PARALLEL LOW-STRETCH SPANNING TREES

August Rønberg, Daniel Nezamabadi, Dennis Buitendijk, Yves Baumann , Zijan Zhang

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Determining Low-Stretch Spanning Trees (LSST) is increasingly used as a subroutine in graph algorithms and linear solvers. Simultaneously, the increasing size of graphs that need to be processed motivates the search for parallel and efficient LSST algorithms. To this end, we implement and evaluate a variety of LSST algorithms. We find that out of our parallel algorithms, "Low-Stretch Parallel Exponential Shift" consistently delivers low-stretch results with a reasonable runtime.

1. INTRODUCTION

We begin by motivating our project and laying out a brief description of relevant related work.

Motivation. Graphs are essential as they allow us to model many natural and social phenomena [1]. While the size of graphs is increasing [1], so is the parallelism in our computing capability. This motivates the development of fast, efficient, and practical parallel graph algorithms.

Our work puts its focus on Low-Stretch Spanning Trees (LSSTs). Determining such trees is of particular interest, as they allow us to determine approximate solutions for graph problems that are NP-hard on general graphs. Additionally, they can be used to improve linear algebra solvers, particularly for symmetric diagonally dominant linear systems [2, 3].

However, determining a LSST for an arbitrary graph in a parallel fashion is not trivial. Common spanning trees, like trees determined by Breadth-First Search (BFS), can perform poorly on certain graphs, a phenomenon that we will explore in more detail. Additionally, determining the optimal solution in parallel is difficult since whether an edge is part of the final LSST influences the stretch of all other edges.

In this work, we implement and evaluate several parallel, shared memory LSST algorithms by learning from the current state of the art [3, 4, 5, 6, 7], which is restricted to the theoretical analysis about runtime and stretch. To our knowledge, we are the first to do this. We evaluate our implementations in two aspects. We measure the average stretch of the resulting spanning trees on various graphs of varying sizes. In addition, we thoroughly test their runtime performance in terms of weak and strong scaling.

Related work. In [8], Papp et al. implement and compare sequential LSST algorithms. We extend their work by implementing and evaluating parallel, shared memory LSST algorithms. Note that Papp et al. consider weighted graphs as well, while our parallel algorithms only consider unweighted graphs.

In [9], Alon et al. introduce the first LSST algorithm. We implement two versions: First, a sequential, unweighted version based on the work of Harvey in [10], which acts as our sequential baseline to evaluate the performance of our parallel algorithms. Secondly, a parallel, unweighted version, which we will call AKPW.

In [3], Blelloch et al. develop a parallel LSST algorithm. We have implemented this algorithm as part of our work.

In [5], Miller et al. introduce a parallel decomposition algorithm for unweighted, undirected graphs based on [3]. This algorithm, among others, has been implemented by the Graph Based Benchmark Suite (GBBS) [11, 12, 13]. We utilize the implementation provided by GBBS to implement an efficient and parallel LSST algorithm, namely Low-Stretch Parallel Exponential Shift (LSPES).

In [4], Becker et al. introduce a distributed algorithm for LSST by combining Star Decomposition with the work of Miller et al. [5] to decompose an undirected graph with integer weights. We adapt and implement this algorithm for the case of undirected, unweighted graphs in shared memory.

2. BACKGROUND

In this section, we introduce definitions and algorithms which are relevant to our problem as well as our methodology.

General definitions. Let $G = (V, E)$ be a connected undirected graph. For a spanning tree T of V we define $\mathrm{dist_T(u, v)}$ to be the length of the unique path between u and v in T, and $d(u, v)$ to be the length of the shortest path between u and v in G. Using this, the stretch of an edge

$(u, v) \in E$ can be defined as

$$\text{stretch}_T(u, v) = \frac{\text{dist}_T(u, v)}{d(u, v)},$$

and the average stretch over E as

$$\text{avg-stretch}_T(E) = \frac{1}{|E|} \sum_{(u,v)\in E} \text{stretch}_T(u, v)$$

Additionally, we define the diameter of a graph G to be the maximum length of the shortest path between any pair of vertices in G.

