Parallel Betweenness Centrality via OpenMP Final Report

SUMMARY

We parallelized Brandes' Algorithm for computing Betweenness Centrality on large-scale social graphs by developing a multi-core CPU implementation using OpenMP. We analyzed their performance on the Gates cluster and Bridges-2 machines to evaluate how different graph topologies (e.g., social networks vs. road networks) and machine architectures affect parallel scaling and memory bottlenecking.

BACKGROUND

Betweenness Centrality (BC) is a metric used in graph theory and network analysis to identify influential nodes by calculating the frequency with which the shortest paths between pairs of other nodes pass through a node. Vertices with high BC scores are reachable from many other vertices on relatively short paths.

Formally, given a directed or undirected graph $G = (V, E)$, let σ_{st} be the number of shortest paths between a vertex pair $(s, t) \in V \times V$ and $\sigma_{st}(v)$ be the number of shortest $s - t$ paths passing through some $v \in V$. Then the betweenness centrality of vertex v is:

$$BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

The best known sequential algorithm to find the BC of all nodes in a graph is Brandes' Algorithm which has O(V · E) runtime and O(V + E) space complexity. It relies on an equivalent formulation of betweenness centrality given below:

$$BC(v) = \sum_{s \neq v \in V} \delta_s(v)$$

where the "pairwise dependency" is defined as $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$ and the "single-source dependency" is defined as $\delta_s(v) = \sum_{t \in V} \delta_{st}(v)$. In other words, summing the pairwise dependencies on v for every possible pair of vertices s and t is equivalent to summing the single-source dependency on v from every source s. Brandes' algorithm relies on the finding that $\delta_s(v)$ can be computed dynamically by working from the leaves to the root of a shortest paths tree rooted at a source s with the following recursive relation:

$$\delta_s(v) = \sum_{w:v \in pred(s,w)} \frac{\sigma_{sv}}{\sigma_{sw}}(1 + \delta_s(w))$$

Brandes' algorithm iterates over all vertices one at a time, and requires two distinct phases.

For each vertex in the graph (source vertex s):

1. Forward Pass: A single-source shortest path algorithm is run from vertex s to compute a tree of shortest paths to every other vertex in the graph. Breadth-First Search (BFS) is used in an unweighted graph.
2. Backward Pass: Working from the leaves of the path tree back to the root, the single-source dependency $\delta_s(v)$ on each intermediate vertex v is calculated, used to update v's centrality score, and propagated backwards up the tree.

Algorithm 1: Betweenness centrality in unweighted graphs

```
C_B[v] ← 0, v ∈ V;
for s ∈ V do
    S ← empty stack;
    P[w] ← empty list, w ∈ V;
    σ[t] ← 0, t ∈ V;   σ[s] ← 1;
    d[t] ← −1, t ∈ V;   d[s] ← 0;
    Q ← empty queue;
    enqueue s → Q;
    while Q not empty do
        dequeue v ← Q;
        push v → S;
        foreach neighbor w of v do
            // w found for the first time?
            if d[w] < 0 then
                enqueue w → Q;
                d[w] ← d[v] + 1;
            end
            // shortest path to w via v?
            if d[w] = d[v] + 1 then
                σ[w] ← σ[w] + σ[v];
                append v → P[w];
            end
        end
    end
    δ[v] ← 0, v ∈ V;
    // S returns vertices in order of non-increasing distance from s
    while S not empty do
        pop w ← S;
        for v ∈ P[w] do δ[v] ← δ[v] + σ[v]/σ[w] · (1 + δ[w]);
        if w ≠ s then C_B[w] ← C_B[w] + δ[w];
    end
end
```

Brandes' algorithm is computationally expensive in terms of runtime and space complexity given a large input graph. For every vertex in the graph, every other vertex must be visited, and space must be allocated to record the traversals, which is used in the backwards pass. However, the algorithm can benefit from parallelization in multiple areas. In the forward pass, running BFS from each source node can be done independently. In a BFS, we iteratively travel to the unexplored neighbors of already visited nodes, visiting "frontiers" of nodes at progressively increasing distance from a source node. The frontier explored in an iteration depends on the vertices visited in the previous iteration, but processing the nodes within the frontier of each iteration is parallelizable. We subsequently refer to these parallelization strategies as "across" and "within" BFS. In the backward pass, updating nodes that are closer to the source depends on data from nodes that are further away which must be updated earlier. However, nodes that are the same distance are independent from each other and thus can be processed in parallel.

