5$, $p = 0.25$).

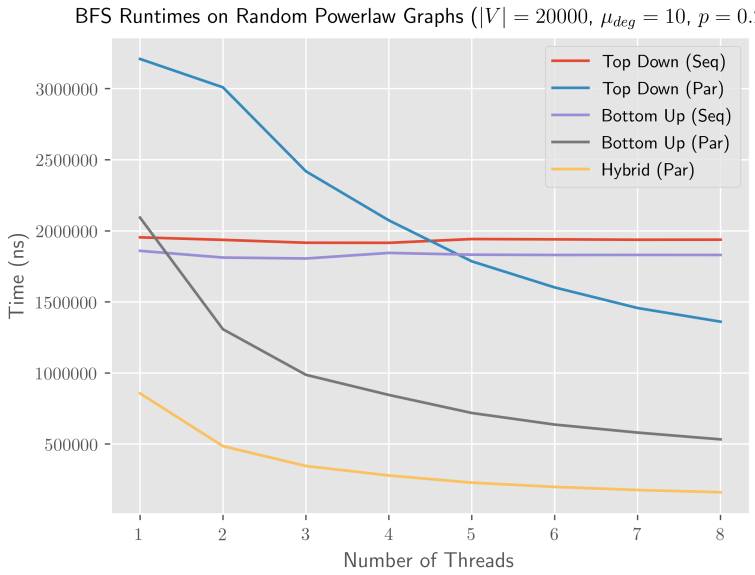

Figure 5: BFS runtimes averaged over 50 random powerlaw cluster graphs ($|V| = 20000$, $\mu_{deg} = 10$, $p = 0.25$).

We present the runtimes for the five BFS implementations on our two powerlaw cluster graph settings in Figures 4 and 5. Immediately, we see that these results affirm that there are performance gains available from employing the hybrid direction-optimizing approach. Interestingly, even though the parallel top-down approach performs quite worse than the parallel bottom-up approach, when the two strategies are used under their preferred conditions, the result is significant speedup. (It is important to note that the direction-optimizing approach actually visibly outperforms the parallel bottom-up implementation; otherwise, it could simply never perform top-down steps to match the bottom-up performance.)

We analyze the results more deeply. Recall that as described in Section 2.2, top-down BFS checks almost all out-going edges (modulo the final iteration) even in the best case. Hence, we see that when the edge-to-vertex ratio exceeds some point, even the sequential bottom-up implementation beats the sequential top-down implementation (due to its early stopping) despite having a generally worse asymptotic complexity. This effect is reflected in the parallel versions as well, as the gap between the parallel top-down and bottom-up implementations is exaggerated from Figure 4 to Figure 5 since the total number of edges in the graph doubles.

To make the gains from incorporating bottom-up steps in addition to top-down steps concrete, in Table 1, we present the number of edges that did not have to be checked because of the early stopping behavior of bottom-up steps in the direction-optimizing implementation.

Graph	Avg. # of Edges Skipped	Avg. Fraction of Edges Skipped
Powerlaw Cluster Type 1	45983	0.46
Powerlaw Cluster Type 2	105784	0.53
Internet AS	16196	0.62
Watts-Strogatz	44609	0.22

Table 1: Edges skipped by bottom-up steps in direction-optimizing BFS.

We see that across our benchmark graphs, anywhere from about one-fourth to one-half of the edges in the graph are not considered, which represents a significant reduction in workload. Further, when we examine the distribution of top-down and bottom-up steps, we find that for the powerlaw cluster graphs as well as the Internet AS graphs, the top-down steps occur in the beginning and end of the BFS, while the middle steps are bottom-up ones. This precisely matches the intuition developed in Section 2.2 since these clustered graphs typically result in a rapid growth in frontier size to begin, a large frontier size in the middle, and a

rapid decay in frontier size to end [1].

There appears to be a positive association between the average degree of the graph and the final speedup using eight threads for these clustered graphs. This is especially evidenced by Table 2, where we present the speedups of the hybrid direction-optimizing implementation compared to the faster of the two sequential BFS implementations. Again, this echoes back to the fact that graph algorithms typically exhibit low arithmetic intensity and workload, so it is difficult to overcome the overhead that comes with parallel execution. Having a higher average degree implies a greater workload for the sequential implementation, which provides more room for parallel speedup.

