NC State University

Department of Electrical and Computer Engineering

ECE 406/506: Fall 2021

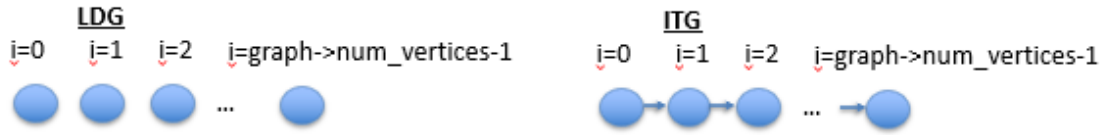
Project #1: Accelerating Sorting in Graph Processing Adjacency Lists

by

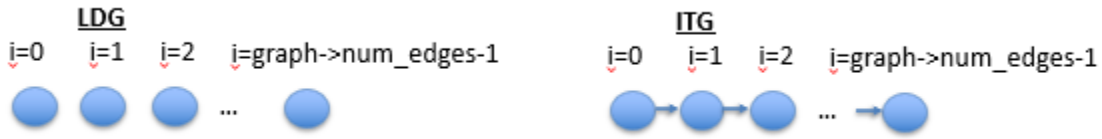
Cristian Hellmer

Serial countsort iteration and dependency graphs

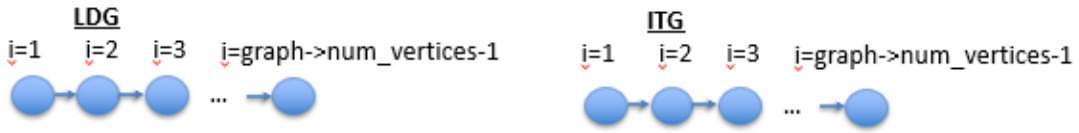
Loop 1



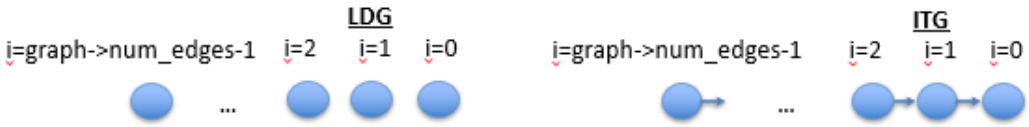
Loop 2