APPROACH

We implemented our parallel brandes' algorithm from scratch in C++ using the OpenMP shared-memory API. We primarily targeted the 8-core GHC machines before deploying profiling runs on the 128-core PSC machines. While our initial implementation relied on OpenMP's high-level lock API, our final implementation utilizes low-level GCC atomic built-ins (__sync_bool_compare_and_swap) and prefix-sums to minimize synchronization stalls.

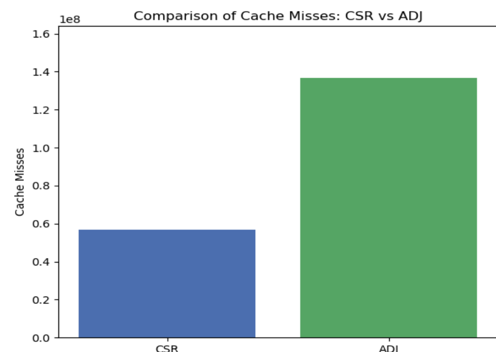
Computing Betweenness Centrality via Brandes' algorithm requires two distinct phases: a forward-pass Breadth-First Search to compute shortest path counts, and a backward pass to accumulate dependencies. We explored two distinct thread mappings to parallelize this process:

- Across-BFS (source-parallel/coarse-grained): We mapped the hardware threads to the outermost source-node loop using #pragma omp for schedule(dynamic). Each thread is assigned a unique source node and executes a complete, independent serial BFS and backward pass using thread-local memory arrays.
- Within-BFS (level-synchronous/fine-grained): To handle massive graphs and avoid duplicating the local memory arrays across threads, we mapped the hardware threads to the inner BFS frontier. All threads cooperate to process a single source node's BFS, strictly synchronizing at implicit and explicit barriers at the end of every BFS depth level. They then cooperate again to process the backward pass, processing each level one at a time to properly accumulate the dependencies.

The across-BFS implementation was relatively straightforward and didn't require much iteration because this approach is embarrassingly parallel. Since each BFS runs independently and modifies thread-local metadata, the only place where we needed synchronization was in updating the final betweenness centrality scores, since that is shared across threads. We made two optimizations to the code here to help reduce the memory bottleneck, which we also applied to the level-synchronous implementation.

The first change we made was to address memory allocation. Our initial Source-Parallel approach declared the necessary BFS arrays (sigma, d, P, delta) inside the per-source BFS loop. This forced all the threads to constantly request the operating system to dynamically allocate and destroy massive vectors. We hoisted all vector instantiations outside the #pragma omp for loop so they are allocated exactly once per thread. Inside the loop, threads simply reuse the memory, significantly reducing the memory overhead of running the algorithm.

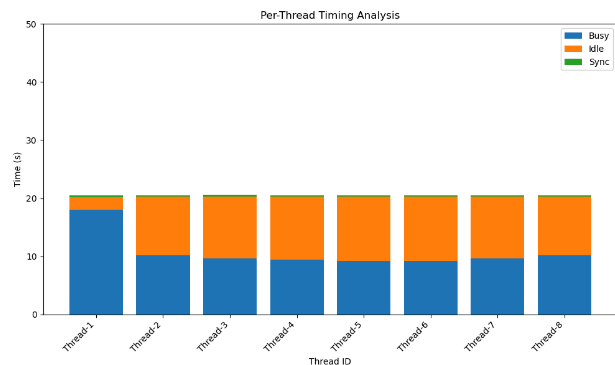
Even with reduced memory allocations, we still thought we could improve the memory bandwidth via the graph representation. To solve this, we converted the graph representation into the Compressed Sparse Row (CSR) format. By packing all graph edges into a single, contiguous 1D array, we greatly improve our spatial locality when iterating through the neighbors of a vertex. We tested each representation on a medium-sized test file on 1 thread and got the following results:

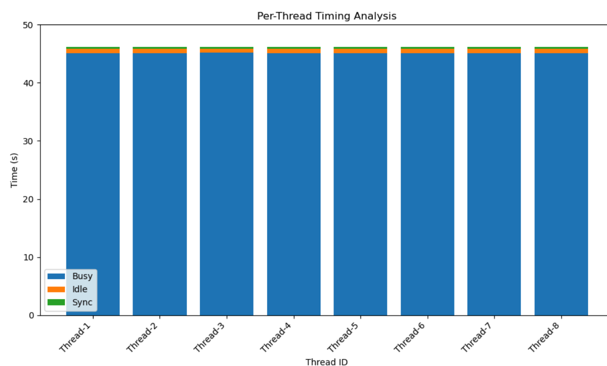


The within-BFS implementation went through several major iterations to achieve the final architecture, as it is much harder to parallelize:

To prevent race conditions when multiple threads discover the same unvisited neighbor in the forward pass, our initial implementation allocated an array of n individual locks (one for each vertex in the graph). Threads had to acquire a node's specific lock before claiming it for the next frontier and processing it. As expected from a naive implementation, it resulted in a large amount of locking overhead. We discarded this approach and replaced it with a lock-free compare-and-swap protocol using the hardware atomics __sync_bool_compare_and_swap() to ensure that multiple threads didn't add the same node to the frontier and __sync_fetch_and_add to ensure that we didn't have any overlapping updates to the predecessor list.

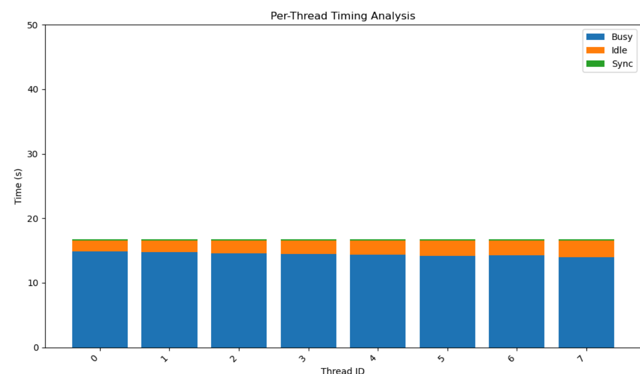
We then investigated if our implementation had any load imbalance. We added per-thread timing code to determine the busy, wait, and sync times and tested a few different schedules. We found that a standard static schedule had quite a poor load balance, with thread 1 taking far longer than the other threads:





We believe this imbalance occurred because thread 1 just so happened to grab a large batch of expensive "hub" nodes, resulting in a much longer computation time than the other threads. We thought we could resolve this load imbalance by switching to a dynamic schedule, and while it fixed the load imbalance, the overhead from managing the work queues and task stealing resulted in a far longer execution time as shown on the left:

We tested a few other scheduling patterns by changing the block sizes, and it turned out that doing an interleaved static schedule gave us the fastest times. It was very well balanced when looking across the entire runtime. We believe this is because large "hub" nodes are very likely to cluster together, with one hub node having many others as neighbors. The default static assignment is blocked, meaning that when processing a hub node, all of its



neighbor hubs would be placed together on the frontier, forcing one thread to process all the hubs on the next frontier. By doing an interleaved assignment on the frontier, these hubs would get distributed evenly between the threads when processing the next frontier. This hypothesis is strengthened by the fact that we still see some idle time in the interleaved assignment compared to the dynamic assignment. This is because each individual frontier is still a little unbalanced since some of the nodes process hubs, but this is balanced out throughout the course of the algorithm since the interleaved assignment will effectively randomize which threads get the hub nodes.

At this point, we still had a slowdown on many of our test graphs. With the load balancing problem mostly solved, we looked at what we could do to reduce the atomic contention of our code. Brandes' original algorithm requires us to have 3 sets of atomics in our forward pass: a compare-and-swap for the frontier, one to update the number of shortest paths, and a second compare-and-swap to build the predecessor array for the backward pass. We realized that we could obtain the same information necessary for the backward pass using a successor array instead of a predecessor one. This meant that during the BFS, instead of updating the predecessor array of other nodes, we only had to update our own node's successor array, which requires no atomics, because each thread “owns” the node that it's processing during the BFS. This greatly reduced the amount of atomic contention in our forward pass and finally gave us positive speedup on various graphs.