Graph	Speedup (8 Threads)	Avg. Degree
Powerlaw Cluster Type 1	5.78×	5
Powerlaw Cluster Type 2	11.99×	10
Internet AS	4.33×	2.6
Watts-Strogatz	4.79×	10

Table 2: Direction-optimizing BFS speedups.

4.3 Low Diameter Decomposition

We implemented the sequential ball growing algorithm in addition to parallel top-down, bottom-up, and direction-optimizing ball growing with exponential delays. While the exact output of the sequential algorithm is not necessarily the same as the parallel algorithm (because fundamentally the algorithms differ), because both algorithms output $(\beta, O(\frac{\log n}{\beta}))$ decompositions, the sequential algorithm still provides meaningful perspective on the performance of the parallel implementations.

We found that the possibility for any vertex to start its own BFS at any iteration effectively voided the top-down approach. Regardless of how we represented the set of unvisited vertices, the top-down approach could not be competitive in terms of performance compared to the bottom-up approach. As discussed in Section 3.3, the unavoidable iteration over the unvisited vertices required each top-down step to perform approximately the same amount of work as a bottom-up step, only with additional synchronization constructs. Hence, we saw that the bottom-up implementation absolutely dominated the top-down implementation, and there was no hope for the hybrid direction-optimizing implementation to then outperform the bottom-up implementation.

Nonetheless, the bottom-up approach yielded reasonable speedup with respect to the number of threads, as visualized in Figure 6. A similar curve is observed when testing on the other benchmark graphs. This follows because the troublesome need to check if a vertex should start its own BFS is naturally folded into the bottom-up approach, as it iterates over all vertices at each step anyway, so the speedup behavior should mirror that of vanilla parallel bottom-up BFS.

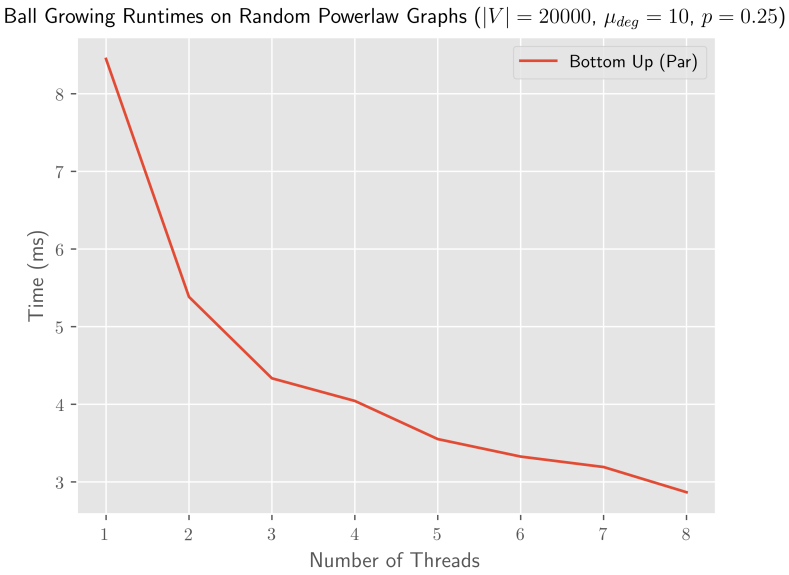

Figure 6: Parallel ball runtimes growing averaged over 50 random powerlaw cluster graphs ($|V| = 20000$, $\mu_{deg} = 5$, $p = 0.25$).

The top-down implementation still outperforms the traditional sequential algorithm that processes each ball one at a time as shown in Table 3, demonstrating the importance of the underlying algorithm design. We assert the correctness of our implementations by measuring the fraction of intercluster edges. Since we ran the algorithms with $\beta = 0.5$, we expect β fraction of the edges to be intercluster [8], and indeed, we see that our implementations achieve such a result. (Note that we fix the random generator's seed across the parallel implementations to check for consistency; hence, their results are the same.)

Implementation	Runtime with 8 Threads (ms)	Fraction of Intercluster Edges
Naïve Ball Growing (Sequential)	213	0.47
Top-Down BG w/ Exp. Delays (Parallel)	53	0.34
Bottom-Up BG w/ Exp. Delays (Parallel)	3	0.34
Hybrid BG w/ Exp. Delays(Parallel)	6	0.34

Table 3: Ball-growing runtimes averaged over 50 random powerlaw cluster graphs ($|V| = 20000$, $\mu_{deg} = 10$, $p = 0.25$); $\beta = 0.5$.

