15-418: Final Project Report - Parallel Minimum Spanning Tree

Mengrou Shou (mshou), Jacky He (junchenh)

---

1 Summary

We explored different parallel programming paradigms for computing minimum spanning trees. Our CUDA of parallel MST algorithm achieves an almost 50x speedup relative to a fast sequential Kruskal's implementation when run on a p3.2xlarge AWS instance (with a Nvidia Tesla V100 GPU). This is around 7 times faster than the default minimum spanning tree algorithm that comes with the CuGraph library. The CPU performance numbers are obtained by running on a m7a.32xlarge AWS instance with a AMD EPYC 9R14 CPU with 64 cores hyperthreaded to 128 processors. Our shared address space implementation with OpenMP achieves sublinear speedup as the number of threads increases from 1 to 128. We also implemented a distributed minimum spanning tree algorithm using message passing. On large graphs, this achieved super-linear speedup as the number of processors increase from 1 processor to 32 processor, but the performance tapered off from 32 processors to 128 processors (possibly also due to hyperthreading not being the same as having a full new core) Finally, we compared the performance of those different implementations and made recommendations for what scenarios or computing contexts each implementation is best suited for.

2 Background

2.1 Graph Definition:

We explored the minimum spanning tree algorithm on connected, undirected, weighted, enumerable graphs with unique edge weights. However, there could be multi-edges (i.e. there might be multiple edges that connect the same two nodes with different weights).

Formally, consider graphs denoted as $G = (V, E)$. V is the set of vertices. E is the set of edges of the form $\{u, v, w\}$, where $u, v \in V$, $u \neq v$, and $w \in \mathbb{Z}$. Note that even though here we defined the weight w of an arbitrary edge to be an integer, our algorithms should also work on real-numbered weights with minor modifications.

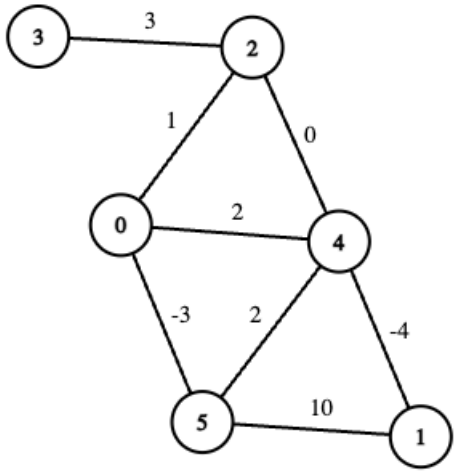
The graphs we consider throughout this project are enumerable, meaning that their vertices are represented by integers numbered from 0 to $|V|-1$. The reason for this restriction is that operations on enumerable graphs representation normally are faster because we can use the vertex numbers as indices into arrays. On the other hand, maintaining a non-enumerable graph might require us to use auxiliary data structures like a hash map or some sort of search tree, the parallelization of which can be themselves an entire separate project.

2.2 Minimum Spanning Tree Problem Definition:

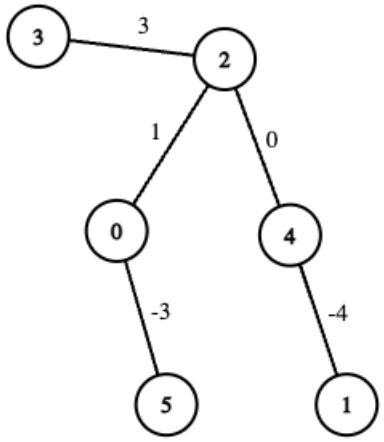
For a graph $G = (V, E)$, a spanning tree is a tree $T = (V, E')$ with $E \subseteq E'$. The minimum spanning tree (MST) problem requires finding a spanning tree of minimum weight, where the weight of a

tree T is defined as: $w(T) = \sum_{e \in E(T)} w(e)$.

To give an example, consider the following graph:



Its MST would be the following, where the weight of the MST is -3:



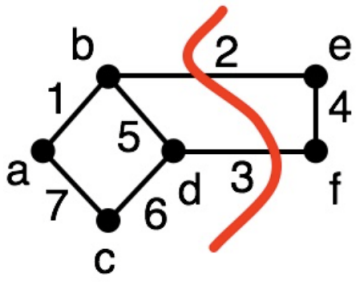
2.3 Light Edge Property

There are many ways for computing the minimum spanning tree, but the correctness of those algorithms are all based on a property about cuts in a graph, which is sometimes called the Light Edge Property. The Light Edge Property says that if we partition the graph into two blocks, the minimum edge between the two blocks is guaranteed to be an MST edge.

More formally, to borrow a definition from the 15-210 textbook, for a graph $G = (V, E)$, a cut is defined in terms of a non-empty proper subset $U \subsetneq V$. The set U partitions the graph into blocks induced by the vertex set U and the vertex set $V \setminus U$, which together are called the cut and written

as the cut $(U, V \setminus U)$. We refer to the edges between the two parts as the **cut edges** written $E(U, V \setminus U)$. We sometimes say that a cut edge **crosses** the cut.

The Light-Edge Property then says that given a undirected, connected, weighted graph $G = (V, E)$ with distinct edge weights, for any cut of G, the minimum weight edge that crosses the cut is contained in the minimum spanning tree of G.



For example, in the figure shown above, we have a graph cut $(\{a, b, c, d\}, \{e, f\})$, and there are two edges across the cut: $\{b, e, 2\}$ and $\{d, f, 3\}$. By the Light Edge Property, the edge $\{b, e, 2\}$ is the lightest edge crossing the cut, so this edge must necessarily be an MST edge.

The algorithms that we use in this project all heavily rely on the light edge property for correctness.

2.4 Cycle Property

There is also a property called the Cycle Property which is complementary to the Light Edge Property. It says that given any undirected, connected, weighted graph $G = (V, E)$ with distinct edge weights, for any cycle in the graph, the heaviest edge on the cycle is not in the MST of G. Whereas the Light Edge Property tells us which edges **are** in the MST, the Cycle Property tells us which edges are **not** in the MST.

2.5 Input and Outputs

Throughout this project, the inputs to our MST algorithms are lists of graph edges (represented using a C++ vector), and our outputs are lists of MST edges. If an algorithm needs a different graph representation than an edge list, the transformation process counts as part of the initialization work.

2.6 Prim's Algorithm

The Prim's Algorithm is a canonical MST algorithm which grows the MST incrementally by considering a cut between the current MST and the rest of the graph. The pseudocode for Prim's is given below:

```
1 prims(G = (V, E)) {
2     T = ∅ // set of MST edges
3     U = {s} // where s is some random source node
```

```
4       while (U ≠ V) {
5           let (u, v) be the least cost edge such that u ∈ U and v ∈ V \ U
6           T = T ∪ {(u, v}}
7           U = U ∪ {v}
8       }
9       return T
10  }
```

On line 7, the least cost edge is typically determined by maintaining a priority queue where the priority of a unvisited node in the frontier is the weight of the edge that that crosses from the set of visited nodes into the frontier.

2.7 Disjoint-Set Union

The Disjoint-Set Union (DSU) data structure is important for the problem of MST because the Kruskal's Algorithm and Boruvka's Algorithm can heavily rely on it.

The set up of the usage of a DSU data structure is as follows: We are given several elements, each of which is a separate set to begin with. A DSU will have an operation (usually called **join** or **union**) to combine any two sets, and it will also have another operation to efficiently tell which set a given element belongs to (usually called **find**).

The operations that a DSU data structure supports are:

- `make_set(v)`: This creates a new set consisting of the new element `v`
- `join(u, v)`: This merges two sets together, one where `u` is located, and another where `v` is located.
- `find(v)`: This returns the representative of the set that contains the element `v`. If two elements are in the same set, then they necessarily have the same representative.

A typical implementation of DSU assuming that the elements are enumerable integers is using an array indexed by the elements to store the representatives of the sets that each element belongs to. Two optimizations are typically enabled to make sure that the DSU has good `join` and `find` complexities: namely path compression and union-by-rank. The pseudocode for the implementation is given below:

```
1  make_set(v) {
2     parent[v] = v
3     rank[v] = 0
4  }
5
6  find(v) {
7      if (v == parent[v]) {
8          return v
9      }
10     return find(parent[v])
11 }
```

```
12
13 join(u, v) {
14     u = find(u)
15     v = find(v)
16     if (u != v) {
17         if (rank[u] < rank[v]) {
18             swap(u, v)
19         }
20         parent[v] = u
21         if (rank[v] == rank[u]) {
22             rank[v]++
23         }
24     }
25 }
```

Listing 1: DSU Pseudocode

It can be shown that with both path compression and union by rank, the amortized time complexity per `join` or `find` operation is $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function. In other words, the operations are nearly constant time amortized.

DSU as applied to MST algorithms will typically involve joining together sets of vertices via MST edges as we discover more and more MST edges, until the entire graph is connected via MST edges.

2.8 Kruskal's Algorithm

Kruskal's algorithm using DSU first starts out with $|V|$ disjoint sets, each set containing just a single vertex in the graph. Kruskal's algorithm first sorts the edges by edge weights, and then it goes through the edges in increasing order of their edge weights and keep choosing the edge that connects together two different sets, joins together the two sets, and adds the edge to the MST. If an edge connects together two vertices that are already in the same set (i.e. it forms a cycle), then that edge can be correctly discarded via the Cycle Property. The pseudocode for Kruskal's algorithm is shown below:

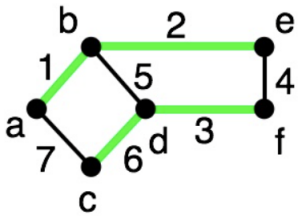
```
1
2  kruskal(G = (V, E)) {
3      T = ∅ // the set of MST edges
4
5      // initialize disjoint sets
6      for v in V:
7          make_set(v);
8
9      sort(E) in increasing order of the edge weights
10
11     for e = (u, v) in E {
12         if (find(u) != find(v)) {
13             join(u, v)
14             T = T ∪ {e}
15         }
```

```
16      }
17      return T
18  }
```

Listing 2: Sequential Kruskal's Pseudocode

2.9 Boruvka's Algorithm

Boruvka's Algorithm is yet another MST algorithm. Boruvka's Algorithm takes advantage of the fact that we can define a cut around each vertex where the vertex itself is a block, and the rest of the graph is in another block. Therefore, the minimum edge out of every vertex (which we call vertex bridges) must be in the MST. Boruvka's algorithm thus proceeds in rounds, adding all vertices bridges into the set of MST edges at every round, and then contracting the graph based on the partitions defined by the vertex bridges before proceeding to the next round. The algorithm terminates when there are no more edges remaining in the graph.