Parallel Graph Decomp. Using Random Shifts. In Parallel Graph Decompositions Using Random Shifts [5], Miller et al. present a parallel algorithm to decompose a graph into clusters with low diameters. The key idea is that "[each] vertex u picks a start time according to some distribution, and if u is not already part of some other cluster at that time, u starts a cluster of its own and performs a breadth first search." (p. 3) [5].

The algorithm is parameterized by a parameter $\beta \leq \frac{1}{2}$. Intuitively, β gives us control over the diameter of the clusters: larger values of β result in smaller clusters (i.e., smaller diameter) , while smaller values of β result in larger clusters (i.e. larger diameter) . A nice visualization can be found on page 4 of [5].

3. METHODOLOGY

This section gives an overview of the various algorithms we implemented and how we implemented them.

Low-Stretch Parallel Exponential Shift (LSPES). On a high level, our implementation of LSPES can be described by Algorithm 1, which follows the general idea presented in Algorithm 5.1 from [3].

Algorithm 1 Low-Stretch Parallel Exponential Shift

Input: Unweighted graph G
Output: Tree edges E

```
G' ← G
E ← ∅
while number of vertices in G' > 0 do
    Determine clustering for G'
    For every cluster, add the tree edges to E
    G' ← Contract clusters in G'
end while
return E
```

We will now describe in more detail how we implemented each of these steps.

Determining a clustering for the graph G' and the tree edges for every resulting cluster is done by calling an undocumented method implemented in GBBS. The implementation of the undocumented method is very similar to the (documented) implementation of Low Diameter Decomposition (LDD) from GBBS [14], which in turn is an implementation of the algorithm presented by Miller et al. in [5].

We note that the algorithmic bounds guaranteed by implementing LDD from GBBS are slightly different from those presented in [5]. We assume this is due to a difference in assumptions, for example, whether $m \gg n$ holds or not. The curious reader can find the bounds provided by the implementation in [14].

To contract the clusters, we again use the methods provided by GBBS. One thing to note is that to contract the clusters, GBBS requires us first to relabel the vertices. This incurs some bookkeeping overhead, as we need to keep track of the edges in the original graph, not the relabeled one. To implement these bookkeeping operations, we again make use of Parlay. For more details, we invite the reader to look at our implementation [15].

At the time of writing, an overview of the graph processing interface used by GBBS and the costs for each primitive were not yet available [16], making a theoretical algorithm analysis difficult.

Besides the fact that the documentation of GBBS in particular was incomplete, another challenge was the presence of bugs in the library, leading to difficult-to-debug behavior like SEGFAULTs, bad_alloc or stack smashing. We invite the reader to look at the GitHub issues we created in this context [17, 18].

Sequential Alon Karp. The original algorithm proposed by Alon et al. in [9] is designed for weighted graphs. We implement an unweighted variant described by Harvey in [10]. The general idea of this algorithm is to take a bottom-up approach by iteratively partitioning the graph into clusters up to a specific diameter and contracting each cluster into a super-vertex. Note that this partitioning process is done sequentially.

We implement this algorithm from scratch using sequential APIs from NetworKit [19].

Parallel Alon Karp (AKPW). Our implementation of the parallel Alon Karp algorithm is a simplified version of the initial algorithm described by Alon et al. [9] adjusted for unweighted graphs. While the algorithm for weighted graphs first divides all edges into subsets with similar edge weights and then finds a tree such that not too many edges overlap between the subsets, the unweighted version works with a single set since all edges have the same weight. Since there is only a single set of edges, the first found tree will have no bad overlapping edges.

The simplified algorithm first chooses c vertices. For each of those vertices it chooses a radius r_i at random. Next, the algorithm computes BFS from each of the c vertices that runs for r_i steps respectively. Any vertices discovered by more than one center are assigned to the center with a

Table 1. Overview: Evaluated Low-Stretch Spanning Tree algorithms
$|V| = n, |E| = m$
* Due to insufficient documentation, we cannot give all bounds for all algorithms.
** Due to an implementation mistake (sequential loop), the depth of our implementation is actually $O(n)$.