4.4 Strongly-Connected Components Detection

We achieved a significant speedup in parallelizing Algorithm 2 by exploiting the parallelism found in the reachability queries and the generation of the subgraphs and strongly-connected components. The runtime for each of our implementations of Algorithm 2 are shown in Figure 7.

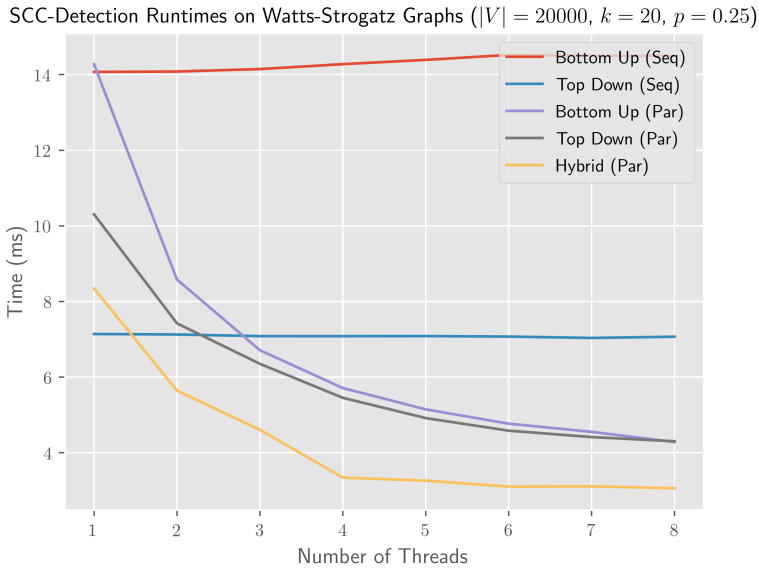

Figure 7: SCC detection runtimes averaged over Watts-Strogatz graphs ($|V| = 20000$, $|E| = 200000$).

We were successful in exploring the advantages and disadvantages of BFS in this application. As mentioned in Section 3.4, after profiling our code and iteratively changing our approach, we were able to achieve a 6 to 11× speedup across all implementations, relative to the first attempt at each approach. The higher average degree of the Watts-Strogatz graphs implied a larger workload, which was important in seeing tangible speedups (*i.e.* the speedups were less pronounced on the lower average degree benchmark graphs).

Still, we notice that the speedups plateau after approximately four threads. We believe that this plateau follows directly from the nature of the algorithm and the structure of the input graphs. While a vanilla BFS reaps the benefits of additional threads (past four) due to a generally nontrivial frontier size, the usage of BFS in our SCC detection algorithm is slightly different. Because the algorithm iteratively partitions the graph into subgraphs, employing the BFS for forward and backward reachability queries, with each partition, the algorithm generates smaller and smaller subgroups for future iterations. Hence, after a certain number of pivot vertices, the size of the subgraphs no longer offer enough work for the additional threads to justify their overhead (*i.e.* the arithmetic intensity is too low). This is made apparent when we compare against runs on smaller versions of the Watts-Strogaz graphs where the speedups are less drastic, as shown in Figure 8 where the parallel runtime curves are much closer to the sequential top-down runtime curve.

Interestingly, we see in Figure 7 that while parallel top-down and bottom-up implementations begin to outperform the faster sequential implementation with three threads, the runtime eventually converges to approximately 4.3 ms. On the other hand, the hybrid approach beats the sequential implementation with fewer threads and converges to a faster runtime of approximately 3 ms. Therefore, we see that the direction optimization yields about a 1.4× speedup when compared to fastest pure parallel implementations. Although direction optimization still outperforms both pure parallel implementations, the relative speedup to the fastest parallel implementation measured on the smaller Watts-Strogatz graph is only about 1.16×, and about 1.6× faster than the fastest sequential implementation. This is in stark contrast to the 2.31× speedup relative to the fastest sequential implementation on the larger Watts-Strogatz graph. Nonetheless, we still see a similar plateauing on the smaller Watts-Strogatz graph at around four threads, once again attributed to the smaller workloads as the algorithm progresses.

