Techniques for Practical Parallel BFS and SSSP

Richard Wen
University of Maryland
rwen1@umd.edu

Quinten De Man
University of Maryland
deman@umd.edu

Laxman Dhulipala
University of Maryland
laxman@umd.edu

Abstract

Breadth-first search (BFS) and single source shortest paths (SSSP) are two fundamental graph problems with countless real-world applications. It is of major interest to develop efficient parallel algorithms and implementations to solve these problems on modern multicore architectures. The challenge is that these computations are often irregular, and there is no known algorithm that guarantees polylogarithmic depth for all graphs. For BFS, this paper describe several performance engineering methods to develop a practical parallel implementation for all classes of real-world graphs. For SSSP, we introduce a unique contraction based preprocessing method which significantly speeds up queries on high diameter graphs, like road networks. Our method is generic and can be used with any algorithm for SSSP queries.

1 Introduction

In the BFS problem, we are given an undirected, unweighted graph $G = (V, E)$ with $|V| = n$ and $|E| = m$. We are asked to construct a data structure that, given any source vertex $u \in V$, produces $Dist(u, v)$, the length of the shortest path from u to each vertex $v \in V$. The runtime is broken into two components. The time to construct the data structure is called ***preprocessing*** time. The time to output the distances of every vertex from a given source is called ***query*** time. The SSSP problem is defined similarly, but the graph is weighted.

In this paper we are primarily interested in the query time of our algorithms and only require the preprocessing to execute within a reasonably small time limit. In Section 2 we describe the numerous performance engineering methods we used to develop a practical parallel implementation of BFS. In Section 3 we describe our contraction based preprocessing method to speed up SSSP queries.

2 Performance Engineering BFS

We parallelize the standard sequential approach to BFS of iteratively computing frontiers of increasing depth. A ***frontier*** is the set of all points with a given ***depth*** or distance from the source. We first initialize the depth-0 frontier to contain the source vertex. Then, the depth-i frontier is all vertices which are neighbors of a vertex in the depth-$(i-1)$ frontier and are not contained in any prior frontier. By computing the frontiers in ascending order, we can guarantee that by the time we compute the depth-i frontier, the vertices contained in frontiers of depth less than i are already known. Although we must construct frontiers sequentially in order of depth, we can parallelize the construction of individual frontiers. To do this, we take any unvisited neighbors of the current frontier and atomically mark them as visited. Any successful candidates are added to the next frontier. This process can be parallelized easily. We provide the pseudo-code below.

Algorithm 1 Breadth-First Search (BFS)

```
1: procedure BFS(G, s)
2:     dists ← [∞ for v ∈ V]
3:     dists[s] ← 0,   d ← 0,   F ← {s}
4:     while F is not empty do
5:         cands ← flatten([neighbors(v) for v ∈ F])
6:         F ← {v ∈ cands if dists[v].CAS(∞, d)}
7:     return dists
```

Real-world graph datasets can be largely categorized into one of two types. The first, which we'll refer to as ***dense*** or ***low diameter***, exhibit properties of social networks: high average degree, low diameter, and scale-freedom. The second, which we'll refer to as ***sparse*** or ***high diameter***, exhibit properties of road networks: low average degree and high diameter. During preprocessing, we use a heuristic to predict the type of graph. We sample a small set R of random points, then perform two rounds of BFS, treating R as a depth-0 frontier. We then compare the size of our sample to the size of the depth-2 frontier. On dense graphs, the blowup in size is on the order of hundreds to thousands. On sparse graphs, the blowup in size is typically less than ten. Equipped with our high level algorithmic approach and a tool to distinguish between dense and sparse graphs, we will now discuss various optimizations we included in our BFS implementation.

Direction Optimization. Due to the high fan-out of dense graphs, frontiers can become very large during BFS. In fact, it is typical for the vast majority of vertices to be contained in the middlemost one or two frontiers of the BFS. When creating or expanding these frontiers, the standard approach to frontier generation can be inefficient. In particular, the number of candidates can exceed n when individual vertices are the neighbors of multiple different frontier vertices. A known optimization on dense graphs is to change the way large frontiers are expanded [1]. Instead of compiling the unvisited neighbors of the frontier, each unvisited vertex checks if it is a neighbor of the frontier. This eliminates the need for synchronization since each candidate is only accessed by a single thread. Moreover, by eliminating duplicate candidates, we reduce the number of writes and the amount of space needed to maintain the list of candidates.