The final iteration we made was to address the bottleneck that occurs when merging thread-local BFS frontiers into the global next_frontier. It was initially written using #pragma omp critical sections to insert the local frontier into the global one. We changed the implementation to use a prefix-sum. Threads record their local queue sizes in a shared thread_offsets array, synchronize at a barrier, and use a #pragma omp single block to compute global array offsets. Threads then all copy their local data directly into their guaranteed block of the global flat array. Since the thread size we're working with is relatively small, doing a sequential prefix sum took less overall time than doing a parallel prefix sum. While this does require a bit of extra synchronization at the end, the impact is not very large since the threads needed to synchronize at each level of the BFS anyways, and the benefit of reducing the contention on the global frontier array outweighed the downside of the extra work from the sequential scan.

RESULTS

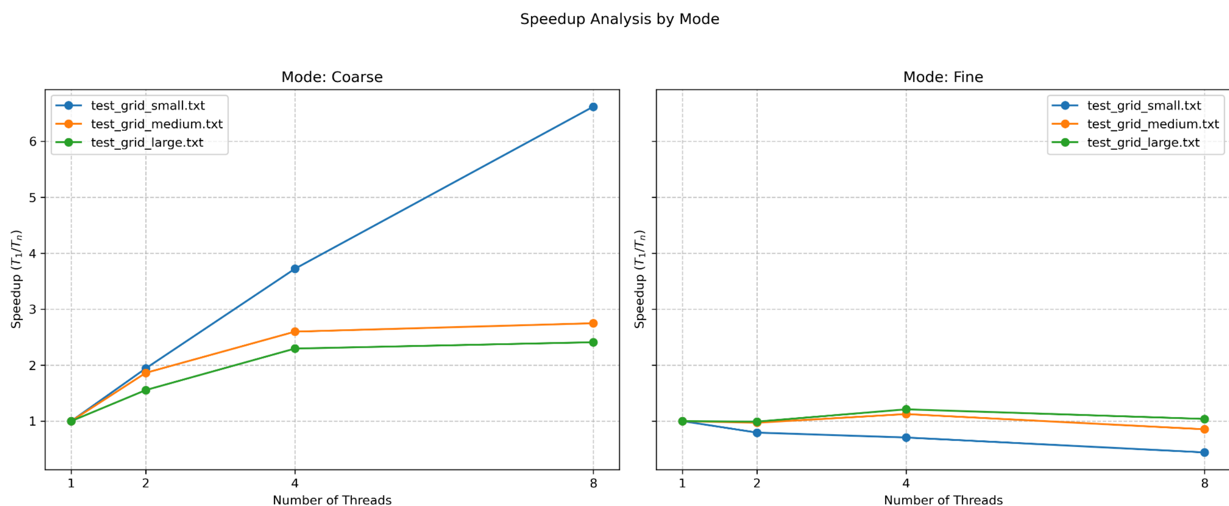
We evaluated our implementations using wall-clock execution time to determine the speedup against an optimized parallel implementation for a single thread. To test the architectural trade-offs of our parallel engines, we benchmarked our code against two distinct classes of graphs: high-diameter networks (such as road networks, which feature deep, narrow BFS trees) and social networks (which feature shallow BFS trees but massive power-law "hub" nodes). We tested three graphs from each category of varying sizes as follows:

High-diameter test graphs:	Number of nodes	Number of edges
test_grid_small	11461	25328
test_grid_medium	52359	133382
test_grid_large	98163	218998