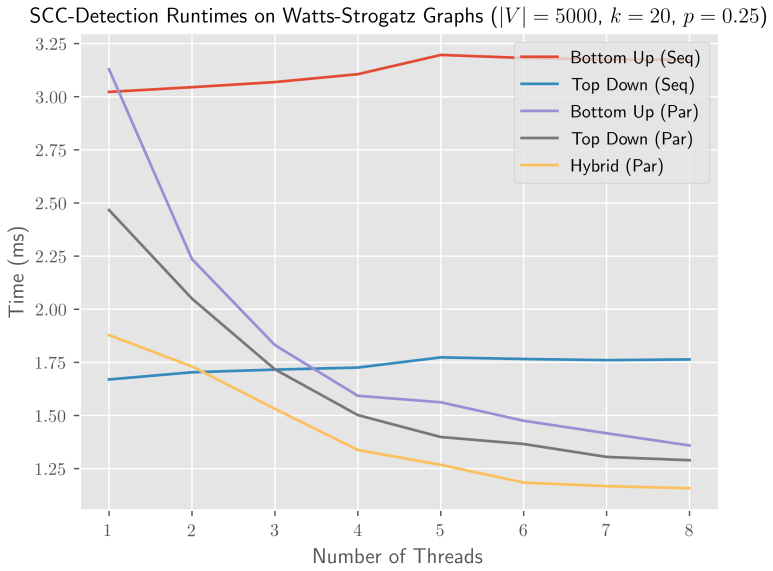

Figure 8: SCC detection runtimes averaged over Watts-Strogatz graphs ($|V| = 5000$, $|E| = 50000$).

4.5 Least-Element Lists Construction

Experimentally, we found the parallel bottom-up approach to be too slow to be viable. We can actually understand why this is upon closer examination of Algorithm 3. Recall that the goal is to output the least-element list for every vertex $v \in V$, so there is inherently an n-iteration loop over the vertices. Further, each bottom-up BFS call to find the set S starts with an iteration over all vertices regardless of how much of the graph actually needs to be searched. As such, the algorithm devolves to require a minimum of n^2 memory accesses even in the *best case*. Originally, we thought that a bit map representation of S would be better for the bottom-up approach since it preserved the absence of contention over a shared variable, but upon experimentation, we found that a queue representation actually performed slightly better (*e.g.* $\approx 10\%$ faster on Powerlaw Cluster Type 2). The reason for this, we found, was that the size of the set S was typically very small, as shown in Table 4.

Graph	Avg. $\lvert S\rvert$
Powerlaw Cluster Type 1	2.47
Powerlaw Cluster Type 2	2.32
Internet AS	2.21

Table 4: Average size of the set S over all LE-lists.

We even tested on an additional random graph on 20,000 vertices and 1,000,000 edges, and the average size of S was only 4.34. Therefore, just to find the small set S, the bottom-up implementation was exploring every vertex at least once for every LE-list. This is in sharp contrast to the top-down approach, which should benefit directly from the limited search space. As explained in Section 2.5, only the set S and its out-edges actually need to be explored by the BFS; *i.e.* if the distance to a vertex exceeds its δ value, then it is not added to the frontier. This can prune a significant portion of the normal BFS tree, so naturally, we expect the top-down approach to profit.

Because the bottom-up steps did not provide any advantage in this application, it followed that the hybrid direction-optimizing implementation (no matter how α and γ were tuned) could not outperform the purely top-down implementation. Even so, it turned out the top-down implementation did not see success in terms of speedup either. This again directly follows from the small average size of S. Each BFS call in the inner loop of Algorithm 3 is actually extremely brief with an average frontier size of approximately 1.6 vertices

on the Powerlaw Cluster Type 2 graphs. As mentioned in Section 3.5, this is because once the δ values have mostly converged to their minimizing values, most visited vertices will no longer be added to the next frontier in the BFS's since their distances will exceed their δ values. Thus, each iteration has extremely poor arithmetic intensity, and the runtime is heavily memory-bound, limited by the updates to the distance and δ arrays and the appends to the LE-lists themselves. Ultimately, the top-down implementation had approximately the same runtime as the sequential implementation even with eight threads.

In contrast, the alternative approach parallelizing over the LE-lists themselves instead of over the BFS's resulted in significant speedup. These results are summarized in Figure 9 and Table 5.