For example, as shown in the graph above, the highlighted green edges are the vertex bridges, and the graph would contract the two components connected by the vertex bridges. Suppose node a is the representative of $\{a, b, e\}$, and c is the representative of $\{c, d, f\}$, then the resulting contracted graph $G' = (V', E')$ would be such that $V' = \{c, a\}$, and $E' = \{(a, c, 7), (a, c, 5), (a, c, 4)\}$.

The pseudocode of the Boruvka's Algorithm is roughly as follows:

```
1  boruvka (G = (V, E)) {
2      T = ∅
3      boruvka'(G = (V, E)) {
4          if (|E| == 0) {
5              return
6          }
7          for v in V {
8              let {a, b} be the vertex bridge out of v
9              if (find(a) != find(b)) {
10                 join(a, b)
11                 T = T ∪ {(a, b)}
12             }
13         }
14
15         V' = { v ∈ V | find(v) == v }
16         E' = { {find(u), find(v)} : {u, v} ∈ E | find(u) != find(v)}
17         G' = (V', E')
```

```
18          boruvka' (G')
19      }
20      boruvka' (G)
21      return T
22  }
```

Listing 3: Sequential Boruvkas Pseudocode

2.10 Parallelism

The MST algorithms such as Prim's and Kruskal's presented above are inherently sequential. Whereas Prim's relies on a visit order of the vertices, Kruskal's relies on a visit order of the edges based on edge weights. However, Boruvka's algorithm presents more opportunity for parallelism. Since every node's vertex bridge can be computed in parallel, it's possible to split up the work of computing vertex bridges across different processing units.

Additionally, there are also parallelism to be gained during the graph contraction step. This step involves something like a filter or map operation, and this can potentially be optimized.

The workload of the algorithm forces several synchronization points. First of all, Boruvka's algorithm proceeds round by round, so the parallel runtime is at least proportional to the total number of rounds the algorithm takes (for sparse graphs one can show that this is $O(\log(n))$). In addition, if we're using DSU to help us with contraction, then that's another potential synchronization bottleneck because concurrent `join`'s and `find`'s to the DSU data structure should occur quite frequently. Alternatively, it's possible to use graph contraction schemes that do not require a DSU data structures, but those schemes typically have much larger overhead that overshadows the potential gain in getting rid of DSU.

There are temporal locality in the problem because it's possible that we access data about the same node multiple times in a row. However spatial locality might be hard to gain because of the random distribution of vertex numbers and edges. It's not necessarily true that the neighbors of vertices will have close vertex numbers as the vertices themselves. It's possible to artificially create locality by renumbering the vertices based on their distances to each other, but this would need to happen at every round of contraction, which can be quite expensive considering a full graph search is likely needed to come up with an effective numbering (and the time it takes to do such a full graph search in parallel could almost rival the time it takes to compute the MST without renumbering the vertices).

3 Approaches

3.1 Baseline

Our baseline approach is uses a sequential Kruskal's implementation, this is the approach that we'd benchmarking against if we'd like our approach to be based on the speedup relative to a fast sequential algorithm.

The initialization step of this approach simply initializes a DSU, and stores the edge list in a private variable.

During computation, the edges are sorted by edge weights, and the MST is computed by looping through the sorted edges and repeatedly trying to join the sets that each the two endpoints of the edge connects. If the joining succeeds (i.e. if the two endpoints were not previously in the same set), then we'd add the edge into the MST.

See Listing 2 for the pseudocode for sequential Kruskal's. Here, the edge set is represented by a C++ vector.

3.2 CUDA

For running the algorithm on a CUDA GPU, we tried to parallelize the sequential Boruvka's algorithm (the pseudocode is in Listing 3).

At a first glance, the most straightforward idea is to parallelize by vertices in line 7. However, this might complicate graph representation a little. The graph is given as a an edge list. However, if we were to parallelize by vertices, then the representation would need to allow efficient lookup of the neighbor list of a given vertex. Most likely, this would involve an adjacency list. However, this approach gives us some performance in the line 7 for-loop, but at the expense of the later step of contraction because now when we contract the graph, we'd need to not only construct a new edge list, we'd also need to construct a new adjacency list. This is very expensive.

Therefore, we decided to shift to an edge-centric approach where we use an auxiliary array to store the minimum weighted edge out of any particular vertex. We would map the edges onto different CUDA threads by letting each thread in a CUDA block take an edge until all the edges have been exhausted by the blocks. For each edge, we'd perform an "atomicMin" operation for its endpoints so that across all edges, we'll be able to find the minimum edge out of all the vertices (i.e. the vertex bridges).

Furthermore, the DSU is likely to become a bottleneck in this algorithm, so we used a wait-free DSU that uses atomic CUDA operations [2]. This lowers the synchronization cost associated with concurrent accesses to the DSU. Contention is expected to be happen, but a lot less if we have a large number of threads because for each edge, we're typically only accessing the two DSU array entries corresponding to the endpoints of the edge. Therefore, contention is low when different threads are working on edges that are not incident to each other.

The wait-free DSU pseudocode is as follows:

```
1  make_set(v) {
2      dsu[v] = CREATE_RECORD(0, v)
3  }
4
5  find(v) {
6      record = dsu[v]
7      while (v != GET_NEXT(record)) {
8          new_par = GET_NEXT(dsu[GET_NEXT(record)])
9          new_record = SET_NEXT(record, new_par)
10         if (record != new_record) {
11             atomicCAS(&dsu[v], record, new_record)
12         }
13         v = new_par
```

```
14          record = dsu[v]
15      }
16      return v
17  }
18
19  update_root(u, oldrank, v, newrank) {
20      record = CREATE_RECORD(oldrank, u)
21      new_record = CREATE_RECORD(newrank, v)
22      return atomicCAS(&dsu[u], record, new_record) == record
23  }
24
25  join(u, v) {
26      while (1) {
27          u = find(u)
28          v = find(v)
29          if (u == v) {
30              return
31          }
32          ur = GET_RANK(dsu[u])
33          vr = GET_RANK(dsu[y])
34          if (ur > vr || (ur == vr && u > v)) {
35              swap(ur, vr)
36          }
37          if (!update_root(u, ur, v, vr)) continue
38          if (ur == vr) {
39              update_root(v, vr, v, vr + 1)
40          }
41          break
42      }
43  }
```

Listing 4: wait-free DSU pseudocode

In the pseudocode above, there's the notion of a "record", this is simply a 64-bit integer representation of the rank at a node and the parent vertex number of that node (which we call "next"). This is so that we're able to use the atomicCAS operation on 64-bit integers to atomically manipulate both the rank and the next (or parent) "pointer" at the same time.

After enabling the parallelization techniques described above, here's the resulting pseudocode:

```
1
2  min_weight := an array that stores the minimum edge weight
3                out of each vertex v at min_weight[v]
4  dsu := an array of records that represents the DSU data structure
5  E := an array of edges in the graph
6  is_mst_edge := an array of size |E| tracking whether or not each edge is in the MST
7
8  // initialization
9  do in parallel for each vertex v:
10     min_weight[v] = LLONG_MAX
```

```
    make_set(v)

do in parallel for each edge e:
    is_mst_edge[e.id] = false

// computation

while (|E| != 0) {
    // compute vertex bridges
    do in parallel for each edge e {
        atomicMin(&min_weight[e.from], e.w)
        atomicMin(&min_weight[e.to], e.w)
    }

    // union together the vertices that are connected by vertex bridges
    do in parallel for each edge e {
        if (min_weight[e.from] == e.w) || (min_weight[e.to] == e.w) {
            join(e.from, e.to)
            is_mst_edge[e.id] = true
        }
    }

    // remap all edges
    do in parallel for each edge e {
        e.from = find(e.from)
        e.to = find(e.to)
        // reinitialize the weights for next iteration
        if (e.from != e.to) {
            min_weight[e.from] = min_weight[e.to] = LLONG_MAX
        }
    }

    filter in parallel such that we remove all edges where e.from == e.to
}
```

Listing 5: edge-centric approach pseudocode

Throughout the CUDA implementation, we use "sparse/large.txt" and "dense/large.txt" to tune the various approaches. The tuning is conducted on a Tesla V100 on a p3.2xlarge AWS instance. We measure the speedup relative to our fast sequential baseline.

We measured the performance of the implementation so far, and the above code gave us a computation speedup of 32.71x on "dense/large.txt" and and a speedup of 44.64x on "sparse/large.txt".

We continued to iterate on this approach in an attempt to achieve even better speedup. We first tried to time each section of the code to see what part is taking up the most time and what we're bottlenecked by.

Below is the time that each part of the algorithm takes when run on "dense/large.txt":

Parallel Operation	Time Spent (ms)
computing vertex bridges	23.3905
joining together disjoint sets	1.4795
re-mapping edges	14.4738
filtering edges	4.8850
other	0.0544
Total	44.2832

Below is the time that each part of the algorithm takes when running on "sparse/large.txt":

Parallel Operation	Time Spent (ms)
computing vertex bridges	31.5371
joining together disjoint sets	24.9202
re-mapping edges	38.0717
filtering edges	10.2718
other	0.0522
Total	104.8529

Looking at those two tables, it appears that much of the costs appear in computing the vertex bridges or re-mapping the edges. Additionally, the edge filter step also incurs some cost. The fact that the re-mapping step takes a long time might suggest that our DSU operation has too much overhead. In addition, the time spent in each of those regions appears to be quite even for at least the sparse graphs. This suggests that we have CUDA launch overhead across all of those steps.

Therefore, we thought of several optimizations that we tried to iterate on this approach:

Optimization 1: Try different block sizes in the kernel launches. Our experience with this optimization is that the block sizes don't matter too much from block sizes of 512 to block sizes of 1024. The performance difference is minimal, although it seems to be better sometimes with a block size of 512, so we decided to use 512 as our block size. This is possibly because 512 provides slightly better load balancing.

