15-418 Final Project

Soren Dupont - sdupont
Aidan Lincke - alincke

April 29, 2025

Contents

1 Summary

We developed a parallelized pipeline for solving the single-source shortest paths (SSSP) problem, achieving significant speedups over sequential methods. By introducing parallelism in both the pre-processing phase (graph contraction) and the path-finding phase, we substantially reduced the overall time to compute shortest paths.

2 Background

Our algorithm works in two phases: a pre-processing phase and a path-finding phase.

2.1 Pre-Processing Phase: Contractions

The goal of the pre-processing phase is to simplify the graph into a "skeleton" or overlay graph so that 'unimportant' nodes (dead ends, nodes that are only really important in connecting two other nodes, and are thus best to be combined with a specific path) don't have to be visited when we service individual queries asking for the shortest path between two nodes. The key feature behind actual road networks that makes this a good idea is that they are highly hierarchical. When we want to know how to get from street A of city 1 to street B of city 2, we can think of our task as being to traverse across "lower importance" roads from street A until we get to the highway, then travel to city 2 via the highway, then take "lower importance" roads to street B. This is essentially what happens in contraction hierarchies. The overlay graph that we produce by contracting away nodes can be thought of as a highway that we build (or isolate) for our graph.

The point of this preprocessing step is to drastically increase the effectiveness of Dijkstra's algorithm in handling these queries by reducing the size of the graph that the algorithm has to check.

- What are the key data structures? What are the key operations on these data structures?

  We represent an input graph (for example, a graph representing all the road intersections and road segments (and their weights) in Pittsburgh) by an adjacency list graph representation; more specifically the C++ type used to represent these "Simple Graphs" is a vector of unordered maps mapping from a vertex's neighbor to Edge, where Edges are structs just containing the weight of the edge (i.e. the length of the road segment, or the cost of traveling along this edge). Thus the type `SimpleGraph` is just `std::vector<std::unordered_map<int, Edge>>`

  In order to contract a graph we use a `ContractedGraph` data structure that has the set of which nodes are active (as nodes get contracted away, they are removed from the set) and a vector of unordered maps from `int`s to `ShortCutEdge`s. Thus the `ContractedGraph` struct contains a `std::unordered_set<int>` and a `std::vector<std::unordered_map<int, ShortCutEdge>>`. Unlike in `SimpleGraph`s, `ShortCutEdge`s contain a path (linked list) that one can take (along nodes from the original graph that have been contracted away) to go from one active node to its 'neighbor' (these active nodes may not be neighbors in the original graph, but in the contracted graph, they are connected by a shortcut), and the total weight of this path.

- What are the algorithm's inputs and outputs?

  The input to the algorithm is an undirected (positive) weighted graph representing the road network of a city, and the output is a contracted version of this graph representing the network of "important" intersections in the road network.

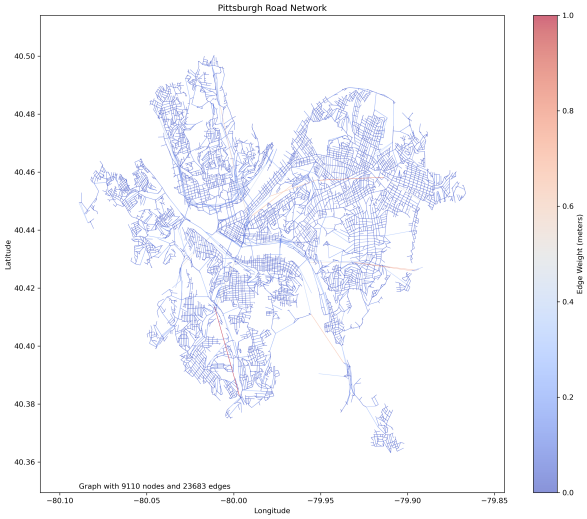


Figure 1: The edges-only Pittsburgh roadmap data before it has been contracted.

- What is the part that is computationally expensive and could benefit from parallelization?

