CSE305 Report

Sunho Hwang and Diego Gomez

June 13, 2021

The shortest path problem has countless applications as a fundamental combinatorial optimization problem. Dijkstra's algorithm is a well known solution to the single source shortest path problem (SSSP). The goal of our project is to therefore explore and implement parallel version of such algorithms. This report will thus outline our approach to the project along with the results and some analysis. The first section will discuss the parallelization of Dijkstra's SSSP as presented by Crauser et al [2] while the second takes the approach of graph partitioning and iterative correcting as shown in [6].

1 A parallelization of Dijkstra's shortest path algorithm

1.1 The algorithm

In this algorithm, we aim to subdivide Dijkstra's algorithm into delete phases and then implement it with parallelization. While there are multiple variants presented in the paper, we will be implementing the OUT-variant where we compute the thresholds defined via the weights of the outgoing edges. The paper also uses a random graph with random edges and we will do the same with a graph model $\mathcal{G}(N, \frac{d}{N})$ where N is the number of nodes and each theoretically possible edge is included into the graph with probability $\frac{d}{N}$. We also denote $c(v, w)$ as the cost of going from node v to node w. The parallel version of the algorithm can then be implemented as follows:

1. The algorithm handles one global array of tentative distances while each processor possesses two sequential priority queue (Q, Q^*). Q handles the tentative distances while Q^* is defined by $tent(v) + \delta_0(v)$ where $\delta_0(v)$ is the minimum weight of the outgoing edges from the node v.
2. We then compute the global minimum L of all Q^*s and each processor removes from their queues nodes with $tent(v) \leq L$. We denote by R the union of these deleted nodes.
3. The processors then cooperate to generate the set $Req := \{(w, tent(v) + c(v, w)), v \in R\}$. We can then update $tent(w)$ if needed with the values in Req while calling **decreaseKey** on our priority queues accordingly.

We repeat this process until all the queues are empty. The implementation of this algorithm can thus be found in the `Dijkstra` folder at the `ParDijkstra.h` file. Before we present the results, it is important to note that all of the complexities presented in the paper heavily relies on the complexities of the operations related to the relaxed heap as presented in [3]. Most importantly their **findMin**, **insert** and **decreaseKey** can be done in constant time while deleting nodes can be done in logarithmic time depending on the size of the queues. Due to the complications related to implementing this data structure, we have opted for implementing a min binary heap by using the STL `makeheap` function, which can be found in `Heap.h`. In this implementation, we have that **findMin** runs in constant time but **insert** is logarithmic while deleting nodes is upto twice logarithmic. Most importantly, **decreaseKey** can go up to linear time. More detail on the complexities related to this structure can be found in [1]. We can thus expect these aspects to have a significant impact on our performance. Finally, for the tools required to parallelise, since the algorithm was written with parallelisation in mind, we simply had to use locks with `std::mutex` at the appropriate places. Most notably, we used it before calling **decreaseKey** as each processor may make changes to queues handled by other processors. We can now move onto the results of our simulations.

1.2 Results

The correctness of the following results were verified by directly comparing with the output of the greedy sequential Dijkstra algorithm. In this section, we will be fixing the probability of a node to have an outgoing edge as 0.3.

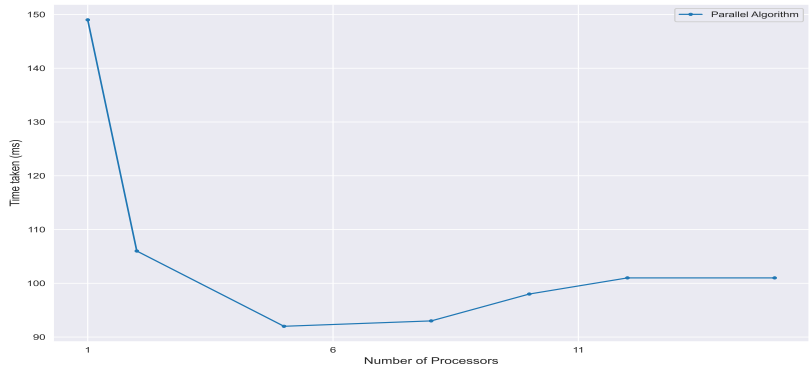

Figure 1: Time taken for by the algorithm for 1000 nodes with varying number of processors.

