SOFTENG 751: Group 14 - Project 5

Aorthi Afroza, Blain Cribb, Kelsey Murray
Department of Electrical, Computer, and Software Engineering, University of Auckland
GitHub Repo: https://github.com/KelseyRM/SOFTENG751_Group_14

Contribution to Work Item	Aorthi	Blain	Kelsey
Initial Research into Julia and Graphing Algorithms	33%	33%	33%
Presentation	33%	33%	33%
Implementation of Prim's I	100%	0%	0%
Implementation of Prim's II	0%	100%	0%
*Implementation of A**	0%	0%	100%
Benchmarking and ReadMe	20%	45%	35%
Report	45%	20%	35%

1. Introduction

Graphing algorithms can be used in a number of contexts to solve real-life problems, such as creating communication networks and roading networks. These contexts often require such graphing algorithms to be fast, however with large and/or complex graphs the algorithms can be slow. One way to speed up these algorithms is to parallelise them and we were tasked with implementing this in a relatively new programming language called Julia. This report details the parallel implementation of two graphing algorithms in Julia, and evaluates the speedup and correctness of these parallel versions. These chosen algorithms are the A* shortest path algorithm and Prim's Minimum Spanning Tree algorithm.

2. Background

2.1. Julia

Julia is a high level, general purpose language designed for high-performance numerical analysis and computational science. It is an open-sourced language with a vast amount of formal documentation that is constantly developing and evolving. First released in 2012, Julia is considered to be a relatively young language and its current major release version is 1.0 (as of 2018). Currently, multithreading is described to be "experimental" in the official Julia documentation. It has methods implemented that allow the parallelisation of loops on multiple threads, which ensure all threads have completed a loop before execution continues. For thread safety, it also has a number of types of locks and atomic variables available.

2.2. Graphing Algorithms

2.2.1. Prim's Minimum Spanning Tree

Prim's Algorithm constructs a minimum spanning tree (MST) for a graph; that is, it creates a tree that connects all nodes in a graph such that it has the least total cost among all possible trees that connect the nodes. It is a greedy algorithm that builds the MST incrementally; in each iteration it chooses the lowest-weighted path of all currently available paths.

2.2.2. A* Shortest Path
A* Algorithm generates a shortest path by finding the lowest-weighted edges between given start and end vertices. The path returned is the true shortest path from start to end. It is a best first search that is not greedy, since it considers the current best option without relying on it exclusively. The path is built incrementally by exploring the reachable vertices with the lowest expected cost.

3. Design Decisions

3.1. Datasets

3.1.1. Prim's
The dataset for testing both implementations of Prim's algorithm are collections of randomly generated graphs with varying number of vertices and connections between them. These graphs are generated using a built in barabasi / albert method with additional functionality for randomising weights of edges. Each weight within the graph should be unique, as this ensures there is only one possible MST which allows us to compare our solution against the base library's to verify correctness.

3.1.2. A*
The A* algorithm dataset we have used is a collection of randomly generated mazes on an X-by-X grid. These graphs are generated using a modified version of Prim's algorithm which adds a random edge to the tree, rather than the lowest cost edge. The edges of the graph have uniform edge weights meaning equal length paths are equal in cost.

3.2. Graphing Algorithms
When deciding which algorithms to implement, a number of factors were considered. Firstly, we wanted to pick two different types of graphing algorithms as we hoped this would allow us to experiment with a wider range of Julia's tools. Secondly, we wanted to choose algorithms that had substantial existing literature around parallelising them, with at least one successful implementation.

During our initial research into Julia, we discovered the LightGraphs.jl package, which contains parallel implementations of a number of popular graphing algorithms. Since this project is a research project, we decided to eliminate any algorithms that were in this package from our options as we felt there was little point in attempting to implement algorithms that were already done. As a result of this, we chose to implement Prim's (an MST) and A* (a shortest path). The literature surrounding parallelising Prim's primarily focussed on two different approaches, one of which produced some conflicting results. Taking this into consideration, we decided to implement two different approaches to Prim's algorithm as we wanted to explore the results of these differing approaches.

3.3. JuliaBox
The development for this project was largely done on JuliaBox, a Jupyter notebook that allows users to write and run Julia scripts in browser without installation. This made it easy to take a fail-fast approach to our development process as sections of a script could easily be run in isolation to test its correctness. JuliaBox also automatically saves the current script as it is being written and allows the user to revert to earlier versions if necessary which allowed for easy version control.

