Collin Paiz

Professor Anastasiu

COEN 145

6 December 2022

Assignment 3

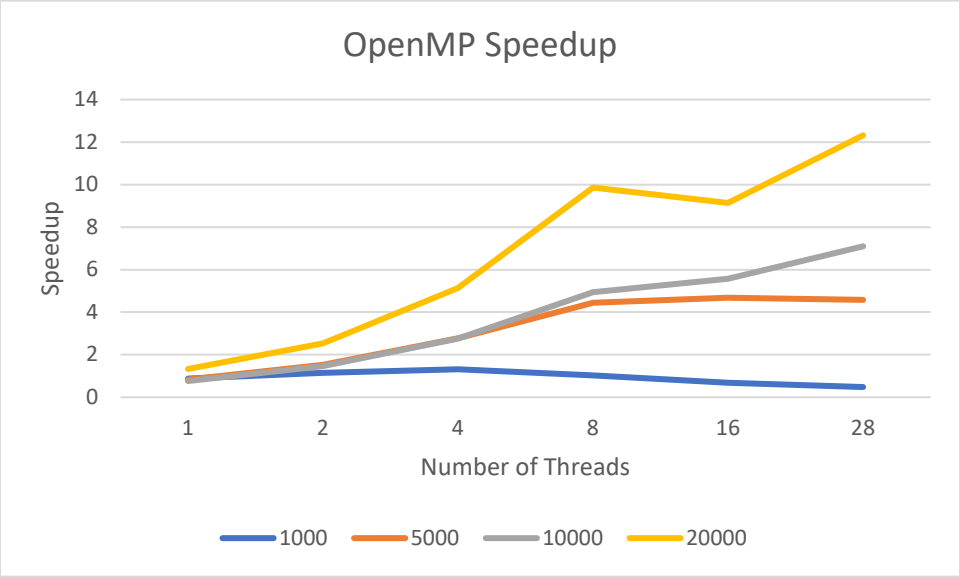
Parallelizing Sparse Matrix Multiplication

Description

To parallelize Dijkstra's algorithm using OpenMP, I chose to use *#pragma omp parallel for* with static scheduling on the loops within the *dijkstra* function. I decided to parallelize the two inner for loops which are used to find the local minimum as well as record the new distances to the other nodes based on the local minimum, respectively. I chose to parallelize the inner loops rather than the outer loop because the two inner loops perform a lot of work; so by prioritizing parallelizing the outer loop, each thread would need to perform a lot of work individually. I also parallelized the first for loop in the *dijkstra* function which is used to obtain random sources to search from and put it into an array.

I was unable to parallelize Dijkstra's algorithm using MPI. I was on the track of utilizing a scatter/gather approach before running out of time on the assignment. I think the biggest issue I faced was trying to figure out how to fulfill the requirement of working with graphs large enough that they don't fit in the memory of a single node. I was not sure on how to either split up this work or on the best approach to send it to other processes evenly/efficiently.

Results



Analysis

Increasing the number of threads, resulted in significant speedup across all graph sizes and thread counts, aside from the smallest graph, *1000.graph*. The reason for this anomaly is likely due to the small size of this graph; as you can see in the results, the execution time increased beyond 4 threads. This likely means that there is not enough work to be done for each thread to see any kind of benefit for a graph this small and a thread count that high, resulting in slowdown.

As the size of the problem increased, so did the runtime. This is due to the fact that as we increase the graph size (node count), we have an exponential increase in node connections to consider, resulting in more loop iterations, slowing down the overall runtime.

With that said, as we increased the thread count, speedup seemed to continue to increase for all tests except for *1000.graph.* In fact, as the problem size increased, speedup relative to the serial program increased. For example, using 28 threads for the *5000.graph* test yielded a speedup of 4.580804x, while using the same thread count for the *20000.graph* test yielded a speedup of 12.31841x. This likely means that this parallelization strategy scales well with the Dijkstra algorithm. From the tests that were performed, it is likely that we can continue to increase the number of processes/threads to continue increasing performance, at least for graph sizes above that of *1000.graph* which showed an increase in runtime beyond 4 threads, and *5000.graph* which showed a plateau in runtime between 16 and 28 threads. We can hypothesize that for graphs beyond this size, such as *10000.graph, 20000.graph,* and larger that increasing the thread/process count will continue to show speedup, as supported by the decrease in runtime

between 16 and 28 threads for both graph sizes. Although, as shown in the results of the smaller graphs, this increase in speedup will likely plateau eventually.