ABSTRACT: Graphing algorithms are fundamental to modern computing. Despite this, many algorithms have undesirable complexities and long run times, presenting a serious problem in certain applications. Parallelisation of algorithm execution is a technique used to improve runtimes, particularly with large datasets. However, the programming language used is fundamental in the effectiveness of parallelisation due to overhead introduced in any such techniques. Despite Julia language claiming to be lightweight and high performant, we were unable to achieve any speedup using parallelisation techniques which had proven beneficial in lower level languages. In this report, we investigate parallel implementations of Prim's and Floyd-Warshall's algorithms in Julia, exploring possible reasons for the poor results.

1. Introduction

Graphs algorithms are very common in modern day computing, with common applications having long runtimes and some algorithms exhibiting complexities of up to $O(n^3)$. As a result, parallel execution of these algorithms can be extremely beneficial in achieving faster run times. Despite this, many of the common algorithms are not suitable for efficient parallelisation, due to either the inherent sequential nature of the algorithms, or the fine granularity and short runtimes of parallelisable inner loops.

In this project, we evaluate the implementation of parallel Prim's and Floyd-Warshall's algorithms in Julia. Julia is powerful, high performant, and compiles to native code, resulting in performance claimed to be similar to that of statically-typed languages like C++ or Fortran [1]. It is a flexible dynamic language, alleviating many of the problems associated with traditional dynamically-typed languages like Python or R [1].

2. Julia Language

As Julia is a language aimed at the scientific community, there are various parallel computing options available. In particular, they fall under three main categories; Tasks, Multi-Threading, and Multi-Core/Distributed Processing [2]. Multi-Threading in Julia allows compute bound tasks to achieve parallel execution as you are given the ability to utilise all of the OS threads [2]. Julia allocates the iteration space via block allocation amongst the number of threads specified on start-up. If the number of threads is not provided, it defaults to a single thread [2]. It is important to note that Julia only supports multi-threading on shared memory systems, and this is still experimental at its current stage. Every thread involved in the execution of a loop utilises the same version of variables, hence concurrent access must be taken into consideration.

The Julia multi-core environment is based on the message passing paradigm, facilitating distributed execution on systems which do not have a shared memory domain [2]. Unlike traditional message passing environments, Julia provides one-sided message passing [2]. This means that the programmer only manages messaging on one of the two processors involved in the operation. Distributed processing in Julia harnesses the ideas of workers, where parallel regions are distributed among workers. As this is based on distributed memory message passing, each worker has their own private version of variables, hence requiring the use of SharedArrays to allow multiple workers access to a single array and visible results after execution.

3. Parallel Graph Algorithms

3.1. Approach

Before implementing any graphing algorithms in Julia, we first consulted the existing literature to learn about existing parallelisation techniques for these kinds of algorithms. The algorithms we selected for parallelisation (Prim's algorithm and Floyd-Warshall) were chosen for two main reasons. The first being that both have well known parallel implementations in the literature, meaning we did not have to reinvent the wheel and come up with a new parallel algorithm. The second being that these algorithms work quite differently, where Prim's is a greedy algorithm and Floyd-Warshall works by repeating the same calculation many times until it has found the desired result. Exploring these two algorithms allowed us to learn more about different parallel graph algorithms while also giving us more room to focus on the capabilities of Julia's parallel computing tools and experiment with different approaches.

3.2. Prims Algorithm

3.2.1. Algorithm Description

Prim's algorithm solves the problem of finding a minimum spanning tree for a weighted graph. A minimum spanning tree is a tree that includes all nodes of the graph such that its sum of edges is the minimal sum of all possible trees. The algorithm progresses iteratively, where at each iteration the "optimal" node is added to the minimum spanning tree. This is the node that has the smallest edge weight such that the edge connects a node in the minimum spanning tree with a node outside of the tree. The algorithm progresses until the minimum spanning tree contains all nodes in the graph.