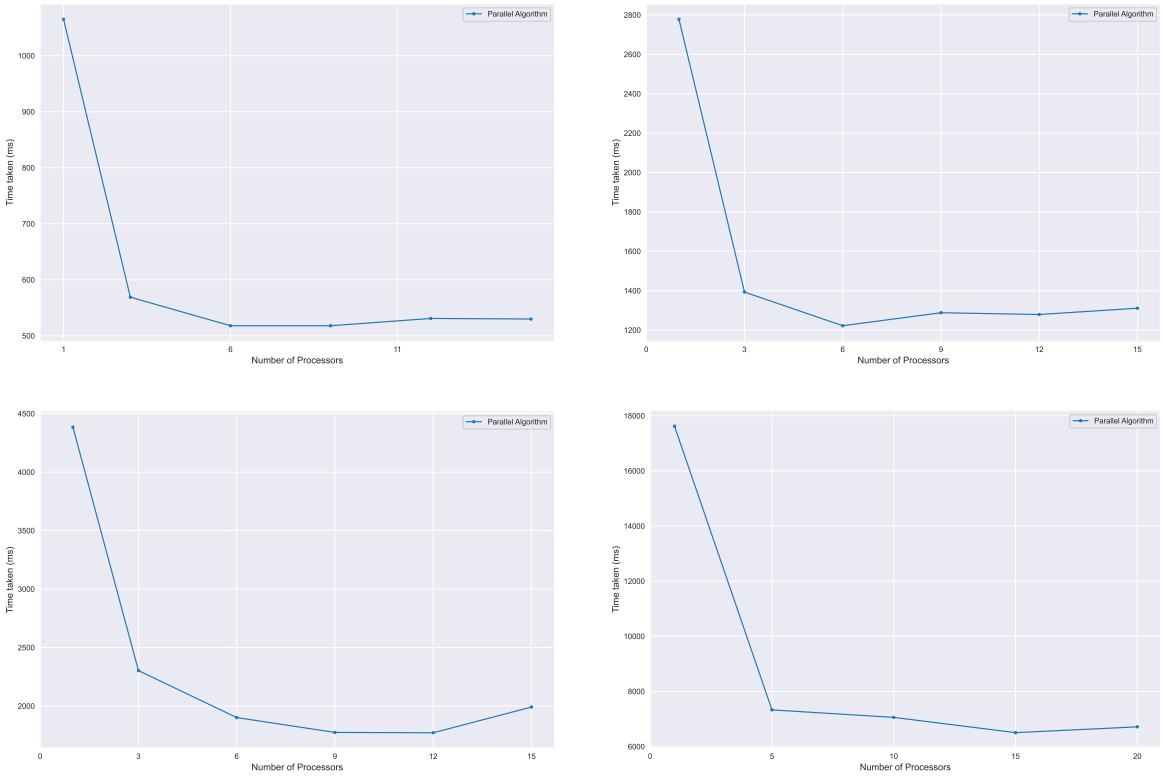

Figure 2: Results with $2500, 4000, 5000$ and 10000 nodes from left to right.

As we can see from both Figure 1 and 2 this algorithm runs faster than its sequential version. However, simply using more processors does not automatically mean better results. Indeed, the paper mentions using $p = \frac{n}{rlogn}$ processors where $r = \mathcal{O}(\sqrt{N})$ phases. Plotting this with varying number of processors shows us that: By comparing the values as in Figure 3 and the number of nodes that yield the best times in Figure 2, we can see that they are indeed coherent.

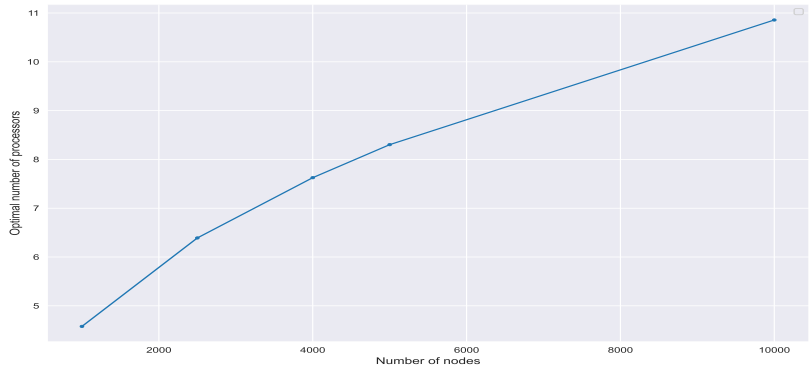

Figure 3: Optimal number of processors with varying number of nodes.