4. Implementation

4.1. Prim’s I

4.1.1. Existing Literature

Our first approach to parallelising Prim’s algorithm was based off three implementations of the same approach, which had produced mixed results. The implementations done in OpenMP at PES University [1] and University of California, Irvine [2] both achieved speedup with their parallel algorithms. However, the same algorithm implemented in Python at Reed College, Oregon [3] did not achieved speedup. These studies show that the implementation of the same algorithm in different languages can give different results and motivated us to try and see what results we might get when implementing the same algorithm in Julia.

4.1.2. Our Implementation

Our implementation is adapted from the literature described above. The minimum spanning tree is built by first finding the current lowest available weight using the minKey function. This function takes in the current key array and each thread it allocated a section of this array, where it calculates its local minimum. This is then reduced to a global minimum using a mutex to protect the variables. In the main function, we then set this node to be visited. Threads are then used to iterate through all the possible nodes that can be visited from this node. If an unvisited node can now be reached with a lower weight than what is currently stored in the key array, then this value is updated. This continues until there are no longer unvisited nodes.

The main parallel features of Julia utilised in this implementation are the use of the @threads macro to parallelise for loops and the use of a mutex to create a critical section. We also experimented with using atomic variables, however this did not achieve the same level of correctness as using a mutex lock due to the experimental nature of Julia’s parallelisation features.

4.2. Prim’s II

4.2.1. Existing Literature

The second approach to parallelising Prim’s algorithm was based off an implementation developed at the National University of Singapore [4]. This implementation achieved speedup using C++ and some GPU parallelisation. We were motivated to attempt this algorithm on Julia so that we could compare the performance of a higher level language with a lower level language like C++.

4.2.2. Our Implementation

This implementation is adapted from the pseudocode present in the literature described above. The only shared variable between threads is a successor array of atomic variables which keeps track of which vertices are connected. Each thread independently creates a minimum spanning tree until they meet another tree at which point they join and restart with a new tree. This process is repeated until all vertices in the graph have been visited. At this point there is still a chance of unconnected islands of vertices which need to be joined to achieve the overall MST. To amalgamate these islands we run a unification step where we trace through the successor array to find which vertices are separated and join them with the minimum edge to another island.

A parallel features of Julia used for this implementation are the @threads macro to parallelise the creation of subtrees. We also use atomic variables to ensure the successor array is accessed in a thread safe manner.

4.3. A*

4.3.1. Existing Literature

A number of approaches to investigate the A* algorithm were discovered while researching previous attempts to parallelise this algorithm. The two main approaches differed essentially in the loop that was to be parallelised - the outer loop, which would mean more of the algorithm could run in parallel, and the inner loop, which has the potential to save overhead from having to protect shared data [5]-[6]. Both approaches achieved speedup while remaining correct [5]-[6], but since having a larger proportion of code running parallel is ideal when parallelising code we chose the first approach in our implementation.

4.3.2. Our Implementation

Our implementation is adapted from an implemented sequential version of the A*algorithm, and uses concepts and recommendations drawn from the research to make the algorithm parallelisable[5]-[7]-[8]. A priority queue is used to track the list of “reachable” vertices, since the remove-best and insertion nature that the algorithm prefers matches well with this. While the next element on the list of “reachable” vertices is not the goal node, the next vertex is dequeued off this list, and its neighbours are investigated to check if they could lead to an optimal path. The inner loop investigates these neighbours, and while some literature indicated speedup could still be achieved with this there was more literary support for parallelising the outer loop, which we have done using Julia’s @threads macro.

Since Julia’s range of thread-safe data structures is quite limited at this stage, locking is used to ensure the shared data is protected. Two locks were considered during implementation - the Mutex and the SpinLock. Julia documentation states that SpinLock *can* be more lightweight and efficient for fewer threads and finer-grained locking, while Mutexes are better for instances where the lock is to be held for a “considerable length of time”. Since we had no indication of what “considerable” relatively meant, both locks were tested. The Mutex was selected since it halved the execution time - we have attributed this to likely be due to some sections of the code where there are multiple accesses to the shared data. The entire section must therefore be thread-safe to avoid ABA problem.