FCPC '25, March 1–5, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1446-7/2025/03.
https://doi.org/10.1145/3711708.3723448

Observe that for an unvisited vertex, we may add it to the next frontier as soon as it fins a neighbor in the current frontier. Thus, the earlier such a neighbor is found, the fewer neighbors need to be checked. To increase the chances of finding a neighbor in the frontier, we check the neighbors in order of descending degree as a heuristic. We sort the vertices in each adjacency list accordingly during preprocessing.

We use a heuristic to decide whether to use standard or direction-optimized frontier expansion. We use direction-optimization on frontiers where the product of its depth, its size, and the global average vertex degree exceeds the number of vertices. Multiplying the frontier size by the average degree estimates the number of vertices in the next frontier in the first half of the BFS. We include the depth in the calculation because the already-visited vertices, which grow more numerous at higher depths, are filtered out first. We observe experimentally that the cost of direction-optimization decreases substantially at higher depth. Thus, in the latter half of the BFS, the direction-optimized approach is more efficient even on medium-sized frontiers. We find on our dense datasets that including the frontier depth as part of the product seems to estimate the more efficient approach with good accuracy. We disable direction optimization on sparse graphs, as the frontiers are never large enough for it to be worth using over standard frontier expansion.

Reducing Synchronization Overhead. In sparse graphs, the maximum frontier size remains low throughout the BFS, so the parallel steps of the BFS have limited work. As a result, the BFS is often unable to saturate the majority of processors available. In fact, we find that on some of our data, lowering the maximum number of worker threads to 8-16 yields a significant speedup over using all threads. This is due to additional synchronization overheads from managing more threads, most of which may be unproductive. Some schedulers, such as ParlayLib's, can limit the number of worker threads available at runtime. However, other schedulers, such as OpenCilk's, are limited in this functionality.

To compensate for this, we implement a lightweight job pool that works on any scheduler. We break the jobs of a frontier into fixed-size blocks and assign single blocks to processors, which will in run the standard frontier construction procedure and push block-sized partial frontiers back into the pool for the next level of BFS. We find that this job pool gives us almost all the speedup that limiting the number of threads gives us. Using both the pool and limiting threads is better than either approach individually.

Memory Tricks. Several of the optimizations we implemented are directly related to reducing cache misses, page fetches, and unnecessary writes. First, we simply reuse the list of distances to indicate whether vertices are visited, saving space. We initialize all distances to infinity. We atomically set the distance of a vertex to the depth of its frontier once known. A vertex is visited if its distance is not infinite.

Graph	Type	n	m
Soc-LiveJournal	Social	4.8M	42M
ENWiki-2023	Web	6.6M	150M
Hollywood-2009	Collaboration	1.1M	55M
Erdos-Renyi	Random	9.9M	490M
South-America	Road	22M	29M
North-America	Road	86M	110M
GeoLifeNoScale-5	K-nn	25M	77M
Grid-1000-10000	Synthetic	9.9M	20M

Table 1. Input graph datasets used in our experiments.

We also preprocess an ordering for the vertices and relabel the vertices accordingly. By ordering the vertices so that two vertices are more likely to be nearby in the ordering when they appear in the same frontier, we can reduce the number of cache misses when fetching adjacency lists. We tested a number of different orderings cognizant of spacial locality, including shingling and layered label propagation [4] [3]. However, the ordering that we found had the best results was simply sorting the vertices by degree. We have two suspected reasons for the effectiveness of this ordering. The first is that high degree vertices are more likely to appear earlier in the search. The second is that high degree vertices may appear together in "cities" where high degrees are more characteristic of their members. Either scenario would lead to fewer cache misses using this ordering. We also found that using this ordering with CSR is more effective than using other storage formats that may pack data in a more cache-friendly way, such as keeping a cacheline-length prefix of a vertex's adjacency list next to its CSR row pointer.

Our final optimization limits TLB misses by aligning data to 2MB chunks and using Linux's `madvise` system call to use huge pages, storing the data in larger memory blocks.

2.1 BFS Experimental Results

All of the experiments presented in this paper were run on a 96-core machine (with two-way hyperthreading), with 4 × 2.1 GHz Intel Xeon(R) Platinum 8160 CPUs (each with 33MiB L3 cache) and 1.5TB of main memory. Table 1 describes the graph datasets that we used in our experiments. All edge weights were generated uniformly at random for SSSP.