3.2.2. Parallelisation Techniques

Prim's algorithm consists of an "outer loop" that is inherently sequential and hence cannot be parallelised. This "outer loop" refers to the logic that determines whether a new iteration needs to be run after each iteration finishes. A well-known approach to parallelising the "inner loop" of this algorithm exists that uses data partitioning of the adjacency matrix (the data structure used to represent the graph) and the distance vector (an array that maintains the cheapest edge for adding each node to the tree). [Fig. 1] illustrates how this data partitioning occurs if p processors are used [3]. The distance vector and adjacency matrix are simply split into p equal-sized chunks. To determine the node to be added to the minimum spanning tree at each iteration, all processors calculate their local minimum and broadcast this to the main thread to "reduce" these local minima into the global minimum. Once the node with the minimal edge weight is added to the tree, this information is broadcast to all processors and each one will update its chunk of the distance vector before the next iteration begins. This is the same approach used in [4], which implemented this parallel algorithm using Fortran, where speedups ranging from 0.98 to 2.79 were observed across 2 to 10 processors respectively.

3.2.3. Implementation in Julia

For this project, a sequential and parallel formulation of Prim's algorithm were developed using Julia. The core principles of the implemented algorithms follow those discussed in sections 3.2.1 and 3.2.2. To achieve calculation of the minimum distance vector value in parallel, the @distributed macro was used, with a reduction parameter "min" specifying the local minima to be reduced into a global minimum [Fig. 2].

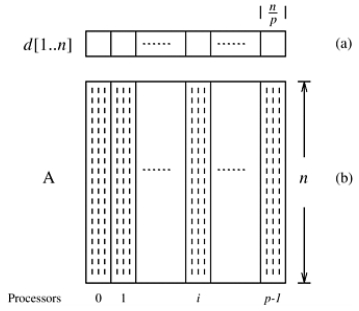

Figure 1. Work Division [3]

```
function cheapestNodePN(distanceVector, nodes)
    cheapest = @distributed min for i in collect(nodes)
        (distanceVector[i], i)
    end

    return cheapest[2]
end
```

Figure 2. Parallelising a for loop with reduction in Julia

Two variants of Prim's algorithm are present in the project repository. The difference between the two is how they keep track of the nodes that exist in the minimum spanning tree. The first variant, "PrimsSequential", uses a set that contains all the nodes in the tree. The second, "PrimsSequentialNodes", uses a set to keep track of the remaining nodes. This second variant reduces the search space when determining the next node to add to the tree because this set of remaining nodes can simply be iterated through, rather than iterating over all nodes and checking if they are in the tree or not. The parallelisation techniques on both these variants still follow the same principles discussed.

3.3. Floyd-Warshall Algorithm

3.3.1. Algorithm Description

The Floyd-Warshall algorithm solves the problem of shortest path between all nodes in a weighted directed graph. This is done by comparing all possible paths in the graph, comparing each pair of nodes. As a result, the complexity of the algorithm is $O(|V^3|)$. The most basic implementation of the algorithm consists of a triple nested loop, incrementally improving an estimate on the shortest path between two nodes in the graph. At each point, the shortest path between two nodes has the opportunity to use the direct edge between nodes (assuming it exists), or to use a combination of shortest paths between other nodes to reach the destination node. As a result, after termination of the triple nested loop, all shortest paths have been determined incrementally against all possible paths between nodes.

3.3.2. Parallelisation Techniques

We followed the implementation approach used in a research paper from the National Technical University of Athens [5]. Using their parallelisation approaches on the Floyd-Warshall algorithm, they were able to achieve an almost 50% reduction in run times using parallel for in OpenMP using C++. It is not possible to parallelise the k loop as there are possible dependencies between iterations due to the incremental estimation of shortest path built using new combinations of edges. As a result, the outer i loop (second level in triple nested loop) was parallelised using OpenMP's parallel for, as seen in [Fig. 3]. This achieves reasonable granularity of parallelisation as each iteration of the parallelised loop iterates through the entire set of edges.

3.3.3. Implementation in Julia

Julia provides a similar style parallelism approach to the parallel for in OpenMP with its multithreading @threads macro. As a result, we tackled the parallel implementation in a similar approach to the research paper. We used the @threads macro on the i loop to distribute the iteration space in block form across the available threads as seen in [Fig. 4]. In addition to this, we implemented a version using the Julia's multi core @distributed macro. As each worker is allowed its own local version of the variables, we had to use a shared array (a special data structure in Julia) in order to maintain an array across all workers.

```
for (k = 0; k < n; k++)
   parallel_for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         // handle min update
```

Figure 3. Research Paper Implementation

```
for k = 1:len
   Threads.@threads for i = 1:len
      for j = 1:len
         # handle min update
```