Algorithm	Average Stretch(n)	Work	Depth	Parallel
Sequential Alon Karp [9]	*	$\mathcal{O}(m \log \log n)$	-	No
RandBFS	$\mathcal{O}(dia(G))$	$\mathcal{O}(n+m)$	$\mathcal{O}(dia(G) * \log(n))^{**}$	Yes
LSPES	*	*	*	Yes
AKPW [3]	$2^{\mathcal{O}(\sqrt{\log n \log \log n})}$	$\tilde{\mathcal{O}}(m)$	$\mathcal{O}(\log^{\mathcal{O}(1)} n \cdot 2^{\mathcal{O}(\sqrt{\log n \log \log n})})$	Yes
Star Decomposition [4]	$\mathcal{O}(log^3(n))$	$\mathcal{O}(dia(G) * (n+m))$	$\mathcal{O}(dia^2(G) * log(n))$	Yes

lower radius. By growing the different BFS in parallel, we can guarantee that a single BFS does not create multiple connected components. We can contract all the individual connected components created by the different BFS'. Finally, the algorithm runs one last BFS on the now contracted graph, to find the final tree.

In our implementation, we use the parallel functions for BFS and cluster contraction provided by GBBS and Parlay.

Star Decomposition (StarDecomp). While there exist multiple variants of Star Decomposition, we chose to implement a version meant for distributed computing. The exact algorithm is described in [4], but the basic idea is to grow a ball based on the radius of the current graph. Afterwards, we decompose the rest of the graph by adding a source vertex, and connect it to each vertex in the ball shell, which are all the vertices directly connected to the ball, with a length based on an exponential distribution. We then run BFS starting on the source vertex, and for each vertex we keep track from which vertex in the ball shell it was discovered. We call the set of nodes that were discovered from the same vertex a cluster. For each of these clusters, we add the edge connecting it to the ball to the spanning tree, and we perform the algorithm recursively on each cluster and the ball until each cluster has at most one edge.

As with LSPES and AKPW, we implemented StarDecomp using GBBS and Parlay.

Random BFS (RandBFS). To implement RandBFS, we first randomly choose a start vertex. Then, we utilize the non-deterministic parallel implementation of BFS provided by GBBS [20] to compute a tree of the given graph, which we use as a LSST.

4. EXPERIMENTAL RESULTS

In this section we will first introduce our experimental setup, followed by our results.

Experimental setup. We ran all our experiments on an AWS m6a.xlarge node running Ubuntu 20.04, containing an AMD EPYC 7R13 processor with 32 cores, running at 3.6GHz, 512 KiB L1i and L1d cache, 8 MiB L2 cache and a 64 MiB L3 cache. For compilation, we used Bazel 1.15, which used gcc 8.2.0 to compile the code. The gcc flags used were *-std=c++17, -march=native, -O2*. Since we ran our experiments on a NUMA machine, we ran all our benchmarks using *numactl -i all* as recommended by GBBS [13].

We structured our experiments into 3 categories: strong scaling, weak scaling, and special graphs. For each combination of algorithm, graph, and number of processors, we took measurements until the 95% confidence interval (CI) of the median was within 5% of the median for both the runtime and the average stretch. We did this for processor numbers 1-8, 16, 32. This means that in our plots the number of processors on the x-axis increase linearly for the first eight coordinates, while the last two coordinates double the previous coordinate.

We evaluate our algorithms with both synthetic and real-world graphs. For Barabasi-Albert graphs, we fix the number of attachments per node to be 4. For dense Erdos-Renyi graphs, we fix the probability of existence for each edge to be 0.01. For sparse Erdos-Renyi graphs, we fix the probability of existence for each edge to be $\frac{10}{\#\text{vertices}}$.

For strong scaling, we ran our algorithms on Barabasi-Albert graphs with 12 million vertices, dense Erdos-Renyi graphs with 100,000 vertices, and finally sparse Erdos-Renyi graphs with 10 million vertices.