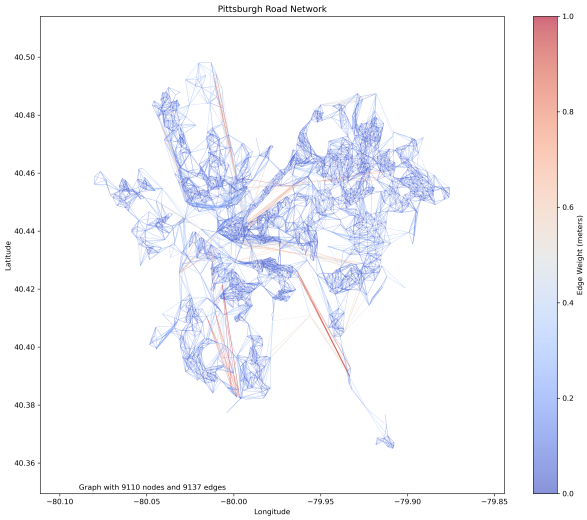


Figure 2: The edges-only Pittsburgh roadmap data after a computationally light contraction has been applied. Many extra node and edge clusters that will make path finding difficult have been left uncontracted.

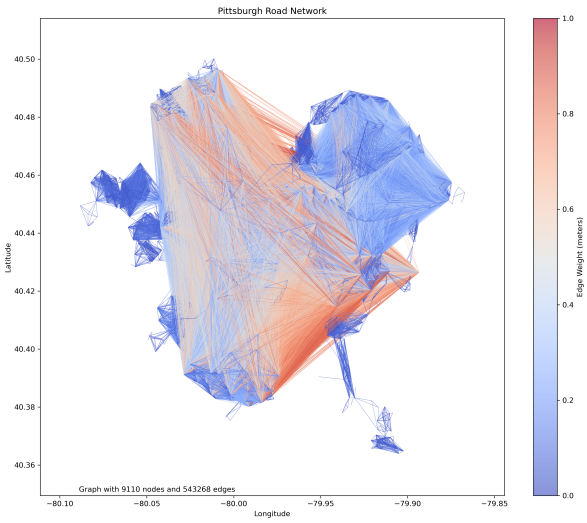


Figure 3: The edges-only Pittsburgh roadmap after an expensive contraction policy has been applied (randomly contracting 75% of the nodes). High opportunity for parallelism.

One of the main difficulties of the contraction hierarchies algorithm is in picking a good heuristic for what nodes we should consider "unimportant" (and contract first). The different proxies that we looked at for "importance of a node" led to different workloads throughout the program, and thus different parallelization approaches being necessary between the different heurisitcs for node importance.

For the most part, the computationally expensive part of contractions comes from the fact that when a node is contracted, each path that one could have taken through the node from one of its neighbors to another has to be considered and added as a shortcut (if this path is sufficiently cheap) to the contracted graph. As we contract more and more nodes, the amount of edges that each node remaining is connected to increases, meaning this part of contractions gets more and more expensive as more pairs of neighbors of a node we want to contract need to be considered.

Contracting away the great bulk of a graphs nodes itself is also of course an expensive part of the algorithm, as it requires looping through the vertices of the graph, but as the graph contraction gets more and more condensed, vertices are connected with more and more graphs nodes, and it becomes harder to find independent work (independent nodes) that can be done. Thus we saw greater success parallelizing within contractions than across.

In one of our heuristics for importance, we considered how many neighbors a given node has. If a neighbor has say 4 or less neighbors, we consider it "unimportant" and thus contract it. This leads to a very light workload in the contraction stage because if a node has many neighbors that means that it will take a while to contract, but it also means that we deem it "too important" to contract. Parallelism is only likely to be beneficial in this case if it is does across node contractions, as very few paths need to be tested within a single node contraction.

- Break down of workload. Where are the dependencies in the program? How much parallelism is there? Is it data-parallel? Where is the locality? Is it amenable to SIMD execution?