Figure 4. Our Julia Implementation

3.4. Breadth-First Search

3.4.1. Algorithm Description

Breadth-first search (BFS) is a graph traversal algorithm, which is an algorithm that aims to visit all nodes in a graph. BFS visits all nodes by moving out layer by layer from the source node. One approach to implementing BFS is to use two collections, one representing the current layer and one representing the next layer. This approach lends itself well to parallelisation due to the ability to split up the current layer between processors to parallelise visiting the layer

3.4.2. Parallelisation Techniques

Like Prim's algorithm, BFS consists of an outer loop which is inherently sequential and cannot be parallelised. However, as mentioned in section 3.4.1, the inner loop where the nodes in the current layer is visited can be parallelised by assigning a subset of the current layer to each processor. The BFS algorithms that use this approach for parallelisation are considered "level-synchronous" due to the fact that visiting each

layer is parallelised, but the layers are still visited in order [6]. There is a possible race condition with the parallelisation where two processors might discover the same node due to it being neighbours of both nodes currently being visited. This requires synchronising the operations that are used to mark a node as visited, which could cause contention [7]. This contention, combined with the fact that only a small amount of work in the inner loop can be parallelised, makes it very difficult to effectively parallelise the BFS algorithm and to achieve speedup over the sequential version. Although, this parallelisation could be beneficial on very wide graphs where exploring each level would take long enough sequentially for the parallelisation to be worth it.

4. Benchmarking

4.1. Setup

The general setup for running benchmarks were to run each algorithm on a fixed set of randomly generated graphs. These graphs range in size, with sizes being chosen based on their total runtime. The algorithms are run 20 to 30 times (also depending on runtime) per graph.

Machine specs:

- Intel i5-4570 @ 3.20GHz CPU
- 4 physical cores, no hyperthreading
- 8GB DDR3 RAM

4.2. Prims Algorithm

Our benchmarking for Prim's algorithm consisted of testing three variations in implementation, purely sequential, parallel search (cheapest node) with sequential update for distance vector, and fully parallel (@distributed for search and @threads for update). An interesting observation is that @threads pushed CPU utilisation to 100%, while sequential update with parallel search only had around 45% CPU utilisation. However, @threads interestingly performed roughly the same as its sequential version but using @sync @distributed (omitted) for updating was extremely slow. We can however see that the "speedup" is increasing with larger graphs [Fig. 6]; while this speedup appears to be plateauing, it shows that improvements may be possible. One method of improvement may be using @threads for search parallelisation, yet some work is required to do this correctly due to @threads not currently supporting reduction. The final results were taken from 30 sample runs for each of the three smallest graph sizes (per implementation) and 20 samples for the 20,000-node graph. These are summarised in [Fig. 5].

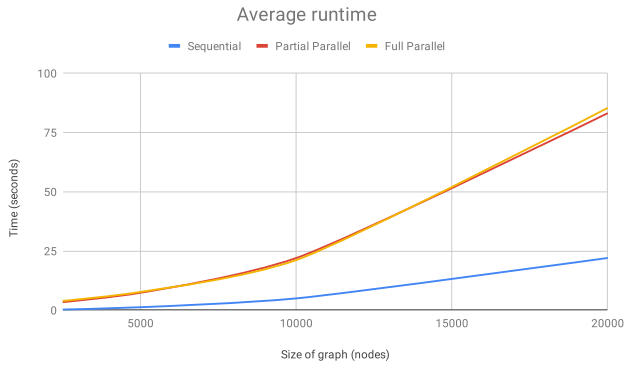

Figure 5. Prims algorithm average runtimes

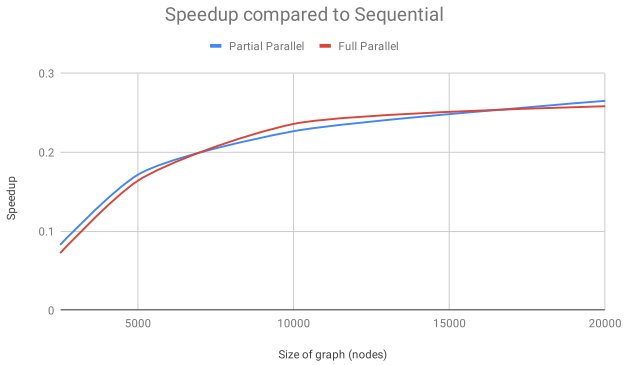

Figure 6. "Speedup" of parallel Prim's algorithm

4.3. Floyd-Warshall Algorithm

Our benchmarking for Floyd-Warshall's all paths algorithm consisted of testing the two implementations, sequential, and fully parallel with @threads. Similar results were shown here as with Prim's, we were unable to achieve any speedup however the slowdown was much smaller [Fig. 8]. This is likely due to the coarser granularity of parallelisation used in this algorithm. We do however see that the speedup drops

at the 1000 node graph, we were unsure why this was the case, yet further tests and benchmarking did not change this outcome. The results were taken from 30 sample runs for the two smaller graph sizes and 20 sample runs for the other two. The average runtimes are shown in [Fig. 7].