5. Results and Evaluation

When evaluating the outcomes of our implementations we chose to focus on two aspects, correctness and runtime. The testing environment used to conduct the testing for our results was on Windows on a Intel Core i1-4790 @ 3.60GHz (4 CPUs). The tests were run in isolation with only the Julia command line present and no other processes running the background.

5.1. Approach

5.1.1. Correctness

As an incorrect algorithm is not viable, we ensured that our implementations consistently gave correct results before moving forward with any other kind of benchmarking. This was done by generating a number of graphs and running the base library’s implementations of Prim’s and A* on these. We then ran our implementations of these algorithms on the generated graphs and compared the respective outputs using an IsSame() function that we wrote. This function checks if the MSTs and paths generated from our implementation match the edges generated by the base library, regardless of their direction.

5.1.2. Runtime

Since the purpose of parallelising graphing algorithms is to reduce runtime, we wanted to correctly benchmark our implementations so that we could compare our results with their sequential versions. To do this we used the built-in BenchmarkTools package, which runs a function a number of times and returns statistical timing values. In each benchmark the algorithm was run using varying graph sizes, from 100 to 10,000 vertices, so that we could get an idea of how the runtime scaled with larger datasets. Each sample generated a different graph - this generation time was not considered as part of the algorithms runtime. The benchmark would run the function 1000 times per graph-size, and we recorded the mean runtime of the algorithm for later use in evaluating speedup. The algorithms were benchmarked first on a single thread, and then on four threads so that an accurate comparison could be made. For Prim's algorithm we also considered dense graphs - defined here as graphs with fifty edges per vertex - against their less dense counterparts (eight edges per vertex) to see how each implementation scaled with edge complexity.

