Final Paper: Parallel BFS Algorithms and Optimizations

Kevin Lin
klin5@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Abstract

As real-world networks continue to scale to billions of vertices and edges, breadth-first search has become one of the fundamental algorithms to parallelize and optimize. However, optimizing BFS can be difficult due to the variance in graph topology. This paper evaluates the performance of three approaches to parallel BFS: top-down, bottom-up, and hybrid. Implementation is done using C++ and OpenMP, and testing is done on an AMD EPYC 7313P processor using sparse and dense datasets. The results show top-down traversal being the most optimal for high-diameter, low-degree, sparse graphs, while bottom-up traversal being most optimal for small-world graphs. On high-diameter traversals for small-world graphs, the hybrid approach achieves significant speedups by switching between traversal approaches. There are also significant improvements in the top-down approach when switching from critical sections to atomic operations of up to a 16x speedup on a dense graph and 13x speedup on a sparse graph.

1 Introduction

Breadth-first search is one of the oldest and well-studied path finding algorithms. It is used in the Graph500, a benchmark test for supercomputers, and is one of the building blocks for many other complex algorithms. The parallelized version of this algorithm comes in many shapes and sizes, each having their own pros and cons, but I wanted to investigate the bulk sync strategy and some of its optimizations.

A challenge for BFS is its adaptability to different graph topologies, such as sparse and dense graphs [6]. Real world data rarely come in a uniform pattern, so I was curious how we could create an algorithm that could not only take advantage of parallel computing, but also be efficient across all kinds of inputs and hardware.

I will be implementing and testing three parallel BFS approaches: top-down, bottom-up, and hybrid, and then also comparing critical section and atomic implementations.

Top-down is the traditional and more intuitive way to perform BFS, where we iterate through each node in the current frontier to discover the frontier of the next layer. Already visited nodes are skipped, while unvisited nodes are appended to the frontier queue.

Bottom-up takes the opposite approach, where a loop iterates through all unvisited nodes instead. This aims to find the vertices that are children of nodes in the frontier. This method is of significant benefit because it can discover fewer edges on average compared to the top-down strategy.

The hybrid algorithm is a top-down and bottom-up merge. It uses a heuristic algorithm to find the optimal time to switch between the two, using the proportion of the frontier size to the graph size.

NCSU-CSC, Raleigh, NC
2025. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Algorithm 1 Top-Down BFS. Adapted from [2].

Require: G: Graph, $ActiveFrontier$: Queue, $Parent$: Array
Ensure: $NextFrontier$: Queue

```
1:  function TopDown(G, ActiveFrontier, Parent)
2:      NextFrontier ← ∅
3:      for each u ∈ ActiveFrontier do
4:          for each neighbor v of u do
5:              if Parent[v] = −1 then          ▷ If v is unvisited
6:                  Parent[v] ← u
7:                  NextFrontier ← NextFrontier ∪ {v}
8:              end if
9:          end for
10:     end for
11:     return NextFrontier
12: end function
```

This dynamic will allow the algorithm to efficiently manage a wide range of graph topologies.

I will be testing these algorithms on the NCSU ARC cluster with OpenMP. I will use two data sets to cover sparse and dense graphs and different numbers of threads for comparison.

2 Implementations and Theoretical Expectations

2.1 Sequential

The sequential implementation of BFS has the work complexity of O(vertices + edges) and a depth complexity of O(diameter). The means that its execution time will scale linearly with the input size, making it not a very suitable solution for real-life graphs with billions of edges.

The parallel implementation will have the total work spread across multiple processors, making its time complexity equal to Work/Processor + Depth. However, we should expect diminishing returns with more threads as a result of synchronization costs and thread overhead.

2.2 Top-down

The Algorithm 1 top-down implementation uses a double loop, with the outer loop iterating through the nodes in the current frontier and the inner loop iterating through the neighbors in search of an unvisited node. The outer loop is parallelized, dividing the frontier up for each thread.

We expect top-down to perform well on high-diameter, sparse graphs but suffer significantly on dense, small-world graphs where the frontier becomes too large. When the frontier grows to a significant portion of the graph and each vertex has many edges, there are bound to be edge checks that point to the same vertex. This

Algorithm 2 Bottom-Up BFS. Adapted from [2].