Optimization 2: Shrink the number of kernel launches by weaving the "filtering" step into other kernel launches. This requires us to maintain a separate auxiliary work list for storing the new edge list after filtering [3]. In addition, we'd make use of the "atomicAdd" operation to resolve contention when different threads want to concurrently add to the new edge list. We then swap the two lists after the filtering is done. This cuts down the number of kernel launches from 4 launches per iteration to 3 launches per iteration. The pseudocode for this is provided below:

```
1 min_weight := an array that stores the minimum edge weight
2               out of each vertex v at min_weight[v]
3 dsu := an array of records that represents the DSU data structure
4 E := an array of edges in the graph
```

```
wl := an array of size |E| that helps with storing newly filtered edges.
wl_len := the length of the worklist
is_mst_edge := an array of size |E| tracking whether or not each edge is in the MST

// initialization
wl_len = 0

do in parallel for each vertex v:
    min_weight[v] = LLONG_MAX
    make_set(v)

do in parallel for each edge e:
    is_mst_edge[e.id] = false

// computation
while (|E| != 0) {
    // compute vertex bridges
    do in parallel for each edge e {
        idx = atomicAdd(&wl_len, 1)
        wl[idx] = {find(e.from), find(e.to), e.w, e.id}
        atomicMin(&min_weight[e.from], e.w)
        atomicMin(&min_weight[e.to], e.w)
    }

    // swap edge list and work list
    swap(E, wl)
    wl_len = 0;

    if (|E| != 0) {
        // union together the vertices that are connected by vertex bridges
        do in parallel for each edge e {
            if (min_weight[e.from] == e.w) || (min_weight[e.to] == e.w) {
                join(e.from, e.to)
                is_mst_edge[e.id] = true
            }
        }

        // reset the min weights array
        do in parallel for each edge e {
            min_weight[e.from] = min_weight[e.to] = LLONG_MAX
        }
    }
}
```

Listing 6: work list approach pseudocode

After including this optimization, our code now achieves a significant better speedup of 45.36x on "dense/large.txt‘ as opposed to 32.71x before the optimization.

Optimization 3: Disable path compression during the "find" operation. This might seem counter-intuitive at first because without path compression, the theoretical worst case of DSU operations becomes $O(\log(n))$ instead of $O(\alpha(n))$. However, this might actually bring us performance improvement since path-compression is not the most memory friendly. Path-compression requires repeated atomic write operations to memory locations, which needs special hardware synchronization mechanisms that might introduce overhead. On the other hand, if we do not perform path compression, then our wait-free DSU is still correct, but there's no longer needs to perform atomic write operations. Regular read operations would suffice. After including this optimization, our code was sped up by a little more again to around 48.87x on "dense/large.txt" (note: actual numbers fluctuate a across runs, but this is consistently better than without the optimization)

Our final approach includes all of the above 3 optimizations.

3.3 OpenMP

The shared address space approach also takes inspiration from Boruvka's algorithm.

Similar to the CUDA approach, it parallelizes via an edge-centric approach, with each thread being allocated multiple edges. This approach consists of 4 components. The first component involves computing the minimum weight outgoing edge for each vertex. These minimum weight outgoing edges are MST edges. The second component uses DSU to join endpoints of MST edges, putting the two endpoints of a MST edge in the same set. The third component maps all vertices to its set representatives. For vertices that are endpoints of a MST edge, this could make the MST edge a self loop, since the set representatives of those vertices are the same. The fourth component naturally follows — it filters out all self loop or MST edges.

Here is the pseudocode for the shared addresss space approach:

```
min_weight := an array of length |no. of vertices| that stores the
              weight of the minimum edge out of each vertex
while (there are still exist edges not considered):
    do in parallel for each edge:
        vertex_from, vertex_to = edge
        atomic_min(min_weight[vertex_from], edge.weight)
        atomic_min(min_weight[vertex_to], edge.weight)

    do in parallel for each edge:
        vertex_from, vertex_to = edge
        if (both the min_weight out of vertex_from and vertex_to equal edge.weight):
            merge the set containing vertex_from and the set containing vertex_to
            flag edge as an MST edge

    do in parallel for each edge:
        vertex_from, vertex_to = edge
        find the set representatives for vertex_from and vertex_to
        update edge to use the set representatives as its endpoints
        if vertex_from and vertex_to come from different sets, reset the weight
        of the minimum edge out of both vertices (i.e. set to infinity)

```

```
22      do in parallel for each edge:
23          filter out the edge if its endpoints are the same i.e. it is a "self-loop"
24
25      set the number of edges that are yet to be considered to be the no. of edges
26      left after filtering
```

Listing 7: Shared address space approach pseudocode

The time taken for the each component of the algorithm when run on dense and sparse graphs is shown below:

Parallel Operation	Time Spent (ms)
computing the min edge out of every vertex	719.2
joining together disjoint sets	1262.6
mapping vertices to their representatives	4409.2
filtering out edges that are self loops	621.7

Figure 1: Time taken for Various Parallel Operations on sparse/large.txt for OpenMP approach

Parallel Operation	Time Spent (ms)
computing the min edge out of every vertex	88.5
joining together disjoint sets	96.5
mapping vertices to their representatives	536.5
filtering out edges that are self loops	726.0

Figure 2: Time taken for Various Parallel Operations on dense/large.txt for OpenMP approach

From Figure 1, we can see that the main bottleneck for sparse graphs is the mapping of vertices to their representatives. This is to be expected as sparse graphs have a considerably lower number of edges than the maximum, which means that operations that have to be done on vertices will take up the largest part of the cost.

From Figure 2, we can see that there exist two major bottleneck, namely the mapping of vertices to their representatives and filtering out self loops. Thus, we outline the following optimizations in an attempt to address these bottlenecks.

Optimization 1a: Parallelize filtering of self-loop edges. The original approach sequentially filters out edges that represent a self loop by checking the if an edge is a self loop and adding it to a result vector. The main impediment to parallelizing this is contention among the threads to update the result vector with the filtered edges. The solution to this is inspired from the ticket lock. Similar to the ticket lock, we propose that each thread gets an index into the result vector by atomically incrementing an atomic index variable. Consequently, each thread gets a unique index into the resulting vector, where it can write the filtered edge to. As opposed to a single thread holding on to a lock for the entire result vector for every write, this allows multiple threads to write concurrently to their corresponding index in the vector once they get their unique index.

Optimization 1b: Parallelize the process of populating the vector of MST edges. The original approach populated the vector of MST edges sequentially based on the boolean array of whether an edge is an MST edge. This optimization is similar to optimization 1a, where there is an atomic index that each thread increments to get an index into the MST edge vector.

Optimization 2: Combining operations. As shown in 7, the original approach has 4 inner for loops within the while loop, each of which does different parallelizable operations. This can be reduced to 3 inner loops.

To do so, the filtering of the self-loop edges in the 4th loop can be instead done within the 1st loop, which is in charge of finding the minimum outgoing edge from each vertex. To maintain correctness, updating the edge's endpoints with the set representatives of the endpoints also has to be moved to the 1st loop. As a result, the 1st loop now does mapping of endpoints to set representatives, filtering out self-loops and computing the minimum outgoing edge of every vertex. The 2nd loop still does disjoint set union while the 3rd loop resets the weight of the minimum weight out of the endpoints from different sets.

Reducing the number of parallel regions could reduce the overhead of launching new threads for the region, hence reducing computation time.

Final Approach We decided to use optimizations 1a and 1b in our final approach. We chose these two optimizations as they resulted in the best performance after ablation and benchmarking. We also used dynamic assignment with a chunk size of 1024 after benchmarking.

Optimization 3: Removing path compression. Path compression is implemented in the `find` function, which is a function that finds the set representative of a vertex. Path compression involves an atomic CAS operation to update the DSU data structure (described in 2.7). However, this involves updating the memory of a shared data structure. This can take a significant amount of time if there are a large number of threads attempting to do so at the same time, and the thread has to spin until the atomic update is successful. The savings from compressing a path to save time on future calls to `find` may be outweighed by the atomic update to memory. Hence, removing path compression may result in a lower computation time.

3.4 MPI

We attempted to use the message passing paradigm for MST computation. We started again by trying to do something similar to Boruvka's or Kruskal's algorithm by requiring a disjoint set data structure to be maintained across the different approaches. The problem is that unlike a shared address space approach, here we lost the ability to have access to the same disjoint set data structure, so we need to communicate that information in some ways. Our initial approach involves broadcasting the entire edge list to all compute nodes. We would partition work on the edges across all the nodes using static interleaved assignment. For each compute node, we compute locally, the minimum weighted edge that each node is incident to. The observation is that we can treat the minimum weighted edges of all the vertices as a single vector of length D (call it `local_min_edge`), where D is the total number of disjoint sets that are currently present in the graph. `local_min_edge[i]` would store the minimum weighted incident edge for the vertex i our of all the edges that are assigned to that compute node. To get the overall globally minimum weighted incident edge, we just need to do an `AllReduce` across all the compute nodes. After that, each compute node would modify its local disjoint sets according to the minimum edges found and

contract the graph locally.

The pseudocode for this approach is presented below:

```
1
2  // initialization
3  Broadcast G = (V, E) to all processors
4
5  Processor i is assigned edges j where j % nproc == i
6
7  local_min_edge := array storing the local min edge vector
8  global_min_edge := array storing the global min edge vector
9  rep_idx := array index by vertex number of disjoint set representative,
10           and the value is the index to use on local_min_edge and
11           global_min_edge to find the vertex bridge corresponding to
12           that representative
13 mst := list of MST edges
14
15 for each vertex v {
16     make_set(v)
17 }
18
19 // computation
20 do {
21     initialize the local_min_edge entries to LLONG_MAX
22     for each edge e that the processor is in charge of {
23         if (find(e.from) != find(e.to)) {
24             idx = rep_idx[find(e.from)]
25             if (e.w < local_min_edge[idx]) {
26                 local_min_edge[idx] = e;
27             }
28         }
29     }
30
31     AllReduce(local_min_edge, global_min_edge)
32
33     for i = 0..<dsu_size {
34         e = global_min_edge[i]
35         if (join(e.from, e.to)) {
36             mst.add(e)
37         }
38     }
39
40     find new set of representative, updating dsu_size and rep_idx
41     contracting the graph by updating the list of edges and vertices
42     for this processor
43 } while (dsu_size > 0)
```

Listing 8: Pseudocode for message passing with disjoint set all reduce

When we tried to benchmark this algorithm, we realized that this approach had a significant issue. The code does runs super slow, and that it doesn't scale well at all. On large graphs such as 'dense/large.txt', the approach runs a 16GB computer out of memory if we open up 8 processors. This is because this approach needs a copy of the entire graph for each processor. Therefore, the total memory requirement does not scale as the number of processors and if we're doing this on a single machine, we quickly run out of memory.

Significant effort was then put into investigation of a separate approach using the Gallager, Humblet, and Spira (GHS) algorithm, which is a fully distributed way of computing a MST. In an ideal world, the GHS algorithm would be used when each processor is responsible for one graph vertex and its neighbors. However, because we need to make it work in a setting where the total number of processors might be much smaller than the total number of graph nodes, some modification is needed for MPI.

