Parallelizing MSF with Borůvka’s Algorithm – 15-618 Final Project

Shubham Bhargava & Jake Zaia

Summary

We implemented Boruvka’s algorithm in ParlayLib and CUDA achieving average computation speedups of 13.31x and 29.05x on the GHC machines using 8 CPU cores and the GeForce RTX 2080 respectively. ParlayLib is a C++ library written by CMU professors to implement algorithms on multi-core machines. Boruvka’s algorithm is used to find a Minimum Spanning Forest (MSF) and is more amenable to parallelization than Kruskal’s or Prim’s algorithm. We tested the algorithm on a variety of sparse and dense graphs with good performance on both the CPU and GPU. The implementations were benchmarked for performance against a sequential C++ implementation of the algorithm, and tested for correctness against a Python implementation of Kruskal’s algorithm. The sequential implementation was lightly optimized for running on a single core but was not extensively optimized. We also wrote graph generators and a benchmarking framework to support testing the correctness and efficiency of our algorithm.

Background

Boruvka’s algorithm is a greedy algorithm for the Minimum Spanning Tree/Minimum Spanning Forest problem. The Minimum Spanning Tree is a well-studied graph algorithm used for finding a spanning tree for a graph with minimal weight. It has many practical uses: water networks, electric grids, and computer networks. It is also a subroutine of many other algorithms such as algorithms for max-cut min-flow and the traveling salesperson problem. Due to its usefulness and the large number of algorithms to solve it, it is typically taught to undergraduates and usually covered in a parallel algorithms class.

Although there are other algorithms such as the Filter-Kruskal algorithm for parallel machines and the GHS algorithm for distributed machines, we chose to implement Boruvka’s algorithm as it is an interesting algorithm. Not only does it predate other MST algorithms, it was originally designed to be used by humans, not computers. Despite it seeming so simple, neither of us had heard of this algorithm before which excited us to learn more about it. Since the algorithm can be implemented in many ways, it makes for an interesting project to implement the algorithm and test the performance of various approaches. Implementing the algorithm in both ParlayLib and

CUDA allows us to compare and contrast which implementation choices work well on both the CPU and GPU.

Implementing Boruvka's algorithm on the GPU is challenging. Past 15-418 groups have failed to do so, instead settling on using OpenMP. As such, our main challenge is to successfully implement the algorithm on the GPU ensuring correct usage of concurrent data structures and atomics. Moreover, we aim to produce a CUDA implementation that can achieve a significant speedup over both the sequential and CPU-parallel versions on the GHC machines. We chose to use CUDA for this task as both of us are familiar with it and we have access to RTX 2080s for testing on the GHC machines.

ParlayLib is a C++ library for developing efficient parallel algorithms on shared multi-core machines. It was designed by CMU parallel algorithms faculty and provides a large number of highly optimized parallel primitives including a scheduler and concurrent memory allocator. It is written in a highly functional style which makes it easier to convert many parallel algorithms into practical implementations.

We use it for this project as it makes it more manageable to implement the algorithm and try various optimizations quickly. Furthermore, the programming model seems sufficiently different from the various parallel frameworks/libraries we have seen throughout the class which makes for a great learning experience. Since implementing the algorithm in CUDA is hard, the ParlayLib implementation serves as a stepping stone and a guide to a CUDA implementation. Lastly, using ParlayLib allows us to get CPU benchmarks so we can compare our CUDA results.

The algorithm works in two stages. First, we find the cheapest edge (i.e. with least weight) adjacent to each vertex. Then, we add all these edges to our minimum spanning tree. This makes use of the property that the minimum edge for each vertex is guaranteed to be in the MST (with a few caveats when multiple edges have the same weight discussed later). As we do this, we merge both the vertices along the edge. This results in a graph with fewer vertices. We can then rerun the algorithm on these smaller graphs. Since the number of vertices is at most halved every iteration, the algorithm finishes in the log *V* rounds where *V* is the number of vertices. Note that the algorithm is intentionally vague in how these steps are implemented which leaves us with a lot of room to determine the implementation.

The following pseudocode (courtesy of Wikipedia) illustrates the general outline of the algorithm:

```
algorithm Borůvka is
    input: A weighted undirected graph G = (V, E).
    output: F, a minimum spanning forest of G.
    Initialize a forest F to (V, E') where E' = {}.

    completed := false
    while not completed do
        Find the connected components of F and assign to each vertex
its component
        Initialize the cheapest edge for each component to "None"
        for each edge uv in E, where u and v are in different
components of F:
            let wx be the cheapest edge for the component of u
            if is-preferred-over(uv, wx) then
                Set uv as the cheapest edge for the component of u
            let yz be the cheapest edge for the component of v
            if is-preferred-over(uv, yz) then
                Set uv as the cheapest edge for the component of v
        if all components have cheapest edge set to "None" then
            // no more trees can be merged -- we are finished
            completed := true
        else
            completed := false
            for each component whose cheapest edge is not "None" do
                Add its cheapest edge to E'

function is-preferred-over(edge1, edge2) is
    return (edge2 is "None") or
           (weight(edge1) < weight(edge2)) or
           (weight(edge1) = weight(edge2) and
tie-breaking-rule(edge1, edge2))

function tie-breaking-rule(edge1, edge2) is
    The tie-breaking rule; returns true if and only if edge1
    is preferred over edge2 in the case of a tie.
```

The algorithm takes in a list of edges representing an undirected graph and outputs a subset of those edges that form the minimum spanning tree. The edges are represented by a triple, (*u*, *v*, *w*) where *u* and *v* are the endpoints/vertices and *w* is the integer weight of the edge. We assume that the edges are sorted by *u*, and then by *v* in the edge list. We also assume that *u* < *v* i.e. edges are undirected. This is a standard assumption we can make as various graph file formats do so, including competitive programming datasets.

From here on we will discuss the algorithm in terms of optimizing these two steps:

1. Finding the cheapest edge per vertex
2. Contracting the graph (combining vertices)

On large and dense graphs, the first step tends to be more expensive as the total work is proportional to the number of edges. The second step can be implemented in a variety of ways but in our implementations, the total work is roughly proportional to the number of vertices. The difference in time taken becomes especially apparent in the CUDA implementation when there are many edges. Note that these steps are run in a loop and each step is dependent on the completion of the previous one given how we represent a graph. We also need to update the MST between steps 1 and 2 which is handled differently in both the ParlayLib and CUDA implementations.

Approach

To represent the graph, we chose to use a list of edges as opposed to an adjacency list as it becomes very easy to parallelize the first step. As we loop through the edge list, we can check if that edge is the cheapest edge for either of its neighboring vertices. We need to store the cheapest edge per vertex. This would require synchronization to ensure multiple cores do not attempt to update the cheapest edge per vertex simultaneously. If we used an adjacency list, we would have to parallelize over the vertices, but if the vertices had vastly different numbers of edges, this could result in bad load balancing. However, if each core was assigned one vertex, we wouldn't need to deal with synchronization in this step. This approach might have had slightly better locality in updating the cheapest edge array, however, we address this issue with our edge list in a later section. The edge list representation also means that we cannot start the second part of the algorithm until we have gone through all the edges. However, due to the relatively even workload among cores/threads, this turns out to be a non-issue.

We also have to be mindful of breaking ties between edges with the same weight at this step, a cause of much pain when debugging. If edges of the same weight form a cycle, it is possible that all these edges are chosen simultaneously as the cheapest edge per vertex and then added to the MST. Cycles are disallowed for two reasons, the first being that the final result is no longer a valid tree/forest, and the second being that it can cause the program to hang when traversing a part of the tree. As such, we need some way of strictly ordering all the edges (that is consistent with ordering them by weight). If two edges have the same weight, we choose the edge with the lower index in the edge list. This will ensure that we can't select a cycle of edges with the same weight to be a part of the MST in one round of the algorithm. Initially, we used a different ordering which turned out to be buggy for subtle reasons discussed later.

Before we do the graph contraction, we must also add the edges to our MST. This was combined with the second step in our implementations. In ParlayLib, we used the parlay::sequence primitive to store the list of edges in the MST. Every round, we would use a filter on our list of cheapest edges to get rid of edges that might be double counted and then append them to our previous list which Parlay handles efficiently. Since we did not have an easy and efficient filter and append available to us in CUDA, we instead kept a boolean for each of the original edges. As we iterate through the cheapest edges in parallel, we update the boolean corresponding to the edge. We had to be careful about not double counting edges as we kept track of the number of edges added to our MST (which was used to determine when to terminate our algorithm in CUDA implementation). We could check if an edge was already counted by checking our MST boolean array atomically. Alternatively, we could check that the current edge did not count as the cheapest edge for any other vertex. In practice, we didn't notice any significant difference and used the second implementation.