5.2. Results

5.2.1. Prim's I

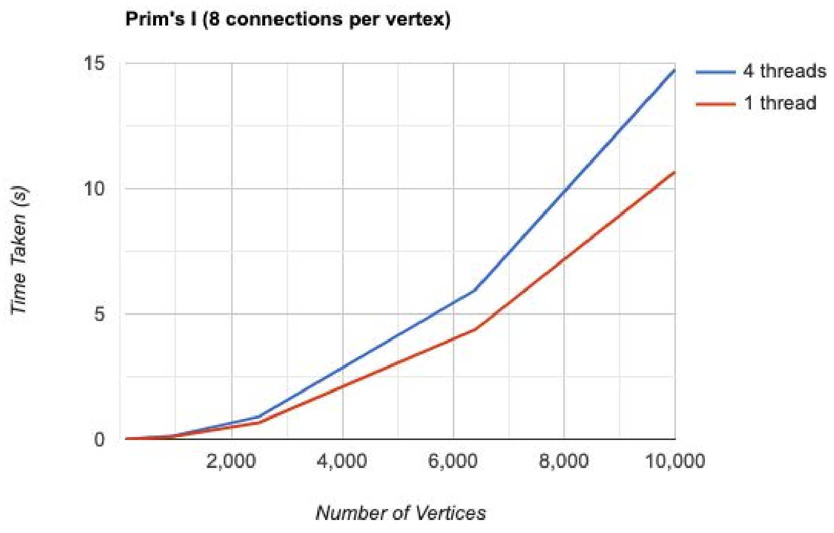

Figure 1.0: Graph of Prim's I Results, 8 connections per node

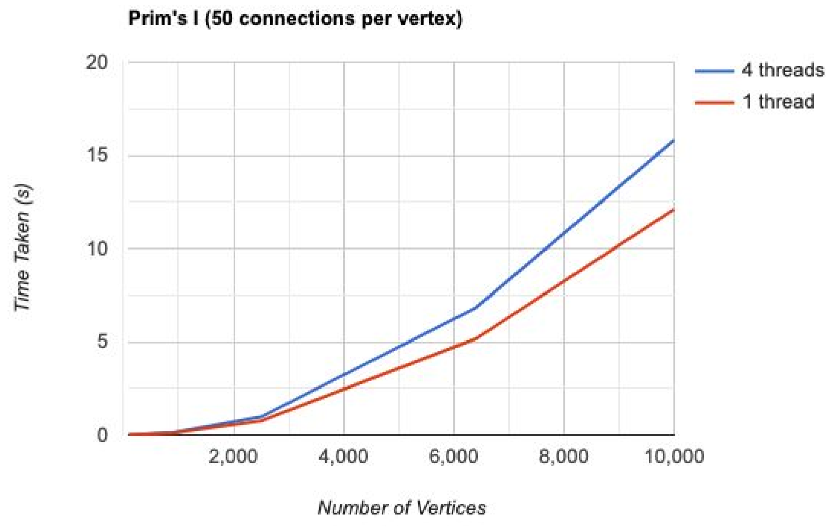

Figure 2.0: Graph of Prim's I Results, 50 connections per node

As shown in Figures 1.0 and 2.0, there was no speedup achieved for the Prim's I implementation between the serial and parallel runs with the both having a speedup of roughly 0.7-0.9. Interestingly, as the number of vertices increased, both the serial and parallel results scaled in a similar manner and this suggests the parallel processing of data in this implementation did little to change the runtime. Furthermore, the Prim's I implementation had quite similar runtimes even while changing the density of the connections between vertices and this suggests that the complexity of the algorithm depends more on the number of vertices rather than the number of edges within a graph.

Compared to Prim's II, this implementation has a significantly higher runtime for both the serial and parallel runs.

