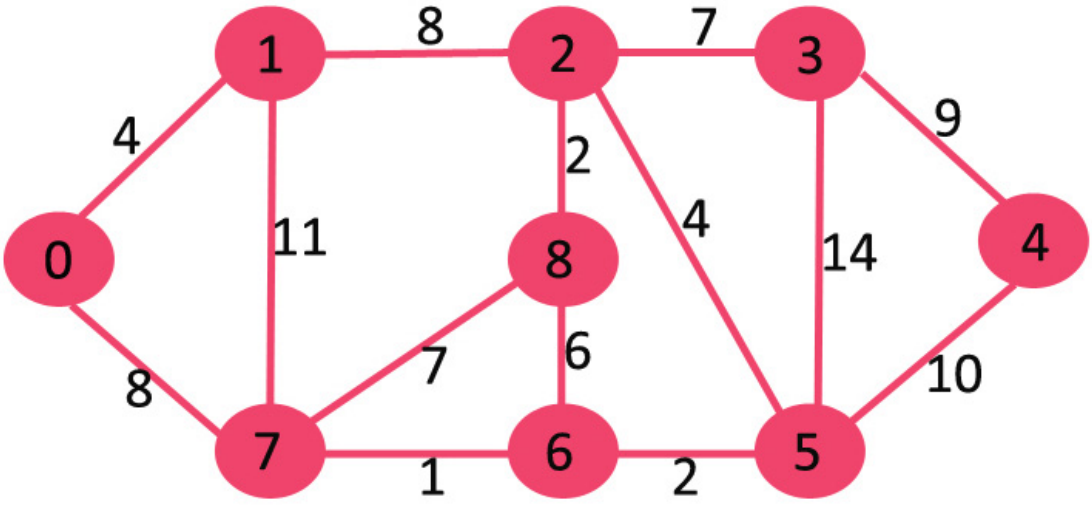
Parallel Shortest Path Problem

Ibnul Jahan (iej) + Ryan Gess (rgess)



OpenMP vs MPI in parallelizing Dijkstra's, Bellman-Ford's, and Johnson's Algorithms.

Summary

We parallelized the shortest path problems (SSSP and APSP) with Dijkstra's, Johnson's, and Bellman-Ford's algorithm. Two separate implementations were created, one in OpenMP and one in MPI, which we used to benchmark against each other and a sequential version. We achieved ~7.4x speedup with 8 GHC cores using OpenMP and ~5.5x speedup using MPI.

Github

https://github.com/ibnullify/Parallel-Shortest-Path

Background

There are really three separate algorithms that we are working to parallelize here, with the added caveat that one of them is built on top of subcalls to the remaining two. Each are graph problems, where we are attempting to find some variation of a minimum weight path across the graph. We decided on the following properties for our input graphs, due to the corresponding justifications:

- **Fully-connected**
    - An edge was included between every two vertices in all of the generated graphs for each algorithm. The reason for this was to maximize the computational complexity of the problem, as having more edges to expand would lead to longer sequential runtimes and more areas to parallelize.
    - Along with this, a fully-connected graph guaranteed that there would always be a minimum distance path between the source and every other vertex.
- **Undirected Edges**
    - For the input graphs, it was decided that the edge from vertex x to vertex y would have the same weight as the edge from the same vertex y to vertex x. This was done to keep our program aligned with the TSP program earlier in the course.
    - Having directed edges would not have led to a different implementation or change in our parallelization, which is why we determined it a better decision to use undirected edges. This made graph generation slightly faster, and also allowed us to test on larger graphs more efficiently.
- **Only Positive-weight Edges**
    - All of the edge weights across our test graphs are solely of positive weight. The reason for this is that Dijkstra's cannot run properly on graphs with negative weights. In order to be able to test out all three algorithms on the same graphs, we decided to ensure there were no negative edge weights.
    - Allowing negative edge weights would not have changed the complexity of the algorithms or the validity of our parallelization scheme in any way, which is why we felt it was reasonable to leave out. The implementations of Bellman-Ford's and Johnson's Algorithm are able to handle negative edge weights, but these weights were left out to make testing more efficient.

Along with this, we decided on an adjacency matrix representation for the graphs we were working with. An example graph and the corresponding matrix are as follows:

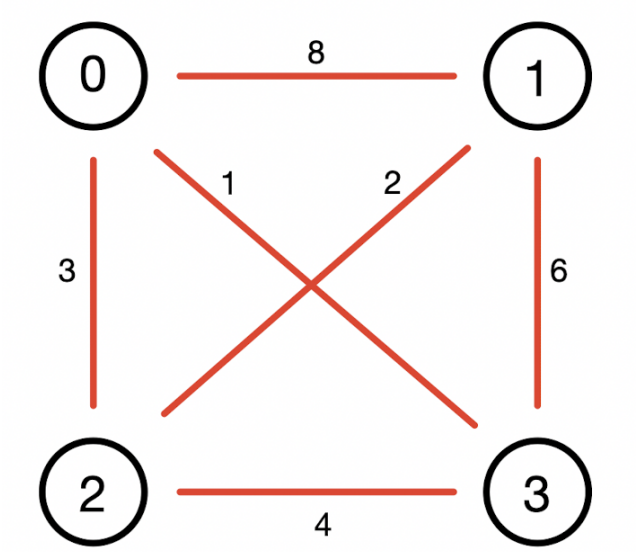


Figure 1: An example graph with 4 vertices

0	**8**	**3**	**1**
8	**0**	**2**	**6**
3	**2**	**0**	**4**
1	**6**	**4**	**0**

Figure 2: An adjacency matrix corresponding to the graph in Figure 1

(For notation's sake, the number of edges in a graph is typically denoted as M and the number of vertices in a graph is denoted as N.)

We can provide a **background for each algorithm** below, based on the problem type they are attempting to solve:

The Single Source Shortest Path problem (SSSP)

The SSSP problem is a graph question that attempts to find the path of shortest weight between one specified node (typically denoted s) and every other node in a graph. We will solve this problem using

two different algorithms: Dijkstra's and Bellman-Ford's. The output for this problem is a list of distances, where the entry at an index is the minimum path distance from the source to the corresponding vertex.

Dijkstra's Algorithm

This algorithm solves the SSSP problem (and there are many variants to solve for the smallest possible path from a source, s, to a certain target, t). This algorithm proceeds by starting at the source node, and relaxing the searching distance for paths outward, increasing either the total path weight we are searching for or the total length of the path. In particular, our version of the algorithm works by keeping track of which vertex that has not been processed yet has the current minimum distance from our source, and then expanding from there to find new minimum distances for each of its neighbors.