Require: G: Graph, $ActiveFrontier$: Bitmap, $Parent$: Array
Ensure: $NextFrontier$: Bitmap

```
1:  function BottomUp(G, ActiveFrontier, Parent)
2:      NextFrontier ← ∅
3:      for each v ∈ G.Vertices do
4:          if Parent[v] = −1 then                ▷ If v is unvisited
5:              for each neighbor u of v do
6:                  if u ∈ ActiveFrontier then
7:                      Parent[v] ← u             ▷ Found a parent
8:                      NextFrontier ← NextFrontier ∪ {v}
9:                      break                     ▷ Stop checking neighbors
10:                 end if
11:             end for
12:         end if
13:     end for
14:     return NextFrontier
15: end function
```

means redundant edge checks, which adds unnecessary execution time.

2.3 Bottom-up

The Algorithm 2 bottom up BFS implementation uses a frontier bitmap and a double for-loop. The reason we use a bitmap is to get fast lookups to check if the vertex is part of the frontier. The outer loop is parallelized and iterates through all vertices in the graph. The inner loop will only run for unvisited vertices, and it iterates through the neighbors, checking if any of them are part of the frontier. If there is a match, the algorithms will early exit for the vertex, saving some execution time.

For this algorithm, it will be expected to be inefficient for small frontiers but extremely efficient for large frontiers, since it can skip more edges. In Algorithm 2 line 6, once a neighbor is part of the frontier, we can stop processing the rest of the edges because we know the current vertex is part of the next frontier. This means bottom-up should work very well with dense, small-world graphs, where the degree is high, and we reach a large frontier very fast. For small frontiers, such as a road network, the algorithm will be unnecessarily checking nearly the entire graph repeatedly, causing significant performance issues.

2.4 Hybrid

The Algorithm 3 Hybrid BFS implementation uses alpha and beta parameters to mark thresholds to switch between top-down and bottom-up. In between the switches, the frontier will need to change data structures, between a queue and a bitmap. This is required because the frontiers are of vastly different sizes, and a bitmap allows a constant lookup to check if a vertex is part of the frontier [2].This change should only happen a few times and when the frontier is small, so it is not a significant overhead.

For hybrid implementation, it is expected to provide the best of both worlds, matching the best performance of both top-down and bottom-up in all topologies. The chosen hybrid heuristic will begin in the top-down, since the frontier begins small. Then if the

Algorithm 3 Hybrid BFS. Adapted from [2].

Require: $G = (V, E)$: Graph, $Source$: Vertex, α, β: Tuning Params
Ensure: $Distances$: Array

```
1:  function HybridBFS(G, Source)
2:      Distances ← initialize all to -1
3:      Distances[Source] ← 0
4:      Frontier ← {Source}
5:      N ← |V|                                   ▷ Total vertices
6:      UseTopDown ← true
7:      while Frontier ≠ ∅ do
8:          n_f ← |Frontier|                      ▷ Size of current frontier
9:          if UseTopDown then
10:             if n_f > N/α then                 ▷ Large frontier -> Bottom-Up
11:                 UseTopDown ← false
12:             end if
13:         else
14:             if n_f < N/β then                 ▷ Small frontier -> Top-Down
15:                 UseTopDown ← true
16:             end if
17:         end if
18:         if UseTopDown then
19:             Frontier ← TopDown(G, Frontier, Distances)
20:         else
21:             Frontier ← BottomUp(G, Frontier, Distances)
22:         end if
23:     end while
24:     return Distances
25: end function
```

frontier ever gets larger than 6.67% of the graph size, the strategy will switch to bottom-up. Then, when the frontier reaches a frontier size of less than 4% of the graph size, the strategy will switch back to top-down. These values were taken from evaluations from the direction optimization paper by Beamer [2].

It is predicted that hybrid BFS will work well on graphs that have sparse and dense topologies, and with queries that span across the graph.

3 Experiment Setup

My experiments were run on the NCSU arc cluster. I used the AMD EPYC 7313P 16-Core Processor with 128GB DDR4 ram and wrote my code in C++ using OpenMP for parallel processing.

Regarding the data set, I used two real-life data sets from the Stanford Network Analysis Project (SNAP). One dense, small-world graph and one sparse, high-diameter graph. I chose these datasets primarily for their difference in degree.