During benchmarking, we encountered a strange fault; we noticed that the runtimes for sequential on the 750 and 1000-node netted similar results. This should not have been the case but could possibly be due to the random nature of graph generation; thus, we retested and benchmarked the algorithms, resulting in the 750-node graph to halve in runtime and 1000-node graph to double. These large fluctuations caused concern, so we tested the algorithms on more generated graphs. The 750-node graph runtime remained the same (at half its initial) while the 1000-node graph runtime came back to its original. Over the course of this testing the runtimes were fairly stable with no major fluctuations appearing. We were uncertain what exactly caused this, but we guess that it could be due to the graph's density.

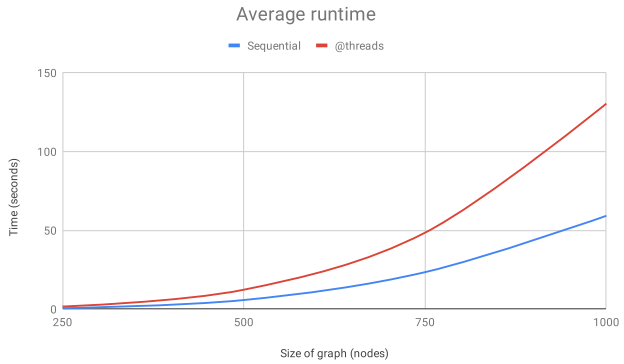

Figure 7. Floyd-Warshall algorithm average runtimes

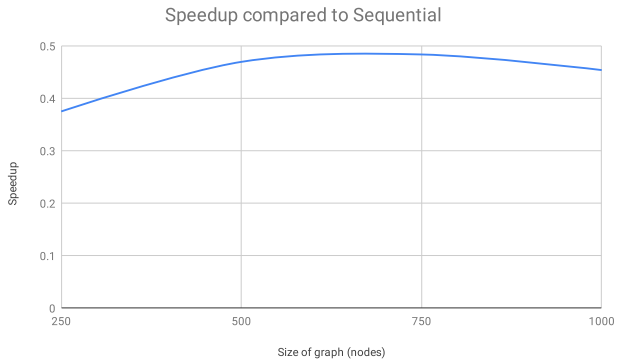

Figure 8. "Speedup" of parallel Floyd-Warshall algorithm

4.4. LightGraphs Comparison

After recording unsatisfying results in benchmarking our implementations, we decided to benchmark a well-known Julia graph library LightGraphs. This library also implemented the Floyd-Warshall algorithm in both sequential and parallel. As a result, we ran benchmarking between their two implementations. As this is an open source project, we were able to read the source code to ensure that the nature of the algorithm was not changed in the parallelisation process. Based on our benchmark results, a dense 1000-node graph which was also used for benchmarking our implementation was slower in parallel than in the sequential version. However, when increasing the nodes to 5000, the parallel version produced a shorter run time, achieving a speedup of 1.29 against the sequential version (median sequential run time: 64.370s, median parallel run time: 49.722s).

4.5. Observations

BenchmarkTools proved very slow for functions with long runtimes, this could be due to some unknown bug or other uncovered issue however certain benchmarks would take far longer than expected. An example being that for many algorithm calls, the benchmark would take at least twice as long to finish as the actual function; one place where this became a major issue was in testing large graphs where we noticed that one benchmark wasn't finishing. The actual algorithm only took 24 seconds (with @time) whereas the @benchmark hadn't finished after over 3 hours later.

4.6. Limitations

We were unable to test larger graphs due to the memory required for Prim's algorithm, as @distributed requires much more memory due to copying of arrays. This problem wasn't encountered when benchmarking Floyd-Warshall's, however, runtime was instead the limiting factor as to get meaningful results multiple samples were needed, but this took very long to complete.