The GHS algorithm is too long to include in this report, so we'll leave out the pseudocode. There's an explanation of what the algorithm does in this Wikipedia article, or in the original GHS paper [4].

However, to summarize what the algorithm does, one can think of each vertex as a state machine that changes its state or sends messsages out to its neighbors based on the messages that it receives. There's a notion of "fragments". A fragment is a collection of nodes connected by MST edges. Similar to Boruvka's algorithm, every node starts out as its own fragment, and they make connection requests to adjacent fragments via the minimum weighted edge out of that fragment. Over time, the fragments merge together into larger fragments until eventually the algorithm converges and the entire graph is a single fragment, after which no more messages are sent. Messages are broadcasted along the branch edges of every fragment in order to determine the minimum edge out of every fragment and to ensure that there can't be concurrent connection requests sent out of one fragment, which would result in race conditions.

To implement this in MPI, each node in the GHS algorithm can be thought of as a logical processors, we'll present a version of the pseudocode below that treats the state machine of each node as a blackbox and it would be given by "peformLoop(node)".

```
1
2  // initialization
3  Processor 0 partitions the vertices to all processors.
4  Processor i is assigned vertex j where j % nproc == i
5
6  do in parallel:
7      Each processor initializes the nodes that it is assigned by
8      initializing the node state and the neighbor list of each node.
9
10 // computation
11 do in parallel:
12     while (timer not expired) {
13         poll messages and update timer if message is received
14         while (message queue is not empty()) {
15             for msg in message queue {
16                 node = msg.dst
17                 performLoop(node, msg)
```

```
18                }
19            }
20        }
```

In the above code, due to the fact that we have fewer processor than the number of graph vertices, we make each processor "simulate" the vertices that it's in charge of. Therefore, when we receive some message, we would only need to process the message for the vertex that the message is targeting.

When a vertex wants to send a message to another vertex, the processor would first check to see whether or not the destination vertex belongs to the same processor. If so, it just puts the message onto its own message queue without using an `MPI_Isend`. Otherwise, it calls `MPI_Isend` to send it to the processor that the vertex belongs to.

The above code achieves linear self-speedup at least when it was run on the GHC machines during exploration of different approaches. However, its performance is poor when compared to the fast sequential solution. However, this is mostly expected because our MPI approach comes with lots of overhead just to make the algorithm fully distributed. For the MPI approach, we're much more concerned about self-speedup and how well the approach would scale when the number of processors increase rather than the absolute time (since we're operating under the assumption that we're memory constrained).

However, we realized that the absolute time can still be improved by cutting down computation cost at each processor (the communication cost represents a very small fraction of the total time). Three optimizations are attempted below:

Optimization 1: We were wondering whether or not maintaining a per-vertex message queue would be more efficient than maintaining a per-processor message queue, so we tried both approaches and looked at the execution time. It turns out that the performance difference is minimal. Since the per-processor message queue is more memory efficient, we ultimately chose to use it instead of a per-vertex message queue.

Optimization 2: Inside of "performLoop", we needed to lookup the index of the edge where the received message came from. This was originally done by doing a linear search through the neighbor list. However, this is then improved to be $O(1)$ by maintaining a mapping of neighbor node number to neighbor index. This is advantageous for dense graphs at the cost of extra memory

Optimization 3: Vertices often times need to send out TEST messages along the first BASIC edge (a BASIC edge is an edge that we currently don't know whether or not it belongs to the MST or not) in its sorted neighbor list. Originally, this was also done via a linear search, but we used a the C++ set interface instead to make the search $O(\log(n))$. Again, this is an optimization for dense graphs. For sparse graphs, this comes at the cost of extra memory overhead.

Our final approach contains both Optimization 2 and Optimization 3.

4 Results

4.1 Test Files Descriptions

The test files that we used throughout are separated into two sets. One is called "sparse" and the other is called "dense". The sparse graphs typically have the number of edges being proportional to the number of vertices whereas the dense graphs typically have the number of edges being almost a perfect square of the number of vertices.

The graphs are randomly generated using a script. The different test files are used to evaluate how our algorithms perform across various problem sizes. Below is a table summarizing the sizes of the different graphs in the different test files:

Test File	Number of Nodes	Number of Edges
dense/small.txt	200	17,847
dense/medium.txt	1,000	449,576
dense/large.txt	5,000	11,245,624
dense/extreme.txt	8,000	28,793,554
sparse/small.txt	100,000	149,960
sparse/medium.txt	1,000,000	1,501,216
sparse/large.txt	10,000,000	15,000,571
sparse/extreme.txt	20,000,000	29,998,467

4.2 Baseline

Here included is the raw execution time for the baseline Kruskal's implementation on the different test files

Test File	Initialization Time (ms)	Computation Time (ms)
dense/small.txt	0.307383	1.162513
dense/medium.txt	8.363402	38.440034
dense/large.txt	201.563059	1230.631403
dense/extreme.txt	522.185191	3347.270322
sparse/small.txt	0.973502	16.733151
sparse/medium.txt	6.11955	217.514365
sparse/large.txt	74.7529	4504.298113
sparse/extreme.txt	588.866202	9599.326241

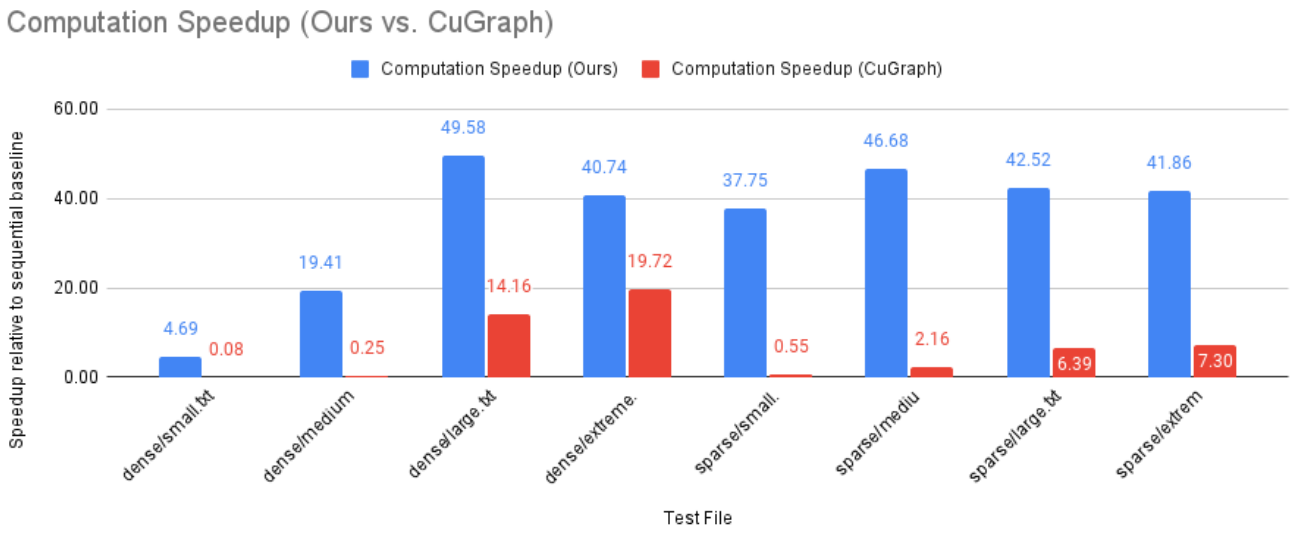
4.3 CUDA

Listed as a possible target goal in our original project proposal, we also investigated the use of the CuGraph library to compute a MST. The CuGraph library comes with a built-in GPU-accelerated minimum spanning tree implementation which can be invoked via `cugraph.minimum_spanning_tree`, so we also included the evaluation of that result here to compare against the results that we achieved for the CUDA implementation.

Below is a table summarizing the raw execution time for both our final CUDA approach and the CuGraph built-in MST library when run on a Tesla V100 on a p3.2xlarge AWS instance.

Test File	Computation Time (ms) (Ours)	Computation Time (ms) (CuGraph)	Computation Time (ms) (Baseline)
dense/small.txt	0.25	14.67	1.16
dense/medium.txt	1.98	155.72	38.44
dense/large.txt	24.82	86.91	1230.63
dense/extreme.txt	82.16	169.71	3347.27
sparse/small.txt	0.44	30.60	16.73
sparse/medium.txt	4.66	100.51	217.51
sparse/large.txt	105.92	704.70	4504.30
sparse/extreme.txt	229.34	1315.63	9599.33

And here is a bar-chart for the speedup of both our final CUDA approach and the CuGraph algorithm relative to the sequential baseline:



Discussion and Limitations: Looking at the figures above, we think that we've reached the goals we set out to achieve for the CUDA implementation of MST algorithms. It appears that performance is the best when the problem size is very large (i.e. for "sparse/large.txt", "sparse/extreme.txt", "dense/large.txt" and "dense/sparse.txt"). All of those graphs have the number of edges in the order of tens of millions. This makes sense because CUDA launch overheads can be significant if the graphs size is small. Furthermore, the block size we chose is 512, and on smaller graphs, this would not be the best granularity since it would be too large to provide good load balancing across the various blocks.

Comparing dense graphs with sparse graphs, it appears that the algorithm is agnostic to the density of the graphs, and performs equally well on both types of graphs. This is in agreement with the

fact that our CUDA approach is edge-centric, meaning that it's runtime is mostly a function of the number of edges rather than the number of vertices. Graphs with the same size names have roughly similar number of edges (e.g. "dense/large.txt" and "sparse/large.txt" both have around 10-20 million edges), and we can see that their speedups are very similar.

However, there might still be room for improvement for our final algorithm. Below is the breakdown of the amount of time the algorithm is spending in each part of its code for "dense/large.txt" and 'sparse/large.txt":

dense/large.txt		sparse/large.txt	
Parallel Operation	**Time Spent (ms)**	**Parallel Operation**	**Time Spent (ms)**
computing vertex bridges & filtering edges	18.7526	computing vertex bridges & filtering edges	55.0012
joining together disjoint sets	1.5076	joining together disjoint sets	22.3842
re-mapping edges	8.5958	re-mapping edges	28.8300
other	0.1330	other	0.1818
Total	28.9889	**Total**	106.3972

From the tables shown, we can see that the bottleneck here appears to the the step where we're computing the vertex bridges and filtering out the edges to put into a new worklist. This is especially significant for dense graphs.