We used a concurrent union-find data structure to keep track of which vertices had been merged into a single vertex. We call each set of vertices in a contracted portion of the graph a "component". A component forms a tree with some root vertex responsible for maintaining information about the component itself. For our particular implementation, the union-find was stored as an array of length *n*, where *n* represents the number of vertices in the entire graph. Each array element tracks the cheapest edge for the given vertex as well as a parent in the component to which the vertex belongs. For root nodes of a component, the vertex stores itself. To *union* 2 components, the root of one component must set its parent to any node contained in the other. To *find* the root of a component, you must loop over the "parent" nodes until the root is reached. However, in our parallel implementation we carefully structured our usage of union and find operations such that we could be sure that the component trees never had a height greater than 2 (this is further detailed in the optimizations section below). This allowed us to omit the loop in the *find* operation entirely, significantly cutting down on the time spent on such operations. Since *find* operations make up a significant amount of the implementation of Boruvka's algorithm, our custom problem-optimized union-find implementation significantly outperformed our CUDA adaptation of the concurrent union-find from Wait-free Parallel Algorithms for the Union-Find Problem, achieving a 1.077x speedup over our implementation of the version from this paper.

The union-find data structure formed a key component of our graph contractions. Once we combine the vertices, we have to update our edge list to reflect the updated contracted graph. One way to do this is to update each edge. We would replace the vertices of an edge with the component the vertex belongs to. As such the edge would now go between 2 components of our updated graph. This can be done efficiently using the find operation. We noticed that this can be combined with the first step of the algorithm: as we iterate through the edges on the algorithm's first step, we could use the union-find to find the component for each edge's endpoint and update the edge in the edge list. This worked well for both our ParlayLib and CUDA implementations. For our CUDA implementation, we also had to worry about warp sync which meant that we would flatten our union-find trees as discussed later.

One might notice that in this case, the number of edges doesn't go down throughout the algorithm. Instead, we obtain self-edges and multi-edges when we merge vertices. Since self-edges can no longer contribute to the spanning tree, it is possible to filter them out. For the ParlayLib implementation, we implemented this which means that in every iteration, the number of edges goes down. However, due to the nature of the algorithm, for many types of random/unstructured graphs, the number of self-edges is low. As such, the number of edges in the edge list goes down very slowly despite the number of vertices nearly halving every iteration. The speedup obtained varied across test cases but was overall small. In addition, filtering the edge list results in having to track extra auxiliary information about the edges which cancels most of the benefits of filtering out self-edges. Implementing this filtering would be tricky in CUDA so we tried using the CUB and Thrust libraries with no success. Due to the negligible benefits seen in the Parlay implementation, we decided not to implement it in CUDA.

In addition to filtering self-edges, we could also replace multi-edges with the shortest edge between the two vertices. However, this could take up to $O(E + n^2)$ space and time where *n* is the current number of components. As such, this is not done in Boruvka's algorithm. However, when the number of components becomes sufficiently small, we can get rid of most of the unnecessary edges in the multi-edge. At this small scale, it likely even makes sense to switch from a CUDA implementation of Boruvka's algorithm to a parallel implementation of the Filter-Kruskal algorithm. This kind of optimization requires a good number of changes to our CUDA implementation and implementing Filter-Kruskal was beyond the scope of this project.

In our ParlayLib implementation, some of the steps mentioned earlier were combined or separated depending on the needs. By combining steps in Parlay, we can avoid doing multiple passes over our data structures. Parlay also provides lazy data structures which allows us to chain certain operations easily and prevents Parlay's scheduler from doing multiple passes over the data. This can partly be seen when we use map_maybe in Parlay which combines a map and a filter. Since our ParlayLib implementation uses edge filtering, we terminate our main loop of the algorithm when there are no more edges left across any disjoint components. This means that each of our connected components has been merged into a single component and as such, we are done.

For the CUDA implementation, we broke the implementation of our main loop into three kernels. These kernels were chosen to combine similar operations and maximize locality. The first kernel resets the cheapest edge array as well as flattens our union-find data structure (this sets up the second kernel to perform *find* on a flat tree). Although it would be possible to merge the functionality of this kernel into the other two, it does not perform as well. The second kernel, assign_cheapest() corresponds to the first portion of the algorithm which involves finding the cheapest step. The third kernel, update_mst() corresponds to contracting the graph. This involves iterating through our cheapest edge array in parallel to merge vertices along those edges. Finally, the host copies a value from the device to determine when the MST is fully formed, and these 3 kernels are iterated until this point.

Tie-Breaking

As mentioned before, the method of tie-breaking for edges with the same weight is extremely important. If the tie-breaking method is incoherent it can cause cycles to form, which presents a correctness issue and can cause the program to hang. At first, we used a method of ordering edges based on the vertices that the edge conjoined, *u* and *v*, choosing that edges with lower values of *u* than *v* would come “first”. This is a coherent method of breaking ties; it is even one of the examples listed in the Wikipedia article on Boruvka’s algorithm. However, this fails to be true when accounting for an optimization we make in the code. When selecting edges we alter the values *u* and *v* to be the root of the associated component. This simplifies comparing edges between components that have many vertices. However, this also introduces multi-edges, meaning that the original method of comparing edges no longer has a strict ordering for some edges with the same weight. The result is undefined behavior wherein cycles may be added. Interestingly, this issue only appears in a parallel implementation, since a sequential implementation will visit all edges in some sequential order, which prevents 2 different but equal-weight edges from being added to 2 components simultaneously, preventing the creation of such cycles.

Synchronization

Synchronization was tricky to get right for our project, especially since we hand-rolled our own wait-free union-find. Our union-find wouldn’t work in a general application but was designed to be specific to our needs. In addition to using a union-find, we had to ensure that we correctly found the cheapest edge for every vertex. This couldn’t simply be done using an atomic minimum so we settled on using CAS. For the CUDA implementation, we were tracking the number of components/vertices in the graph. This was updated anytime we merged two vertices forcing us to atomically handle this as well.

For ParlayLib, we were initially using the union-find provided by ParlayLib. It implemented a variation of path compression (grandparent-linking) for find and provided a simplistic union that only worked in very specific algorithms. To address this, we implemented a simple union ourselves. However, it didn’t mesh well with the find implementation provided by ParlayLib. Occasionally, our program would hang and we couldn’t understand why. Any issues showed up only at 8 cores for large test cases rarely. Eventually, we reimplemented the find function ourselves getting rid of any path compression/shortening. We reasoned by producing small counterexamples that the grandparent-linking interfered with our implementation of the union function. The cause of the bug was loops being created in our union-find implementation under certain circumstances.

We used C++ atomics for the arrays/sequences that could be simultaneously modified by multiple threads. There was a minor implementation issue with how we used compare_exchange_strong due to misunderstanding its unusual semantics. We used a compare-and-compare-and-swap as opposed to a compare-and-swap in an attempt to optimize the code (this made no practical difference in speed across tests). As a result, any bugs arising from misusing compare_exchange_strong were incredibly rare as the first compare would handle most synchronization issues. This made it nearly impossible to debug. We attempted to simplify the code by removing the compare before compare-and-swap leading to many more failing test cases which pointed us toward the problem.

Optimizations for CUDA

Parallelizing Boruvka’s algorithm using CUDA presents a challenge since warps execute instructions as SIMD. Boruvka’s algorithm is composed of many loops that have unpredictable early exit points and branches. Thus, a naive translation to CUDA will produce code that has significant thread divergence and warp stall. Indeed, our original implementation would spend on average over 1000 cycles per operation stalled in the worst-performing kernels. This constituted well over 90% of cycles spent executing the kernel. Through careful optimization, we were able to reduce this to approximately 70 cycles in the worst case by carefully organizing the access patterns of edges.

The first optimization is to ensure that (for threads within a threadblock) all threads are accessing similar memory locations. We were able to accomplish this by ensuring that all edges are ordered in increasing order of the first vertex. This means that sequential edges are very likely to act on at least 1 vertex in common. We split the edge list into chunks where each threadblock was given 1 chunk to process. Then, we had threads within a threadblock traverse this chunk in interleaved order. The result of this is significantly reduced memory stall times, since each edge only occupies 12 bytes and several edges can be read in the same cache line. Moreover, while contention on single vertices is high, this contention is mostly contained within a threadblock (and even more often within a warp), which is significantly preferred to distributing the contention across different threadblocks.