1.3 Final thoughts

Finally, we must mention that most likely due to the complexities related to the data structure used, we were not able to output better performance than the naïve Dijkstra's algorithm. However, it is interesting to note that as we increased the number of nodes, we were able to bridge the gap, going from 4 times slower to 2 times slower from 1000 to 10000 nodes. Furthermore, by decreasing the probability of having an outgoing edge to 0.02 which is more closer to real-world scenarios (a road is most definitely linked to much less than 2% of other roads in large cities), we can essentially perform twice as fast, as shown by 4. Therefore, in order to take this project further, it would be interesting to optimise our data structures and see how much we could improve the performance even more.

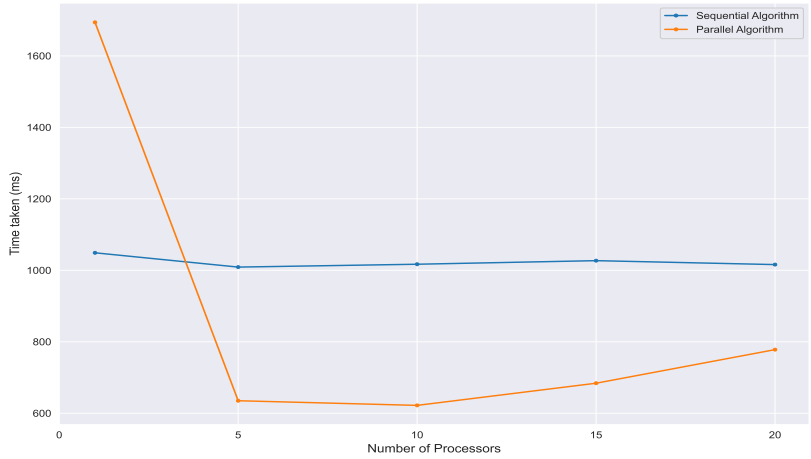

Figure 4: Sequential (blue) and Parallel (orange) Dijkstra algorithm with 5000 nodes and varying processors (only applies to parallel algorithm).

2 Parallel Shortest Path Algorithm Based on Graph-Partitioning and Iterative Correcting

In the following another algorithm that solves the SSSP based on Graph-Partitioning and Iterative Correcting presented by Tang et al [6] was implemented.

2.1 The algorithm

This algorithm is a parallelization of a variation of the well-known Dijkstra algorithm, called "The sequential Dijkstra Algorithm with Two-Queue". As it is implied in the name, this algorithm requires us to introduce a new data structure names Two-Queue. The details of it can be seen in the paper [6], the main asset of this data structure is that we insert in a different manner Nodes that are seen for the first time and already visited nodes. This algorithm has complexity $O(n^2m)$[5], but is proven to be most efficient on sparse graphs [7]. In order to implement the parallel version of this algorithm we must use the help of graph-partitioning [4].

The parallel version of the algorithm [6] can be briefly stated as follows. We start by partitioning the graph, this is the Graph-Partitioning phase. Each Sub graph is then assigned to a process. Afterwards the algorithm iterates the following two steps.

1. Computation Step: Each process compute the local shortest path of their corresponding Sub Graph

2. Communication Step: Processes send Message-Arrays containing information related to the Boundary Nodes, nodes connecting two Sub Graphs together.

the algorithm then iterates until in step 2 no message is exchanged, this is the Iterative Correcting phase.

2.2 Implementation

Let us now discuss our implementation. In the file `Graph.hpp` the basic classes for Node,Edge,Graph and Sub Graph were implemented from scratch. Each of these classes have the usual function and attributes one would expect, with the addition of functions to transform graphs into a desired form, such as a Matrix, or the format used in the METIS library [4].

The implementation of this algorithm can be found in `graph_part.cpp`. In this file we have the implementation of the TwoQueue data structure as well as the sequential version of the algorithm at the top. Afterwards, one can find two auxiliary functions, one for the computation Step of the algorithm and another for the communication step. Finally function `ParallelSSSP` implements the algorithm.

Let us dive into the details of the parallelization tools that were used for this algorithm. Firstly, in the `ComputationStep` function, we start by computing the local distances, and we maintain for each processes a local vector of vector of Edges that will help us transmit the necessary information to the adjecent Sub Graphs. This vector stores in the position k the list of Edges connecting different Sub Graphs involving the $k-th$ Sub Graph. After the processes is done computing the distances and filling the local message array, it is time for it to communicate it to the other processes. In order to do this we maintain a global Message Array analogous to the local one, in which all processes can send and receive the messages. In order to ensure that multiple processes do not send a message to the same slot of the Message Array at the same time we maintain a vector of locks (`message_lock`). Thus, every time a processes i seeks to send a message to a processes j a lock on the $j-th$ position of the Message Array is called.

