David Rohweder and Chris Ackerman
Dijkstra's Algorithm and Bellman-Ford Algorithm Analysis
Penn State University
State College, PA, USA
djr6005@psu.edu, cwa5262@psu.edu
April 15, 2022

1 Introduction

For this research, we will attempt to reduce the execution time of Dijkstra's Algorithm and the Bellman-Ford Algorithm using parallelization. To begin our analysis, we will first need to give the specifics of the equipment and tools used in the experiment. First, the study was executed on the Penn State Roar supercomputer using the ACI-b Standard Core resource profile and queued batch jobs to fully utilize open allocation on the cluster. The allocation for this setting is as follows: 20 available cores, InfiniBand interconnect, and 256 GB total RAM (for open allocation users). More specifically, the VM is using an Intel Xeon CPU E5-2680 v3 @ 2.5GHZ with 32KB L1 cache, 256KB L2 cache, and 25600KB L3 cache. It is important to note that this experiment utilizes multiple cores for program execution because we aim to study the effects of parallelization on code performance. Next, we will further discuss the specific algorithms we aim to explore and compare to further our understanding of parallelization on serial, time-complex code. The first algorithm we will analyze is Dijkstra's algorithm. In short, Dijkstra's algorithm is an algorithm that finds the shortest path from a source vertex to other connected vertices. When analyzing this algorithm, we will seek to understand how a time-complex algorithm with iterative dependencies can still gain several times speedup with the introduction of parallelization. Secondly, we will be analyzing the Bellman-Ford algorithm. Like Dijkstra's algorithm, the Bellman-Ford algorithm is an algorithm that as well finds the shortest path. However, the Bellman-Ford algorithm takes into consideration the usage of negative cycles and can detect if they are present. Furthermore, we will be analyzing how the introduction of parallelization can improve the speedup of the Bellman-Ford algorithm across numerous threads.

2 Analysis

Dijkstra's Algorithm Analysis

In the analysis of Dijkstra's Algorithm, we sought to speed up the original serial implementation using OpenMP.

To begin, in the serial implementation we take a generated adjacency matrix graph of size 20,000 vertices and 20,000 edges and a source vertex of 0. The desired output of our function is to generate a vector containing the shortest paths from the source vertex to any other vertex. This distance vector can be thought of as an alarm clock, with each vertex having a distance of infinity from the source vertex. Intuitively, the source vertex has a distance of 0 from itself, therefore we need to declare the distance of the vertex in the distance vector to 0. Now, we can begin the algorithm implementation. For |V| times we will first find the vertices smallest alarm clock that has not rang yet, i.e they have a value of infinity. Next, we will iterate through each edge containing that vertex and update the distance of that connected vertex if the distance is greater than

the current distance plus the weight. Then, we will update the distance to the vertex. When implementing the graph, we saw a serial performance of 1.76s on the 20,000 x 20,000 adjacency matrix.

To improve the performance of Dijkstra's Algorithm through OpenMP we need to be aware of the iterative dependencies. Simply, when using the for work-sharing construct in OpenMP the loop iterations will be divided according to the scheduling method. The tasks given to each thread in the thread pool will be updating the connected nodes of the given vertices in the task. However, the caution is that the current vertex in the task given to the thread may have not been updated to the correct path length when updating the distance lengths of connected vertices. As shown in *Figure 1*, we can see a steady performance increase as more threads are added to the thread pool; however, we do not achieve a perfect speedup ratio. When using 20 threads on the 20,000 x 20,000 adjacency matrix we saw a speedup of 1.76s / .21s = 8.1X.