  Because this algorithm does not deal with highly data-parallel computations, but rather unpredictable graph accesses, it does not lend itself well to SIMD execution.

  There is major opportunity for parallelism when a node is contracted because independent paths connecting neighbors of the node being contracted need to be checked and added. There are some small dependencies that arise from this, and updates to edge lists need to be done safely, but overall this represents a major opportunity for parallelism.

  For parallelism across node contractions (rather than within, as discussed above), the dependencies in the program come from the fact that as you are contracting nodes, you are modifying the shared graph. Attempting to contract away two nodes that are neighbors or that share a neighbor could create inconsistencies in the graph, such as nodes having neighbors that have actually

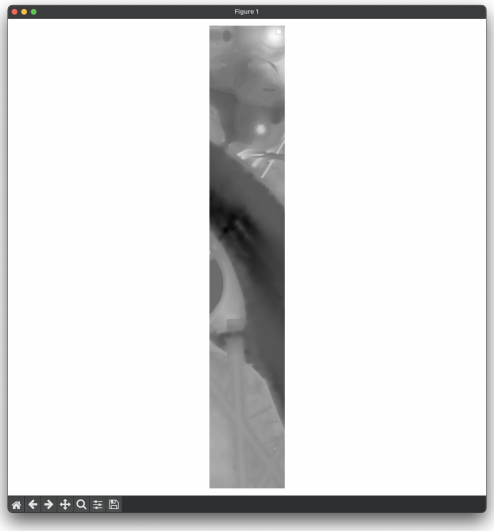
been contracted away. If a graph remains fairly spread out, and degrees of vertices are fairly low, transactional memory could be an effective approach to utilizing this kind of parallelism as the odds of a transaction (node contraction) aborting in this case would be fairly low; nodes would be unlikely to neighbor eachother or share neighbors resulting in aborts.

2.2 Path-Finding Phase: Delta-Stepping

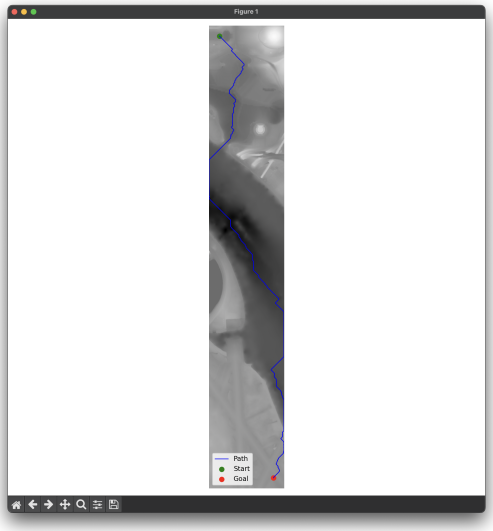
In the next phase of the algorithm, we use the contracted graph to find optimal paths. To achieve this, we opted to use the delta-stepping algorithm, rather than Dijkstra's or another path-finding algorithm, because it lends itself particularly well to parallelization. In delta-stepping, edges are categorized into light edges and heavy edges based on a threshold value Δ, hence the name "delta-stepping." Light edges have a weight less than Δ, while heavy edges have weights greater than Δ. The concept of buckets is also important in delta-stepping: buckets group vertices based on their tentative distances, with lighter edges contributing to lower-indexed buckets and heavier edges contributing to higher-indexed buckets.

The algorithm proceeds by iterating through buckets in order of their index. Within each bucket, the distinction between light and heavy edges enables parallelism: light edges, which may require multiple relaxations, can be processed concurrently. Heavy edges are also processed in parallel but often move nodes to future buckets due to their larger weights. Thus, the Δ parameter plays a crucial role in tuning the algorithm. Although a larger Δ might initially appear to unlock more parallelism, it can also lead to unnecessary work and redundant relaxations. Therefore, a carefully balanced Δ is necessary to maximize performance.