Dijiksta's is largely regarded as a highly sequential algorithm, and this makes it difficult to parallelize. Many implementations of the algorithm work by using a priority queue to maintain all edges on the frontier being explored, and by nature having to compare every edge in the iteration makes it heavily sequential. We modified this algorithm to instead expand edges based on the minimum cost vertex we have already found, and this allows us to update their neighbors with potential new values. Overall, updating our current shortest paths to each vertex is the computationally complex part of this algorithm, because it requires comparing many edges against each other at once. Parallelizing how path costs get updated would prove to be extremely beneficial.

```
dijkstraPQ G s =
let
  dijkstra X Q =            (* X maps vertices to distances. *)
    case PQ.deleteMin Q of
      (None, _) ⇒ X          (* Queue empty, finished. *)
    | (Some (d, v), Q') ⇒
        if (v, _) ∈ X then dijkstra X Q'      (* Already visited, skip. *)
        else
          let
            X' = X ∪ {(v, d)}          (* Set final distance of v to d. *)
            relax (Q, (u, w)) = PQ.insert (d + w, u) Q
            Q'' = iterate relax Q' (N⁺_G(v))    (* Add neighbors to Q. *)
          in dijkstra X' Q'' end
  Q₀ = PQ.insert (0, s) PQ.empty      (* Initial Q with source. *)
in dijkstra {} Q₀ end
```

Figure 3: Pseudo-code for Dijkstra's Algorithm

Bellman-Ford's Algorithm

This algorithm also solves the SSSP problem, and also does so by relaxing edges, starting at our source node. The particular difference between the approach in Bellman-Ford's and Dijkstra's is that where the latter relaxes edges based off of the closest unprocessed vertex, Bellman-Ford's relaxes all edges *at once*. This means that we only need to have N iterations in the algorithm, as at the i-th iteration we will account for all vertices with a path length (note: length, not cost) of i to the source.

Like Dijkstra's, the computational complexity of the algorithm is in relaxing the frontier so that we can explore paths of different size lengths. This is where the algorithm can benefit from parallelization, as there will be many different paths of the same length to consider at once, and many will be independent of

each other (some will have no similar nodes other than the source). There is a dependency in that the paths of length i+1 can only be considered after the paths of length i, so we must do all parallelization within each level.

```
BellmanFord (G = (V, E)) s =
  let
    BF D k =
      let
        D_in(v) = min_{u∈N_G^-(v)}(D[u] + w(u, v))     (* Min in distance. *)
        D' = {v ↦ min(D[v], D_in(v)) : v ∈ V}     (* New distances. *)
      in
        if (k = |V|) then None          (* Negative cycle, quit. *)
        else if (all {D[v] = D'[v] : v ∈ V}) then
          Some D        (* No change so return distances. *)
        else BF D' (k + 1)              (* Repeat. *)
      end
    D_0 = {v ↦ ∞ : v ∈ V \ {s}} ∪ {s ↦ 0}     (* Initial distances. *)
  in BF D_0 0 end
```

Figure 4: Pseudo-code for Bellman-Ford Algorithm

The All Pairs Shortest Path problem (APSP)

This problem builds on top of the SSSP, and attempts to find the path of shortest weight between *every* single pair of vertices. Naively, this can be done by running an SSSP solver, like Bellman-Ford's, on every single vertex. But, there exist other algorithms, in this case Johnson's, that work to greatly reduce the time complexity and repetitiveness of this naive approach.

Johnson's Algorithm

This algorithm solves the APSP problem by using Bellman-Ford's algorithm to reweight the graph and make it so there are no longer any negative edges, but the shortest path is preserved. By removing negative weights, we are then able to use Dijkstra's algorithm on the graph. In our implementation, there are no negative weights so the transformation of the graph does not need to occur, but the algorithm is implemented in its entirety and will complete the call to Bellman-Ford's algorithm. The same amount of computation occurs regardless of whether there is a negative weight edge.

Johnson's algorithm has built in calls to both Bellman-Ford and Dijkstra's, both of which will have parallel implementations to increase the performance of Johnson's. Apart from this, Johnson's can also benefit from parallelism in that it works by having calls to Dijkstra's on every node in the graph, which we can parallelize the calls of. Each of these calls are independent of each other, but are dependent on the modified graph created by Bellman-Ford's so we can parallelize after the former algorithm completes.

```
JohnsonAPSP (G = (V, E, w)) =
let
  G⁺ = Add a dummy vertex s to G,
       and a zero weight edge from s to all v ∈ V.
  D = BellmanFord(G⁺, s)
  w'(u, v) = w(u, v) + D[u] − D[v]    (* w'(u, v) ≥ 0 *)
  G' = (V, E, w')

  Dijkstra' u =
    let Δ_u = Dijkstra G' u
    in {(u, v) ↦ (d − D[u] + D[v]) : (v ↦ d) ∈ Δ_u} end

in ⋃_{u∈V} (Dijkstra' u) end
```

Figure 5: Pseudo-code for Johnson's Algorithm

Approach

Due to having two separate sets of implementations (in addition to our sequential versions), we have different approaches for both OpenMP and MPI. Within these, we parallelized three distinct algorithms and each of those had different approaches as well. These were largely based on the computational intensity of certain parts of the algorithm as detailed in the background, with small additions when possible, and parts missing when they proved to be infeasible.

Serial Algorithm Change - Dijkstra's

Dijksta's algorithm is largely considered to be sequential because of the use of a priority queue to typically track the next edge relaxation that we are considering. After noting this issue, we decided to switch our implementation of our sequential algorithm so that it could better translate to and benefit from parallelization. Particularly, we made it so that instead of watching every edge on our frontier at once, we first found the closest vertex that we have definitely found the shortest path for, and then relaxed the edges to its neighbors. This introduced computation that was not dependent on each other, and gave us an area to parallelize.

Dijkstra's Algorithm

With our version of Dijkstra's, we were able to achieve parallelization in one large location, with small changes being needed outside of this to make it possible. This large area is allowing us to search all edges adjacent to our specified minimum-distance and finalized vertex in parallel. There are a few data structures that were used to implement this, specifically two arrays that indicated whether a vertex had a finalized path cost ("Processed"), and what the current minimum path cost to a vertex is ("Distance"). An example of two iterations of this is specified below:

- Steps are in order from top-left to top-right to bottom-left
- The green vertex is being relaxed from, which is the minimum cost vertex not yet finalized
- The blue edges are all edges being considered in a single iteration - ***this is what we parallelize***
- The red values are the ones that have been updated in the current iteration