The step in question contains many CUDA atomic operations [1] such as "atomicMin" and "atomicAdd", which can become a bottleneck if memory accesses on the same location is frequent. This is especially the case for denser graphs because each node have higher degrees, so given a memory location that stores the minimum edge weight out of some vertex, that location will be operated on much more frequently by the "atomicMin" operation than it would've been if the graph is sparse. This observation agrees with the numbers presented in the above tables. Furthermore, the "atomicAdd" operation on the memory location the maintains the worklist length is effectively serialized across all "atomicAdd" operations. This can present large synchronization overhead if the number of edges filtered out is small.

Given the observations above, future areas of improvements includes investigating schemes that can contract more edges each round. Not only will this reduce synchronization overhead, it will also reduce the total number of rounds that the algorithm will need to run. The rounds depend on each other, reducing the number of rounds will greatly increase the amount of parallelism.

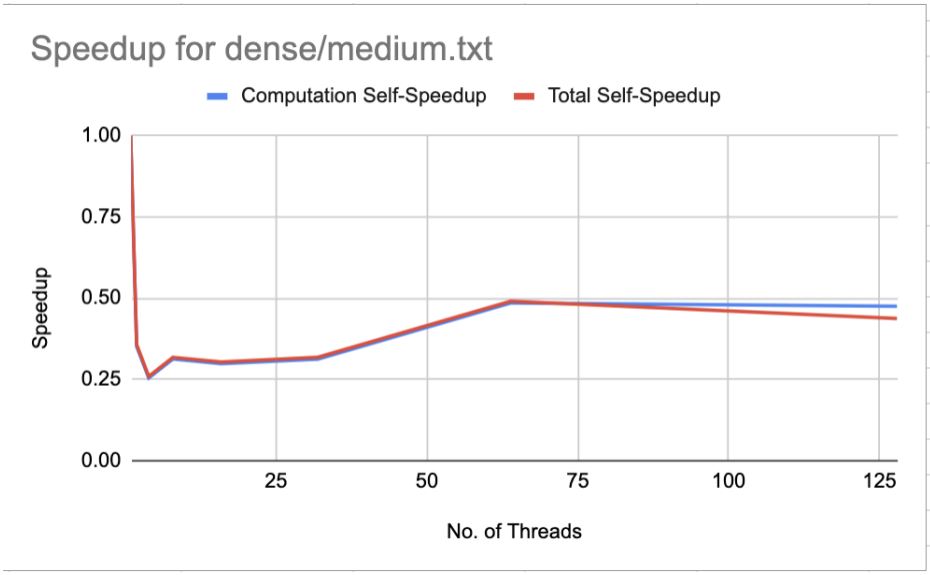
Moreover, the "atomicMin" operation bottleneck is somewhat unique to the edge-centric approach. If we were instead vertex-centric, then by parallelizing across vertices, each vertex can look at its neighbor list in parallel and pick the minimum weighted edge without having to access the same memory location as another vertex (if we map vertices to CUDA threads, then this means that we'd reduce sharing across threads). Then we'd be more friendly on GPU memory hardware.

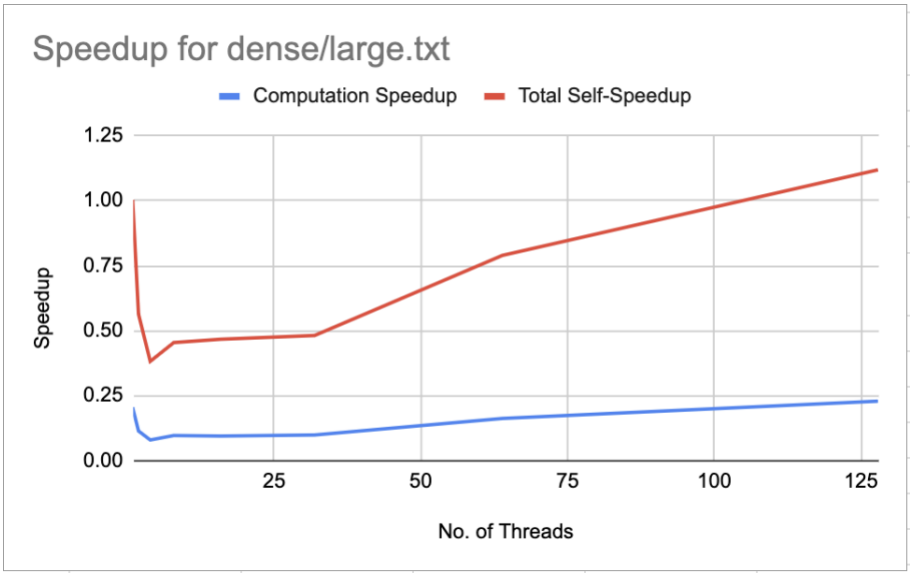
4.4 OpenMP

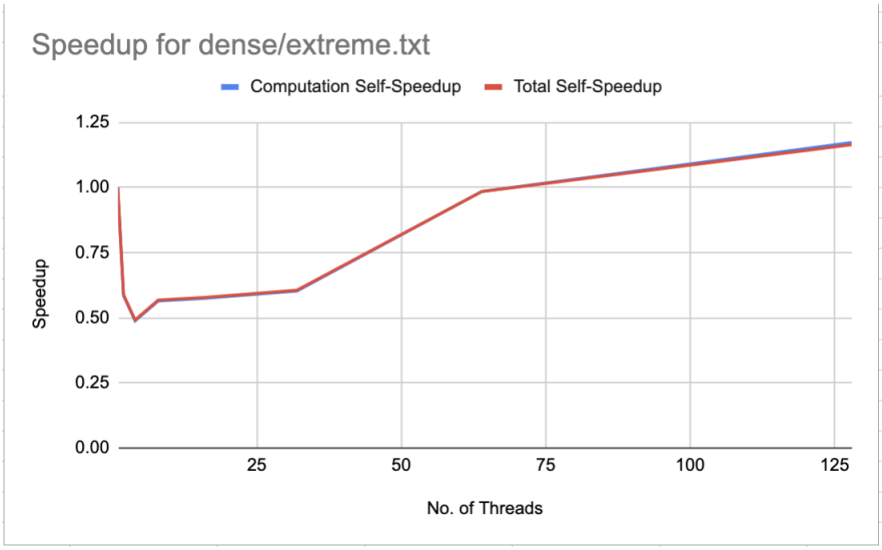
We measured the initialization and computation time using 1 to 128 number of threads. The times were measured for different problem sizes. In particular, we have measurements for medium, large

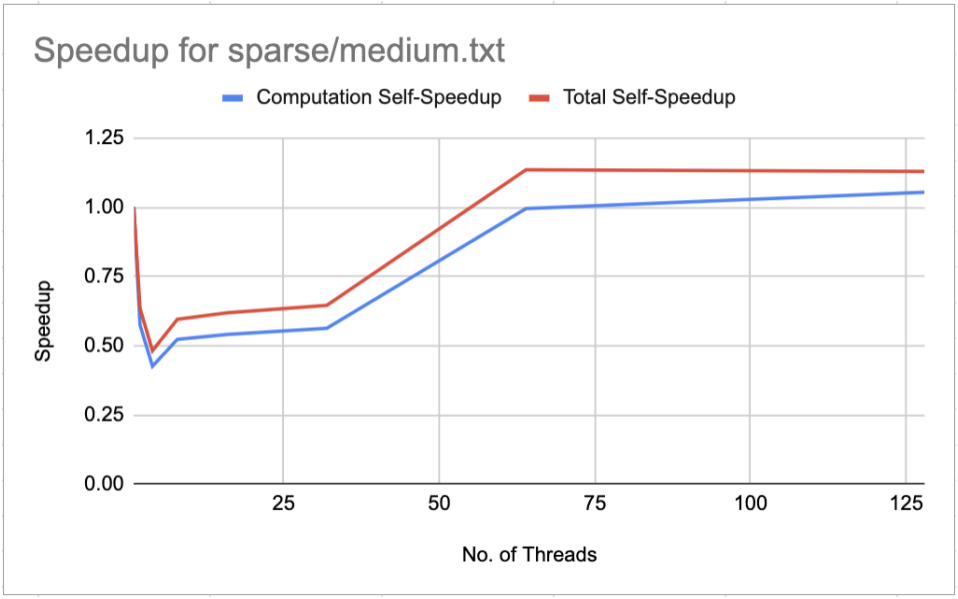
and extreme graphs and also sparse and dense graphs.

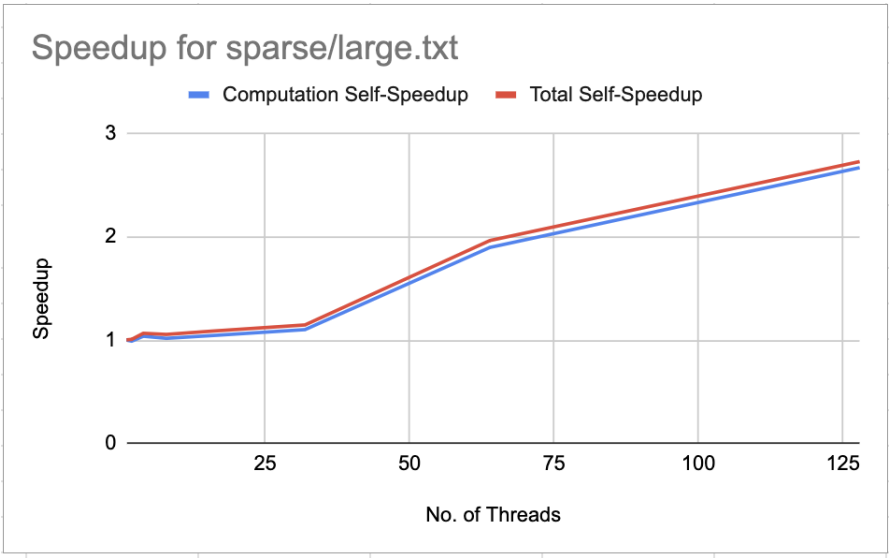
Based on the initialization and computation time, we then computed speedup (computational, total) relative to a baseline of an optimized parallel implementation for a single CPU.