An algorithmic optimization that is often made for Boruvka’s algorithm is to flatten the component trees as they are traversed. Ordinarily, this can be done as each component is accessed. This is typically done by updating the parent pointers as the component tree is traversed. However, for SIMD execution this is suboptimal since it means that some threads will spend many more iterations looping than others, which can cause thread divergence. Moreover, checking the component associated with a vertex is one of the most common subroutines used in Boruvka’s algorithm, so it must be a quick operation. We eliminated the looping behavior by introducing an additional “flatten” step that can be performed in parallel between executions of the main portion of the algorithm. In essence, after the MST is updated, each vertex traverses its component tree to its root and stores its root, flattening the tree maximally. Then, until the next merge step every single component is guaranteed to be either a root or of depth 1. Notably, for this to remain threadsafe, there is one more precaution that must be taken: when merging components, each component must only be merged as the “child” of another component *once*. If this is not the case, then merges will overwrite the parent since it cannot loop to retrieve the root component. This is different from the typical optimization for a sequential version of a union-find which would instead track a rank or depth of the component trees and merge the smaller as the parent.

Another optimization we attempted was to keep two copies of every edge in our edge list. As mentioned before, using an adjacency list to represent the graph would mean that we would experience better locality as we tried to find the cheapest edge per vertex. This is because, for each vertex, we can iterate through its neighbors picking the cheapest one. Instead, parallelizing over the edges, (u, v, w), we would attempt to update the cheapest edge for u and then v. This involved accessing different portions of the cheapest edge array. Note that the edges were already sorted by u which means that updating the cheapest edge for vertex u had temporal locality. However, updating the cheapest edge for vertex v didn’t have the same locality. For a given edge, (u, v, w), if we stored both (u, v, w) and (v, u, w) in our edge list, we could only update the cheapest edge array for the first vertex in the edge. If these edges were all sorted by the first component, we would gain the same locality benefits as an adjacency list. However, implementing this involved a lot of tradeoffs. To create this sorted edge list efficiently, we had to convert our old edge list to an adjacency list and then back to this new “directed” edge list representation. Additionally, we had to store extra auxiliary information for each edge. Previously, we would memory-map our large binary graph files directly into memory. That was no longer possible due to the changed edge format. We also had to update our edge tie-breaking function. Furthermore, our initial communication from the CPU to the GPU increased as we had to send more auxiliary information. Although we saw a slight improvement in CUDA running time, our preprocessing time and our total execution time suffered greatly. It also made the code more convoluted and as such, we opted not to do this.

Project Infrastructure

A significant portion of time was spent producing infrastructure to back this project. Namely, gathering a collection of graphs that are undirected, weighted, large, and representative of various types of graphs represented several hours of careful work. Moreover, devising ways to store these graphs, which were in many cases extremely large, and efficiently load their contents into the programs was an active challenge. During the testing phase, we encountered correctness issues that would only appear on specific input graphs, and these issues may not have been corrected if not for the comprehensive benchmarking tools we created.

Ultimately, we created a benchmarking script (mstbench) that would generate random graphs of specific structures on the fly and then execute our various implementations several times on these graphs. These graphs were generated using various graph generation algorithms from networkx, and written into a custom binary file format. These files (which were in many cases several hundred megabytes) were then saved into tmpfs and fed into the different implementations. Since simply reading the file was the source of significant overhead, each implementation would read the file once, and then execute the algorithm a specified number of times. This reduced the amount of time spent loading generated graphs into memory.

Results

Benchmark Graph Types

Our benchmark set is composed of various types of graphs generated with random weights from 1 to 1000 on each edge. Edge weights are kept to this range to ensure that ties occur since edges that have tied weights can pose an issue for incorrect implementations. Moreover, edge weights are allowed to vary in this range such that the set of correct MSTs remains small to reduce non-determinism. Further elaboration on each graph type is as follows.

Circulant Graph: A set of cycles on n vertices. The d degree circulant links every vertex to all vertices that are ≤ ±d away from the current vertex. It is a very sparse ($2d$)-regular graph.

Hypercube Graph: A graph whose edges form the hypercube in a d-dimensional space. It is a d-regular graph on 2^d vertices and is highly sparse.

Caveman Graph: A graph composed of several disjoint cliques. The caveman graph actually does not have a minimum-spanning tree, but instead a minimum-spanning forest. Since it is composed of many cliques, it is relatively dense, however the density depends on the number of groups and the size of each group. The caveman graph on g groups of size k has $g*k$ vertices and O(gk^2) edges.

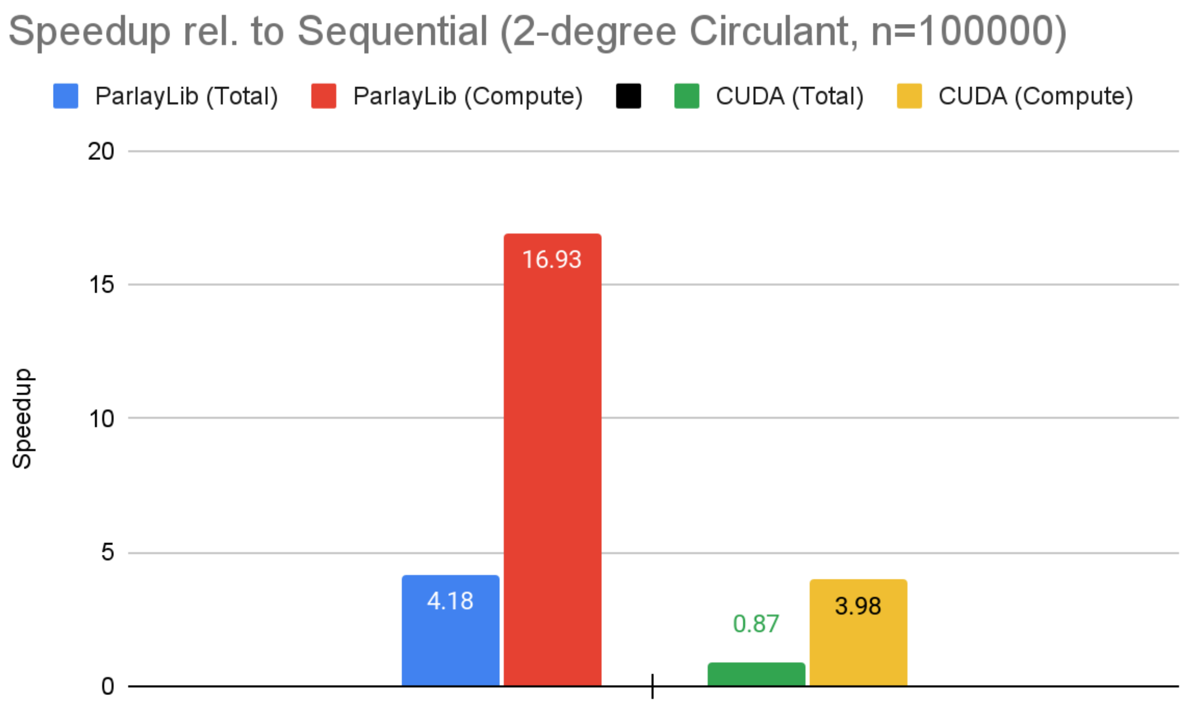
Connected Caveman Graph: A graph composed of several cliques joined by 2 edges in a cycle. This graph is extremely similar to the caveman graph, however, now 2 edges in each clique are used to connect to other cliques making the entire graph contain a cycle. It has the same number of vertices and edges as the caveman graph.

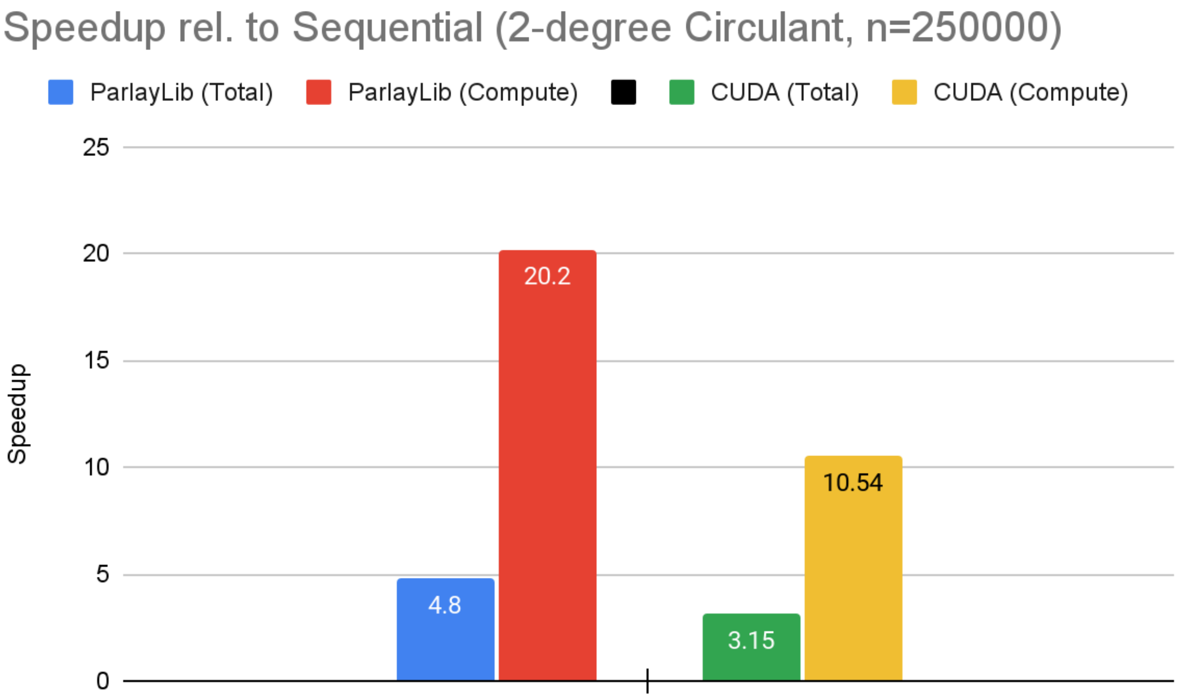
Erdős–Rényi Graph: A random graph of a specified density (sometimes also called the binomial graph). This graph is completely unstructured and is used to represent a more realistic graph layout. It has n vertices and O(pn^2) edges. In our case, we consider it a dense graph since we will set p to be comparatively high and the number of edges scales as O(n^2).