For weak scaling we ran on Barabasi-Albert graphs with a vertex count of $12000000 \times$ #processors, dense Erdos-Renyi graphs with a vertex count of $100000 \times$ #processors, and on sparse Erdos-Renyi graphs with a vertex count of $10000000 \times$ #processors.

For the special graphs, we used a graph we designed called BadBFS, which we will describe in more detail in Section 5. Additionally, we evaluate three graphs from the SNAP dataset [21]. We chose to evaluate on the patent citation [22], LiveJournal [23], and California road network [24] graphs.

Results. In all the following plots we used boxplots. The whiskers indicate the 1.5 IRQ, and the notches indicate the 95% CI around the median.

As a first step, we investigate weak scaling. More specifically, we want to examine the amount of overhead our algorithms have when increasing the number of processors while keeping the ratio of input size to the number of pro-

cessors constant. As seen in Figure 1, Star Decomposition fared very poorly in our benchmarks, and as a result, we did not continue to evaluate the algorithm beyond eight processors.

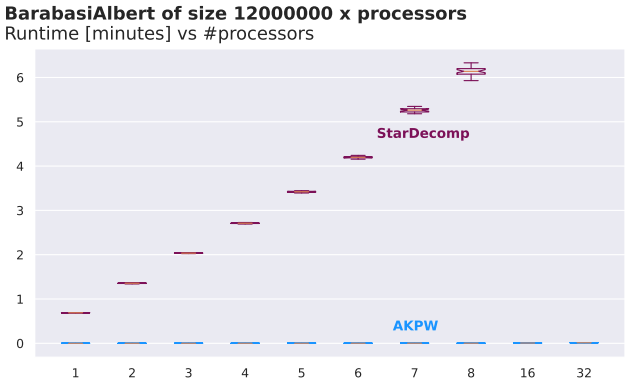

Fig. 1. Weak scaling on Barabasi-Albert graphs with Star Decomposition and AKPW

To investigate the performance of the other algorithms, we turn our attention to Figure 2. As we can see, each algorithm has a linear overhead when increasing the number of processors. RandBFS has the least overhead of these algorithms as it is the simplest, and AKPW has the highest overhead as it requires a lot of coordination across the different threads.

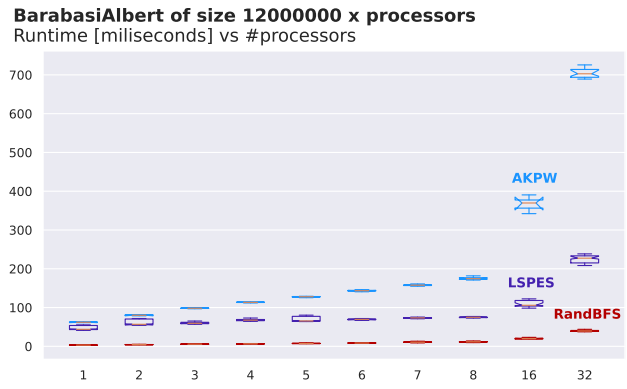

Fig. 2. Weak scaling on Barabasi-Albert graphs with AKPW, LSPES, and RandBFS

After looking at the runtime of our algorithms, we will now look at our second benchmarking result: the average stretch. We again start by comparing Star Decomposition to AKPW in Figure 3. We can see that the average stretch of the Star Decomposition increases with the input size. It performed significantly worse than AKPW, which is stable with increasing input size.