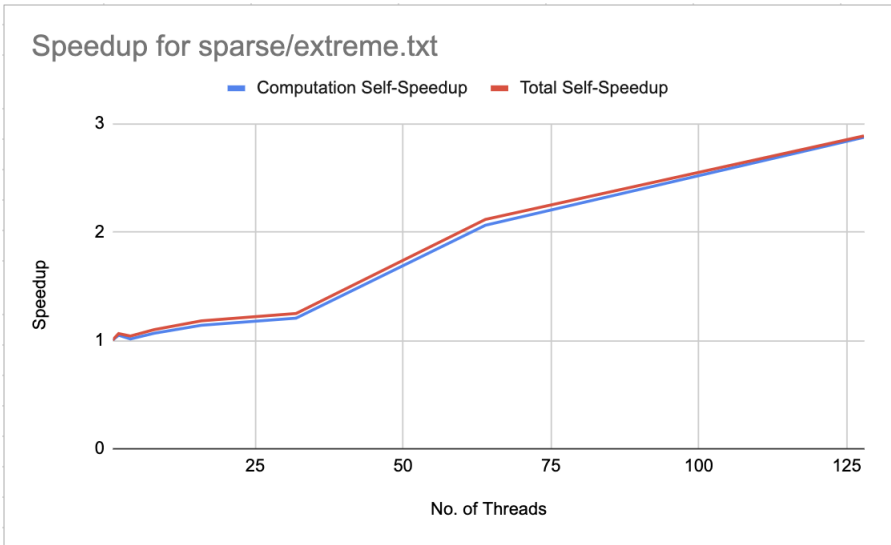





Speedup for dense/extreme.txt
Computation Self-Speedup
Total Self-Speedup
1.25
1.00
0.75
0.50
0.25
0.00
Speedup
25
50
75
100
125
No. of Threads


Speedup for sparse/medium.txt
Computation Self-Speedup
Total Self-Speedup
1.25
1.00
0.75
0.50
0.25
0.00
Speedup
25
50
75
100
125
No. of Threads


Speedup for sparse/large.txt
Computation Self-Speedup
Total Self-Speedup
3
2
1
0
Speedup
25
50
75
100
125
No. of Threads



Discussion and Limitations: We observe mostly an increasing trend in the total and computational speedup with increasing number of threads. In particular, we observe sublinear speedup for sparse large and sparse extreme graphs. However, for the other types of graphs, we actually experience a fractional speedup, i.e. the algorithm gets slower with increasing number of threads.

Parallel Operation	Time Spent (ms)	Percentage Time (%)
computing the min edge out of every vertex	91.2	7.062822537
joining together disjoint sets	92.9	7.192663048
mapping vertices to their representatives	535.4	41.44635407
filtering out edges that are self loops	376.3	29.13054729

Figure 3: Time taken for Various Parallel Operations on dense/large.txt for final approach

Parallel Operation	Time Spent (ms)	Percentage Time (%)
computing the min edge out of every vertex	708.5	9.486651288
joining together disjoint sets	1283.1	17.18126628
mapping vertices to their representatives	4399.0	58.90451241
filtering out edges that are self loops	344.1	4.607597134

Figure 4: Time taken for Various Parallel Operations on sparse/large.txt for final approach

These trends can be explained by the presence of dependencies in the parallel regions, that hindered the reduction in computational time. From Figure 3 and Figure 4, we can see that the main bottlenecks for dense graphs are the mapping and filtering operations (taking up 70.58% of total time), while that for sparse graphs is primarily the mapping operation (taking up 58.90% of total time).

Hence, we direct our attention to the operation to map all vertices to their set representatives. In the algorithm, this operation is parallelized across edges. For each endpoint of each edge, `find` is called to find the representative for that endpoint. `find` utilizes path compression, which requires an `atomic_compare_exchange` operation to atomically modify the element corresponding to the endpoint in the DSU data structure. However, as each vertex can have multiple edges incident to it, the `atomic_compare_exchange` can quickly become a bottleneck, as each thread is only successful at updating after several attempts. This is especially an issue for dense graphs, where each vertex has a large number of edges incident to it. This is supported by our results above, where dense graphs of all sizes exhibit fractional speedup.

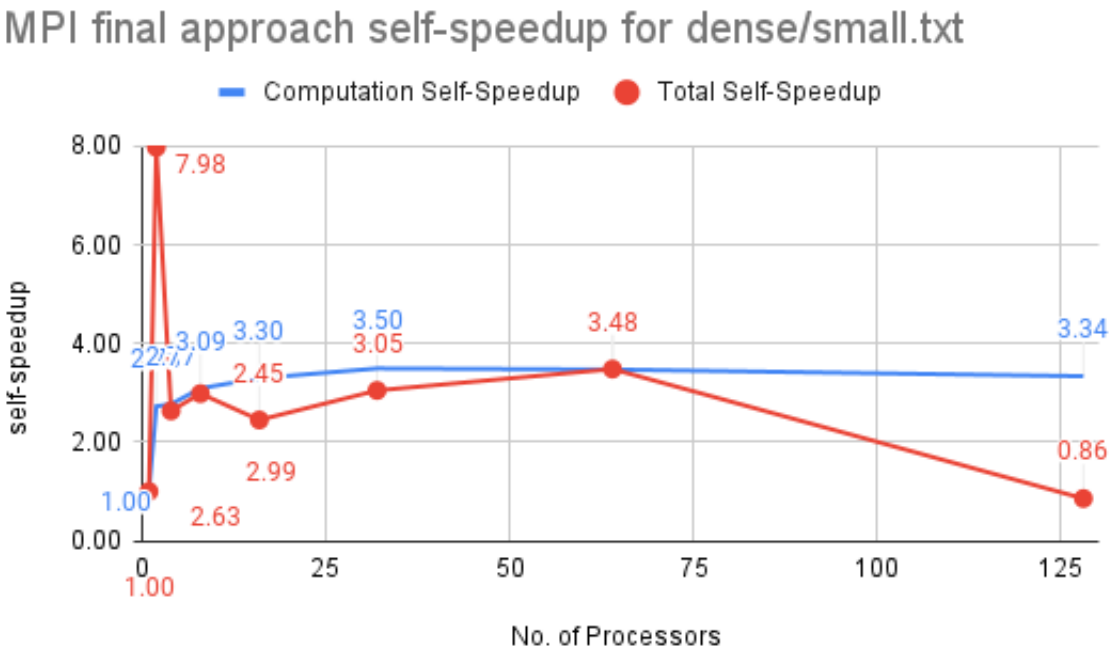
Next, we examine the next biggest bottleneck on dense graphs: the filtering operation. As described in Section 3.3, the filtering operation uses an atomic index and each thread will increment that index to obtain a unique index into the result array it needs to write to. We note that outside of incrementing the atomic index, the operations are to check if an edge is a self loop and writing the edge to the new edges array. As we are using a single machine, these are not very time intensive operations, which means that the bulk of the time would be spent on incrementing an atomic index. This synchronization overhead would significantly add to the computation time.

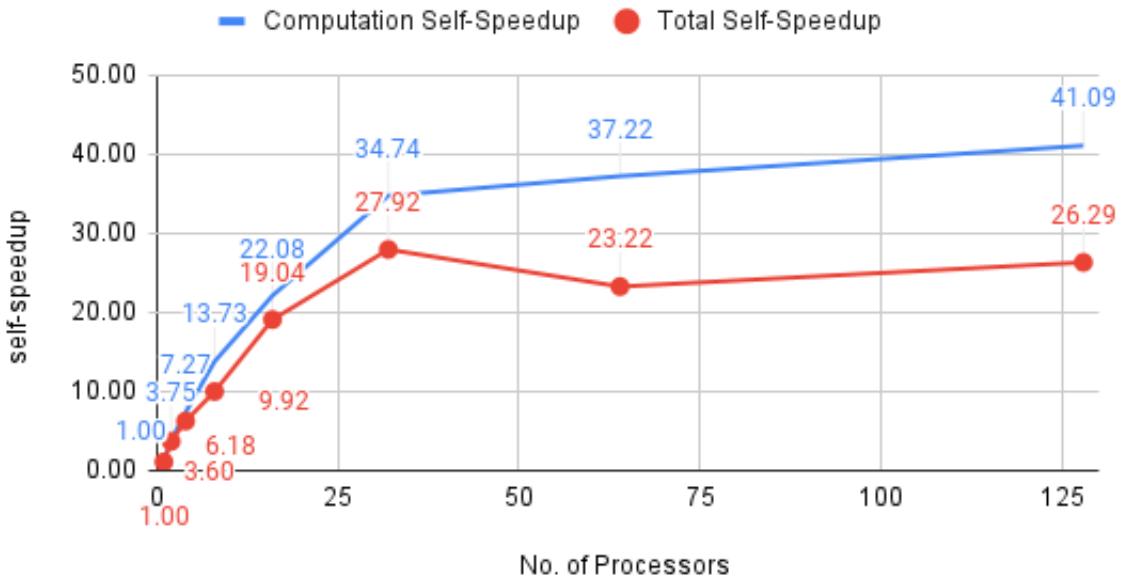
Due to synchronization overheads in the parallel regions of the code, the gains from more threads is sometimes outweighed by the costs introduced by more threads. Hence, we observe that our speedup falls below linear speedup.

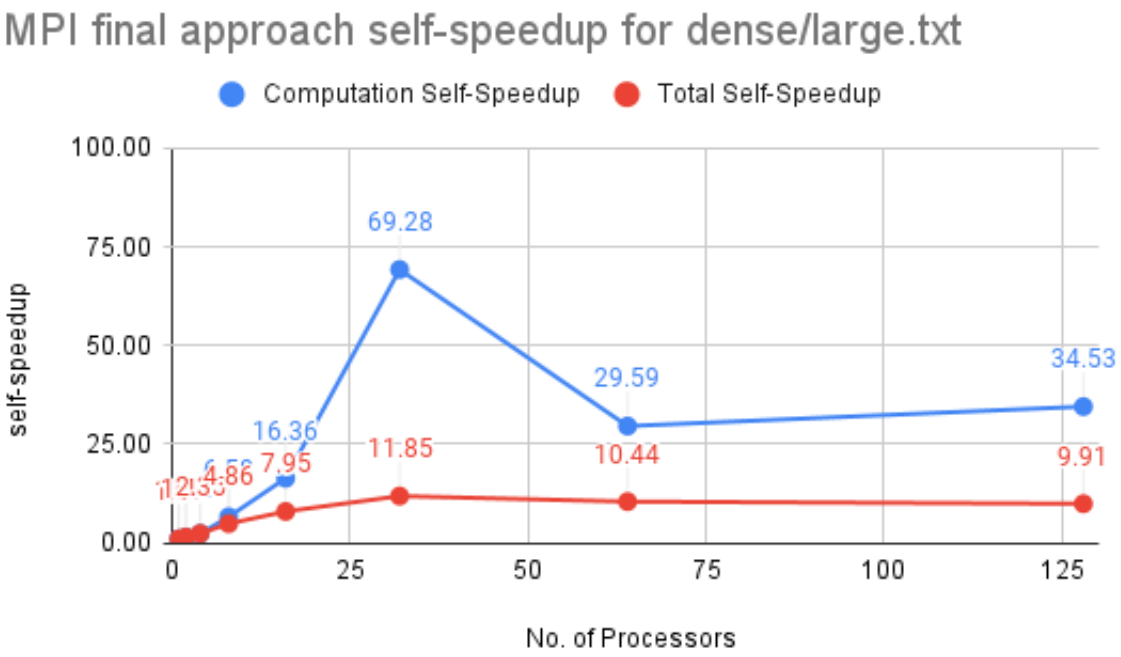
We hypothesize that the issue of contention due to a vertex having multiple edges by switching to a vertex-centric approach. This would help reduce contention for the atomic operations as each edge only has two endpoints. However, as the edges come in an edge list format, there would be additional initialization cost of converting the edge lists to adjacency lists. We would have to perform more measurements to see if the savings from a vertex-centric approach outweights the increase in initialization cost.