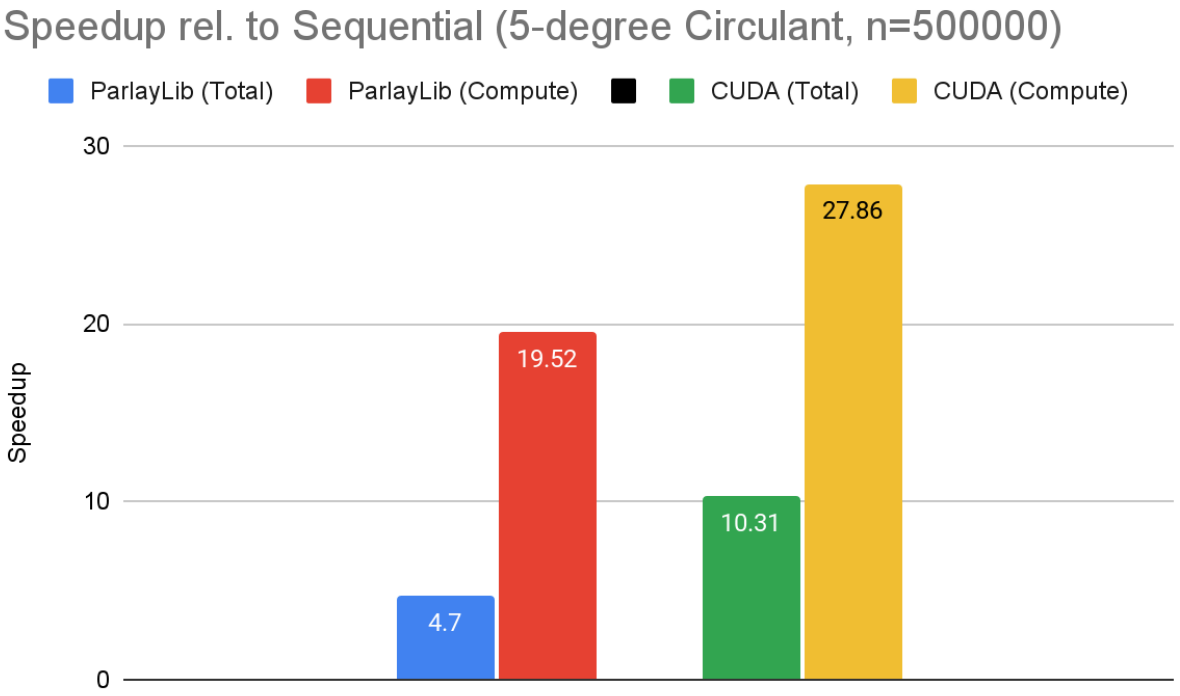
These graph types include graphs that are both highly structured and unstructured and graphs of both high and low densities. As such, they are a representative sample of several families of graphs that one may want to compute using.

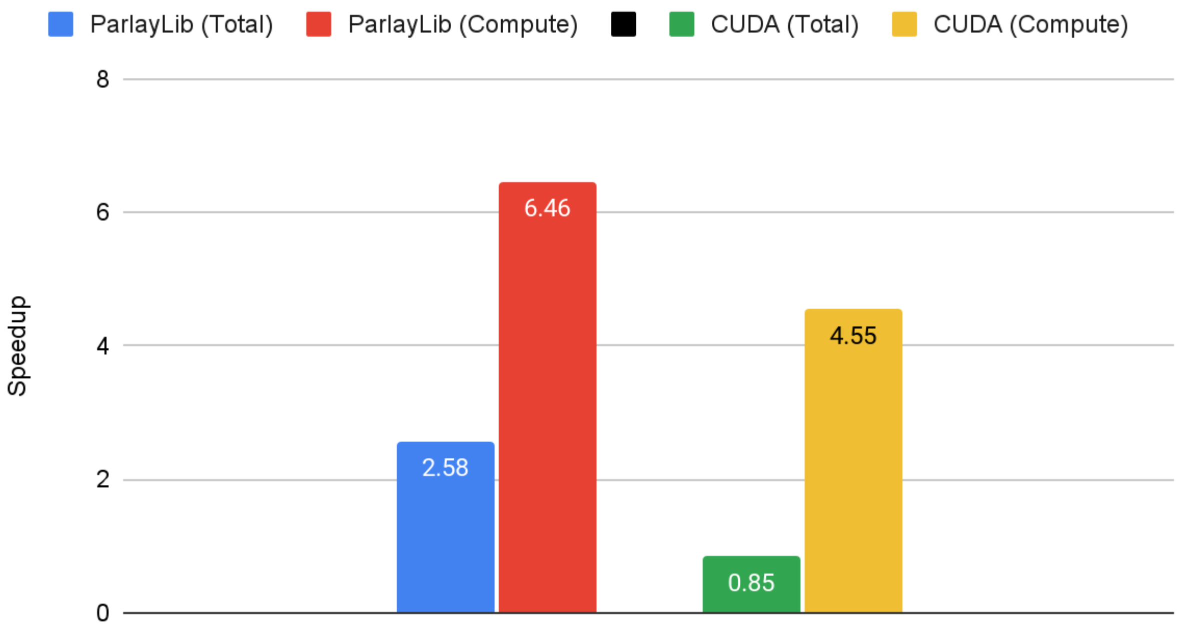
Speedups on Benchmark Graphs

All benchmarks were collected on the GHC machines. The ParlayLib implementation was run using 8 cores and the CUDA version was run on a GeForce RTX 2080. The graphs are presented with speedup relative to an optimized sequential C++ implementation of Boruvka's algorithm. Values are the average speedup obtained over 10 repetitions of each implementation on each graph.

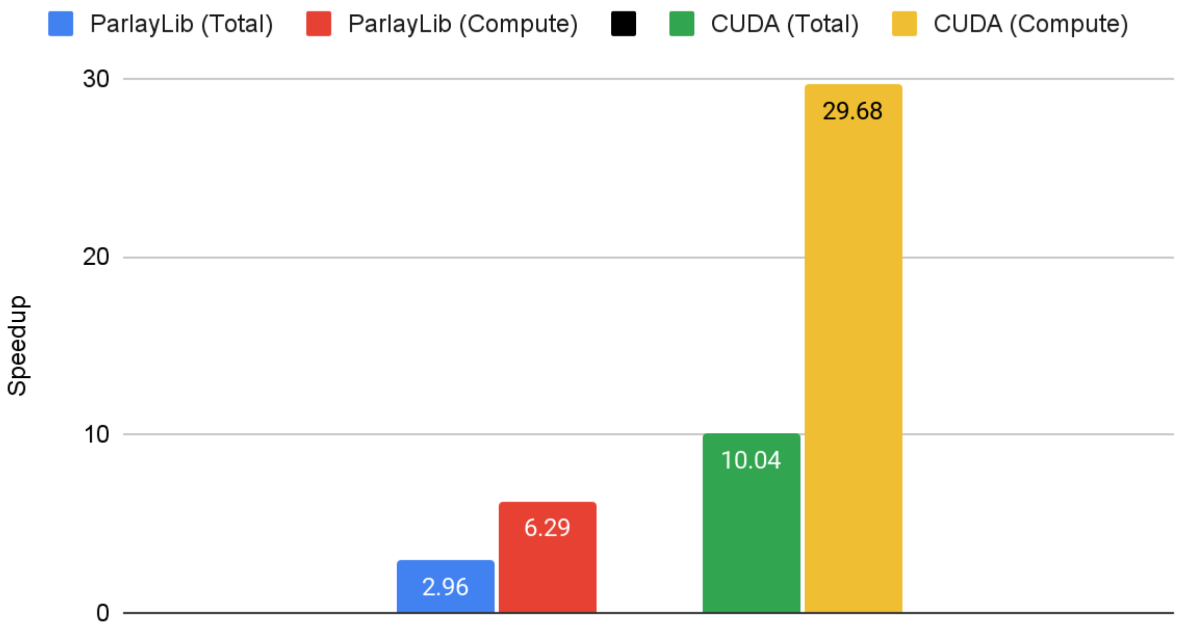



Speedup rel. to Sequential (2-degree Circulant, n=250000)
ParlayLib (Total)
ParlayLib (Compute)
CUDA (Total)
CUDA (Compute)
Speedup
25
20
15
10
5
0
4.8
20.2
3.15
10.54