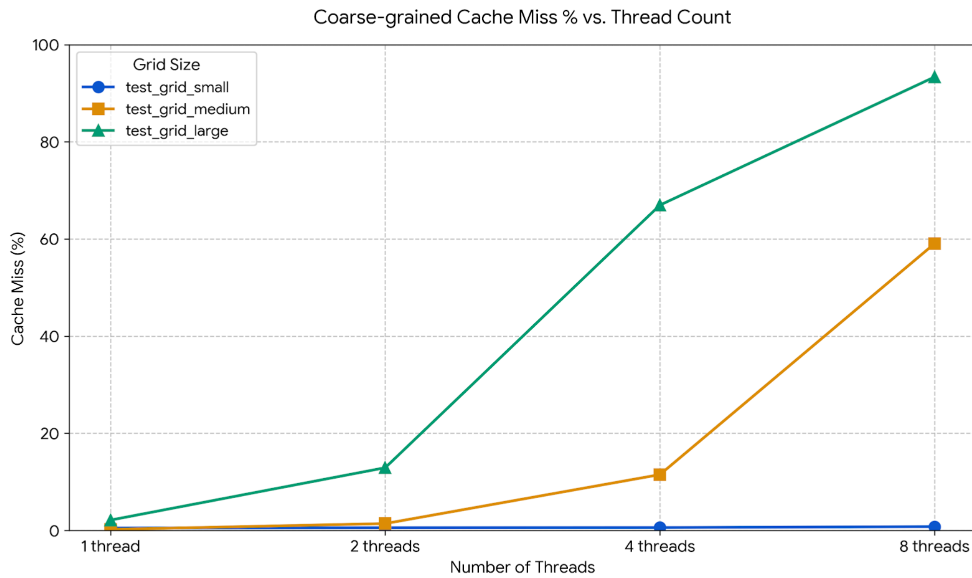
Scale-free test graphs:	Number of nodes	Number of edges
wiki-Vote (sanitized)	7115	103689
email-Enron	36692	367662
ego-Twitter (sanitized)	81306	1768135

We couldn't find a wide range of sizes for road networks, so we decided to randomly generate the data for the high-diameter test graphs. They were generated using a python script by starting with a 2d grid where each node is connected to its neighbors and then randomly removing a certain percentage of vertices and edges. The scale-free test graphs were found on the Stanford Large Network Dataset Collection (SNAP), and then sanitized using a python script to normalize the vertex numbering and remove duplicate edges.

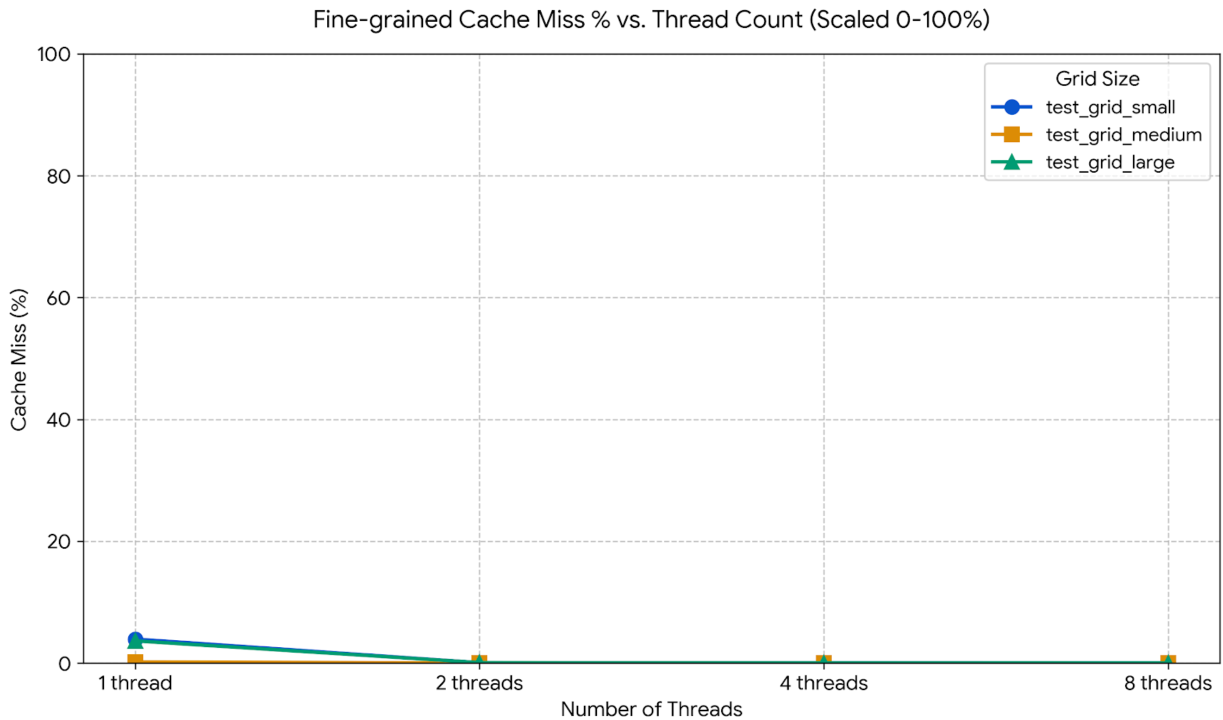
Below are the speedup graphs for each of our implementations run on the high-diameter graphs when run on the GHC machines at 1, 2, 4, and 8 threads:



The coarse-grained (source-parallel) implementation scales extremely well on the tiny graph while performing poorly on the larger graphs. We believed that this was due to the graph and each thread's O(V) thread-local vectors exceeding the L3 cache capacity, resulting in the implementation thrashing DRAM. We confirmed this by running perf stat and recording the percentage of L3 cache misses at each thread count. This gave us the following data:



We also profiled the L3 cache misses from our fine-grained (level-synchronous) implementation, which shouldn't have this issue because it doesn't copy the per-node metadata that Brandes' requires.



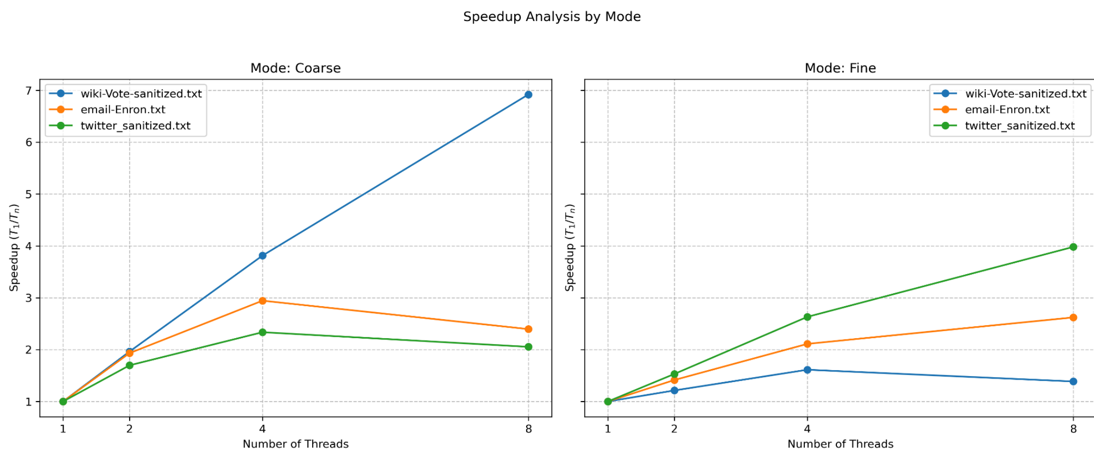
This practically confirms our hypothesis that the slowdown at higher thread counts is the result of exceeding the L3 cache capacity, as the extra memory needed due to the larger graphs in combination with extra threads copying extra data generate a large amount of L3 cache misses.

The speedup for the fine-grained (level-synchronous) code is poor due to the nature of the test data itself. When BFS trees are extremely deep like in high-diameter graphs, they require many more levels to compute, which greatly increases the synchronization time that occurs at the end of each level. Additionally, the level-synchronous implementation requires a wide frontier to efficiently parallelize the computation, or else there is insufficient computation for the threads to do, making them just sit idle. We confirmed this by checking what percentage of the frontiers in the BFS are strictly smaller than the following thresholds:

Frontier thresholds	<2	<4	<8	<16	<32
Small	0.8308%	2.3537%	8.2914%	26.5108%	66.7937%
Medium	0.4432%	1.6828%	5.4001%	9.5064%	15.7358%
Large	0.9172%	2.4601%	4.9650%	9.9208%	20.4314%

This is consistent with the small graph showing the worst slowdown, while the medium and large grids show a nearly constant runtime.