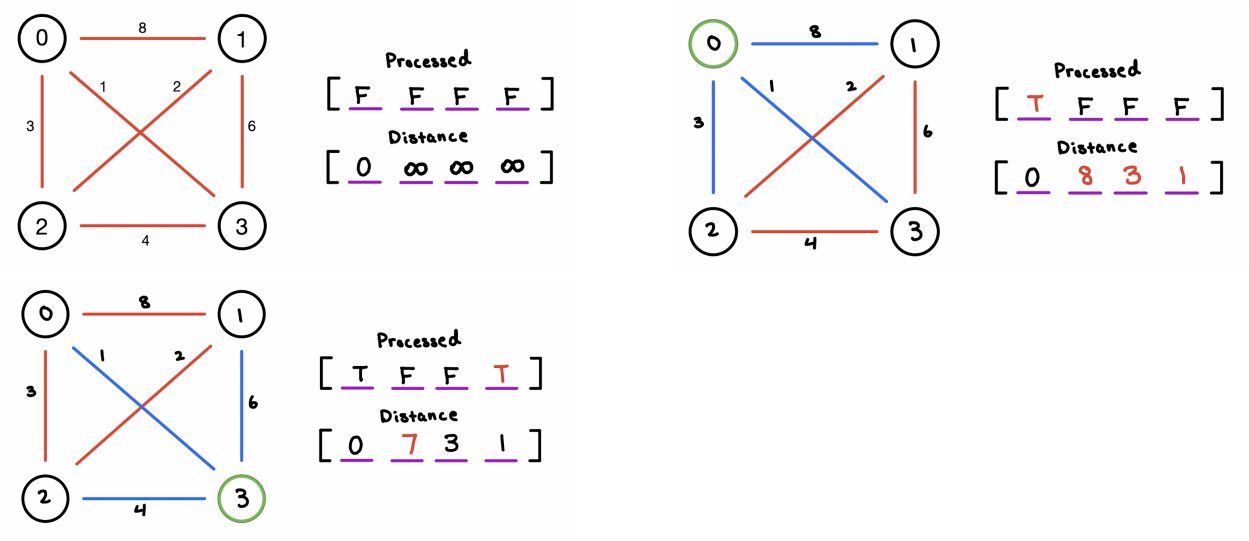


Figure 6: Two iterations of Dijkstra's Algorithm.

OpenMP

To parallelize our blue edges using OpenMP, we made use of `#pragma omp parallel for`. This worked well, because of a few algorithm design choices we made. Particularly, the data structure being updated on each iteration was the distances list. The updates being made were considering the immediate edge from the green vertex and the vertices it was neighboring. Because of this, none of these operations required knowledge of the others and were independent. This was determined to be a sufficient amount of work to parallelize due to the size of the graphs we were working with. For the example above it does not seem like much, but this is where the majority of the complexity of the algorithm comes from, which is why it was the main focus for our parallelization

MPI

To parallelize our blue edges using MPI, we assigned a process to handle collections of vertices. The control flow was tweaked to be a little different, in that now instead of the green vertex reaching out to its neighbors to update them, now our processes would search at every iteration to see if they had a vertex connected to a green one. In this case, they would know that they could update their own values, based on what the globally agreed upon green vertex's value was. At the end of the iteration, every vertex would have its new updated value thanks to the processor who's group it was a part of. They would then signal to each other that the iteration was complete, allowing every process to continue to the next iteration. It was important that every process completes every iteration together because the minimum distances are being updated at every iteration, and these are what are used to determine the green vertex in the next iteration.

Considered Alternatives

Initially, we attempted to parallelize the algorithm underneath either the first or second iteration. This would have entailed doing the first iteration to give some edges a temporary minimum (not necessarily the final minimum) distance to the source, and then running Dijkstra from those points. This would have led to more ease in our parallelism, but we noticed very early on that this required tons of work to be

computed multiple times. Along with this, it would often allow for large chunks of work to be done, just for us to find out the distance was not optimal resulting in all of the work being invalid.
Following this, we attempted to find a solution where we separated all of the vertices into groups, weight each vertex by its distance to the source, and then ran a small Dijkstra algorithm within each group to find the minimum cost distance this way. This would have led to a lot of complications in merging the results of the smaller recursive calls, and also would have introduced a lot of overhead that was not needed in the problem, causing us to avoid this idea as well.

Bellman-Ford's Algorithm

For the Bellman-Ford algorithm, we realized that the computationally complex portion of the program that could benefit the most from parallelism was again when we relax the edges. This works differently than in Dijkstra's, however, with us now relaxing every edge at a time, effectively growing out the sizes of the paths we are checking by one in each iteration. Bellman-Ford's algorithm only requires N iterations (where N is the number of vertices in the graph), because N-1 should be the maximum size of a minimum path due to no redundancy. We realized that in these N iterations, as long as we made sure to process all edges properly, they were not dependent on each other within the iteration. Edges would affect the distances we were tracking to each city and would therefore be dependent on the edges we processed in prior iterations, but within an iteration the edges could be processed in an arbitrary order and the algorithm would work properly.

OpenMP

Our parallelization in OpenMP worked by doing precisely this, parallelizing the expansion of our path lengths by considered edges asynchronously. Particularly, we set up the program to have a pragma parallelize the for loop where we iterated through and checked all of our edges to update distances. This represented the bulk of the work in the algorithm, so making this change was able to give us significant amounts of speedup. After doing this, however, we also had to account for a lot of race conditions as different threads attempted to update the distances to the same vertex. To remedy this, we had to put the accesses to our minimum distances list in a critical region to avoid these issues. This largely would have required the majority of the iteration to be encased in a critical region, so we fixed this by storing the previous values of the vertices we were considering initially, and then solely doing the update in a critical region based off of the value we computed using these.

MPI

Our implementation in MPI for this algorithm proved to be a bit more complicated. First, because the majority of our runs occur with very large graphs, we split up the initialization of the dist array to the different processors we had available. This proved to help only very slightly, but we expect that for even larger graphs it would have a greater effect. With the graph sizes we tested on, we expected the communication overhead counteracted the benefit from this shared work. Apart from this, we approached our MPI implementation by having different threads be responsible for different vertices. Based on this, that made it so they would partition the set of edges, and therefore we could parallelize the edges that we were expanding (This worked because all edges (u, v) would be assigned to the process that handled the vertex v, and thus that process would carry the updated value for this vertex). Then, after we completed an

iteration all of the processes would communicate using an MPI_Allreduce and synchronize on a list of global minimums that had been viewed thus far.