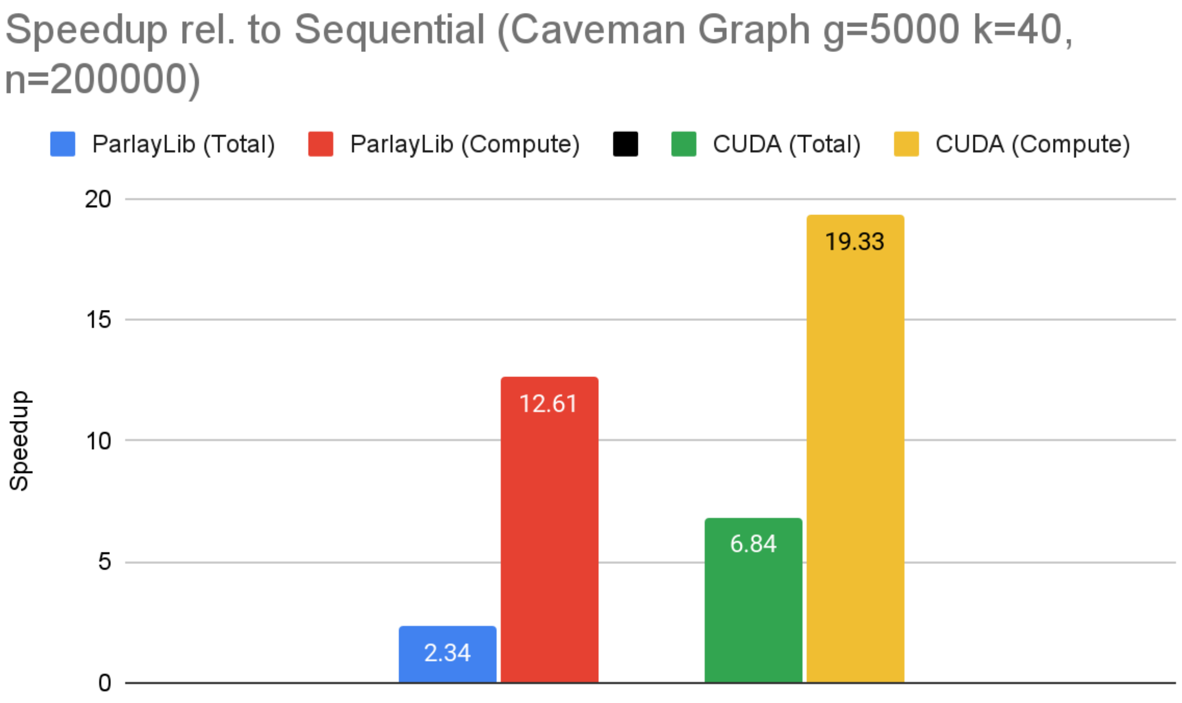

Speedup rel. to Sequential (5-degree Circulant, n=500000)
ParlayLib (Total)
ParlayLib (Compute)
CUDA (Total)
CUDA (Compute)
Speedup
30
20
10
0
4.7
19.52
10.31
27.86

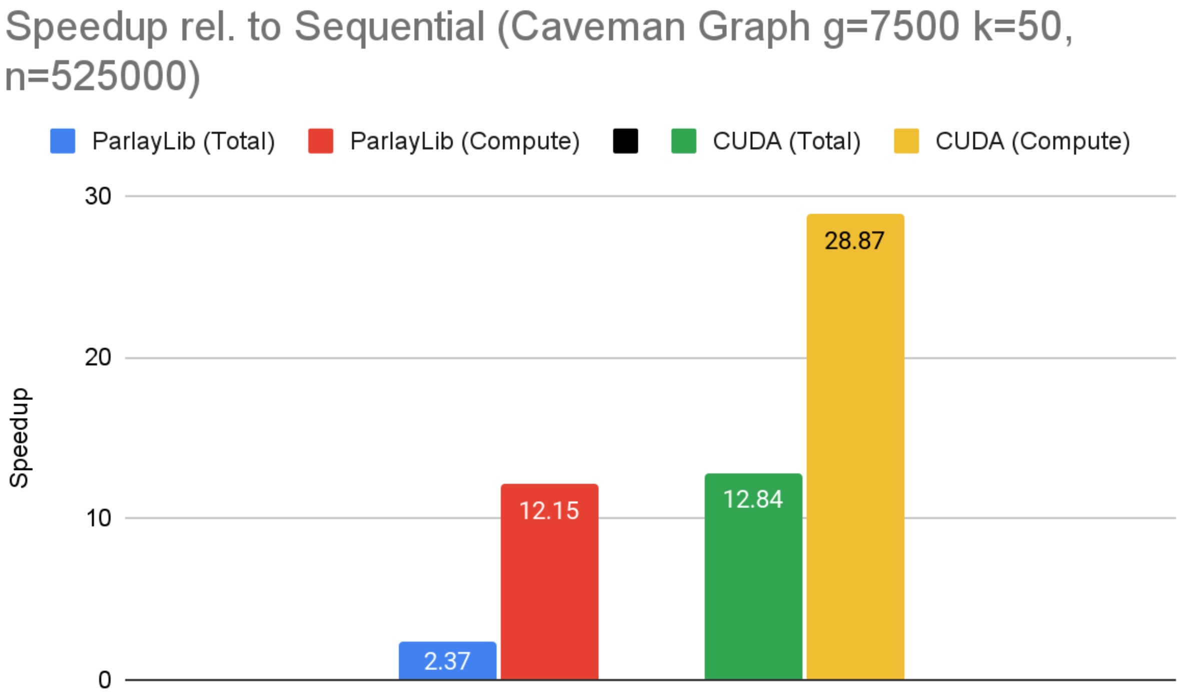


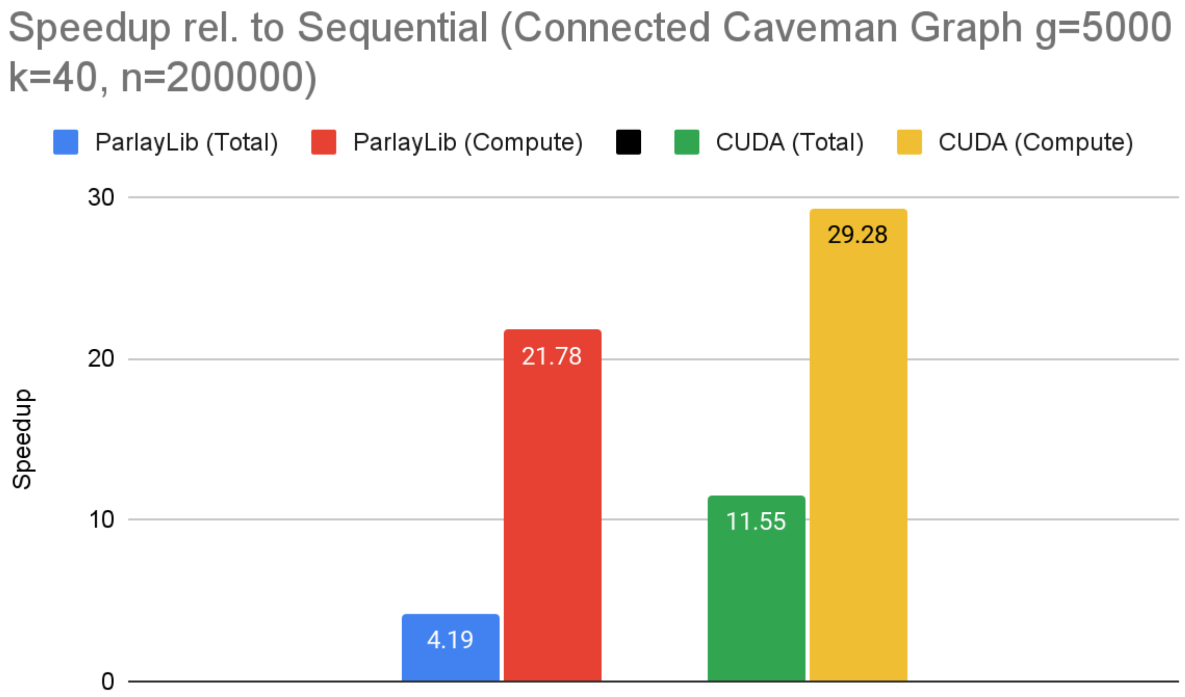
Speedup rel. to Sequential (Hypercube d=15, n=32768)