5. Discussion

Overall, we were disappointed with the poor results we gathered from this project. Despite attempting many of the "performance tips" as indicated in the Julia Wiki, we were unable to achieve many improvements. As a result, we investigated some of the reasons this may have occurred, as well as taking additional approaches to mitigate the problems. In particular, the parallel environment in Julia appears to be a serious problem. Despite JuliaLang claiming the environment to be extremely lightweight, providing performance similar to C++ or Fortran, we were unable to achieve anything close. This was backed up by a number of articles online reporting similar performance in parallel implementations. Both of the algorithms which we chose to parallelise have successfully achieved speedup using the same algorithmic approaches in lighter weight languages such as C. This suggests the main reason for our results was not necessarily the algorithmic approach, but rather the parallel environment we were working with. The effects of the parallel environment may have been amplified as a result of needing fine-grained inner loop parallelisation, as this could cause a bigger impact from the overhead associated with thread creation and destruction. The sections below go into more detail about problems encountered, subsequent research carried out, and the approaches taken to mitigate them.

5.1. @threads vs. @distributed

The two main techniques used in this project for parallelisation were the @threads and @distributed macros. We found that when using @distributed, the program ran far slower than the sequential version. When using @threads, the program ran slightly slower, but not to the same degree as the @distributed approach. It is largely unclear why the distributed macro had such an impact on performance when all it is expected to do is split up the iterations of a loop to be executed independently by different threads, which in theory would not cause as much overhead as was observed. A possible reason for this behaviour may be due to the need to use shared arrays. The Julia documentation notes that when using @distributed, the array accessed by the loop must be converted to a shared array so that the output from all threads' execution is visible after the loop has terminated. There could be a large overhead associated with either this conversion or the accessing of the shared array that is not well documented. Alternatively, the reason could be how workers are managed by Julia.

5.2. Memory Usage

An interesting issue we discovered in benchmarking was the extremely large amount of memory which the parallel version was using. The difference was so significant that the parallel version sometimes used up to 15 times as much memory compared to the sequential implementation (Sequential: 2GB vs. Parallel: 43GB). This was surprising as multithreading in Julia is shared memory, so no additional variables were used as a result of the parallelisation. Large amounts of memory allocations clearly indicate a problem with the implementation, however further testing and research showed that this is an inherent problem in Julia multithreading, with memory leaks occurring even merely iterating over a basic loop using @threads. We found some issue comments on the Julia repository suggesting that garbage collection was unable to run with multithreading in Julia. If this is the case, then the results which we experienced with memory allocations can only be expected in the size of the graphs we were benchmarking with. This demonstrates an inherent limitation of the parallel environment in the language.

5.3. Inner Loop Parallelisation

A significant reason for not achieving speedup with the parallel implementation of Prim's algorithm is the fact that the algorithm consists of an outer loop that is inherently sequential and cannot be parallelised. The inner loop of Prim's algorithm is in fact parallelisable through the techniques mentioned in section 3.2.2. However, this parallelisation is very fine-grained so the overhead of creating threads and allocating work to them can easily outweigh the benefits of the parallelisation, leading to the algorithm running for just as long, if not longer than the sequential implementation. This sequential outer loop is a very common issue with parallelisation of graph algorithms and in general makes it difficult to achieve good speedup through using parallelisation. This issue is also present to a lesser degree in Floyd-Warshall, where the "inner loop" is actually

a nested for loop that operates on the whole adjacency matrix. This is less fine-grained than Prim's algorithm, so the slowdown is not as extreme, but this still makes it difficult to achieve good speedup.

5.4. Julia Limitations

Julia has a range of limitations which hindered the parallelisation of algorithms in the project. Firstly, the documentation for Julia is very poor, especially in regard to the parallel features. This made it extremely difficult to determine how functions actually worked, and the pros and cons between them. This was amplified by the "newness" of the Julia language as there were very few additional resources online. Despite looking through all the documentation and carrying out many google searches; we were unable to find accurate documentation on the thread life cycle. As a result, we had to run our own tests in an attempt to determine the lifecycle of threads in Julia. It appears as if threads are created and destroyed each time a parallel region is executed, which would explain the overhead we experienced, although we were not able to definitively confirm this. In comparison, tests carried out on the @distributed macro appeared to show that workers stay alive, which is very confusing considering the extreme overhead exhibited while using this functionality.