- What are the key data structures? What are the key operations on these data structures?
  - `S`, `R`, and `nextR` are `tbb::concurrent_vector<Position>`'s containing the light and heavy edges, respectively, of the current bucket. While exploring edges in parallel, additional edges can be added to `S` or `nextR`, so race conditions must be avoided.
  - `prev` and `dist` are both `tbb::concurrent_unordered_map<Position, ...>`. They store the least distance encountered so far from the starting node and the previous node that led to that minimum distance. These fields are frequently updated across different threads (for example, whenever a new best path is found), so race conditions must be avoided.
- What are the algorithm's inputs and outputs?
  - The delta-stepping algorithm takes in a graph as input and a starting point. It outputs the `dist` and `prev` fields to thus solve the SSSP problem. Using the `dist`, `prev`, and starting point fields, the minimum-cost path can be determined by simply following the `prev` node from the goal until reaching the start.



(a) Inputs to the delta-stepping algorithm (elevation data of the downtown Washington, DC area).



(b) An example best-cost path from the starting point of the Jefferson Memorial to the DCA airport.

Figure 4: Inputs and outputs to the delta-stepping algorithm. With close inspection, you can see the runways of the DCA airport at the bottom-left, the circular Jefferson memorial at the top-left, and the Potomac River cutting through the middle. Our path minimizes the cost of elevation and thus follows the bottom of the Potomac River.

- What is the part that computationally expensive and could benefit from parallelization?
  - The process of delta-stepping is equally computationally expensive throughout its length because it performs the same operations of exploring the graph. Processing light edges and processing heavy edges can both be parallelized, so they both benefit. The processing of light and heavy edges consitutes the vast majority of the algorithm, so the algorithm generally benefits as a whole.
- Break down the workload. Where are the dependencies in the program? How much parallelism is there? Is it data-parallel? Where is the locality? Is it amenable to SIMD execution?
  - As described, most of the workload can be parallelized through the parallelization of light and heavy edges. However, the challenges arise due to the plethora of variables that are shared among the threads. Additional challenges arise from the frequent times that the threads must re-sync with each other before continuing on to further buckets or heavy edges.
  - Yes, the algorithm is data-parallel. With OpenMP, the loops of edges are split among threads, and the edges are the input provided to the function.
  - Locality is difficult to exploit in these graph problems because edges may travel to anywhere else in the graph. These edges often travel to far-away sections that aren't in cache.
  - The program isn't amenable to SIMD execution due to the locality issue described above. Due to the lack of locality inherently associated with graphs of this nature, certain loads from far-away sections of the graph could take a long time while other sections may already be present in the cache. Requiring SIMD lanes to wait for a specific lane requiring accesses to a far-away section would be wasteful and reduce speedup.

3 Approach

To create graphs, we used Python because it lends itself to simple programs that can be quickly developed. We prioritized speed of development for graph creation because we wanted to spend the majority of our time on parallelization. For the delta-stepping algorithm explorations, we downloaded a file of the Washington, DC area's elevation data from the USGS to use as a cost grid, with the idea of minimizing elevation while solving the SSSP problem. However, we wanted to incorporate graph contractions into our project, but a simple 8-connected grid was too simplistic for contractions to help in any meaningful way due to the hierarchical structure of the graphs that they are applied on. Thus, for our exploration of contractions, we switched to using road map data of the Pittsburgh, PA area because these roads would create nodes with more complex edges that could vary in distance. The graph we wound up using to test contraction hierarchies on had roughly 9,000 nodes and 23,000 edges.

We used C++ on the GHC machines for our parallelization work. We chose this setup because our parallelism was well-suited to a shared-memory model, which allowed efficient use of OpenMP on a single machine. We've also become familiar with C++ and OpenMP running on the GHC machines from previous assignments. In addition, we used Intel's Thread Buildling Blocks (TBB) API. This helpful library provides lock-free implementations of concurrent data structures, such as `tbb::concurrent_vector`.

3.1 Graph Contractions Approach

The way we contract a single node `n` is we loop through each of its neighbors `v`, and for each `v`, we loop through every other neighbor of `n`, `w`, and update edge `vw` if it is cheaper to go from `v` to `w` via `n`. Thus we have to update `v`'s map from its neighbors to the corresponding edge out of `v`, and update `w`'s map from its neighbors to the corresponding edge out of `w`.