5.2.2. Prim's II

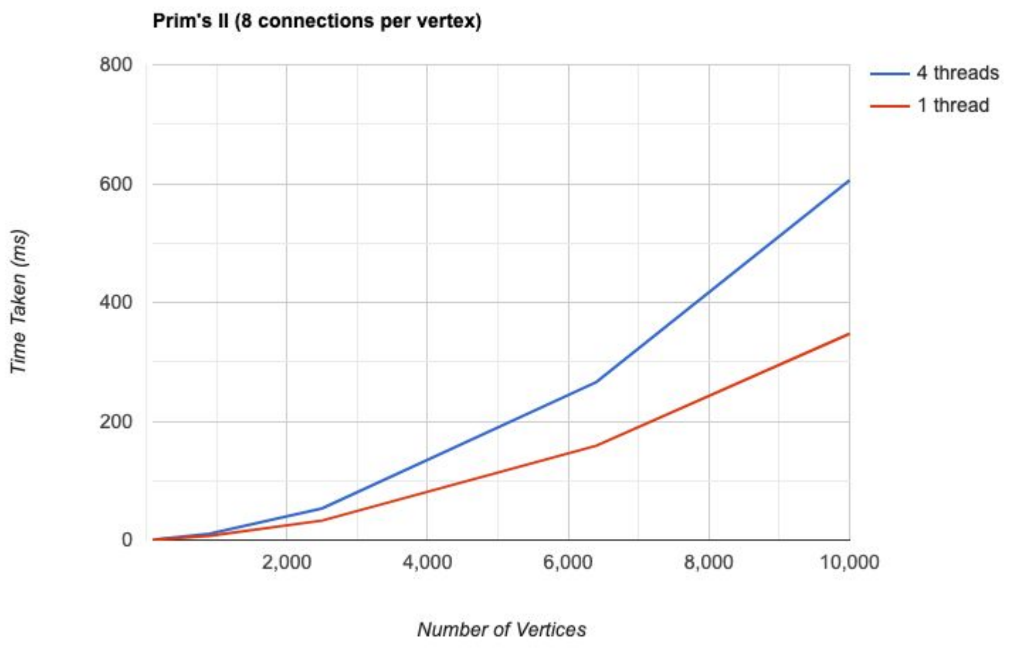

Figure 3.0: Graph of Prim's II Results, 8 connections per node

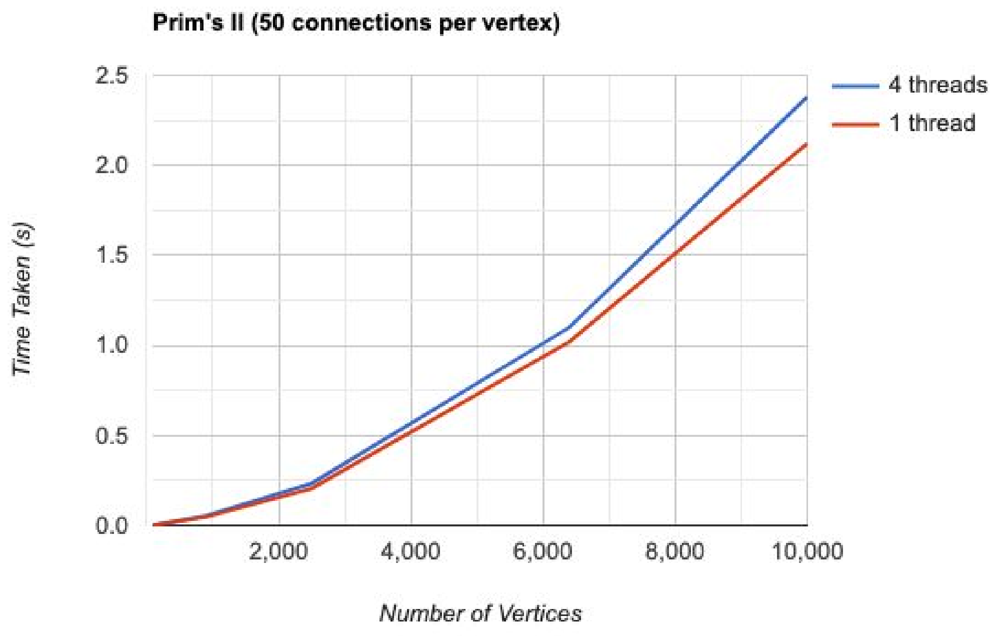

Figure 4.0: Graph of Prim's II Results, 50 connections per node

Figures 3.0 and 4.0 show that no speedup was achieved in the Prim's II algorithm for both densely and not densely connected graphs, the speedups of these lying between roughly 0.84 - 0.92 and 0.57 - 0.71 respectively. In both cases, the threaded version seems to scale worse for larger numbers of vertices compared with the serial version, though densely connected graphs appear to be less affected. As for increasing the number of edges, Prim's II scaled much worse for densely connected graphs, slowing down almost five times as much in some cases.

Compared with Prim's I, Prim's II achieved some speedup for similar sized graphs, in terms of vertices, though appears to scale much worse when increasing the edge density.

5.2.3. A*

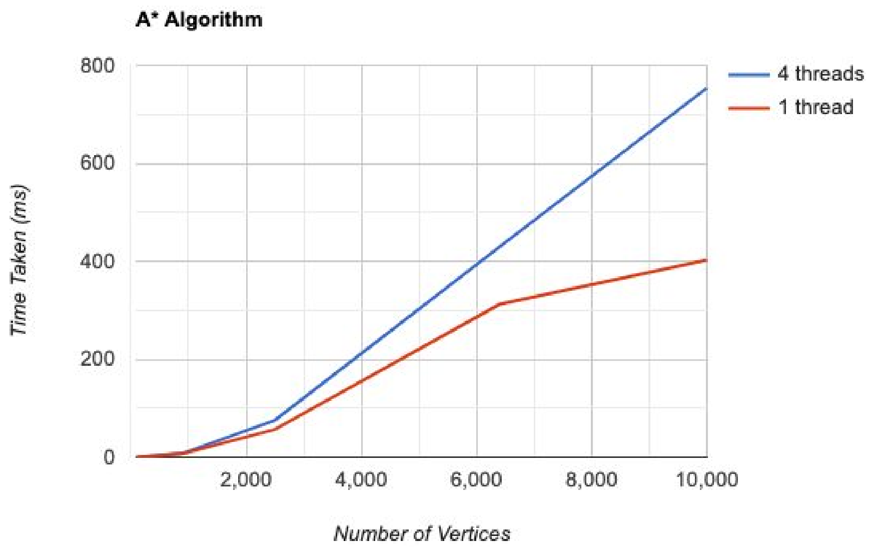

Figure 5.0: Graph of A Results*

As shown in Figure 5.0, there was no speedup achieved for A* algorithm between the serial and parallel runs. Neither implementation scales particularly well, and interestingly the multi-threaded version does this worse than the serial, contrary to expectations. The speedup achieved on smaller graphs, i.e. 100-1000 vertices was in the range of 0.96-0.99, which indicates a very similar run time between the versions with these sizes. Bigger graphs, i.e. 2,500-10,000 vertices, achieved much less speedup, which progressed linearly to be in the range 0.75-0.53.