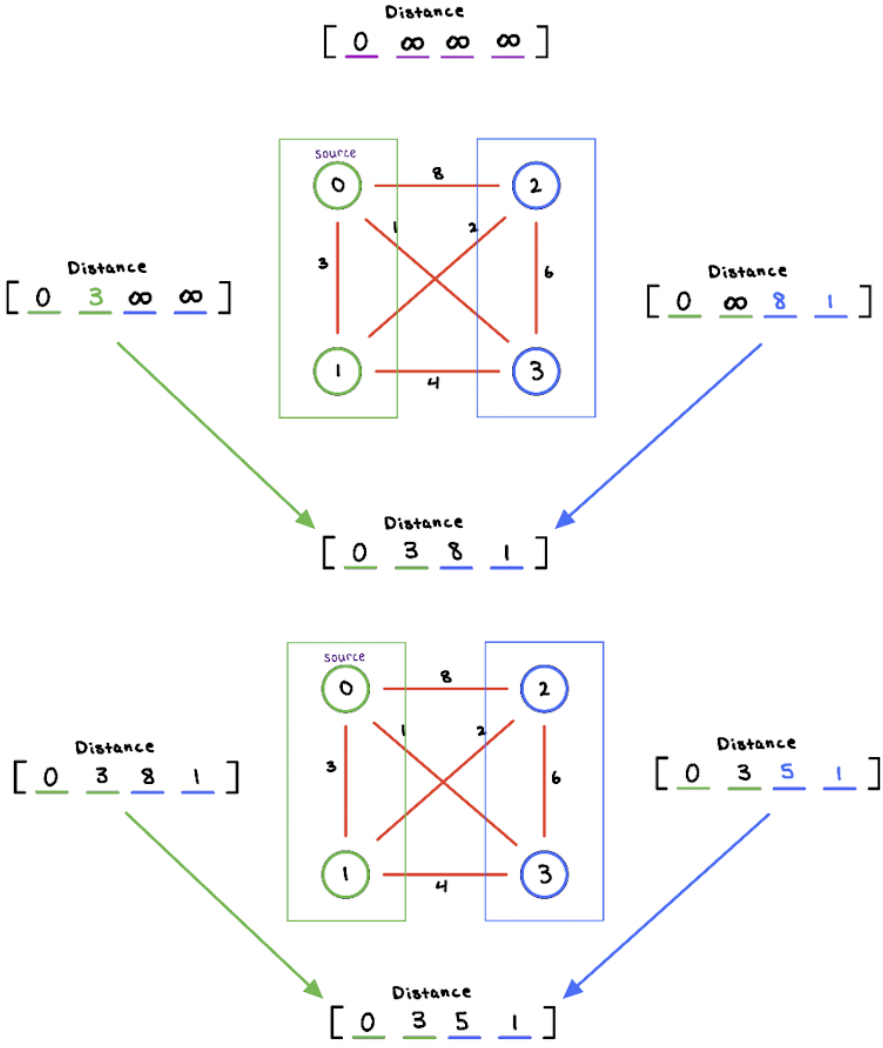


Figure 7: Two iterations of MPI Bellman-Ford's Algorithm.

Green vertices belong to one process, and blue vertices belong to another. Both begin with a distance of 0 for the source (Node 0), and distance of infinity for every other vertex. At each iteration, the process will consider every edge that has a destination in its vertex set. Particularly, for the initial iteration the following edges are considered by the processes to get their updated distance array:

Green	Blue
(1,0) - Weight 3 | (2,0) - Weight 8 | (3,0) - Weight 1	(0,2) - Weight 8 | (1,2) - Weight 2 | (3,2) - Weight 6
(0,1) - Weight 3 | (2,1) - Weight 2 | (3,1) - Weight 4	(0,3) - Weight 1 | (1,3) - Weight 4 | (2,3) - Weight 6

Considered Alternatives

Although we properly identified the area to parallelize in this algorithm relatively early on, we approached the problem a number of incorrect ways initially. In particular, before agreeing on this final version of our MPI implementation, we initially made the difference of assigning edges to our processes based on the *source* node of the edge. Although this seems like it would be trivially different and functionally equivalent, it actually resulted in much worse and less elegant code. This is because it is only the destination node in the edge that we are updating the values for, and by assigning by source, this made it so that each process was updating the distances of vertices outside of its partition. This required tons more communication, because each process would have to send over its entire distance list and take the minimum at each entry for all of the lists it received. By restricting all of these updates to only be in the vertex list, we were able to make an MPI_Allgather all that we needed for all the processes to synchronize.

Johnson's Algorithm

For Johnson's algorithm, we knew right away that the algorithm would achieve speedup limited by what Dijkstra's and Bellman-Ford's were able to achieve. Apart from the parallelization that we implemented in those two methods, we also searched for areas of Johnson's algorithm itself that could be parallelized. We were able to find two main portions of the algorithm that we decided to work on:

- After the initial run of Bellman-Ford's in the beginning of the algorithm, we have the list of weights and are tasked with modifying our graph. We decided to transform our graph in parallel because the modified weights are based on the algorithm we have already completed, and thus are not dependent on each other
- The ending part of Johnson's algorithm works by making calls to Dijkstra's algorithm across the graph, so that we can find the shortest path from every source node. We also decided to parallelize these calls, so that we would process the shortest path from a source node separate to the paths from a different source.

OpenMP

In our OpenMP implementation, we were able to achieve the first of these two changes by having different threads be responsible for different vertices. This was done by using a pragma to loop through the creation of our modified graph, and make the weight updates in parallel with other threads. Because the work being done by each thread was independent after the call to Bellman-Ford's, we were able to do this without problem. This resulted in modest speedup, as modifying the graph was not the most computationally expensive part of this algorithm. The second of the changes had a larger speedup attributed to it, as running Dijkstra's is the computational bottleneck in this algorithm. We parallelized by using a pragma to parallelize a for loop where we iterated through all of our nodes and ran Dijkstra's from them as the source node.

MPI

In our MPI implementation, we were able to parallelize the creation of our modified graph by partitioning the vertices across the different processes we were spawning. After the Bellman-Ford run, every process was in agreement about what the weights were, so they each modified the weights on the vertices in their

vertex set. This allowed us to use an MPI_Allgather to collect the updates in the different threads, and have them agree on one global shared modified graph. Following this, we had each process run Dijkstra's for the vertices in it's assigned vertex set. This would allow each one to generate all of the shortest path pairs for any path that started in its set.

Considered Alternatives

One initial alternative that we began to consider was due to the potential for workload imbalance. We believed the runs to Dijksta's may be too coarse grained, even though they were the natural task points based on this algorithm. We realized that this was a necessary way to split up the work, however, as the Dijkstra calls were built into this program and would all take roughly the same amount of time to compute. This was because Dijkstra's would expand the same number of edges regardless of the starting vertex (due to our fully connected graph), so we knew that this would be a balanced operation. This allowed us to reduce the overhead that would have been accrued by trying to parallelize in a finer-grained way than our function subcalls.

Results

With three algorithms being run against two different parallel APIs, we wanted to compare as many things against each other as possible. In particular, the goal was to have comparisons between sequential, OpenMP, and MPI for each algorithm, and then also see overall how each API was able to perform across the different tasks. Performance was measured by comparing the sequential runs of each algorithm to the time of the parallelized versions in each API, and then plotting times and speedup. The measurements are therefore being conducted using wall-clock time, and the speedups are with respect to the core count.

Setup

After testing across many different graph sizes, we ended up deciding that the optimal size to test on was a graph with **1024 vertices** (> 500,000 edges). We decided on this because it gave us a large enough run time to allow us to experience speedup properly, but also was quick enough that we would be able to run many trials together.
Benchmark data was generated by running our algorithms on 5 different graphs of sizes N = 512, 1024, 2048, and then taking the average times for these runs. Benchmark data was created by using a baseline of the sequential run of our parallelized implementation of each algorithm under this metric. This sequential run time was done using single-threaded CPU code on the GHC machines. Then, after measuring our OpenMP and MPI implementations, we would calculate the speedup that was being experienced under every number of cores in the following: {1, 2, 4, 8}.