As stated before, this creates a major opportunity for parallelism because in this step, we only have to make sure that we're not unsafely updating the same unordered map from different processes. This would be even easier if the graph we were working with was directed, meaning only 1 map would need to be updated for a given value of `v`. Then we could just distribute neighbors of `n` to different processes in the outer loop, and each process would only have to make updates to it's own map. This is one of the approaches that we tried for exploiting this kind of parallelism. I refer to this as the "Directed Graph" approach (as here we are essentially treating the graph as if it were directed, and therefore you decouple the part of the program where it updates both the edges `vw` and `wv` into being done separately).

Another way to exploit this kind of parallelism is simply using thread safe unordered_maps to represent each node's list of incoming / outgoing neighbors. We used Intel's TBB library to implement this approach as well.

For this form of parallelism, the same graph will be produced deterministically based on the contraction policy that is used, thus the only relevant performance gain metric for this section is speedup (quality of result, etc. does not change as a result of using different number of threads).

Note that dynamic scheduling was preferred for these techniques (we saw about a 10% better performance for dynamic vs. static scheduling in these cases). In the case of using the thread-safe unordered_maps, this is likely because neighbors that occur earlier in the outer for-loop have all the remaining neighbors to check, while neighbors that occur later have fewer (later occurring) neighbors to check. Though, the distribution of work in the "Directed Graph" approach should be essentially uniform, we still found a noticeable advantage for using dynamic rather than static scheduling.

We also wrote a function to verify that the graphs that we are generating are indeed valid contractions, and that no data races or corruptions have occurred.

We also tried various policies for which nodes will be contracted. As alluded to before, one approach is to only contract nodes with a limited number of neighbors. This is a fairly reasonable proxy for how important a node is, and thus a more intuitive heuristic for which nodes we should be contracting, and this results in very light workloads for generating contracted graphs (as we have found, the reduced graphs from this policy may have just as few or fewer nodes than with other more expensive policies) as well as less opportunity for speedup via parallelism. Thus it was also useful for us to test less intelligent policies, such as simply statically choosing which nodes will be contracted away. With this policy, execution time is much slower, because as contractions continue, each node accumulates more and more edges that need to be considered when contracting future nodes. By parallelizing these

approaches, we develop a parallelization technique that will work for whichever policy is chosen as it does not rely on usable speed simply from the policy chosen being computationally light.

Because we still wanted to see some speedup in this relatively computationally light case, where there is little parallel work within a single node contraction, we attempted to implement across node parallelization.

Our initial approach was to do this with transactional memory, as stated previously. A later approach had a very similar idea where instead we have each thread essentially look for independent work that it can do. This means that when a thread picks up a chunk of work, it needs to log it so that other threads can see that they cannot do anything that "touches" a chunk of work that another thread is working on. Each thread passes over nodes that it cannot contract until it finds a piece of independent work and logs it, etc. Below is a code block illustrating this idea.

```
ContractedGraph contractedGraph(graph);
std::unordered_set<int> dontTouch;
std::mutex mux;

for (int k = 0; k < 5; k++) {

    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < graph.size(); ++i) {

        if (contractedGraph.edges[i].size() < 8) {
            bool doit = true;

            mux.lock();
            std::unordered_set<int> replacement(dontTouch);

            for (const auto& [neighbor, edge] : contractedGraph.edges[i]) {
                if (!replacement.insert(neighbor).second) {
                    doit = false;
                    break;
                }
            }
            if (!replacement.insert(i).second) {
                doit = false;
            }

            if (doit) dontTouch = replacement;
            mux.unlock();

            if (doit) {
                std::cout << "Contracting node " << i << std::endl;
                contractNode(contractedGraph, i);

                mux.lock();
                for (const auto& [neighbor, edge] : contractedGraph.edges[i])
                    {
                    dontTouch.erase(neighbor);
                }
                dontTouch.erase(i);
                mux.unlock();
            }
        }
    }
}
```

Listing 1: Across nodes approach to parallelism; looking for independent work

Unfortunately, neither of these approaches ended up producing valid results. On the bright side, within-node-contraction-parallelism is highly effective for computationally expensive workloads.