Below are the speedup graphs for each of our implementations run on the social network graphs when run on the GHC machines at 1, 2, 4, and 8 threads:



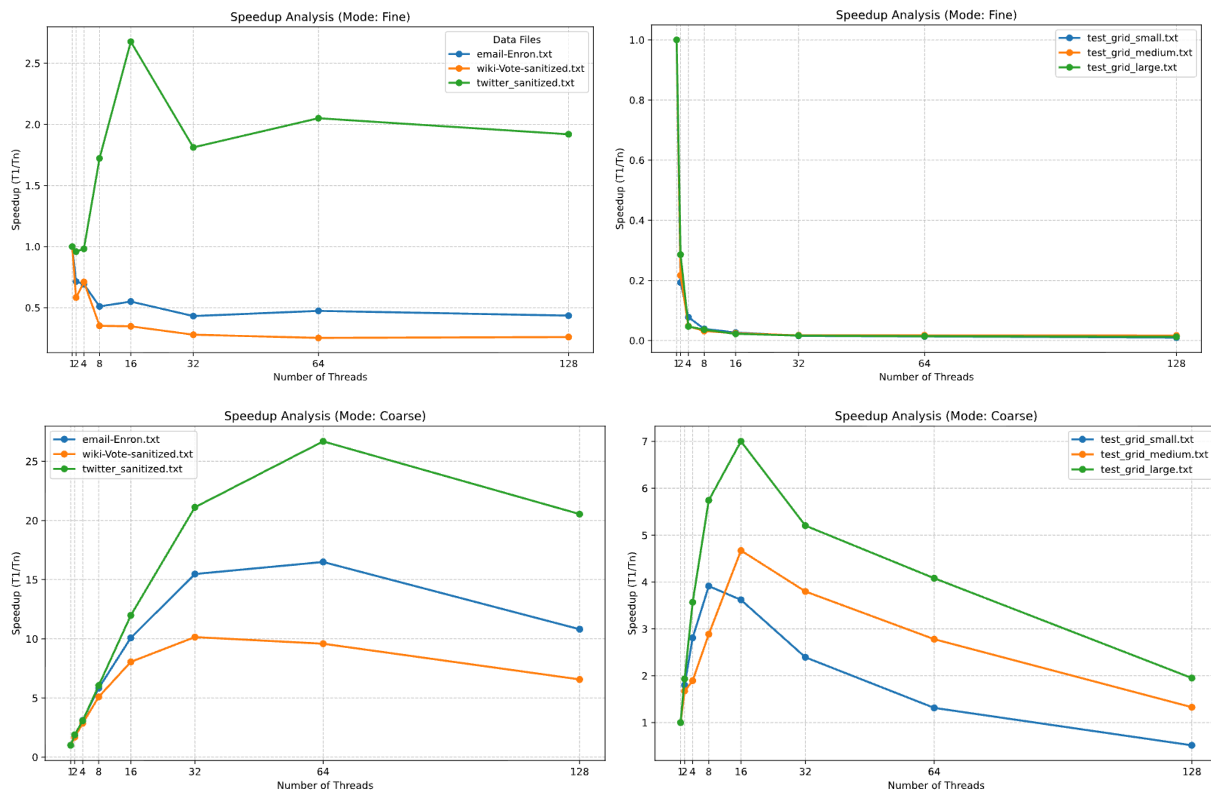
Similarly to the testing on high-diameter graphs, the source-parallel implementation performs well on the smallest test case, but performs poorly as the test case grows in size. The reason is the same, with the larger test cases and thread counts resulting in L3 cache misses.

The opposite is true for the level-synchronous implementation: It performs better on larger graphs. The same reasons apply from the high-diameter test data: the frontiers on large social networks get extremely large very quickly, giving us many nodes to parallelize over within the frontier, while also keeping the BFS depth very shallow, reducing the number of synchronization steps we need to do at the end of each level.

We can see the effects of the frontier being too small in the wiki-Vote test data, where we see the speedup drop off at 8 threads. Running the same frontier size profiling again on this gives us that 41% of frontiers are less than 8 large. This severely limits the amount of speedup we can achieve because the frontiers are not large enough for all the threads to have enough work. In contrast to the high-diameter graphs however, this is a significantly lower percentage of frontiers, and yet we don't see a negative speedup. This is because our BFS depth is still relatively small, with the algorithm needing to process over 3 times more frontiers for test_grid_small than for wiki-Vote, and thus needing 3 times more total synchronization over the course of the algorithm. As a result, we see our speedup drop in the wiki-Vote test case, but not to the levels of the high-diameter graphs.