Conducting Measurements + Complications

When measuring sequential runtime, it was done by doing 5 trials each on graphs of size N = 512, 1024, and 2048. Because the three algorithms are different in nature, we experienced vastly different runtimes while analyzing them. Particularly, Johnson's algorithm itself makes N different calls to Dijkstra's which immediately makes them difficult to test performance for while using the same size graphs. We devised a work around for this by trying to equalize the amount of work done by each test by running Dijkstra's N times, one for each source, for each test case. Bellman-Ford's and Johnson's we let remain at one run each. All results are measured in seconds, and all speedups are measured with respect to the same implementation on a single core.

Sequential Results:

Sequential			
	N = 512	N = 1024	N = 2048
Dijkstra's	0.476	3.597	29.649
Bellman Ford's	0.252	3.011	24.738
Johnson's	0.555	5.604	51.435

We can see that when we double the problem size, the work in each of these algorithms goes up by roughly 10x. This is a reasonable amount as doubling the number of vertices in a graph will increase the number of edges by over 4x (due to being a fully connected graph). This therefore increases both the number and size of the iterations we must do. We can keep this in mind as we begin to analyze the different speedups observed by our parallel implementations.

OpenMP Results:

OpenMP					
Dijkstra's		P = 1	P = 2	P = 4	P = 8
	N=512	0.386	0.369	0.345	0.354
	Speedup	1.000	**1.046**	**1.119**	**1.090**
	N = 1024	3.331	2.165	1.372	1.004
	Speedup	1.000	**1.539**	**2.428**	**3.318**
	N = 2048	27.960	16.440	10.623	7.251
	Speedup	1.000	**1.701**	**2.632**	**3.856**
Bellman-Ford's		P = 1	P = 2	P = 4	P = 8
	N=512	0.247	0.127	0.065	0.033
	Speedup	1.000	**1.945**	**3.800**	**7.485**
	N = 1024	2.856	1.460	0.727	0.386
	Speedup	1.000	**1.956**	**3.928**	**7.399**
	N = 2048	23.850	11.749	5.901	3.234
	Speedup	1.000	**2.030**	**4.042**	**7.375**
Johnson's		P = 1	P = 2	P = 4	P = 8
	N=512	1.902	1.007	0.506	0.262
	Speedup	1.000	**1.889**	**3.759**	**7.260**
	N = 1024	15.525	7.923	4.001	2.074
	Speedup	1.000	**1.959**	**3.880**	**7.486**
	N = 2048	122.486	62.680	31.867	16.383
	Speedup	1.000	**1.954**	**3.844**	**7.476**

N=2048 OpenMP

We can see some key observations about each of our implementations:

Dijkstra's:
We can see that for the graph of size N = 512, we achieved very minimal speedup regardless of how many processors we used. Even before testing, we expected less speedup on a smaller graph for Dijkstra's because it is reliant on parallelizing the blue edges as we explained in the Approach section. When we have smaller graphs, there are less edges for us to parallelize and the load imbalance between threads has a much larger impact on the overall performance. Because we parallelized the work within an iteration, any straggler thread that is slower to expand its set of edges will force the remainder of the iteration to halt until it finishes. With small numbers of edges to parallelize over, the amount of work in our slowest threads ends up being costly, and prevents the larger benefits of speedup from occurring. Also, with such a small graph, a lot of the cost of the algorithm is no longer in expanding the edges which we worked to parallelize, but also in conducting the iterations and initializing our values. Because these were not the portions we parallelized, the lack of large speedup on small graphs is reasonable as these represent more of the algorithm.
Overall, we were able to achieve roughly 4x speedup on 8 cores, which although sub-linear, is a good amount of speedup for a largely sequential algorithm. Because we were limited to only turning the code within an iteration synchronous, this decreased the opportunity for our parallelization, and is a large part of why we were not able to achieve even better results.

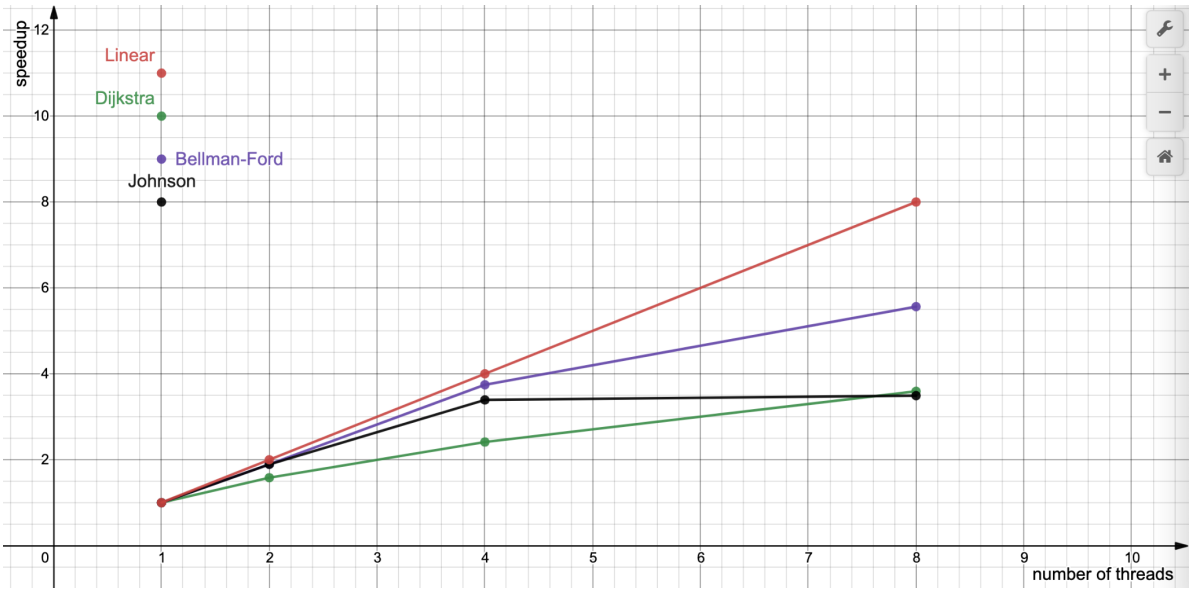
Bellman-Ford's:
We were able to achieve around 7.4x speedup on 8 cores for all graph sizes. This was very good performance, and we were able to reach this by capitalizing on a good parallelization strategy for the algorithm. By being able to consider all edges in the frontier asynchronously, this allowed us to introduce large amounts of concurrency into our implementation. And in addition to this, considering a single edge is a fine-grained operation so we did not need to worry about stragglers and load imbalance. Initially, we expected that a larger grain of work (considering 4 edges at once) would lead to better performance, but we reached the best overall performance by leaving it at one edge at a time (although 4 edges performed slightly better at graphs of size N = 2048).