4 Result

4.1 RoadNet-Cal

In Figures 1, 2, and 3, we see that the top-down approach yielded the best performance. Their execution time remained lower compared to all other algorithms. Interestingly, for some figures, the execution time increased even as the thread count increased. This can be attributed to thread overhead and synchronization costs. It

Table 1: Experimentation Dataset

Abbreviation	Graph	# Vertices (M)	# Edges (M)	Degree	Diameter	Directed	References
livejournal	soc-LiveJournal1	4,847,571	68,993,773	28.46	16	No	[4]
calnet	roadNet-CA	1,965,206	2,766,607	2.82	849	No	[5]

Figure 1: Sparse Network (Short Distance)

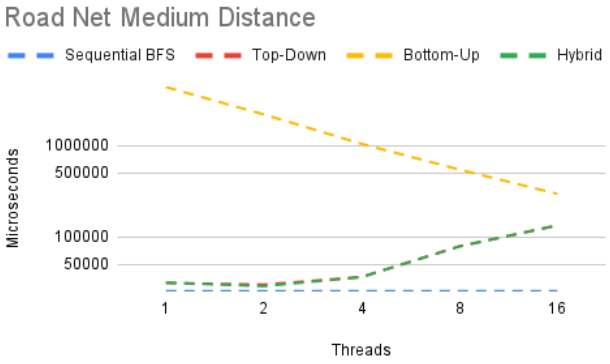

Figure 2: Sparse Network (Medium Distance)

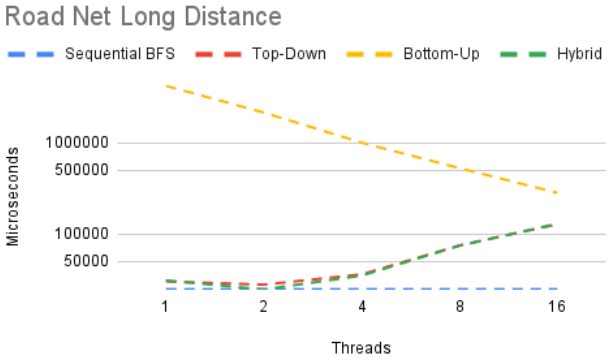

Figure 3: Sparse Network (Long Distance)

seems that the best performance for the query would be around 2-4 threads.

The bottom up approach in this dataset performed significantly worse, and this confirms our expectations that this approach performs more work, and the difference is more clear with fewer threads for parallelization. However, there is a strong trend that with more threads, this approach can execute significantly faster, and may even begin to perform better than the top-down approach. In Figures 1 and 2, we see that at 16 threads, the execution time gap between top-down and bottom up has narrowed to a few hundred microseconds. If we continue to experiment with more threads, there is a likely chance that bottom-up will be the better choice.

The hybrid approach recognizes the topology and closely adheres to the top-down approach, achieving the same performance as top-down. Since road networks have a high diameter and a low average degree, the frontier never becomes a significant portion of the total nodes in the graph. This could also be a critical fault, as the different algorithms seem to perform better/worse with more threads. The ongoing trend with more threads shows that when bottom-up begins to perform much better than top-down, the hybrid approach will still select the top-down step for each layer, because the frontier size is still proportionally small.

4.2 Soc-LiveJournal

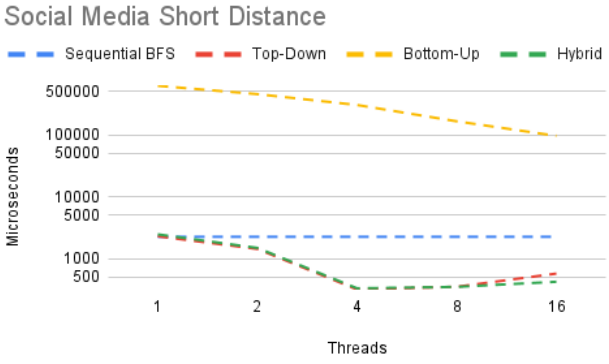

Figure 4: Dense Network (Short Distance)

The livejournal results were very informative and highlighted the strength of the hybrid approach. In Figures 4 and 5, the hybrid continued to follow the top-down time curve, but in Figure 6, we see that it begins to break away and perform much faster than the other two algorithms. Shorter distance queries do not give the hybrid approach enough time and layers to take advantage of switching strategies. In the longer distance test, the hybrid approach is able to take advantage of the efficient queuing approach for small frontiers