6. Limitations and Challenges

From our results it is clear that our implementations of parallel graphing algorithms in Julia did not achieve the speedup we set out to achieve. There are a number of reasons behind these findings, outlined in this section.

6.1. Algorithm Options

The given brief was to parallelise graphing using Julia, however majority of the easily parallelised graphing algorithms have already been implemented in the LightGraphs package. This package was developed and peer-reviewed by contributors in the Julia community who most likely have significantly more experience and familiarity with Julia and its capabilities compared to us. It would have been highly unlikely that we would be able to produce algorithms with greater speedup than they had already achieved within the given timeframe. As a result, our options for algorithms to parallelise were greatly limited, which meant the ones we chose were not the most suitable to parallelise.

6.2. Parallelising Graphing Algorithms

When parallelising an algorithm, it important to balance the speedup achieved by the parallel processing of the data, and the overhead introduced by the use of threads. This is especially true for high level languages like Python and Julia as they introduce even more overhead than lower level languages like C [3]. A significant issue with the Prim's algorithms was that the nature of this algorithm made it impossible to parallelise its outer loop. Instead we had to parallelise the inner loop, which introduced a large amount of overhead, slowing the algorithm down.

6.3. Julia

Another limitation of this project was Julia itself. Julia is considered to be a young language and there are areas of it that are still developing, meaning it does not have the full breadth of capabilities needed for the context we are using it in. In particular, since multithreading is still considered to be experimental there are a lot of features found in other languages, such as Java's ExecutorService and OpenMP's reductions, that do not exist or are not implemented yet in Julia. Julia also does not have a wide range of thread-safety options - a lot of data structures are not thread-safe on their own, and more complex ones cannot be made atomic. This means the only option is to use locks, which adds a lot of contention between threads and introduces overhead for lock creation and use. This affected A* significantly, since it requires a substantial amount of shared data to be protected, but this cannot be done effectively in Julia where the thread-safety capabilities are very limited.

Furthermore, since Julia is a constantly evolving, open source tool it contains some packages that are not stable and this creates a number of issues. It can mean that some functionality is not necessarily as expected, as we found with our attempts at using atomic variables, and it also means that documentation and community support can become outdated.

7. Discussion and Conclusion

The parallelisation of the graphing algorithms Prim's MST and A* in Julia did not produce speedup. This is largely due to the fact that a large portion of algorithms that more readily lend themselves to being parallelised have already been implemented in Julia's LightGraphs package. The algorithms chosen had a number of limitations, with Prim's being limited by the sections of the algorithm that are able to be parallelised introducing overhead and A* being limited by Julia's capabilities. Julia is still in an experimental stage when it comes to multithreading and this introduces a number of limitations to our implementations. One such limitation is the lack of fast thread-safe data structures which forced us to use less ideal methods to handle synchronization errors, such as the mutexes within the A* implementation.

Given more time, there is definitely room for improvement in our implementations. In general, the code could be more optimised for the Julia compiler and use some of Julia's more obscure functions that may produce better results but are not as well documented. Additionally, using a language with more multithreading support, such as Java, would likely produce better results than the ones achieved using Julia.

References

[1] S. Dutta, *et al*, "Development of Gis Tool for the Solution of Minimum Spanning Tree Problem using Prim's Algorithm," no. 8, pp. 1105–1114, Nov. 2014.
[2] P. Rupala, "Parallel Implementation of Prim's Algorithm Using OpenMP."
[3] P. Shah, "Parallel implementation of Prim's algorithm," 2019
[4] S. Nobari, T.-T. Cao, P. Karras, and S. Bressan, "Scalable parallel minimum spanning forest computation," vol. 47, no. 8, p. 205, Sep. 2012.
[5] Zaghloul, S., Al-Jami, H., Bakalla, M., Al-Jebreen, L., Arshad, M. and Al-Issa, A. (2017). Parallelizing A* Path Finding Algorithm. International Journal Of Engineering And Computer Science, 6(9), pp.22469-22476.
[6]R. Inam, "A* Algorithm for Multicore Graphics Processors," 2009
[7] A. Fukunaga, A. Botea, Y. Jinnai, and A. Kishimoto, "A Survey of Parallel A*," 2017
[8] William Miller, "Applying Parallel Programming to Path-Finding With the A* Algorithm," 2017, pp. 237–238.