Another key limitation which exists in Julia is the lack of customisation available in their parallel package. Both @threads and @distributed do not support any form of manual control like what is possible with OpenMP. As a result, we are forced to use the default block allocation for available threads/workers. In the case of both Prim's and Floyd-Warshall algorithms, the load is fairly balanced so it is unlikely that this would have directly impacted on the performance we obtained; however, it could have been informative to test out other allocation techniques.

6. Conclusions

Parallelisation of graphing algorithms is an important area of research due to the wide applications of the algorithms and the fact that they often need to process large amounts of data, causing them to have slow runtimes. We surveyed existing parallel graphing algorithms and adapted two (Prim's and Floyd-Warshall) to run in Julia, a recent high-performance dynamic programming language aimed at the scientific community.

When implementing these algorithms in Julia, we experienced several issues that caused undesirable performance results. In fact, for both algorithms we observed significant slowdown in the parallel implementations compared to the sequential. To a certain degree, this was not surprising because the graphs we chose to implement (as with many graphing algorithms) consist of an outer loop that is inherently sequential and cannot be parallelised. The parallelisation in these cases occurs only in the inner loop, which often already runs very fast sequentially. This fine-grained inner loop parallelisation can make it very difficult to achieve speedup because the overhead associated with setting up threads quickly outweighs any potential speedup. Even with the inherent sequential nature of the graphing algorithms in mind, the slowdown we experienced was more extreme than it should have been, especially with our Prim's algorithm implementation. We believe these further issues arose from problems in the Julia language, such as poor documentation on the built-in parallelisation tools, lack of control given to the developer, and bugs in the Julia code with their more experimental tools.

Overall, using Julia as a parallel computing tool was not a good experience. The language made it very challenging to observe speed-up when parallelising the graphing algorithms, which is already a challenge in its own right. Much of the Julia language has the makings of a good parallel computing tool, such as its simple syntax to parallelise a loop that mirrors the style of OpenMP, yet many problems need to be solved before it would be better to use for parallelising graph algorithms than a lower-level language like C.

References

[1] JuliaLang, "Home - The Julia Language", *JuliaLang*, 2019. [Online]. Available: https://docs.julialang.org/en/v1/index.html.

[2] JuliaLang, "Parallel Computing · The Julia Language", JuliaLang, 2019. [Online]. Available: https://docs.julialang.org/en/v1/manual/parallel-computing/index.html.

[3] Ananth, A., Kumar, V., Karypis, G. and Gupta, A. ed., (2003). Minimum Spanning Tree: Prim's Algorithm. In: Introduction to parallel computing, 2nd ed. Addison-Wesley Professional.

[4] R. Barr, R. Helgaon and J. Kennington, "Minimal spanning trees: An empirical investigation of parallel algorithms", *Parallel Computing*, vol. 12, no. 1, pp. 45-52, 1989. Available: 10.1016/0167-8191(89)90005-7.

[5] Parallelizing the Floyd-Warshall Algorithm on Modern Multicore Platforms: Lessons learned, Students of the Parallel Processing Systems course, School of Electrical & Computer Engineering, National Technical University of Athens

[6] R. Berrendorf and M. Makulla, "Level-Synchronous Parallel Breadth-First Search Algorithms For Multicore and Multiprocessor Systems", 2014.

[7] E. Elbre, "A Tale of BFS: Going Parallel", Medium, 2018. [Online]. Available: https://medium.com/@egonelbre/a-tale-of-bfs-going-parallel-cdca89b9b295.

Appendices

Task	Luke	Darcy	George (Zhi)
Research			
Prims	40%	30%	30%
BFS	0%	100%	0%
Floyd-Warshall	40%	40%	20%
Julia Parallelism	40%	30%	30%
Implementation			
Graph Parsing / Writing	95%	5%	0%
Graph Generation	0%	0%	100%
Test Runner	95%	5%	0%
Benchmarking	5%	5%	90%
Run Script	0%	0%	100%
Prims - Sequential	50%	30%	20%
Prims - Parallel	33.3%	33.3%	33.3%
Floyd - Sequential	100%	0%	0%
Floyd - Parallel	5%	95%	0%
Other			
Report	33.3%	33.3%	33.3%

Appendix. 1. Contributions by group member

Appendix. 2. Repository Link: https://github.com/lukethompsxn/751-Project