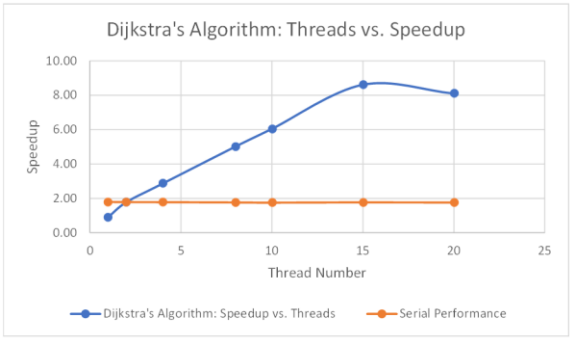

Figure 1: Dijkstra's Algorithm - Threads vs Speedup

Bellman-Ford Algorithm Analysis

For the analysis portion of the Bellman-Ford algorithm, we implemented OpenMP to speed up the serial implementation of the algorithm. To start off, we began by creating an N * N matrix graph with 20,000 vertices by 20,000 edges and a source vertex of 0. The two implementations of the Bellman-Ford algorithm are then expected to take the adjacency matrix graph and attempt to find the shortest paths from the source vertex to connected vertices, while at the same time being able to detect negative cycles. To achieve the shortest path, all edges on the said path are to be relaxed at |V| - 1 times. For the first implementation, Bellman-Ford's algorithm was integrated serially and had an execution time of 4.5s on the input graph. For the second implementation, Bellman-Ford's algorithm was integrated using parallelization. For the parallelized Bellman-Ford algorithm, OpenMP was used to implement loop parallelism in which dynamic scheduling was used to assign chunks of the for loop to each thread. Once the thread was finished with its chunk, it would then move on to the next available chunk until it eventually reaches completion. The chunk given to each thread computed a set of given vertices in the for loop, each vertex then updates each edge in the graph. The results of the two algorithms were then collected and compared to one another for each thread tested to ensure accurate results. This process was performed across all the tested thread numbers up to 20 threads. For each thread number entered, the serial and parallelized implementation's execution times are then displayed for that specific thread number. The results were then taken and the speedup for each tested thread number was calculated (serial/threaded). The overall speedup began relatively low at about 0.96 times at 1 thread and the execution time was reduced to 4.2s serial 4.2/ 0.4s parallelized, roughly a 10 times speedup as 20 threads was approached.

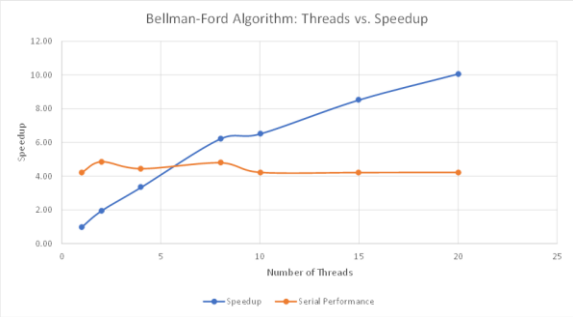

Figure 2: Bellman-Ford Algorithm - Threads vs Speedup

Comparison Analysis

While the Bellman-Ford algorithm is a more generalized version of Dijkstra's algorithm it suffers from a worse performance. The Bellman-Ford algorithm must check for negative cycles caused by negative weights albeit at the cost of performance. Hence, the Bellman-Ford algorithm is more time-complex than Dijkstra's algorithm. Therefore, as shown in the graphs with the introduction of parallelization we can see slightly more improvement in the Bellman-Ford algorithm as more threads are introduced. Moreover, the way distances from a source vertex are computed favor the parallelization improvement possibilities from the Bellman-Ford Algorithm. This is because in each iteration of Bellman-Ford's algorithm, every edge is updated, so we don't have to consider the iterative dependencies found in Dijkstra's algorithm.

3 Conclusion

In conclusion, we tested two different shortest path finding algorithms to better understand the impact that parallelization has on their performance. The results gathered clearly demonstrate that although more threads exist, the resulting speedup isn't a perfect divide by the number of processors. Moreover, while the program might be parallelized, it does not necessarily guarantee massive speed improvements due to algorithmic dependencies. Although, there were still clear and noticeable impacts on execution time from the parallelization performed on the algorithms. Both Dijkstra's algorithm and the Bellman-Ford algorithm are similar in nature, but we found that due to algorithmic dependencies more prevalent in Dijkstra's algorithm that the Bellman-Ford algorithm found better results from parallelization while suffering from a worse time complexity.