3.2 Delta-Stepping Approach

Parallelization of the delta-stepping algorithm required a few different approaches, and we eventually settled on the best one. Initially, once we had implemented the delta-stepping algorithm based on its general psuedocode found on Wikipedia (cited in our references), we began parallelizing the processing of light edges. There were a few major challenges, such as ensuring timely but safe updates to variables shared across threads, and also deciding on the optimal thread-scheduling policy. Initially, we tried to use static scheduling and a critical section to update the shared variables. However, as we somewhat anticipated, no meaningful speedup was achieved with this method. The delta-stepping algorithm simply spends too much time updating shared variables, and thus each thread is constantly fighting over the critical section instead of making progress on the problem.

Next, we tried to coalesce updates to the shared variables and update them after all of the threads completed. We created a few data structures to represent the updates that needed to be pushed to each different shared variable, and threads would create these updates during their work. Finally, at the end of the multi-threaded section, a single thread would publish these updates to the shared variables. However, this, too, yielded poor speedup. We believe it failed to achieve meaningful speedup because, again, the delta-stepping consists of publishing these changes to shared variables. Although some time could be saved by parallelizing the determination of what each update *is*, if the update is saved and must be published later, these updates add up to too much single-threaded work that defeats the purpose of the parallelism and any speedup that might be achieved.

Finally, upon some research online, we came across Intel's TBB library (installed on the GHC machines) which provides several lock-free data structures. We hypothesized that these could be very valuable to our parallelism because they'd allow us to publish updates as soon as they're created, preventing a pileup of single-threaded work. Indeed, upon switching our data structures to use the concurrent versions by TBB, we began to notice speedups from parallelism.

Next, we parallelized the heavy-edge section of our code beyond just the light-edge section, which yielded additional speedup gains. Parallelizing this section was relatively straightforward because the delta-stepping algorithm is designed with this type of parallelism also in mind.

Finally, we tuned the schedule of our parallelism. We had initially implemented static scheduling, which yielded the decent speedups mentioned above. However, we theorized that dynamic scheduling might be better in this case due to the randomness of graph exploration. For example, certain far-away map accesses may take a long time, but certain other map accesses may be quick because they're already in cache. As the search radius expands, these map accesses become farther and farther from each other, making this scenario more likely. Thus, we began using dynamic scheduling and tuned the batch size parameter to around 8, which we found to be optimal, likely because it balances the advantages of fine-grained with the advantages of coarse-grained batch sizes. It avoids the problems of too much overhead associated with too small of a grain, while also avoiding the problems of work imbalance that arise with a large grain size.

4 Results

4.1 Individual Results of Pre-Processing Contractions Phase

We begin with results on speedups for if we are simply contracting every node that is indexed by a number that is not divisible by 4, so we are statically choosing 75% of the nodes that we want to contract. While this may not be a very good proxy for whether or not a node should be contracted, this gives us a sense of what can happen as later in the computation the graph becomes extremely interconnected (with each node being connected with a majority of the other nodes in the contracted graph).

The following are the times we measured for the Directed Graph approach for exploiting within-contraction parallelism for different numbers of processors with this policy.