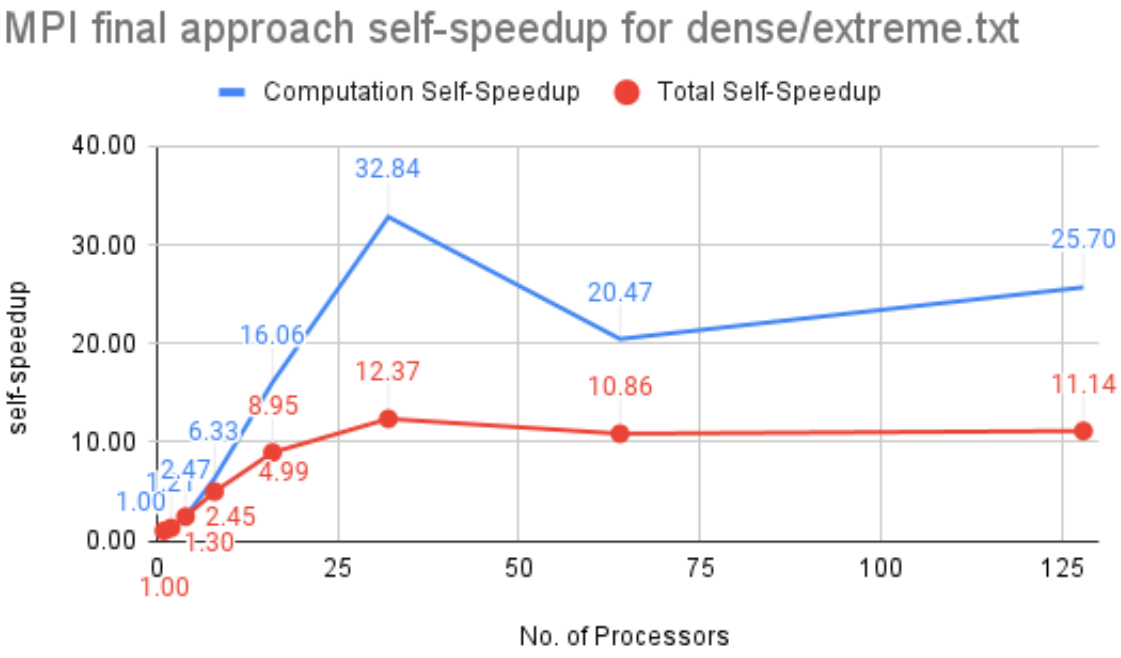
4.5 MPI

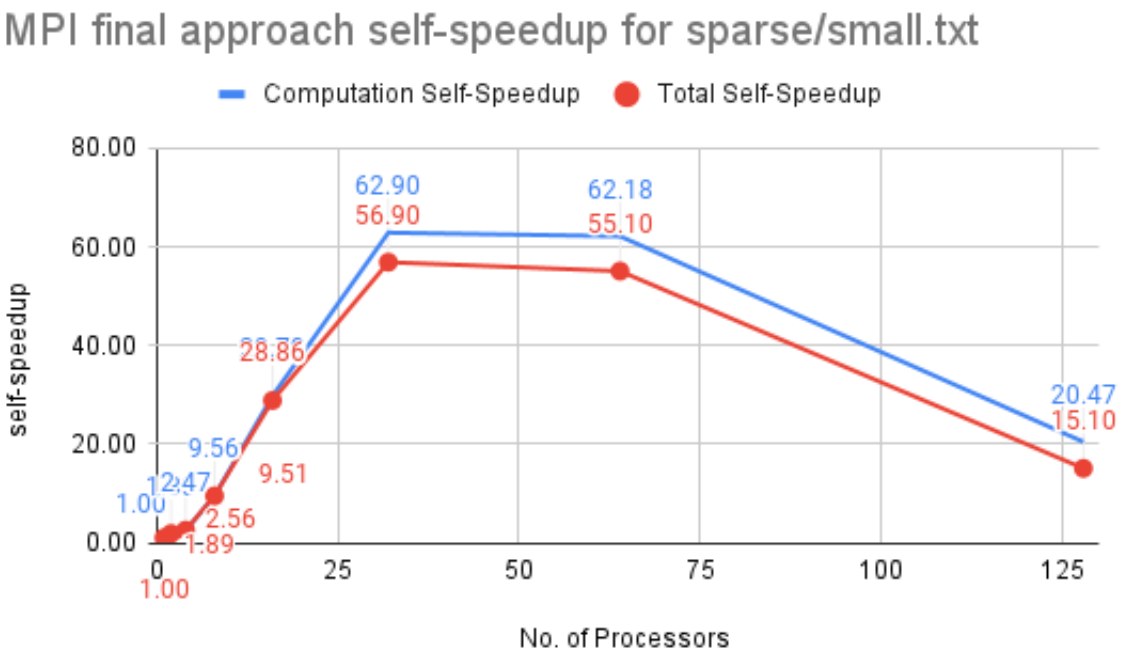
For our MPI approach, we measured self-speedup for the different problem sizes. Below are the graphs for the speedup graphs on the different test files. We did not include the results for "sparse/large.txt" and "sparse/extreme.txt" due to time constraint and the fact that they take way too long to run when the number of processors is low:

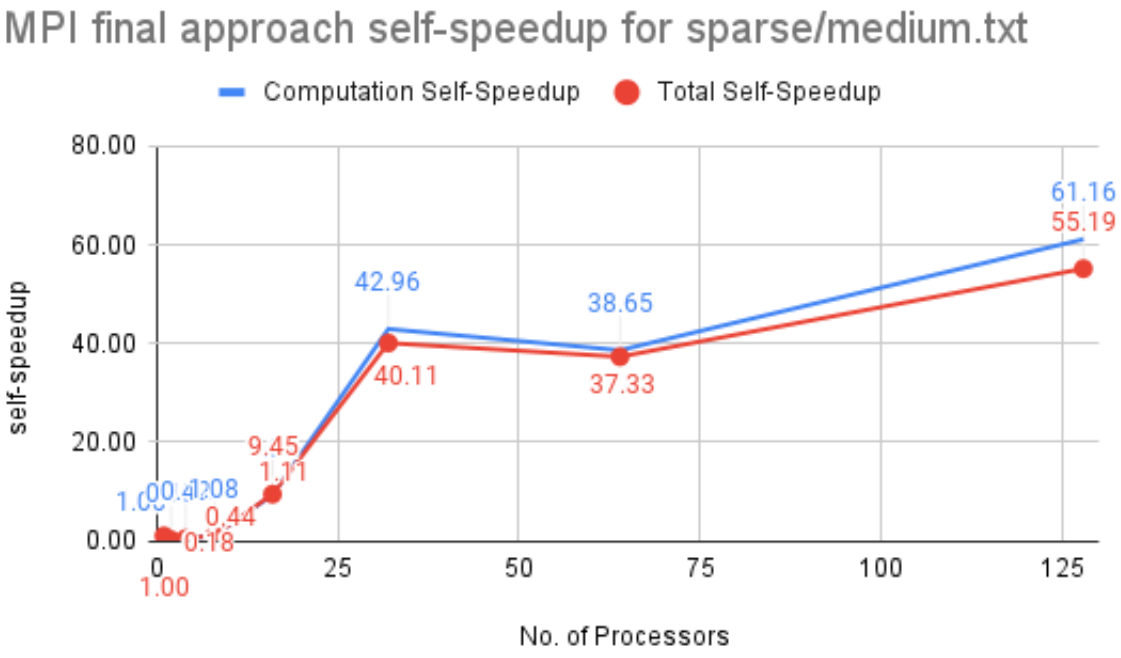

MPI final approach self-speedup for dense/small.txt
Computation Self-Speedup
Total Self-Speedup
self-speedup
No. of Processors
8.00
6.00
4.00
2.00
0.00
0
25
50
75
100
125
7.98
3.09
3.30
3.50
3.05
3.48
3.34
2.45
2.99
0.86
1.00
2.63
1.00


MPI final approach self-speedup for dense/medium.txt
Computation Self-Speedup
Total Self-Speedup
self-speedup
No. of Processors
50.00
40.00
30.00
20.00
10.00
0.00
0
25
50
75
100
125
41.09
37.22
34.74
27.92
26.29
23.22
22.08
19.04
13.73
7.27
3.75
9.92
1.00
6.18
3.60
1.00


MPI final approach self-speedup for dense/large.txt
Computation Self-Speedup
Total Self-Speedup
self-speedup
No. of Processors
100.00
75.00
50.00
25.00
0.00
0
25
50
75
100
125
69.28
29.59
34.53
16.36
7.95
11.85
10.44
9.91
4.86


MPI final approach self-speedup for dense/extreme.txt
Computation Self-Speedup
Total Self-Speedup
self-speedup
No. of Processors
40.00
30.00
20.00
10.00
0.00
0
25
50
75
100
125
32.84
25.70
20.47
16.06
12.37
10.86
11.14
8.95
6.33
4.99
2.47
2.45
1.00
1.00