Then in the `CommunicationStep` function. As one can see in the `ParallelSSSP` function, we have to wait for all threads to be done with `ComputationStep` before moving on to the `CommunicationStep`. Therefore here we are sure that only one process k will read MessageArray[k]. Nevertheless another problem arises, multiple processes are reading and writing in the same slots of the final distance array d at the same time, since now we are dealing with the Boundary Nodes. Thus we must lock the read and written distance, which we do by maintaining a vector of locks for all slots of d. In order to avoid deadlocks we take the general rule of locking the node with the highest key first.

2.3 Correctness and Performance

The SSSP problem is well-known to be able to be solved by the Dijkstra algorithm. Thus in order to test the correctness of our algorithm we implemented said algorithm, and compared its results with our algorithm's results. Moreover let us specify that since the TwoQueue Dijkstra algorithm is most efficient in sparse graphs we will be using highly sparse graphs. In the following for a graph to have a sparsity of $x\%$ will mean that each node has a probability of $x\%$ of having an edge with any given different node.

Let us now take a look at the performance analysis. This algorithm was ran in Macbook Pro 2020 with the following processor, 1.4 GHz Quad-Core Intel Core i5. We can see in figure 4 that as expected the time of execution starts by quickly dropping and stagnates when it reaches the number of cores of the machine, 8 in our case.

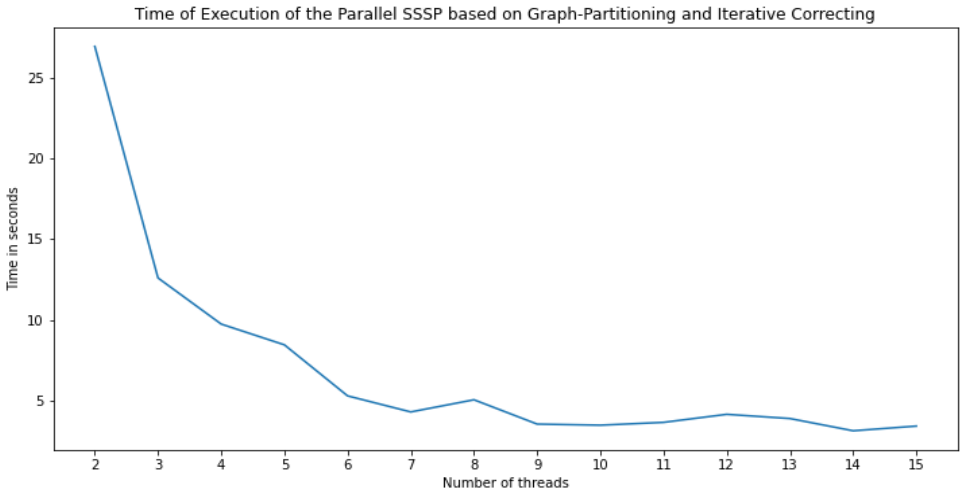

Figure 5: A graph with 20k nodes was used.With a Sparsity of 10%

Nevertheless, it is important to note that the Time of Execution for this algorithm deeply depends on the sparsity of the given graph. Indeed, as one can see in Figure 5 the difference is astonishing. As a reference keep in mind that the traditional Dijkstra algorithm executes in 5 with a graph composed of 20k nodes, regardless of the sparsity.

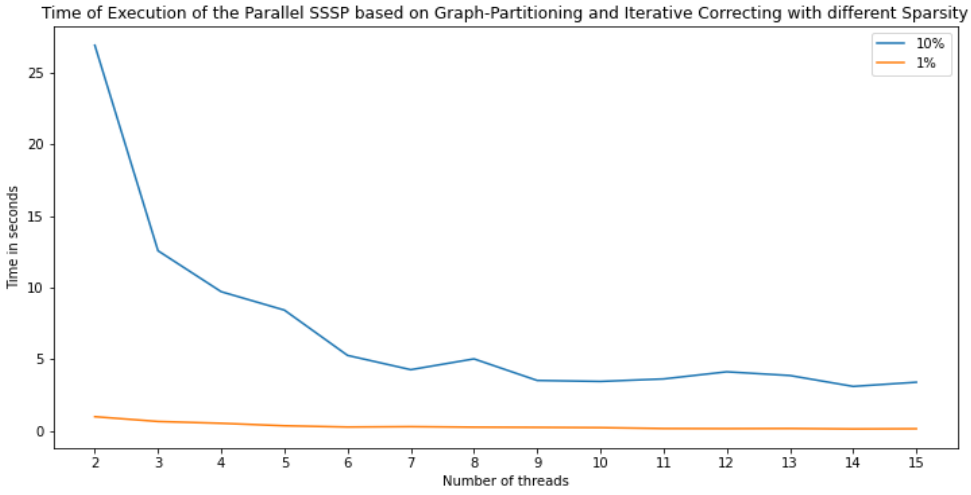

Figure 6: A graph with 20k nodes was used. Comparing 1% and 10% Sparsity.

