Parallel Betweenness Centrality via Brandes' Algorithm

Team Members: Jerry Cheng, Julie Wu

URL: https://jwjulie.github.io/15418-betweenness-centrality/project-proposal

SUMMARY

We will implement and optimize a parallel version of Brandes' Algorithm for computing Betweenness Centrality on large-scale graphs. We plan to develop a multi-core CPU implementation using OpenMP and a many-core GPU implementation using CUDA to analyze how different graph topologies (e.g., social networks vs. road networks) affect parallel scaling and memory bottlenecking.

BACKGROUND

Betweenness Centrality (BC) is a metric used in network science to identify influential nodes by calculating the fraction of all-pairs shortest paths (APSP) that pass through a specific vertex. Vertices with high BC scores are reachable from many other vertices on relatively short paths. In a 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 Brandes' Algorithm 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. $\delta_s(v)$ can be computed dynamically by

working from the leaves of a shortest paths tree rooted at a source s with the following recursive relation:

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

It iterates over all vertices one at a time, and requires two distinct phases. For a source vertex s:

1. Forward Pass: A single-source shortest path algorithm (SSSP) 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 on each intermediate vertices $\delta_s(v)$ 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
```

Parallelism Opportunities:

- Coarse-grained: Each root node's BFS is independent and can be run separately.
- Fine-grained: Multiple cores can execute a single BFS in parallel along its frontier.
- Level-synchronization: During the backward pass, nodes that are further away depend on nodes that are closer. However, nodes that are the same distance are independent from each other and thus can be processed in parallel.

THE CHALLENGE

Parallelizing Brandes' Algorithm is challenging due to several factors:

- Workload Imbalance: Real-world graphs often follow a Power-Law distribution. A high-degree node with thousands of neighbors requires far more compute time than isolated leaf nodes, leading to workload imbalance in naive thread partitioning schemes.
- Memory Access Characteristics: Graph traversals typically require a lot of pointer chasing and irregular memory accesses. This results in poor cache locality and high memory latency. We will need to utilize a more cache-friendly representation like compressed sparse row (CSR).
- Memory constraints on large graphs: On very large graphs, many threads with their own metadata (distances, path counts, dependencies, etc.) can shift the problem from compute-bound to memory-bound if we aren't careful.
- Synchronization: During the forward pass, multiple threads discovering the same neighbor node simultaneously create high lock contention when updating the frontier.

RESOURCES

- Hardware: We will use the GHC Clusters and PSC machines (for multi-core CPU/OpenMP testing).
- Codebase: We are starting from scratch, though we will reference the sequential implementation logic from the original 2001 Brandes paper.
- An experimental evaluation demonstrating speedup on different graph topologies like Roads vs. Social Networks.
- 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

GOALS AND DELIVERABLES

PLAN TO ACHIEVE (Success)

- A working sequential baseline for Brandes' algorithm.
- A working multi-core OpenMP implementation
    - Coarse-grained version where multiple BFSs are assigned to different cores
    - Fine-grained version where multiple cores operate on the same BFS

HOPE TO ACHIEVE (Reach Goals)

- Implementation on weighted graphs using Dijkstra's for the SSSP algorithm on the forward pass
- CUDA-based GPU implementation and comparison with CPU implementation
- Comparing different update methods for the backward pass

PLATFORM CHOICE

We chose C++/OpenMP (shared address space model) as our primary platform because it provides the most effective tools for the bottlenecks found in Brandes' algorithm. Its built-in dynamic scheduling helps us deal with the irregular workloads found in many graphs. It also provides us with efficient primitives to handle shared state between threads.

SCHEDULE

Week	Task
Mar 26 - Apr 2	Implement a sequential baseline and draft CSR representation.
Apr 3 - Apr 9	Implement coarse-grained OpenMP implementation. Test initial speedup on GHC machines.
Apr 10 - Apr 16	Implement fine-grained OpenMP implementation. Test initial speedup on GHC machines. Intermediate Milestone Due (Apr 15).
Apr 17 - Apr 23	Work on improving OpenMP implementation to improve speedup. Also work on CUDA implementation if we have time.
Apr 24 - Apr 30	Collect data from PSC machines. Write the final report. Create the presentation poster.