Number of Processors	Time to Run (s)	Speedup
1	29.774	1.00
2	15.768	1.89
4	8.216	3.62
8	4.548	6.55

Table 1: Speedup from "Directed Graph" parallelization approach on compute heavy contraction policy

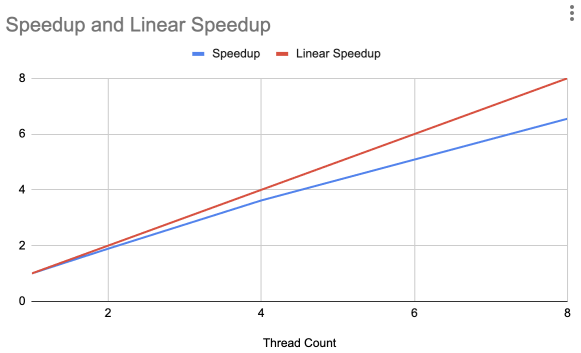


Figure 5: Graph showing speedup "Directed Graph" parallelization approach

The approach of using thread-safe unordered maps yielded very similar speedup results:

Number of Processors	Time to Run (s)	Speedup
1	29.592	1.00
2	15.858	1.87
4	8.379	3.53
8	4.662	6.35

Table 2: Speedup from parallelization approach using thread-safe unordered maps on compute heavy contraction policy

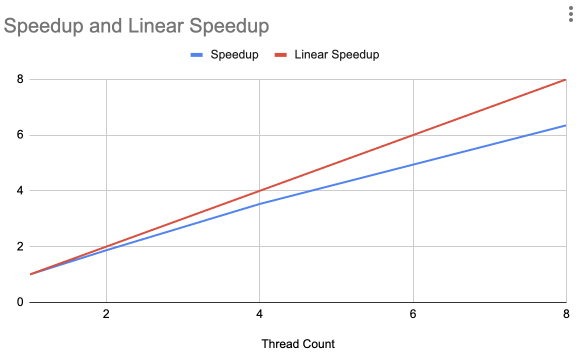


Figure 6: Graph showing speedup from thread-safe unordered maps

For both of these approaches we see fairly high quality speedup (somewhat close to linear speedup). Speedup remains limited, however, by some amount of redundant work that is done in the first approach to treat the undirected graphs as directed (i.e. not taking advantage of the symmetry of an undirected graph with respect to nodes both mapping to eachother if they are neighbors), and implicit locking to keep the `unordered_maps` in the latter approach safe. Additionally, early on in the program's execution there is less parallelism for this general parallelism within-contraction approach to take advantage of because it is only when a significant amount of contacting has been done that the graph no longer resembles a road network (with each node having few, rarely more than 4, neighbors), and becomes much more interconnected, with each node being connected to a majority of the other nodes, and thus presenting the real opportunities for parallelism within a single node contraction.

Finally, if we make our policy contracting only nodes that have less than 8 neighbors, we are able to wind up with contracted graphs with even fewer nodes than the ones obtained with the policy above (1919 nodes remaining vs. 2278), and this is done in only 115ms using just 1 thread. With the above results in mind, we essentially have the freedom to use whatever policy we want (ideally, whichever one will happen to produce the most helpful contraction graph), because thanks to our speedup through parallelism, we aren't stuck relying on only those policies which will finish making the contraction in the shortest amount of time. Below is a table showing the speedup we attain on this contraction policy using the "Directed Graph" parallelization approach.

Number of Processors	**Time to Run (ms)**	**Speedup**
1	115	1.00
2	95	1.21
4	94	1.22
8	92	1.25

Table 3: Speedup from "Directed Graph" parallelization approach on computationally light contraction policy

And finally, the table for parallelization using thread-safe unordered maps on this lighter contraction workload:

Number of Processors	**Time to Run (ms)**	**Speedup**
1	77	1.00
2	64	1.20
4	56	1.38
8	89	0.87

Table 4: Parallelization through thread-safe unordered maps on lighter contraction workload

As we see, while these times are much shorter (note units are in milliseconds), the speedups we see when there is almost no parallel work that we are able to exploit is much lower, and certainly does not achieve the same kind of scaling (continual growth in speedup) as we add more processors.

4.2 Results of Path-Finding Delta-Stepping Phase

After different approaches, we settled on a method of parallelism of delta-stepping that yields a strong speedup.