By comparing the social network and spatial grid results, we can definitively characterize our system's boundaries: Source-Parallel is universally superior across all graph topologies, provided the dataset is small enough that T copies of the O(V) state arrays do not exceed the machine's memory bandwidth. This makes sense because source-parallelism is embarrassingly parallel in the first place. Level-Synchronous is much more sensitive to graph topology. While it is necessary for graphs that are too large for source-parallelism it is only viable on low-diameter graphs with massive frontiers like social networks. On high-diameter graphs, the barrier synchronization required between levels eclipses the parallel compute gains.

Profiling results on PSC machines



We noticed that our within-BFS implementation (mode: fine) consistently suffered from negative speedup at increasing thread counts on the PSC machines regardless for both the high-diameter graphs and the scale-free graphs. This trend was not seen when running the across-BS implementation (mode: coarse) on the PSC machines, and neither implementation behaved this way when run on the Gates cluster machines. We investigated this using perf stat and found that even on test_grid_small, the stalled-cycles-backend metric was extremely high, reaching 90% of backend cycles spent idling on 64 threads, and the cache miss rate reached 80% of all cache references. Even on 1 thread, the stalled-cycles-backend metric was quite high at 36%, and the cache miss rate was over 10 times higher than it was on the GHC machines. This data was incredibly inconsistent with the GHC machines, so we suspected that differences in the architecture between the PSC and Gates cluster machines could be the culprit for the high background idling time.

On the PSC Bridges-2 Regular Memory (RM) partition machines, each node's CPU is two AMD EPYC 7742 64-core processors. Unlike with the GHC CPUs wherein all cores share a single L3 cache, a PSC node has a NUMA architecture the L3 cache is divided among 4-core CPU clusters called Core Complexes (CCXs). Each cluster has a shared 16 MB size L3 cache, and the CCXs are linked with one another by an interconnect.

In the within-BFS implementation, all threads read and write to the same data structures: the BFS frontier queue, the distances array, the shortest path counts array, and the delta pairwise dependencies array. This true sharing also results in large amounts of cache invalidations as the different clusters compete to get exclusive write access to the shared arrays. It is extremely likely that the cache line holding the requested node data would be located in a separate CCX, which is the extremely high miss rate we observed. Retrieval of that data, involving sending the data through an interconnect would be responsible for the poor runtime on the PSC machines.

Conversely with the across-BFS implementation, all the threads only read from the shared graph data, but they read and write to their own thread-local copies of the data structures. This meant that the thread-local data specifically could be cached in the local CCX's L3 cache, resulting in comparatively better performance since a higher proportion of memory accesses don't need to pull from DRAM or across the interconnect, which resulted in the relatively higher speedup compared to the within-BFS implementation's performance on the GHC machines.

REFERENCES

AMD EPYC 7742 Specs. (n.d.). TechPowerUp. https://www.techpowerup.com/cpu-specs/epyc-7742.c2245

Brandes, U. (2001). *A faster algorithm for betweenness centrality* . The Journal of Mathematical Sociology, 25(2), 163–177. https://doi.org/10.1080/0022250X.2001.9990249

Intel® CoreTM i7-9700 Processor (12M Cache, up to 4.70 GHz) - Product Specifications. (n.d.). Intel. https://www.intel.com/content/www/us/en/products/sku/191792/intel-core-i79700-processor-12m-cache-up-to-4-70-ghz/specifications.html

Stanford Large Network Dataset Collection (SNAP):

```
@misc{snapnets,
  author       = {Jure Leskovec and Andrej Krevl},
  title        = {{SNAP Datasets}: {Stanford} Large Network Dataset Collection},
  howpublished = {\url{http://snap.stanford.edu/data}},
  month        = jun,
  year         = 2014
}
```


WORK DISTRIBUTION

Task	Jerry	Julie
Initial serial implementation		Julie
CSR graph representation	Jerry	
Initial coarse-grained implementation		Julie
Initial fine-grained implementation	Jerry	
Optimizing coarse-grained	Both	Both
Optimizing fine-grained	Both	Both
GHC testing & data	Jerry	
PSC testing & data		Julie
Report writeup	Jerry	Julie
Total:	50%	50%