Figure 1 compares the time taken to run BFS on our datasets using different implementations. The first four graphs are "dense" and the remaining "sparse". We compare against the BFS implementation included in ParlayLib's examples and the BFS implementation from the GBBS benchmarking suite [6]. Measurements were taken as the average of 10 queries with random starting points, after a single warmup query. Our implementation is $4-8\times$ faster than the ParlayLib example and $1.15-2\times$ faster than GBBS on dense graphs. On sparse graphs, our implementation is $1.25-2\times$ faster than the ParlayLib example and $3-6\times$ faster than GBBS.

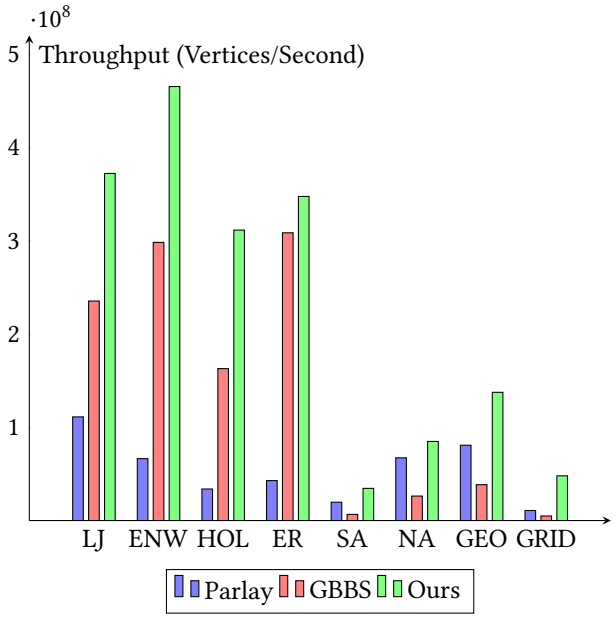

Figure 1. Throughput of BFS with various implementations.

3 Contraction-Based SSSP

SSSP Related Work. Two approaches to parallel SSSP computation are stepping algorithms [2, 7, 10] and contraction hierarchies [5, 8, 11]. Our technique combines the power of these two approaches. Stepping algorithms are highly effective on low diameter and well connected graphs, yielding highly parallel and work-efficient algorithms. However, they struggle on high diameter graphs such as road networks where the depth of any shortest path tree is high. Intuitively, this is because if the frontiers are not large enough, there is not enough work to saturate all of the threads, but if the frontier is too large the total work of the algorithm increases. Parallel contraction hierarchies (PCH) solve this issue by contracting the input graph with several rounds of contraction while constructing a new auxiliary graph. Effectively, any shortest path tree in the auxiliary graph has reduced depth. However, PCH is still outperformed by stepping algorithms on low diameter graphs such as web graphs.

3.1 Our Approach For SSSP

At a high level, our approach is to perform a few rounds of contraction on the input graph, and then use a stepping algorithm to answer queries on the smaller and denser contracted graph. Our contraction method shares similarities to contraction hierarchies, but differs in a few key ways: (1) we do not contract the graph to completion, (2) we perform queries in the contracted graph and not the resulting auxiliary graph, (3) we use ρ-stepping for queries rather than the typical bi-directional search in contraction hierarchies, and (4) we use a much simpler (and easier to compute) criteria for contracting vertices.

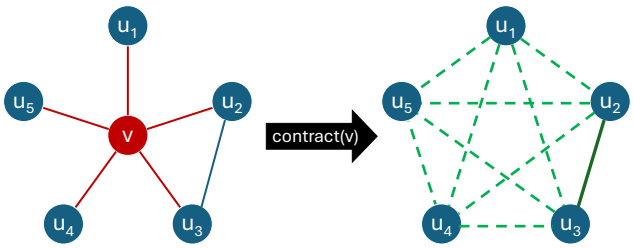

Figure 2. A vertex v is contracted. The red edges and v are removed from the graph. The dashed green edges are inserted. The solid green edge has its weight updated.

Contracting Vertices. When a vertex v is ***contracted***, it and all its incident edges are removed from the graph. The length of any paths through v are preserved by adding ***shortcut edges*** between each pair of neighbors of v. Formally, for all $e = (u_1, u_2) \in N(v)^2$ the edge (u_1, u_2) is added to $\mathcal{E}$ with weight $w(e) = w(v, u_1) + w(v, u_2)$. If $e \in \mathcal{E}$ already, its weight is set to $\min(w(e), w(v, u_1) + w(v, u_2))$. Figure 2 shows an example of a vertex being contracted. Note that any independent set of vertices can be contracted in parallel (performing a reduction over all the weights of parallel edges inserted). Thus the graph can be contracted in multiple ***rounds*** where each round a large independent set of the vertices contracts.