Our experimental setup consisted of running the delta-stepping algorithm from the DCA airport to the Jefferson Memorial in Washington, DC. The program essentially finds the path that minimizes the path length and elevation of the path taken from place to place. In addition, the Washington, DC area elevation data from the USGS is extremely high-resolution, so we can adjust the resolution of the generated maps to thereby adjust the difficulty of the problems. Higher resolution maps lead to longer paths and thus longer planning times to find those paths.

Here, we demonstrate that our dynamic scheduling policy yields much better speedups than static scheduling, as explained in the approach section.

Threads	Time (s)	Path Cost	Speedup
1	100.944	20192.0	1.000
2	57.597	20192.1	1.753
4	31.451	20192.2	3.210
8	17.841	20192.3	5.658

Table 5: Performance results of parallelized delta-stepping with varying thread counts using dynamic scheduling.

Threads	Time (s)	Path Cost	Speedup
1	100.237	20192	1.000
2	67.477	20192	1.486
4	40.137	20192	2.497
8	26.342	20192	3.805

Table 6: Performance results of parallelized delta-stepping with varying thread counts using static scheduling.

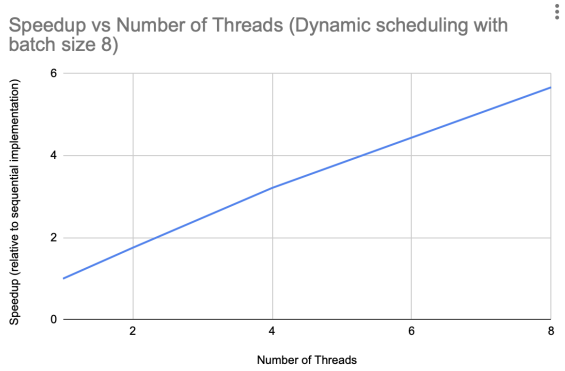


Figure 7: Graph of speedup with dynamic scheduling.

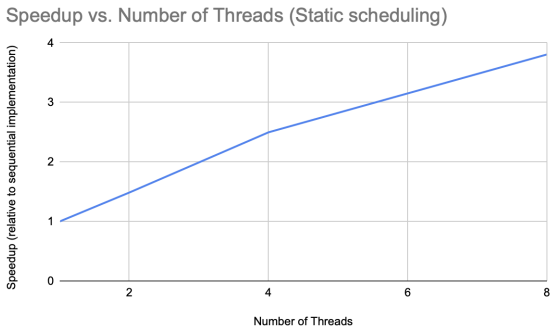


Figure 8: Graph of speedup with static scheduling.

Our speedup was inherently limited by the synchronization steps that are required by the delta-stepping algorithm. Although the delta-stepping algorithm clearly allows opportunities for parallelism, it still requires edges to be explored in a systematic way ranging from close to the start to far away from the start. Thus, although edges can be explored in parallel, there's a limit to the amount of parallelism before the threads must re-join so that the next batch of edges is processed together (instead of too early, causing low-quality paths).

We believe our choice of the GHC machines as our target were sound. OpenMP on a shared-memory machine such as the GHC machines yielded strong speedups for our problems, and we never

found ourselves wishing for extreme parallelism such as from a GPU or communication with multiple nodes such as from a distributed memory machine.

5 References

1. Description and Pseudocode for Delta-Stepping: https://en.wikipedia.org/wiki/Parallel_single-source_shortest_path_algorithm
2. Contraction Hierarchies Overview on Wikipedia: https://en.wikipedia.org/wiki/Contraction_hierarchies
3. CH-based Route Planning Project (Section on Contraction): https://jlazarsfeld.github.io/ch.150.project/sections/8-contraction/
4. Space efficient, Fast and Exact Routing in Time dependent Road Networks: https://www.youtube.com/watch?v=URhbkWsi_vo

6 Work Completed by Each Student

We split up the work between implementing Dijkstra's algorithm and learning about and extracting speedup within Dijkstra's, and learning about graph contractions in general, and more specifically which graph contraction techniques can be useful in path-planning, and how can they be parallelized! We believe that these two halves amounted to roughly the same amount of work.