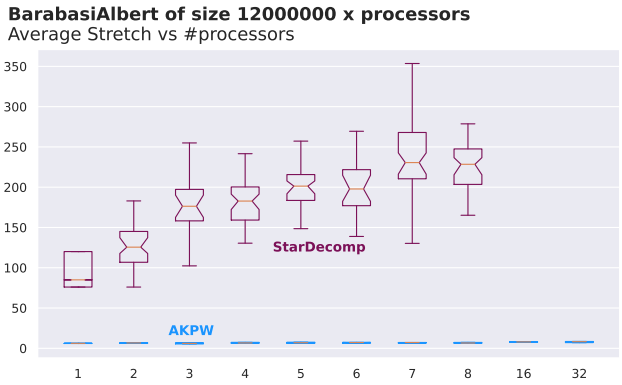

Fig. 3. Average stretch on Barabasi-Albert graphs with AKPW and Star Decomposition

We will now compare the average stretch of the other algorithms to AKPW in Figure 4. In general, all algorithms have an almost constant stretch, even though the graphs are increasing in size. Of the three algorithms, RandBFS and AKPW seem to have almost the same average stretch with LSPES having a slightly higher average stretch most of the time.

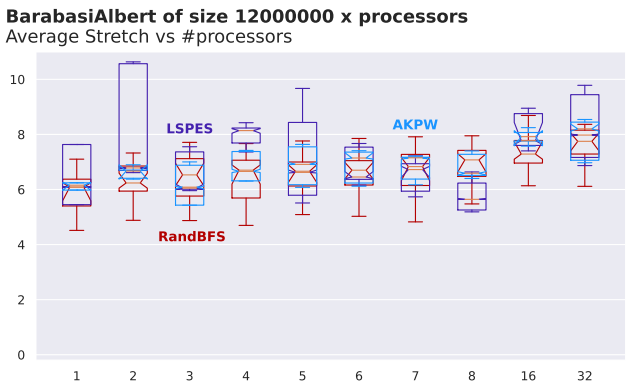

Fig. 4. Average stretch on Barabasi-Albert graphs with AKPW, LSPES, and RandBFS

After looking at weak scaling on Barabasi-Albert graphs, we will now investigate strong scaling using sparse Erdos-Renyi graphs as inputs. Figure 5 shows the performance of AKPW, LSPES, and RandBFS on such graphs with 10 million vertices. RandBFS once again performs best, followed by LSPES, and lastly, by AKPW. The most significant performance gain for all three algorithms was going from one to two processors, which almost halved the runtime. They all continue to scale up to 16 processors, after which the runtime of all three algorithms increases again.

As for the stretch of said algorithms on sparse Erdos-Renyi graphs with 10 million vertices, Figure 6 shows that

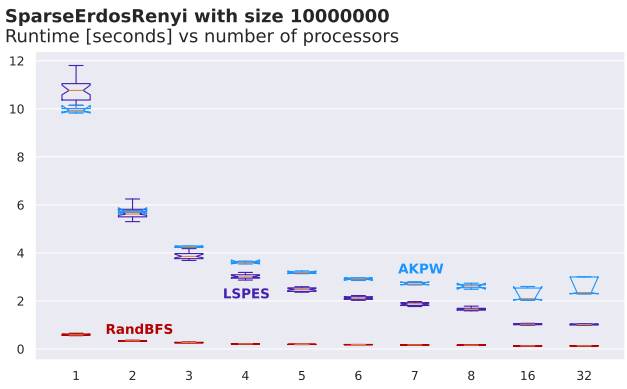

Fig. 5. Strong scaling on Sparse Erdos-Renyi graphs with AKPW, LSPES, and RandBFS

Fig. 7. Runtime of the sequential Alon Karp against the AKPW

increasing the number of processors does not affect the quality of the returned spanning trees. Additionally, it shows that RandBFS and AKPW are almost identical in terms of average stretch, with LSPES having an approximately 50% worse average stretch.