Contraction Criteria. When a vertex v with degree d contracts, $e^- = d$ edges are removed from the graph and up to $e^+ \leq d(d-1)/2$ edges are added. Thus to minimize the amount of extra edges added to the graph PCH algorithms simulate contracting vertices and attempt to find an independent set of vertices whose contraction minimizes $e^+ - e^-$. Prior work [7] has found that this simulation of contracting vertices is the most expensive step in the algorithm.

Instead, our algorithm works with a small fixed constant ***degree threshold*** c such that only vertices with degree $\leq c$ may be contracted. This ensures that each contraction only increases the total edges by at most $c(c-1)/2 - c$. Additionally, determining if a vertex has degree $\leq c$ is trivial and much easier to compute. Surprisingly, in our experiments we found that the total number of edges almost always decreases, indicating that the shortcut edges often already exist in the graph. We use $c = 4$ in our implementation.

Contraction Rounds. We call a vertex ***live*** if it has not been contracted and has had degree $\leq c$ in every round. Our algorithm repeatedly computes a large independent set of the live vertices to contract each round, until their are no live vertices. Thus once a vertex's degree exceeds c it is no longer considered for contraction (although its degree could decrease again). We can compute a large independent set of live vertices in $O(c \cdot n_i)$ work and $O(1)$ depth (where n_i is the number of live vertices in round i) using a single round of Luby's algorithm for maximal independent set [9]. The size of each independent set is at least n_i/c in expectation, thus

		LJ	ENW	HOL	ER	SA	NA	GEO	GRID
Original Graph	Num. Vertices	4.8M	6.6M	1.1M	9.9M	22M	86M	25M	9.9M
	Num. Edges	42M	150M	55M	490M	29M	110M	77M	20M
	SSSP Depth	50	26	35	37	17,995	12,447	11,798	7,248
	ρ-Stepping Time	0.25	**0.26**	**0.048**	**0.54**	0.81	8.3	**0.42**	0.51
Contracted Graph	Num. Vertices	2.8M	5.9M	1.0M	9.9M	5.7M	13M	24M	6.3M
	Num. Edges	40M	150M	55M	490M	16M	40M	77M	22M
	SSSP Depth	50	26	35	37	5,875	3,630	11,558	4,633
	ρ-Stepping Time	**0.093**	0.23	0.045	0.54	**0.20**	**1.3**	0.38	**0.29**
Query Times	Uncontraction Time	0.071	0.060	0.0092	0.027	0.092	0.56	0.084	0.023
	Total Query Time	**0.16**	0.29	0.055	0.57	**0.29**	**1.9**	0.46	**0.32**

Table 2. Statistics for the contracted graphs and the running times for ρ-stepping and total query time.

this process takes $O(\log n_0)$ depth w.h.p. The work in each round decreases geometrically resulting in a total expected work of $O(n_0)$. In our implementation we stop contraction once the number of live vertices is $< 10^6$.

SSSP Queries. To answer an SSSP query from source s, we must first undo some contractions to ensure that s is present in the graph and that its shortest distances to other vertices are still correct. We call this process ***exposing*** vertex s. During preprocessing, we store the vertices that contracted in each round along with the edges that were deleted. We call the vertices that these edges connect to, the ***dependencies*** of the contracted vertex. To expose a vertex v, first its dependencies are exposed recursively. The base case of this recursion is a vertex that is already present in the graph. After exposing all of v's dependencies, v and all of the edges that contracted with v are reinserted into the graph. Thus the state of the graph is as if v (and a small local subgraph of its recursive dependencies) were never contracted.

After exposing s, the algorithm performs ρ-stepping [7] to compute the shortest path distance from s to every vertex present in the contracted graph. For road networks, the contracted graph has much fewer vertices, fewer edges, and is low diameter, so ρ-stepping is extremely practical. For already low-diameter graphs the contraction process may have had little effect, but ρ-stepping is known to be effective on these graphs already. Notably our algorithm can use any other highly optimized SSSP algorithm for low-diameter graphs as a subroutine. Our implementation uses $\rho = 2^{20}$.