Johnson's:
Similar to Bellman-Ford's we were able to achieve over 7x speedup on 8 cores for all different graph sizes. This was expected due to how closely related this algorithm is to the prior two. Because of Amdahl's law, we know that the sequential portion of a program has a large effect in limiting the potential speedup of a program. In this case, this would have been the bottleneck in the program that Dijkstra's represented, but we were able to avoid this by parallelizing our calls to Dijkstra's as well. This is why we were able to achieve such good speedup even though we are calling a function that had worse speedup. Johnson's lended us a great opportunity to do a deeper analysis into the parallelization that we were experiencing, because of its position as a combination of the other two functions. In particular, we hoped to find out which portion of the algorithm was providing the bottleneck in its speedup. We found that largely, the majority of the time the algorithm spent doing was running Dijkstra's algorithm, which worked well for us being these calls were parallelized. Bellman-Ford's algorithm took roughly 1/7th the amount of time (~2 secs vs ~14 secs on a N=1024 graph), and modifying our graphs took roughly 1/28th of this (~0.5 secs).

MPI Results:

MPI					
Dijkstra's		P = 1	P = 2	P = 4	P = 8
	N=512	0.377	0.337	0.302	0.282
	Speedup	1.000	**1.119**	**1.248**	**1.337**
	N = 1024	3.106	1.856	1.122	0.992
	Speedup	1.000	**1.673**	**2.768**	**3.131**
	N = 2048	28.679	18.136	11.890	7.981
	Speedup	1.000	**1.581**	**2.412**	**3.593**
Bellman-Ford's		P = 1	P = 2	P = 4	P = 8
	N=512	0.245	0.134	0.076	0.045
	Speedup	1.000	**1.828**	**3.224**	**5.444**
	N = 1024	2.816	1.597	0.767	0.538
	Speedup	1.000	**1.763**	**3.673**	**5.232**
	N = 2048	23.620	12.463	6.307	4.246
	Speedup	1.000	**1.895**	**3.745**	**5.563**
Johnson's		P = 1	P = 2	P = 4	P = 8
	N=512	0.554	0.329	0.162	0.100
	Speedup	1.000	**1.683**	**3.425**	**5.535**
	N = 1024	5.260	3.014	1.514	0.970
	Speedup	1.000	**1.745**	**3.474**	**5.425**
	N = 2048	52.942	27.978	15.606	9.710
	Speedup	1.000	**1.892**	**3.392**	**5.452**

N=2048 MPI



We can see some key observations about each of our implementations:

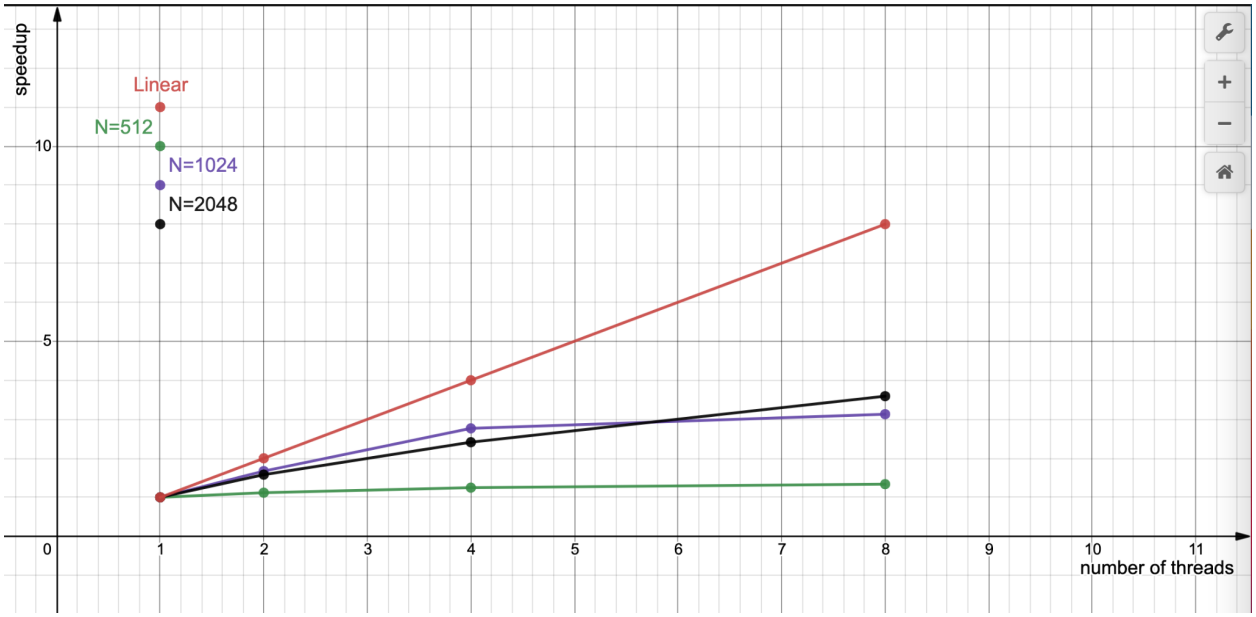
Dijkstra's:
Similarly to our OpenMP analysis, we can see that for the graph of size N = 512, we achieved less speedup regardless of how many processors we used. The previous analysis still holds in that we expected Dijkstra's algorithm in general (and more so on smaller graphs) to have less than ideal speedup. Having smaller graphs introduces less available areas to parallelize, and ultimately makes it difficult to achieve anywhere close to linear speedup. In MPI, we also now have to factor in large communication costs as well now, which plays a large effect as overhead in slowing down our speedup. As mentioned in the Approach, we are still limited in our MPI implementation in that a straggler process will force the iteration to wait for it to terminate. Another overhead for this implementation is also an inefficiency in searching that leads to duplicated work. As mentioned, different processes are responsible for unique partitions of the nodes, and therefore have to search through each to see if there is an edge connected to the green vertex we are expanding. This has noticeable more work than if we were to just iterate looking for edges adjacent to the green vertex, but that implementation was not feasible to parallelize. At the expense of extra repetition of work, we were able to switch the algorithm to achieve parallelization, but the cost of this extra work still had a significant impact on our speedup.

Bellman-Ford's:
On this algorithm, we were able to achieve ~5.4x speedup when using 8 cores. This is noticeably less of an improvement compared to our OpenMP version, but this was still a success given our expectations of what the limiting factors would be. For this algorithm, we must consider every edge at every iteration because this allows us to grow out the sizes of all the different paths we are considering. In our MPI implementation, we let a process be responsible for a subset of our vertices, and this allowed them to only search edges terminating in one of the vertices in their set. This itself leads to a very ideal speedup, but we were limited by the amount of communication that was needed. The different processes can have varying amounts of work depending on how large the costs of their paths are becoming, and because of this stragglers can be introduced. Because each iteration has a large amount of work, not only can we not proceed to the next iteration until the straggler finishes but none of the remaining processes are able to have complete data until this point either.
Along with this, splitting up the vertices into groups introduced an extra repetition in the amount of work that was being done. Particularly, although every edge was being checked, each process still had to consider every edge to determine which ones were relevant to them. In a sequential program this would happen once, but for our implementation every process had to do this.