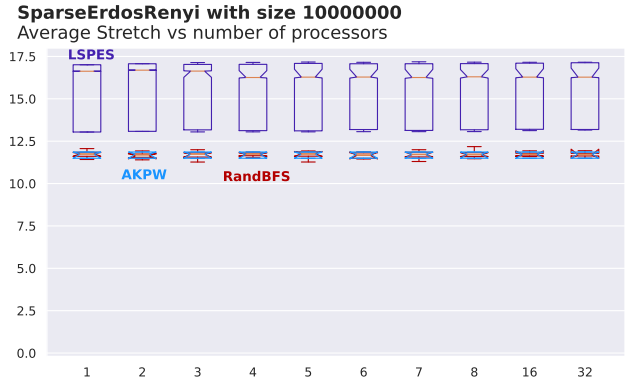

Fig. 6. Stretch on Sparse Erdos-Renyi graphs with AKPW, LSPES, and RandBFS

We will now compare our parallel algorithms to a sequential implementation. Figure 7 shows the sequential implementation of the Alon Karp algorithm, compared to our AKPW implementation. As we can see, the sequential algorithm has a much higher variance and runtime than the AKPW, even with only one core. After further analysis, we believe that the poor performance of sequential Alon Karp is partly due to a naive implementation of the contract method.

Finally, we compare how these algorithms perform on real-world graphs and a specially constructed graph. Figure 8 shows the runtime of AKPW, LSPES, and RandBFS on our constructed graph BadBFS with one million vertices, the *cit-Patents* graph from SNAP with 3'764'116 vertices, the *com-LiveJournal* graph from SNAP with 3'997'961 vertices, and the roadNet-CA graph from SNAP with 1'957'026 vertices. Note that the scale of the y-axis is not the same across the different plots. AKPW performs the worst among these algorithms across all four graphs. RandBFS performs the best on all except for the road network, on which LSPES outperforms. LSPES is close to the average of the other two algorithms for the rest of the graphs.

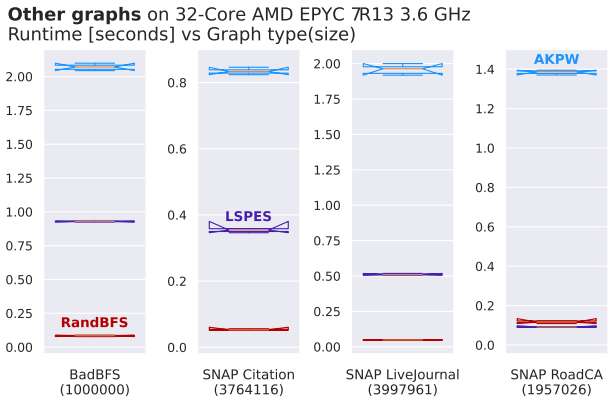

Fig. 8. Runtime on special and real-world graphs

Figure 9 shows the stretch on the same graphs. We note that for BadBFS, the red line of the RandBFS is hidden under the light blue line from AKPW, as they have the same average stretch. Additionally, we again point out that the scale of the y-axis is not the same across the different plots. While on the three SNAP graphs, all three algorithms have a similar average stretch, AKPW and RandBFS perform poorly on BadBFS. In contrast, LSPES produces a reasonable stretch on all evaluated graphs.

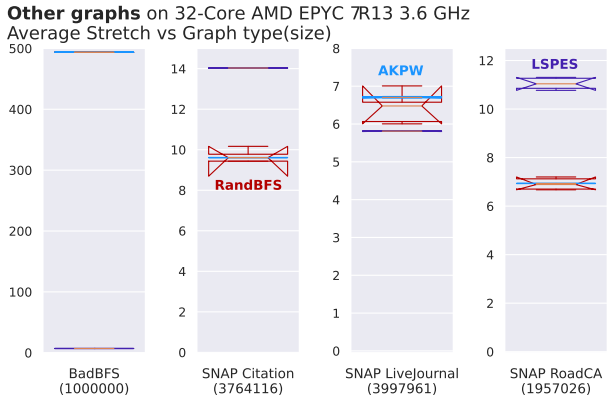

Fig. 9. Stretch on special and real-world graphs

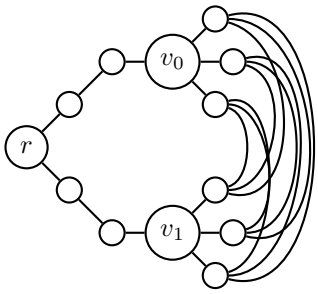

Fig. 10. Example of BadBFS with $n = 13$ and $p = \frac{3}{13}$.