Finally, the algorithm must ***uncontract*** the graph to compute the distances to vertices not present in the contracted graph. The rounds of vertex contractions are "undone" in reverse order, computing the distance from the source to each vertex in the given round. This order ensures that in any given round, we have already computed the distance of the dependencies of every vertex in this round. Then the shortest path distance $Dist[v]$ of v is simply $\min_{u \in N(v)} Dist[u] + w(u, v)$ where $N(v)$ is the dependencies of v. This can be computed in parallel for the vertices in a single round, and it can be done without modifying the graph. Uncontraction takes $O(n_0)$ expected work and $O(\log n_0)$ depth w.h.p.

3.2 SSSP Experimental Results

Table 2 shows statistics about the contraction process and running times for SSSP queries with and without our method. We conclude that **our technique can significantly speed up SSSP queries on high diameter graphs, while adding negligible overhead for low diameter graphs.** We note that it is possible to completely eliminate the overhead on low diameter graphs using the heuristic we described for BFS to determine the type of graph during preprocessing.

On most high diameter inputs (South-America, North-America, and Grid-1000-10000) the contraction process notably decreases the depth of the shortest-path trees. For these graphs and one low diameter graph (Soc-LiveJournal), the number of vertices decreases significantly from contraction, as well as the number of edges in some cases. The number of vertices decreases by 59% on average and the number of edges decreases by 26% on average for these graphs. The lower depth and smaller size of the graph allows for excellent speedups for ρ-stepping, 3.7× on average. The uncontraction is highly parallel and fast, adding little overhead. The total query time using our algorithm is on average 2.6× faster than ρ-stepping. The exceptional high diameter graph is the K-nn graph (GeoLifeNoScale-5), on which our contraction has no impact since each vertex has degree at least 5.

On low diameter inputs (ENWiki-2023, Hollywood-2009, and Erdos-Renyi) the contraction process has little impact. Thus the running time for ρ-stepping stays around the same. Since little contraction occurred, the overheads of uncontracting vertices during query time are very low, and the total query time is only slightly worse than ρ-stepping. Our results indicate that our algorithm retains the excellent query performance of ρ-stepping on low diameter graphs while yielding significant speedups on high diameter graphs.

References

[1] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing Breadth-First Search. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–10. https://doi.org/10.1109/SC.2012.50

[2] Guy E. Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. 2016. Parallel Shortest Paths Using Radius Stepping. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures* (Pacific Grove, California, USA) *(SPAA '16)*. Association for Computing Machinery, New York, NY, USA, 443–454. https://doi.org/10.1145/2935764.2935765

[3] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web* (Hyderabad, India) *(WWW '11)*. Association for Computing Machinery, New York, NY, USA, 587–596. https://doi.org/10.1145/1963405.1963488

[4] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. 2009. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Paris, France) *(KDD '09)*. Association for Computing Machinery, New York, NY, USA, 219–228. https://doi.org/10.1145/1557019.1557049

[5] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. 2011. PHAST: Hardware-Accelerated Shortest Path Trees. In *2011 IEEE International Parallel and Distributed Processing Symposium*. 921–931. https://doi.org/10.1109/IPDPS.2011.89

[6] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. 2020. The Graph Based Benchmark Suite (GBBS). In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Portland, OR, USA) *(GRADES-NDA'20)*. Association for Computing Machinery, New York, NY, USA, Article 11, 8 pages. https://doi.org/10.1145/3398682.3399168

[7] Xiaojun Dong, Yan Gu, Yihan Sun, and Yunming Zhang. 2021. Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '21)*. Association for Computing Machinery, New York, NY, USA, 184–197. https://doi.org/10.1145/3409964.3461782

[8] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th International Conference on Experimental Algorithms* (Provincetown, MA, USA) *(WEA'08)*. Springer-Verlag, Berlin, Heidelberg, 319–333.

[9] Michael Luby. 1986. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.* 15, 4 (Nov. 1986), 1036–1055. https://doi.org/10.1137/0215074

[10] U. Meyer and P. Sanders. 2003. Δ-stepping: a parallelizable shortest path algorithm. *J. Algorithms* 49, 1 (Oct. 2003), 114–152. https://doi.org/10.1016/S0196-6774(03)00076-2

[11] Zijin Wan, Xiaojun Dong, Letong Wang, Enzuo Zhu, Yan Gu, and Yihan Sun. 2024. Parallel Contraction Hierarchies Can Be Efficient and Scalable. arXiv:2412.18008 [cs.DS] https://arxiv.org/abs/2412.18008