In addition, here are the raw execution times with varying number of cores:

dense/small.txt			dense/medium.txt		
No. of Processors	**Initialization Time (ms)**	**Computation Time (ms)**	**No. of Processors**	**Initialization Time (ms)**	**Computation Time (ms)**
1	10.489117	41.549604	1	539.051471	932.224139
2	6.521599	15.229248	2	160.143564	248.60789
4	4.748832	15.00211	4	109.951233	128.296498
8	3.995028	13.436632	8	80.38812	67.908293
16	8.631386	12.602518	16	35.038469	42.217568
32	5.178105	11.888085	32	25.853381	26.835815
64	2.978754	11.962198	64	38.309646	25.047261
128	48.228873	12.446047	128	33.277377	22.689796

dense/large.txt			dense/extreme.txt		
No. of Processors	**Initialization Time (ms)**	**Computation Time (ms)**	**No. of Processors**	**Initialization Time (ms)**	**Computation Time (ms)**
1	7765.071472	16015.73659	1	19573.28083	54156.79335
2	4874.773583	11281.63082	2	11879.29117	44791.86186
4	3285.072848	6845.767801	4	8179.061546	21926.18121
8	2461.544224	2435.094853	8	6225.516325	8561.963934
16	2013.648938	978.879078	16	4864.434751	3372.06989
32	1775.844591	231.190456	32	4312.8525	1648.910222
64	1736.604081	541.269363	64	4143.49188	2646.194824
128	1936.397323	463.844409	128	4511.301218	2107.441141

sparse/small.txt			sparse/medium.txt		
No. of Processors	**Initialization Time (ms)**	**Computation Time (ms)**	**No. of Processors**	**Initialization Time (ms)**	**Computation Time (ms)**
1	201.143759	3261.339358	1	2334.765405	56439.83636
2	56.582553	1778.348517	2	718.804368	321373.7825
4	32.428502	1319.62526	4	414.791431	133069.9103
8	22.601374	341.304592	8	355.278786	52392.56643
16	10.474873	109.485856	16	164.076744	6053.844938
32	8.996802	51.850502	32	151.445566	1313.872262
64	10.38752	52.453534	64	114.148893	1460.328897
128	70.010468	159.311249	128	142.084891	922.887002

Discussion and Limitations: We were hoping to achieve around linear speedup using MPI although we did foresee possible challenges associated with MPI communication. Looking at the

results, we did achieve some good self-speedup on large enough graphs, but the speedup graphs occasionally have some abnormal features.

First, the really small graphs like "dense/small.txt" did not have good speedup because the problem size is simply too small that increasing the number of processors only introduces more overhead rather than helping effecively with computation.

For the other larger dense graphs like "dense/medium.txt", "dense/large.txt" and "dense/extreme.txt", we can see that the speedup are quite linear (and even super-linear) as the number of processors increase. This trend typically continues until we hit 32 processors, after which the speedup tapers off or even start to decrease. This trend also seems to occur on sparse graphs "sparse/small.txt" and "sparse/medium.txt".

The super-linear speedup is likely occuring because of a larger aggregate cache size as we increase the number of processors since each processor has its own cache. Therefore, the algorithm as a whole is able to fit more data into caches and this might speedup the algorithm more than linearly.

As for the performance drop at high processor counts, we think that this is occuring due to a combination of workload imbalance and the nature of communication in our algorithm. First of all, we partitioned our graphs based on vertices randomly. It's possible that asymmetry in the graph can result some processors to proceed faster than other processors (i.e. on some processors, more nodes are part of the same fragment). This can be a small factor contributing to the drop-off. However, the main reason should be how the algorithm is communicating. Over the course of the algorithm, communication within each MST fragment is done by broadcasting along the branch edges. Note that because each vertex only has a local view of their neighbors, they're unable to directly send to each vertex in the fragment, but they're only able to pass along the message to their neighbor until it finally propagates through the entire fragment. This process can take a while, especially on large graphs with millions of nodes. It would take on average some millions of hops until a message is broadcast-ed to every node in the same fragment if the fragment had millions of nodes! On high processor counts, the nodes in a fragment are spread across the different processors, which means that just a single broadcast would be causing millions of messages being sent across the different processors. This can be a significant slow down which is why we think that this is ultimately the cause of the performance drop at high processor counts.

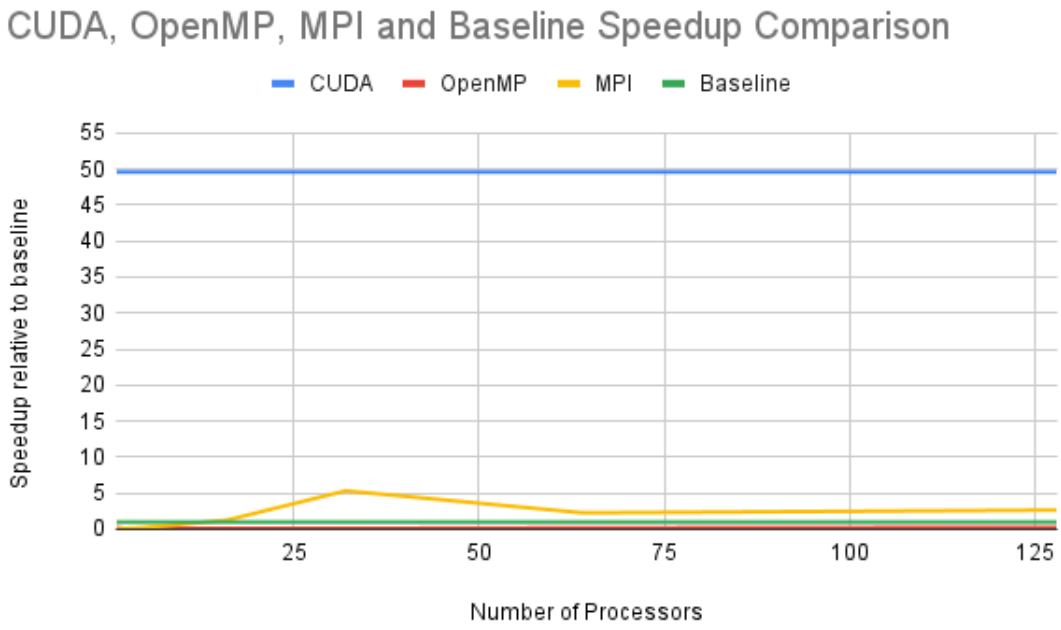
Another note here is that the raw execution time for the MPI approach is a lot higher than the baseline even though the self-speedup is good. This was expected because the main feature of the MPI approach is its low memory overhead and good self-speedup per processor rather than good raw execution time. However, it might still be worth investigating ways to reduce overhead of the MPI approach in the future.

We think that the largest area of improvement would come from more clever ways of sending messages across nodes in the same processor. In this project, we mostly adhered to a fully-distributed MST algorithm. However, consider some node v sending messages to a node u in the same processor. Let's say that v and u belong to the same fragment and the path from v to u is 1 million hops. Then it would be a lot faster if had some local data structure (similar to a DSU) that kept track of whether or not v and u are in the same fragment. If so, we can directly send from v to u rather than going through 1 million extra iterations. This improvement would likely require careful modification of the GHS algorithm so that correctness is not affected, but if this optimization is realized, then it should theoretically speedup the code significantly both on a single processor and also at high processor counts.

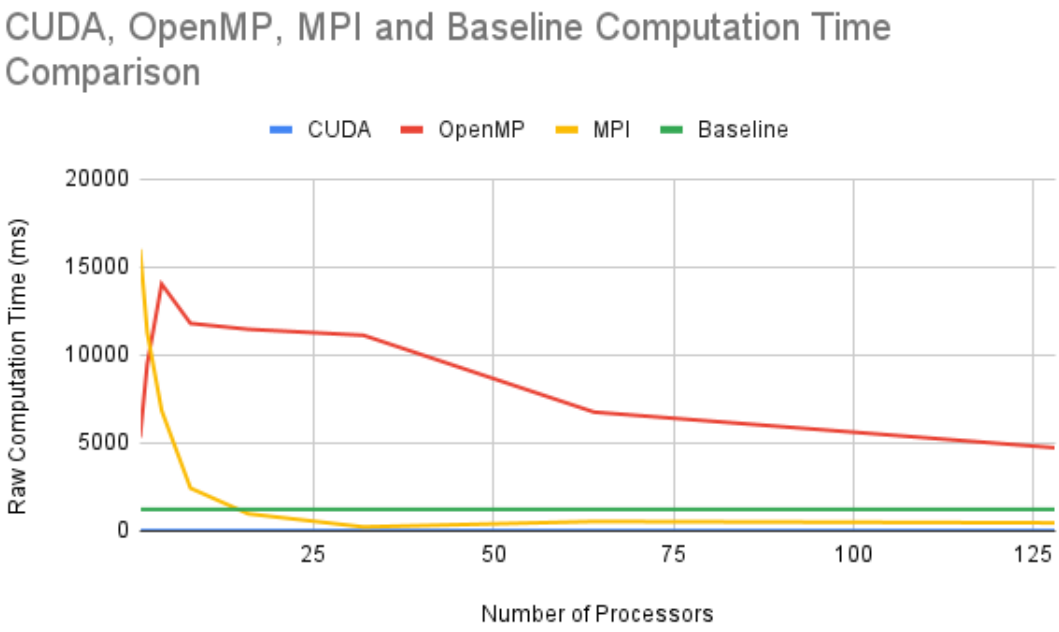
4.6 Cross-Paradigm Comparisons

Here we briefly present how the different paradigms compare to each other when it comes to absolute computation time and speedup. We will use "dense/large.txt" to illustrate the speedup and computation time, but the general trends hold for other large test files as well.

Below is the speedup graph for the different paradigms. The speedup measured here is not the self-speedup, but the speedup relative to the fast sequential baseline:



Below is a graph for the computation time when running algorithms for the different paradigms on "dense/large.txt".



As seen in the figures above, when it comes to raw performance, CUDA seems to be the best performer out of all the platforms that we tried. Surprisingly, OpenMP did not provide an improvement over the baseline, likely due to the high synchronization overheads in the algorithm. On the other hand, even though MPI takes a really long time when running on 1 processor, its computation time significantly improved as the number of processor increased, and eventually became even faster than the sequential baseline. This again shows that the MPI approach scales well to large number of processors.

An additional note about memory consumption: out of all the approaches we tried, despite the CUDA approach being the fastest, achieving a 50x speedup, it is not the most memory efficient if one is provided a large number of processors. The MPI approach's memory efficiency increases if one is to distribute the graph among different computing nodes. This has implication in contexts such as network routing. The MPI approach can potentially be used to find a distributed MST formed by a network of routers. MST can be used to find the path with the maximum minimum edge weight in a weighted graph, and this is equivalent to finding the path with the highest bottleneck bandwidth in a network. While the MPI approach is suited for such distributed settings, the CUDA approach is suited for settings where its possible to have a centralized server with enough memory to store the entire graph.

References

[1] Cuda c++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions.

[2] Richard J. Anderson and Heather Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, STOC '91, page 370–380, New York, NY, USA, 1991. Association for Computing Machinery.

[3] Alex Fallin, Andres Gonzalez, Jarim Seo, and Martin Burtscher. A high-performance mst implementation for gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[4] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, jan 1983.