5. DISCUSSION

In this section we will discuss our results and work. In particular, we explore situations in which BFS may perform poorly, open questions in the context of LSPES, and the poor performance of StarDecomp.

RandBFS. Our results indicate that RandBFS outperforms our other algorithms in many cases, considering both runtime and stretch. In the following paragraphs, we want to show that there exists a graph for which BFS computes a spanning tree with an average stretch equal to the diameter, which is the worst case.

We visualize the proposed graph in Figure 10. The graph is constructed as follows: Let n be an integer and $p < 1$. Define a root vertex r to which we attach two distinct paths of length $n \cdot p$. Denote the last vertices in the path as v_0 and v_1, respectively. Then, we add $n \cdot p$ distinct vertices as children to v_0 and v_1. Finally, we create edges between all newly created children of v_0 and v_1. We denote the resulting graph as T. Note that when we run a BFS on T starting from the root vertex r, the resulting tree consists of both paths and all edges of v_0 and v_1. Therefore, the stretch of all edges between the children of v_0 and v_1 will be $\Theta(np)$. We can now bound the average stretch of this instance of BFS: Let $d(e)$ be the stretch of edge e, then

$$\text{AVG-Stretch} = \frac{\sum_{e \in E} d(e)}{|E|} \geq \frac{(np)^2 \cdot \Theta(np)}{4np + (np)^2} = \Theta(np)$$

To get an arbitrarily bad stretch with BFS, we start at a specific vertex. This, however, is not necessary. Consider a new graph, where we attach more vertices to the root r from T. If BFS discovers r first in T, we will get a bad stretch for the subgraph T. Now, assume we create a copy T' of T and add a single edge between the initial root r and the root in the copy denoted by r'. Regardless of what center we choose for the instance of BFS, either r or r' will be discovered before all other vertices in its copy and hence give a high average stretch.

LSPES. As LSPES uses the algorithm introduced by Miller et al. in [5], it is also parameterized by β. In our experiments, we have set $\beta = 0.2$. We think it would be interesting to repeat some of our experiments with different values for β and evaluate them in quality and runtime. We speculate that decreasing β will lead to results closer to RandBFS, as a smaller β will result in larger clusters, which would be closer to a single run of BFS. If this hypothesis is correct, it would be interesting to conduct further research on whether β can be efficiently tuned, depending on constraints on the runtime, input, or resulting stretch.

StarDecomp. After investigating the poor performance of StarDecomp, we identified a programming mistake. However, the fix resulted "only" in a 2x speedup. Thus, StarDecomp may not be a good fit for the libraries we used or the shared memory model in general. To conclusively determine the viability of StarDecomp, evaluating an implementation using other graph libraries or MPI is necessary.

6. CONCLUSIONS

As finding LSSTs is increasingly used as a subroutine for graph problems and linear solvers, finding efficient algorithms which produce high-quality results is of increasing interest. To this end, we have implemented several state-of-the-art algorithms and evaluated them on their runtime and average stretch. Considering our results, we believe that the best general-purpose algorithm is LSPES, even though it does not have the best runtime or average stretch most of the time. We think that RandBFS can be a viable alternative to LSPES if we first compute the spanning tree produced by BFS and then check whether the stretch is acceptable. We believe that there is no use case for AKPW in the unweighted case due to it defaulting to RandBFS in some cases, meaning that it can perform as poorly or worse than RandBFS due to the added overhead. In regard to StarDecomp, we cannot come to a definite conclusion. From our data, it seems to be the worst algorithm in our lineup; however, this may be due to a non-optimal implementation.

7. REFERENCES

[1] Albert-László Barabási and Márton Pósfai, *Network science*, Cambridge University Press, Cambridge, 2016.

[2] Anupam Gupta, Titouan Rigoudy, Tom Tseng, "15-850: Advanced Algorithms, Lecture #5: Distance Preserving Trees," https://www.cs.cmu.edu/~anupamg/advalgos17/scribes/lec05.pdf, 2017, [Online; accessed 5-January-2023].