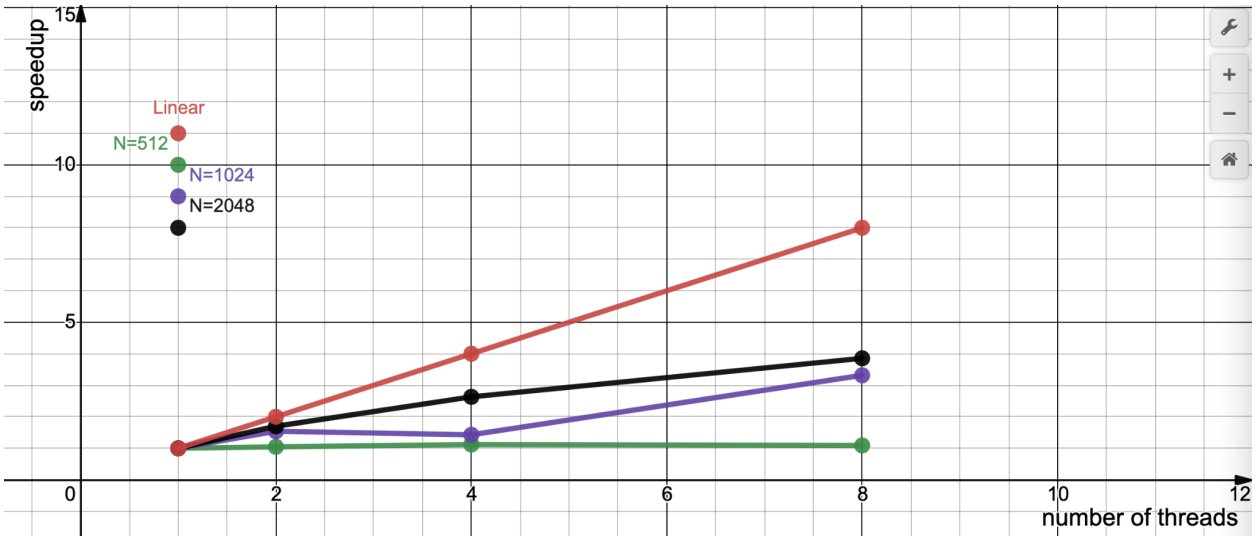
Johnson's:
We were roughly able to achieve 5.4x speedup when using 8 cores for this algorithm as well. This is less than what we experienced for OpenMP, but given this implementation was built off of our slower speedup Bellman-Ford's algorithm, these results make sense. We took some time to do a deeper dive on this analysis as well, and received similar results to our OpenMP analysis. We found that the majority of the work present occurred in the Dijkstra subcalls, which received similar speedup numbers to our OpenMP version. On the other hand, we noted that the slowdown seemed to come from a slower run to Bellman-Ford's algorithm, which took roughly 1/5th of the time that the Dijkstra calls did, and a slower completion of constructing the modified graph, which took roughly 1/12th of the time. The latter of which

was likely due to extra communication costs due to every process having to partially modify their local version and then collect all updates using an MPI_Allgather.