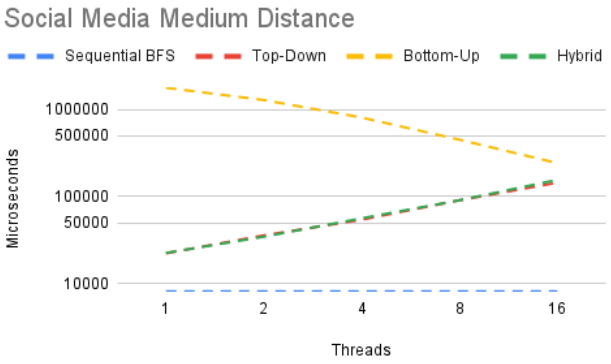

Figure 5: Dense Network (Medium Distance)

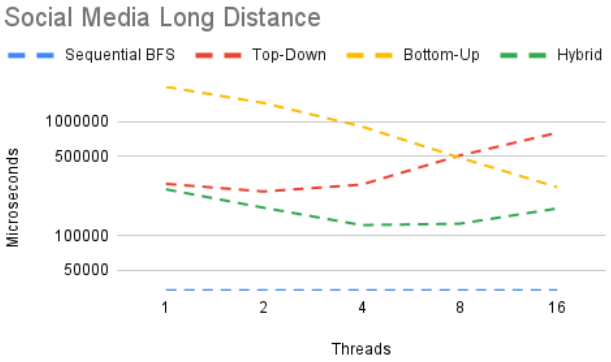

Figure 6: Dense Network (Long Distance)

of top-down, and then switch to bottom-up for the rest of the search to benefit from edge skipping.

The pure top down approach suffered high execution times due to the frontier size exploding. Even as more threads were allocated, the top-down approach began to take longer to finish. Parallelization of work was not enough to compensate for increased thread overhead.

The bottom-up approach here appeared to follow the same pattern as the sparse graph tests. It has very bad efficiency for low threads, but extremely strong performance with many threads. In Figure 6, the gap between bottom-up and hybrid seems to be only a few 100,000 microseconds at 16 threads, which is still nearly a 2x speedup for hybrid compared to bottom up.

4.3 Critical Section vs Atomic Operation Result

Another test I did was to measure the performance between critical sections and atomic operations. In Figure 9, we see that there happens to be a massive 16x speedup with top-down-atomic compared to the standard top-down-critical-section. It seems that implementing compare-and-swap (CAS) or atomic adds is superior to thread-blocking locks for this algorithm.

It also appears that top-down benefits from atomics while bottom up does not. The line for bottom-up atomic initially outperforms the critical section bottom-up, but as we increase the thread count, the

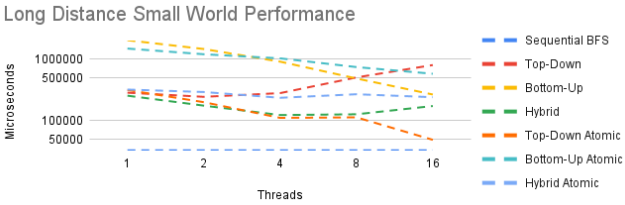

Figure 7: Atomic vs Critical in Small-World

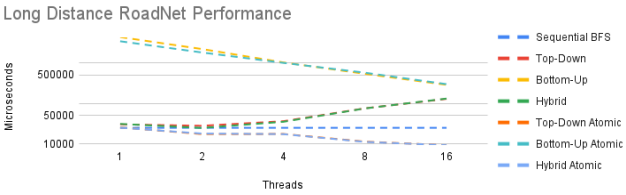

Figure 8: Atomic vs Critical in Sparse Graph

critical section bottom-up outperforms the atomic implementation significantly.

The results for this graph are really surprising because top-down usually performs worse in high degree graphs, and bottom up usually beats top-down in terms of performance, especially with higher thread counts. In this scenario, the only possible explanation would be that the atomic adds for top-down significantly improved performance by negating a lot of synchronization costs and thread overhead. This allowed the algorithm to benefit from more threads.

The hybrid atomic implementation performed worse than its counterpart on all threads, and this is likely due to the algorithm spending a higher percentage of the execution in atomic bottom-up, rather than top-down. In the small world graph, the frontier can reach the threshold to switch in just a few levels.