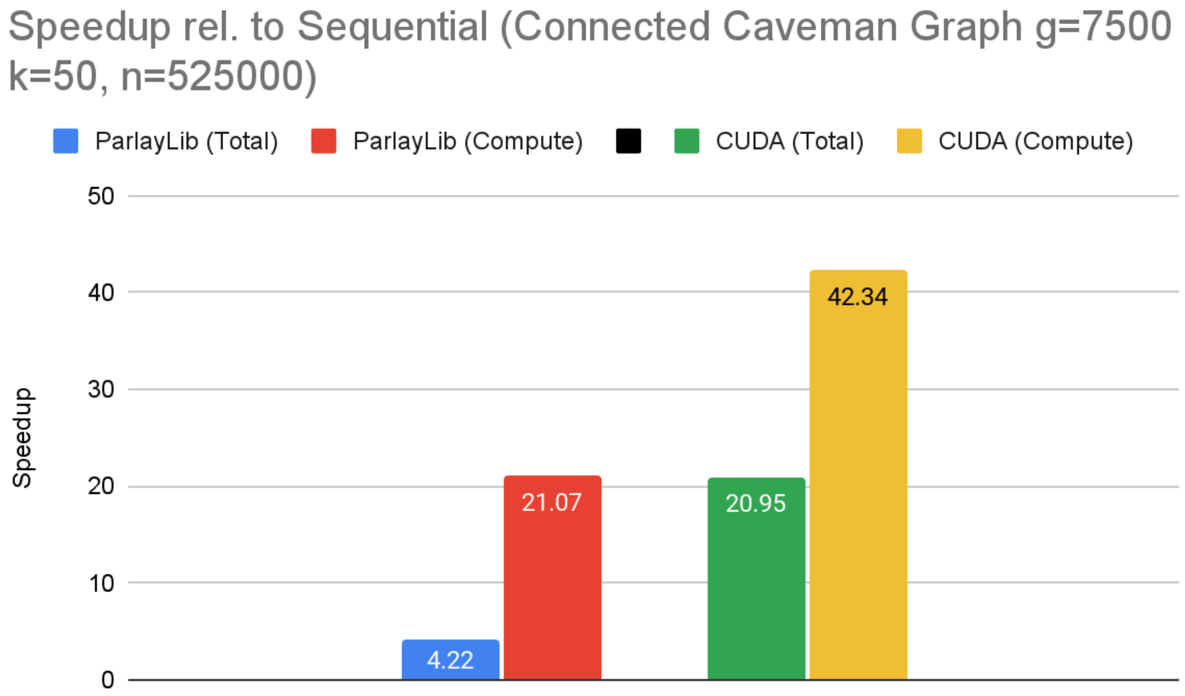
Speedup rel. to Sequential (Hypercube d=18, n=262144)


Speedup rel. to Sequential (Caveman Graph g=5000 k=40, n=200000)

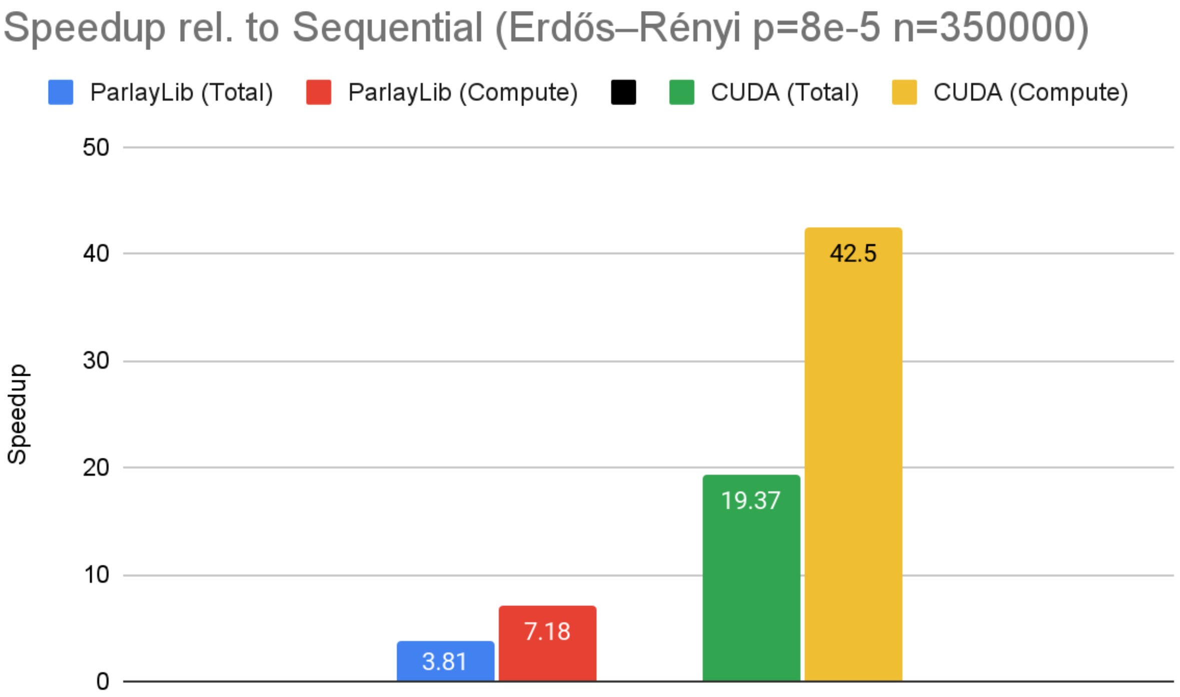

Speedup rel. to Sequential (Caveman Graph g=7500 k=50, n=525000)