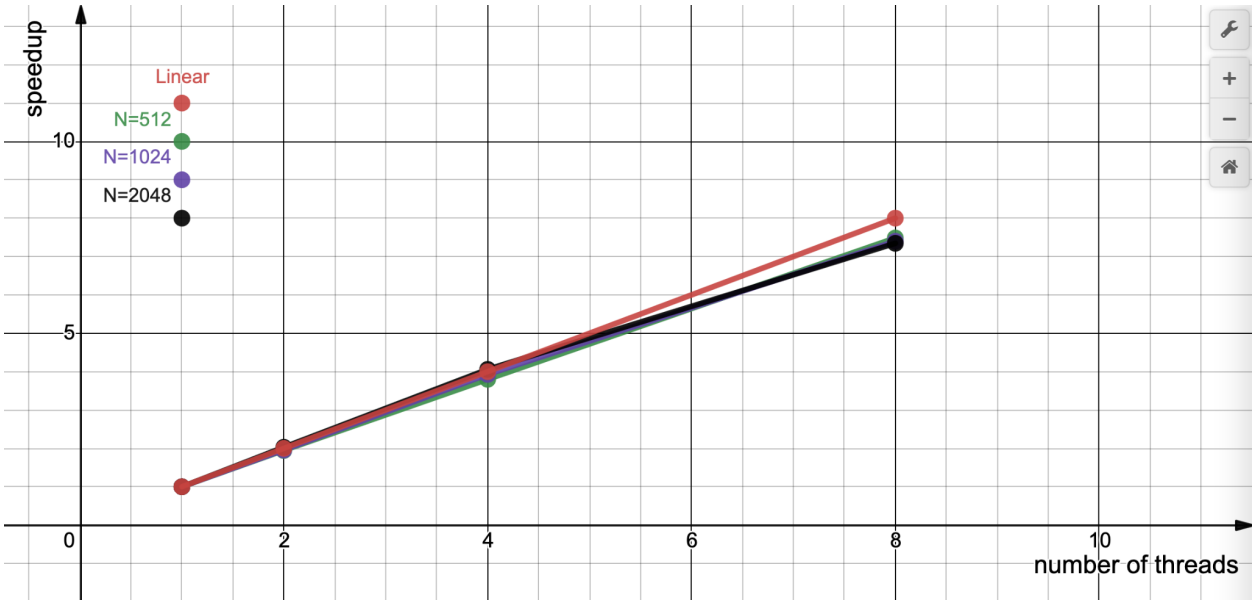
Dijkstra OpenMP



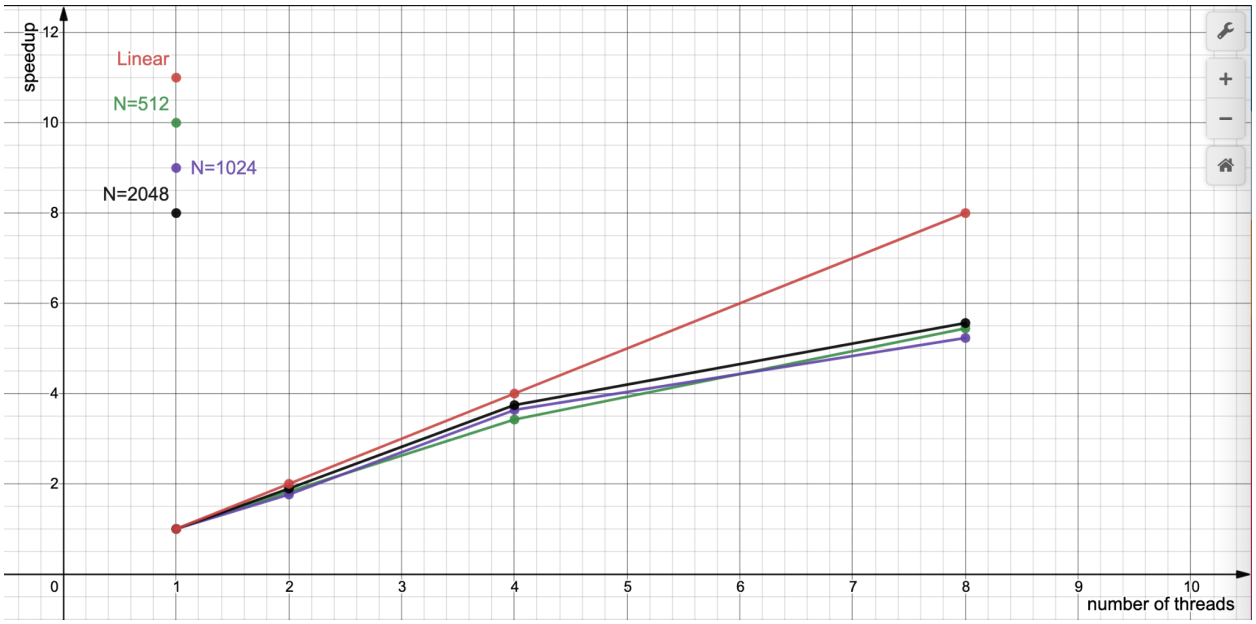
Dijkstra MPI



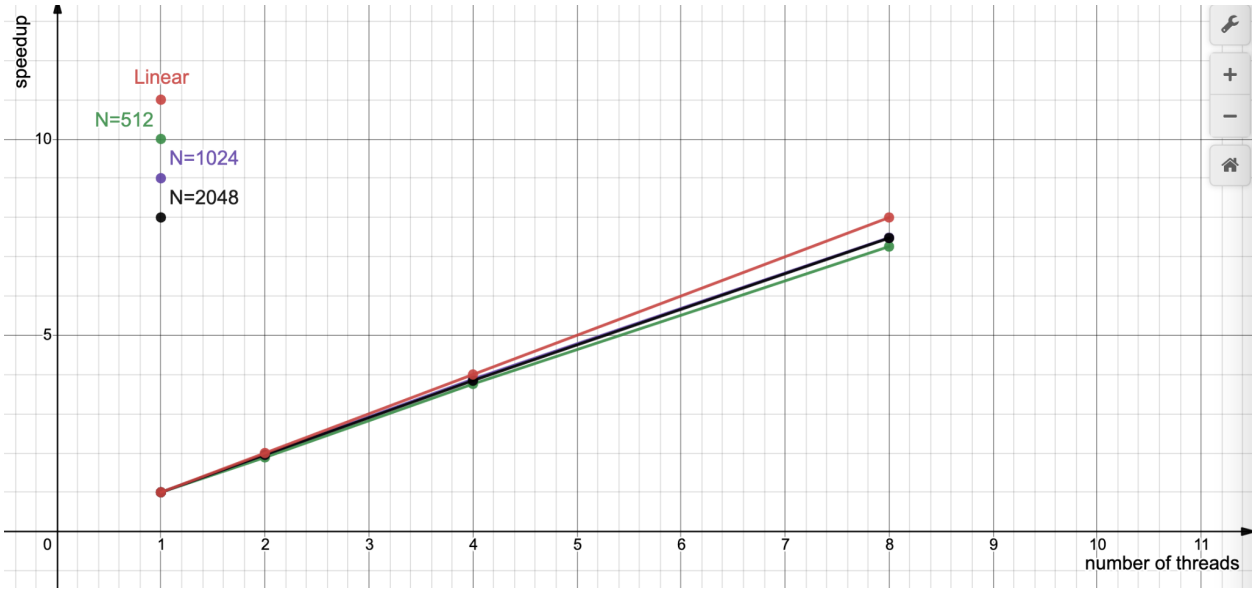
Bellman-Ford OpenMP



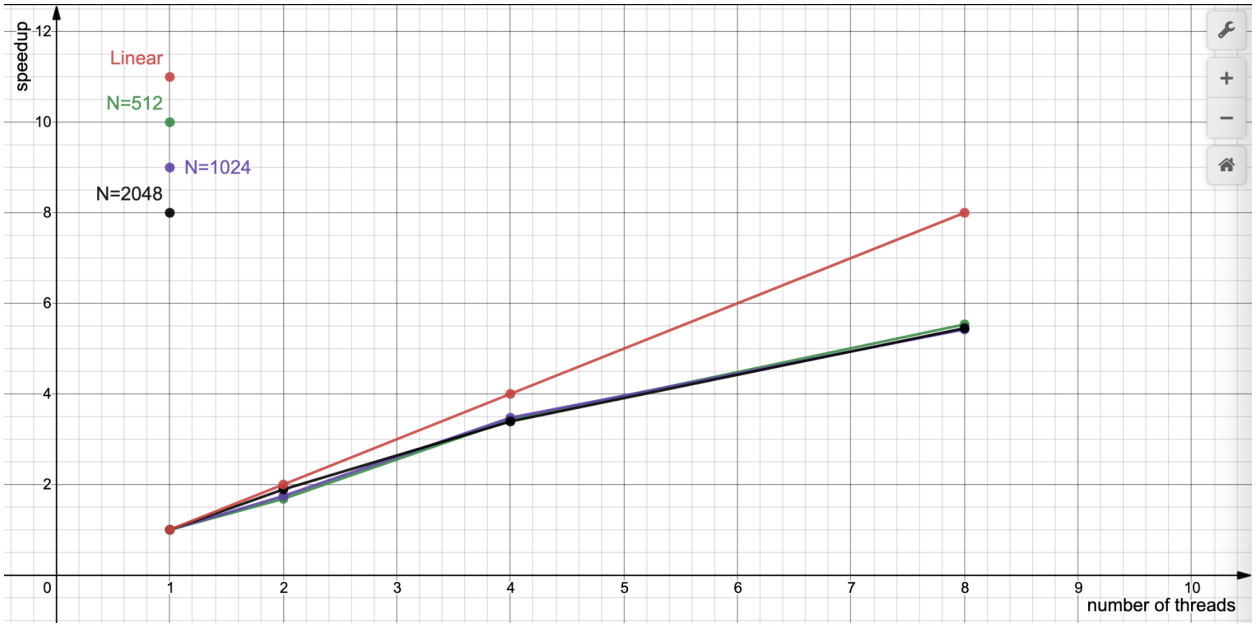
Bellman-Ford MPI



Johnson OpenMP



Johnson MPI



References

https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/ [Cover Image]
https://www.diderot.one/courses/89/books/352/chapter/4575?toc=book [Algorithm Pseudocodes]
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
https://en.wikipedia.org/wiki/Johnson%27s_algorithm
https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm
https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

Division of Work

Equal work was performed by both project members. Luckily, due to living near each other, we were able to largely work together through pair programming. This made design decisions and conversations around our implementations easier to conduct as well.