It seems like the best performer for different topologies would be a hybrid BFS implementation with an atomic top-down implementation and a critical lock bottom-up implementation.

4.4 Surprising Sequential Performance

A notable observation is that the sequential implementation seems to constantly outperform all the parallel implementations of BFS. This was initially very surprising to me, but realized that the main reason is due to the size of the graph. Because the machine on which I was testing was rather powerful, the sequential implementation did not struggle significantly in the queries. The CPU I was using had a 128MB L3 cache that could fit the entire graph for high-speed retrieval [1].

When I introduced parallel counterparts, the benefits of delegating work did not offset thread overhead and synchronization, which causes the execution time to increase instead of decrease when you allocate more work. For small queries, it is not very efficient to parallelize because there is not enough work to save time on, but for larger graphs with billions of edges and vertices, parallel algorithms become much more impactful.

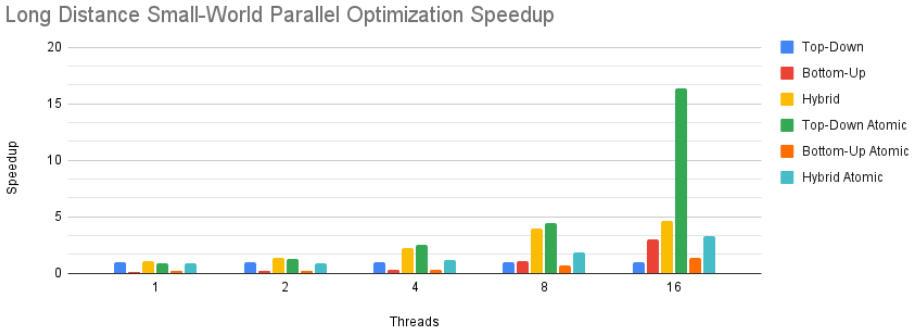

Figure 9: Small-World Optimization Speedup

Another thing is that BFS is naturally memory bound, that is it does very little math and spends most of its time reading memory addresses [3]. Parallelism here only helps calculate faster, but increasing memory bandwidth and speed could decrease BFS query times.

5 Conclusion

This paper evaluated the performance of top-down, bottom-up, and hybrid parallel BFS algorithms across sparse and dense graph topologies and across different numbers of threads. The results confirm that no single traversal strategy is universally optimal. The top-down approach was most efficient for high-diameter, sparse networks (RoadNet-CA), where the frontier remains small relative to the graph size. On the other hand, the hybrid strategy demonstrated superior performance on dense, small-world graphs (LiveJournal), particularly for long-distance queries, where it used bottom-up to skip redundant edge checks during the peak frontier size of the query's traversal.

Another finding is in the optimization between critical sections and atomic operations. We observed a dramatic speedup (up to 16x) when replacing critical sections with atomic operations in the top-down implementation. However, this benefit did not extend to the bottom-up approach. This suggests that an optimal Hybrid implementation should utilize atomic operations during the top-down phases and thread-locking mechanisms during the bottom-up phases.

References

[1] Advanced Micro Devices, Inc. 2021. *AMD EPYC™ 7313P.* https://www.amd.com/en/products/processors/server/epyc/7003-series/amd-epyc-7313p.html Accessed: 2025-12-10.

[2] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-Optimizing Breadth-First Search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12).* IEEE Computer Society Press, Salt Lake City, Utah, USA, Article 12, 10 pages. doi:10.1109/SC.2012.50

[3] Maryia Belova and Ming Ouyang. 2017. Breadth-First Search with A Multi-Core Computer. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* IEEE, Lake Buena Vista, FL, USA, 579–587. doi:10.1109/IPDPSW.2017.48

[4] Jure Leskovec and Andrej Krevl. 2014. *SNAP Datasets: Stanford Large Network Dataset Collection.* https://snap.stanford.edu/data/soc-LiveJournal1.html Accessed: 2025-12-02.

[5] Jure Leskovec and Andrej Krevl. 2014. *SNAP Datasets: Stanford Large Network Dataset Collection.* https://snap.stanford.edu/data/roadNet-CA.html Accessed: 2025-12-02.

[6] Wikipedia Contributors. 2025. *Parallel breadth-first search.* https://en.wikipedia.org/wiki/Parallel_breadth-first_search Accessed: 2025-12-02.