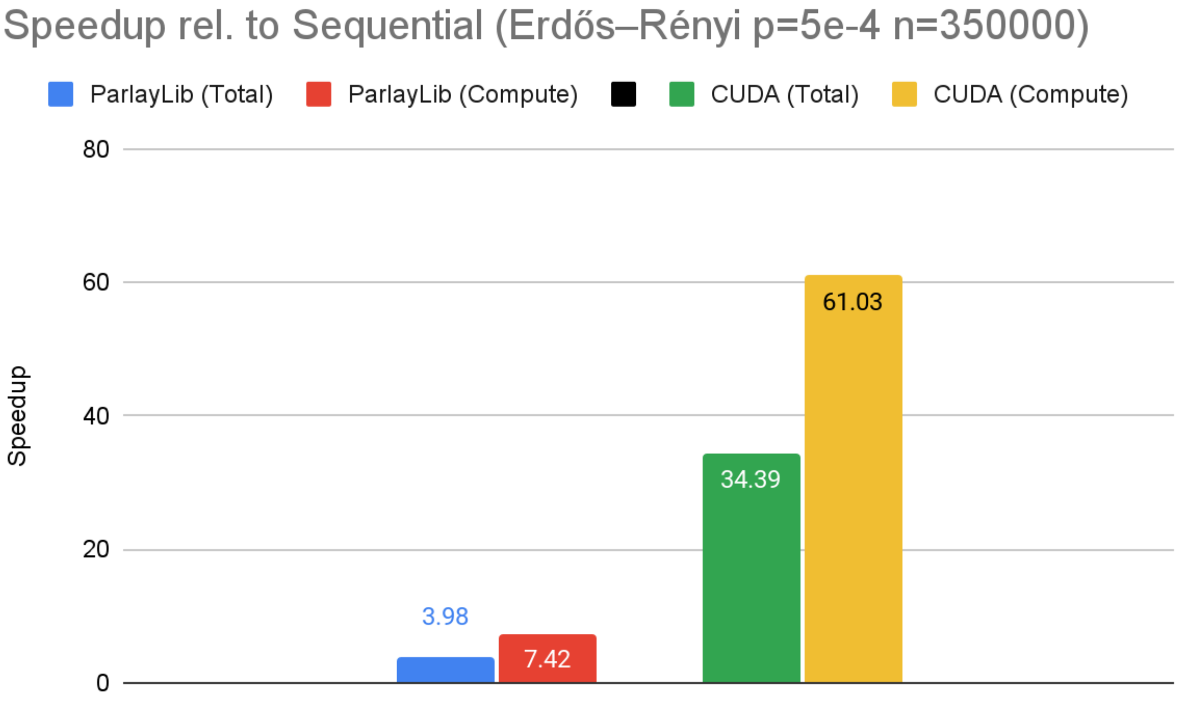


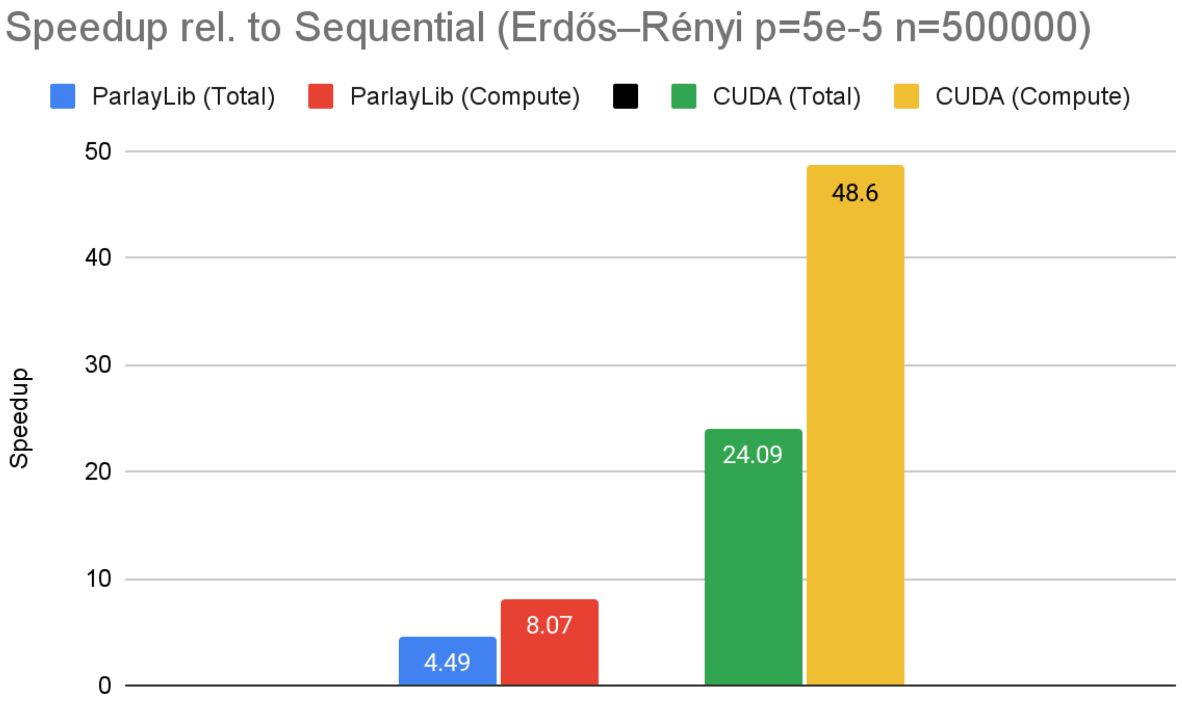
Speedup rel. to Sequential (Connected Caveman Graph g=5000 k=40, n=200000)



Speedup rel. to Sequential (Connected Caveman Graph g=7500 k=50, n=525000)


Speedup rel. to Sequential (Erdős–Rényi p=8e-5 n=350000)
ParlayLib (Total)
ParlayLib (Compute)
CUDA (Total)
CUDA (Compute)
Speedup
50
40
30
20
10
0
3.81
7.18
19.37
42.5


Speedup rel. to Sequential (Erdős–Rényi p=5e-4 n=350000)
ParlayLib (Total)
ParlayLib (Compute)
CUDA (Total)
CUDA (Compute)
Speedup
80
60
40
20
0
3.98
7.42
34.39
61.03



Analysis of Benchmark Results

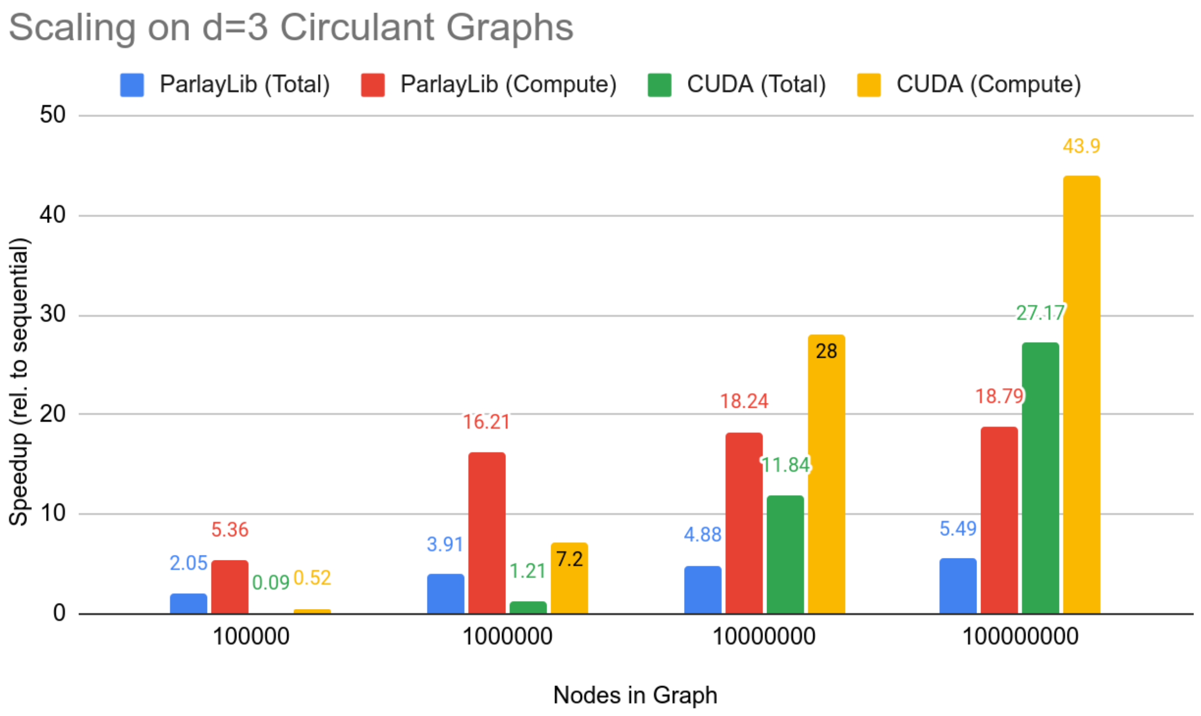
In the graphs above we see that the effects of parallelism are most pronounced on the largest graph types. This is especially true for the CUDA implementation which scales much faster for the large and dense graphs, especially the largest caveman and Erdős–Rényi graphs. On the smallest test cases such as the smallest circulant graph, CUDA falls short of the ParlayLib implementation and in one case even fares worse than the sequential version.

Conversely, the ParlayLib fails to scale significantly, performing only slightly better as graphs are made larger and denser. However, even for the smallest graph, it has a clear speedup of the sequential version. Moreover, given its limited resources of 8 CPU cores, it achieves close to or above a 4x total speedup on a majority of the test cases. The only exceptions to this are the caveman (non-connected) and hypercube graphs. It appears that certain types of highly regular graphs cause issues with the ParlayLib implementation's ability to scale.