Finally, there is still room for improvement in our implementation. As one can see in Figure 6, surprisingly the parallelized version of our algorithm is not always better than the sequential one. This can be explained by the fact that with the parallelized version of the algorithm we make use of our data structures much more. This is a problem since these data structures had to be implemented from scratch and thus are not optimal.

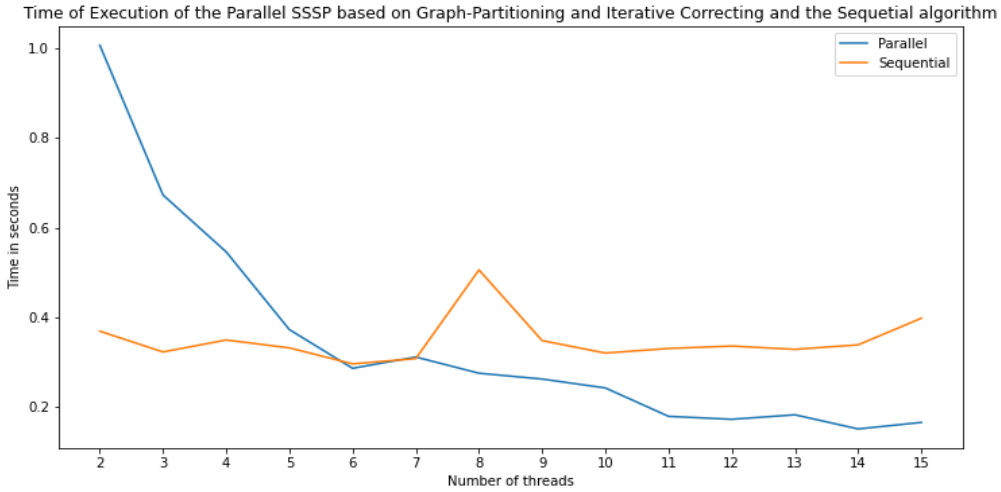

Figure 7: A graph with 20k nodes was used. Sparsity of 1%

2.4 Last Thoughts

As a future work on this part of the project it would be interesting to keep working and optimising the data structures used for the algorithm. This would allow us to then test our project with real life data sets such as Roads Graphs.

As of now we have already a fully working software that can be applied to small really sparse graphs (10k-20k nodes) which yields significantly better results than the traditional Dijkstra algorithm. Moreover, the Graph-Partitioning that was done did not depend on the weight of the edges. Thus this algorithm could be applied to dynamically recompute the shortest path from a node, with changing weights with an acceptable delay (less than half a second), since we do not need to recompute the partition at each change.

References

[1] URL: https://www.cplusplus.com/reference/algorithm/make_heap/.

[2] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. "A parallelization of Dijkstra's shortest path algorithm". In: *Mathematical Foundations of Computer Science 1998*. Ed. by Luboš Brim, Jozef Gruska, and Jiří Zlatuška. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 722–731. ISBN: 978-3-540-68532-6.

[3] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. "Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation". In: *Commun. ACM* 31.11 (Nov. 1988), pp. 1343–1354. ISSN: 0001-0782. DOI: 10.1145/50087.50096. URL: https://doi.org/10.1145/50087.50096.

[4] *METIS*. URL: http://www.cs.umn.edu/~metis.

[5] Stefano Pallottino. "Shortest-path methods: Complexity, interrelations and new propositions". In: *Networks* 14.2 (1984), pp. 257–267.

[6] Yuxin Tang, Yunquan Zhang, and Hu Chen. "A parallel shortest path algorithm based on graph-partitioning and iterative correcting". In: *2008 10th IEEE International Conference on High Performance Computing and Communications*. IEEE. 2008, pp. 155–161.

[7] F Benjamin Zhan. "Three fastest shortest path algorithms on real road networks: Data structures and procedures". In: *Journal of geographic information and decision analysis* 1.1 (1997), pp. 69–82.