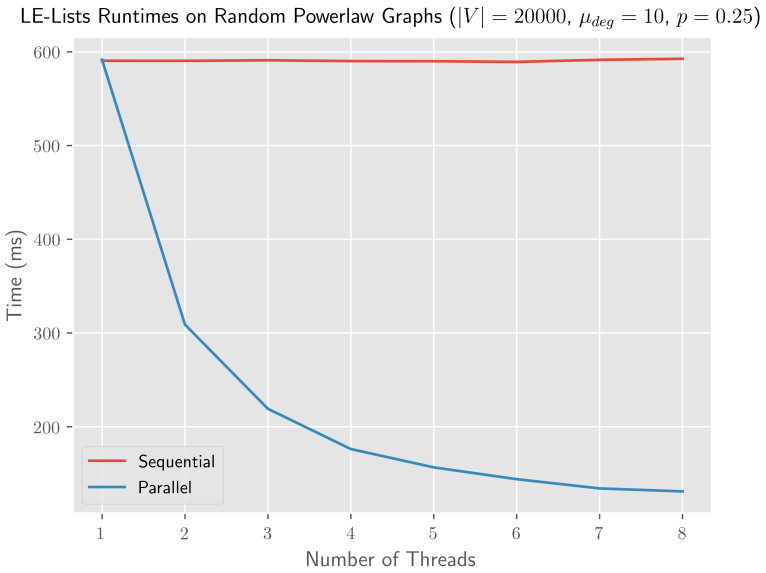

Figure 9: LE-lists construction runtimes averaged over 50 random powerlaw cluster graphs ($|V| = 20000$, $\mu_{deg} = 10$, $p = 0.25$).

Graph	Speedup (8 Threads)	Avg. Degree
Powerlaw Cluster Type 1	4.71×	5
Powerlaw Cluster Type 2	4.52×	10
Internet AS	3.58×	2.6
Watts-Strogatz	4.60×	10

Table 5: Parallel LE-list construction speedups.

4.6 Concluding Remarks

In summary, we began by examining direction optimization in parallelizing BFS. We found that by employing the hybrid approach, the search was able to dynamically adapt to the workload and surpass both the pure top-down and bottom-up approaches. Then, we tried to apply the same technique to three other graph tasks: low diameter decomposition, SCC detection, and LE-lists construction. First, for low diameter decomposition, because the algorithm permitted any vertex to start a BFS at a given time, the top-down approach was ill-suited, rendering a hybrid approach to be infeasible. Next, for SCC detection, we saw meaningful speedup by leveraging the direction-optimizing BFS subroutine, though speedups plateaued due to the decreasing problem size over the algorithm progression. Finally, for LE-lists construction, the limited scope of the BFS's stymied attempts at parallelization, so instead, an entirely orthogonal axis was considered, ultimately resulting in strong speedup. Therefore, we see that the direction optimization technique does not generalize trivially to all algorithms that include BFS's. In fact, considerations for ensuring sufficient arithmetic intensity are more important than ever.

5 Division of Work

Equal work was performed by both project members.

6 Project Code Link

`https://github.com/ptartan21/15418_Project`

References

[1] S. Beamer, K. Asanovic, and D. Patterson. Direction-optimizing breadth-first search. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2012.

[2] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. *CoRR*, abs/1810.05303, 2018.

[3] Aydin Buluç, Scott Beamer, Kamesh Madduri, Krste Asanovic, and David A. Patterson. Distributed-memory breadth-first search on massive graphs. *CoRR*, abs/1705.04590, 2017.

[4] Edith Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. Syst. Sci.*, 55(3):441–453, December 1997.

[5] Edith Cohen and Haim Kaplan. Efficient estimation algorithms for neighborhood variance and other moments. pages 157–166, 01 2004.

[6] Petter Holme and Beom Jun Kim. Growing scale-free networks with tunable clustering. *Physical Review E*, 65(2), Jan 2002.

[7] S. Jaiswal, A. L. Rosenberg, and D. Towsley. Comparing the structure of power-law graphs and the internet as graph. In *Proceedings of the 12th IEEE International Conference on Network Protocols, 2004. ICNP 2004.*, pages 294–303, 2004.

[8] Gary L. Miller, Richard Peng, and Shen Chen Xu. Parallel graph decompositions using random shifts. *CoRR*, abs/1307.3692, 2013.

[9] David J. Pearce. A space-efficient algorithm for finding strongly connected components. *Inf. Process. Lett.*, 116(1):47–52, January 2016.

[10] E. Ábrahám, N. Jansen, R. Wimmer, J. Katoen, and B. Becker. Dtmc model checking by scc reduction. In *2010 Seventh International Conference on the Quantitative Evaluation of Systems*, pages 37–46, 2010.