[3] Guy E. Blelloch, Anupam Gupta, Ioannis Koutis, Gary L. Miller, Richard Peng, and Kanat Tangwongsan, "Near linear-work parallel sdd solvers, low-diameter decomposition, and low-stretch subgraphs," 2011.

[4] Ruben Becker, Yuval Emek, Mohsen Ghaffari, and Christoph Lenzen, "Distributed algorithms for low stretch spanning trees," in *International Symposium on Distributed Computing*, 2019.

[5] Gary L. Miller, Richard Peng, and Shen Chen Xu, "Parallel graph decompositions using random shifts," 2013.

[6] Ittai Abraham, Yair Bartal, and Ofer Neiman, "Nearly tight low stretch spanning trees," 2008.

[7] Michael Elkin, Daniel A. Spielman, and Shang-Hua Teng, "Lower-stretch spanning trees," *CoRR*, vol. cs.DS/0411064, 2004.

[8] Zoltán Király Pál András Papp, Sándor Kisfaludi-Bak, "Low-stretch spanning trees," Undergraduate thesis, Eötvös Loránd University, 2014.

[9] Noga Alon, Richard M. Karp, David Peleg, and Douglas West, "A Graph-Theoretic Game and Its Application to the *k* -Server Problem," *SIAM Journal on Computing*, vol. 24, no. 1, pp. 78–100, Feb. 1995.

[10] Nick Harvey, "Sixth cargese workshop on combinatorial optimization, lecture 1: Low-stretch trees," https://www.cs.ubc.ca/~nickhar/Cargese1.pdf, Accessed: 2023-01-09.

[11] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," in *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[12] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun, "The graph based benchmark suite (GBBS)," in *International Workshop on Graph Data Management Experiences and Systems (GRADES) and Network Data Analytics (NDA)*, 2020, pp. 11:1–11:8.

[13] GBBS Authors, "Gbbs: Graph based benchmark suite," https://github.com/ParAlg/gbbs, 2021, [Online; accessed 12-January-2023].

[14] GBBS Authors, "Low diameter decomposition," https://paralg.github.io/gbbs/docs/benchmarks/connectivity/low_diameter_decomposition/, 2021, [Online; accessed 3-January-2023].

[15] August Rønberg, Daniel Nezamabadi, Dennis Buitendijk, Yves Baumann , Zijan Zhang, "Implementation: Lspes.h," https://gitlab.ethz.ch/dnezamabadi/low-avg-stretch/-/blob/main/benchmarks/LowAvgStretch/LSPES/LSPES.h, 2022, [Online; accessed 13-January-2023].

[16] GBBS Authors, "Overview," https://paralg.github.io/gbbs/docs/library/overview/, 2021, [Online; accessed 3-January-2023].

[17] "Missing return value in graph.h (undefined behavior?)," https://github.com/ParAlg/gbbs/issues/77, [Online; accessed 4-January-2023].

[18] "deletion_fn() is called too late, resulting in segfault," https://github.com/ParAlg/gbbs/issues/80, [Online; accessed 4-January-2023].

[19] Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke, "Networkit: A tool suite for large-scale complex network analysis," 2014.

[20] GBBS Authors, "Bfs," https://paralg.github.io/gbbs/docs/benchmarks/sssp/breadth_first_search, 2021, [Online; accessed 12-January-2023].

[21] Jure Leskovec and Andrej Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, June 2014, [Online; accessed 10-January-2023].

[22] "SNAP Datasets: Patent citation network," https://snap.stanford.edu/data/cit-Patents.html, June 2014, [Online; accessed 11-January-2023].

[23] "SNAP Datasets: Livejournal social network and ground-truth communities," https://snap.stanford.edu/data/com-LiveJournal.

html, June 2014, [Online; accessed 11-January-2023].

[24] "SNAP Datasets: California road network," https://snap.stanford.edu/data/roadNet-CA.html, June 2014, [Online; accessed 11-January-2023].