Results of Varying Problem Size

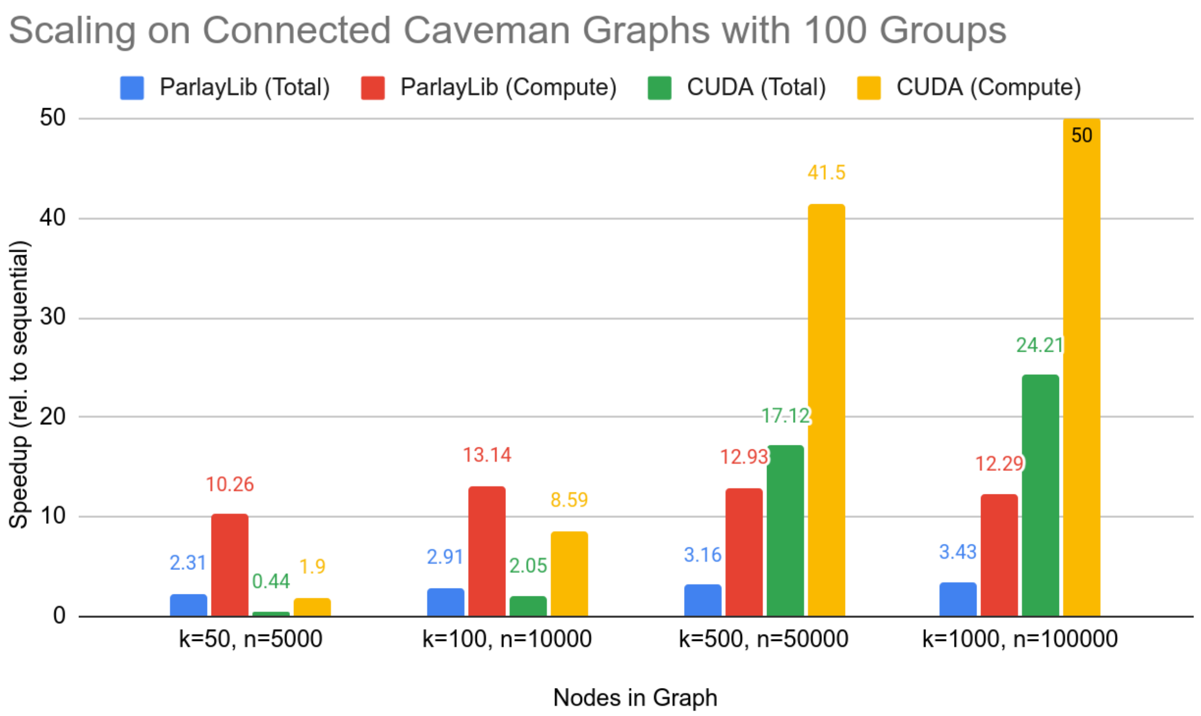
In this section, we analyze how both implementations fair as problem size scales. We evaluate this on the degree 3 circulant graph, which is sparse, as well as the 100-group connected caveman graph, which is comparably dense. Notably, the number of vertices for the dense problem has to stay much lower since the number of edges grows quadratically instead of linearly. More precisely, the circulant graph will have *3n* edges, where *n* is the number of vertices whereas the connected caveman graph will have *50(k^2-k)* edges, where *k* is the group size.

As before, all benchmarks were collected on the GHC machines. The ParlayLib implementation was run using 8 cores and the CUDA version was run on a GeForce RTX 2080. The graphs are presented with speedup relative to an optimized sequential C++ implementation of Boruvka's algorithm. Values are the average speedup obtained over 10 repetitions of each implementation on each graph.



Raw Runtimes in Seconds

Problem Size (n)	ParlayLib (Total)	ParlayLib (Compute)	CUDA (Total)	CUDA (Compute)
100000	0.0028	0.0011	0.0667	0.0112
1000000	0.0173	0.0042	0.056	0.0094
10000000	0.1641	0.0439	0.0676	0.0286
100000000	1.6524	0.4824	0.336	0.2064



Raw Runtimes in Seconds

Problem Size (n)	ParlayLib (Total)	ParlayLib (Compute)	CUDA (Total)	CUDA (Compute)
100000	0.0075	0.0017	0.0396	0.0092
1000000	0.0268	0.0059	0.0381	0.0091
10000000	0.6455	0.1577	0.1185	0.0491
100000000	2.6486	0.7388	0.3752	0.1816

As we can see, the CUDA implementation continues to scale even for extremely large problem sizes. For reference, the input file for the k=1000 connected caveman graph was over half a gigabyte, despite being a minimal binary file. Conversely, while the ParlayLib implementation continues to scale on sparse graphs, it does not continue to scale significantly for graphs with high density.

Since the ParlayLib implementation has edge filtering implemented, sparse graphs with structure will tend to benefit especially. In edge filtering, we remove self-edges from our edge list as we contract our graph. For sparse structured graphs (such as the Circulant), the shortest path between any two random vertices is large which means the algorithm tends to take more iterations, but the number of non-self-edges goes down with the number of vertices. Whereas with denser graphs (such as Connected-Caveman with fixed groups), the algorithm tends to take fewer iterations and the number of non-self-edges doesn't decrease significantly.

Scaling Obstacles for CUDA Implementation

Several obstacles prevent the CUDA implementation from scaling well. Taking the dense graph from above as a case study, we get the following information from nvprof:

```
            Type  Time(%)      Time  Calls       Avg       Min       Max  Name
 GPU activities:   44.39%  228.08ms     36  6.3355ms  5.3347ms  9.2597ms  assign_cheapest(void)
                   43.74%  224.73ms     12  18.727ms  1.1840us  56.243ms  [CUDA memcpy HtoD]
                   11.12%  57.117ms     40  1.4279ms     704ns  14.289ms  [CUDA memcpy DtoH]
                    0.60%  3.0905ms     36  85.846us  74.783us  143.01us  update_mst(void)
                    0.09%  469.79us      4  117.45us  116.42us  118.98us  [CUDA memset]
                    0.06%  323.77us     36  8.9930us  5.5670us  19.999us  reset_arrs(void)
                    0.01%  25.984us      4  6.4960us  6.1120us  7.6160us  init_arrs(void)
      API calls:   41.40%  283.59ms      8  35.448ms  14.694ms  56.250ms  cudaMemcpy
                   33.84%  231.79ms     36  6.4385ms  5.4279ms  9.4178ms  cudaMemcpyFromSymbol
                   23.76%  162.78ms     12  13.565ms  51.866us  160.10ms  cudaMalloc
                    0.86%  5.8650ms     12  488.75us  73.018us  1.0793ms  cudaFree
                    0.05%  367.92us      8  45.989us  8.0250us  85.208us  cudaMemcpyToSymbol
                    0.05%  312.70us    112  2.7910us  2.1020us  10.898us  cudaLaunchKernel
                    0.01%  98.332us    101     973ns      94ns  38.936us  cuDeviceGetAttribute
                    0.01%  90.189us      1  90.189us  90.189us  90.189us  cudaGetDeviceProperties
                    0.01%  45.919us      4  11.479us  8.9970us  18.482us  cudaMemset
                    0.01%  35.013us      1  35.013us  35.013us  35.013us  cuDeviceGetName
                    0.00%  5.9710us      1  5.9710us  5.9710us  5.9710us  cuDeviceGetPCIBusId
                    0.00%  3.2280us      1  3.2280us  3.2280us  3.2280us  cudaGetDeviceCount
                    0.00%     872ns      3     290ns     147ns     578ns  cuDeviceGetCount
                    0.00%     467ns      2     233ns      95ns     372ns  cuDeviceGet
                    0.00%     429ns      1     429ns     429ns     429ns  cuDeviceTotalMem
                    0.00%     224ns      1     224ns     224ns     224ns  cuDeviceGetUuid
                    0.00%     189ns      1     189ns     189ns     189ns  cuModuleGetLoadingMode
```

A significant portion of the time spent "processing" this graph is simply getting the graph into GPU memory. As mentioned before, this particular input graph is over half a gigabyte of memory, so this time must be spent to communicate the full graph to the GPU. Compressed formats for the graph such as adjacency matrices do not parallelize well using Boruvka's algorithm, however, it might be possible to improve by somehow transmitting the graph to the GPU in a compressed format, and decompressing it in parallel somehow. This was beyond the scope of what we had the time to do for this project.

Most of the rest of the time is spent within the kernel assign_cheapest. This is the kernel that does the most work since it is the only kernel that must iterate over the edge list (in this case the edge list is orders of magnitude larger than the number of vertices). While there are likely further small optimizations we can make to assign_cheapest, Amdahl’s law tells us that this can only serve to give at most approximately a factor of 2 speedup.

Moreover, profiling this run in NCU illuminates that most of the slowdown within assign_cheapest is due to inefficient memory access patterns. We were able to optimize to fix this somewhat, gaining a 3-4x speedup on the various benchmark graphs by interleaving memory accesses within a warp. There are likely other places within the code that could be improved to gain a further speedup (although as mentioned before, we are limited to at most a 2x speedup due to Amdahl’s law). It’s also possible that a portion of these slow memory accesses are due to coherency misses when different threads attempt atomic operations on the same vertex simultaneously. Due to the nature of this algorithm, some of these conflict misses will always exist, although there may be some way to further hide the latency.

Results Summary

Overall, this project seems to be a success. The CUDA implementation managed to get a significant speedup over both the sequential and the ParlayLib implementations for sufficiently large graphs. Moreover, a majority of the remaining issues with the speed of the CUDA implementation seem intrinsic to the algorithm or the size of the data being operated on.

References

- A Generic and Highly Efficient Parallel Variant of Boruvka's Algorithm
- Engineering Massively Parallel MST Algorithms
- Wait-Free Union-Find implementation
- 15-210 MST Notes
- Networkx Graph Generators
- ParlayLib
- CUDA Documentation
- Wikipedia: Boruvka's Algorithm (Pseudocode snippet)

Work Distribution

Task	**Shubham Effort**	**Jake Effort**	**Relative Weight in Project**
Sequential Implementation	0%	100%	10%
ParlayLib Implementation	100%	0%	20%

CUDA Implementation (excluding optimization)	60%	40%	30%
CUDA Optimization	40%	60%	30%
Infrastructure/Testing	0%	100%	10%