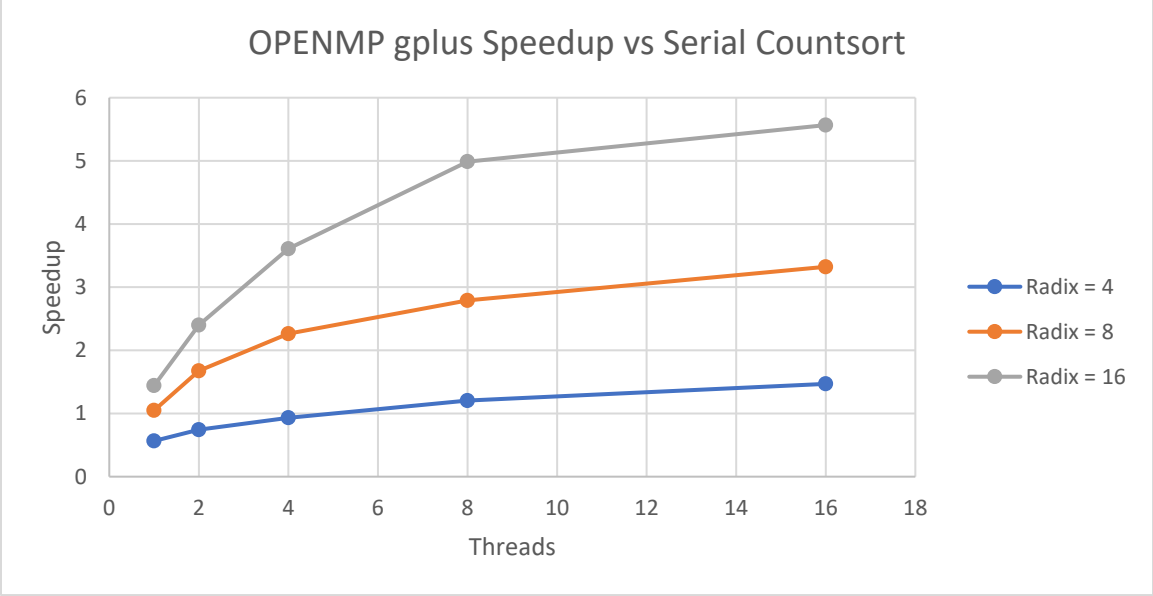
Loop 3



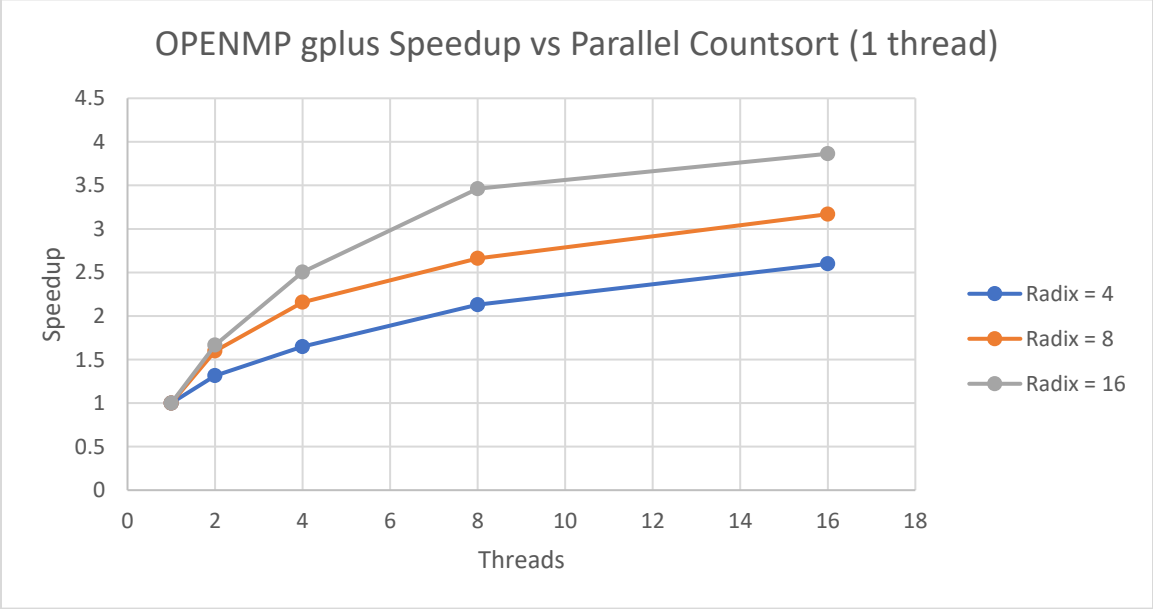
Loop 4



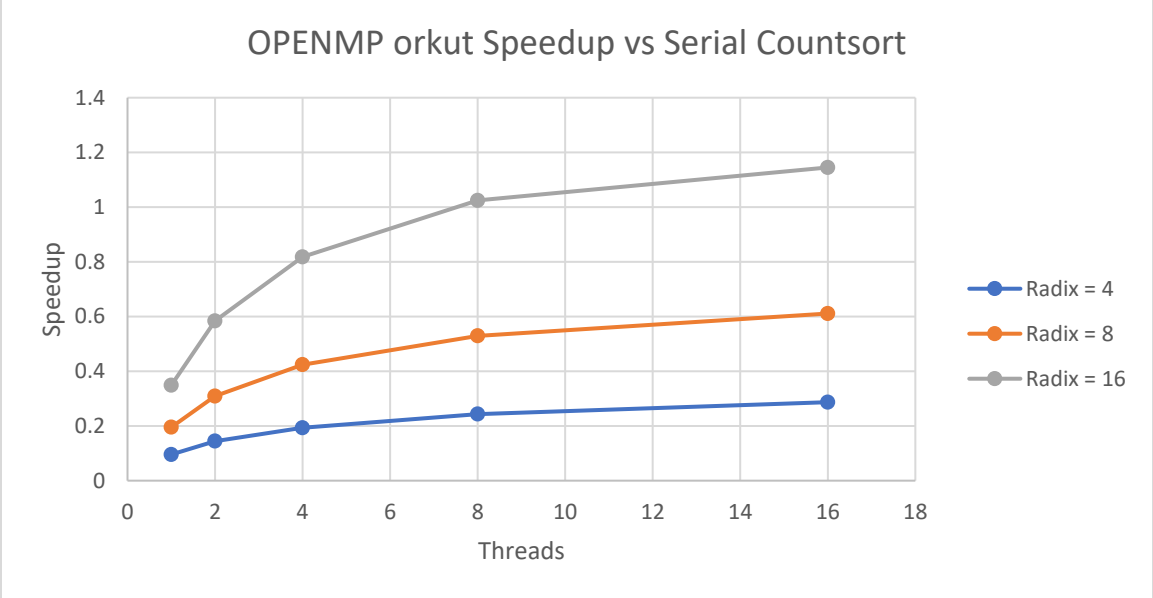
GRAPH #1



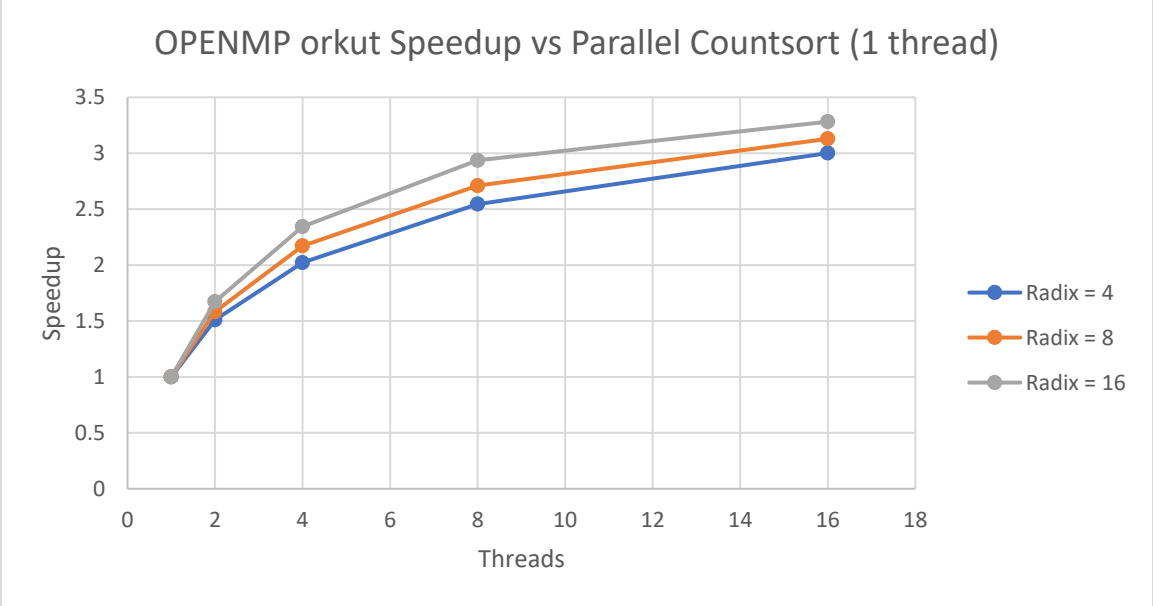
GRAPH #2



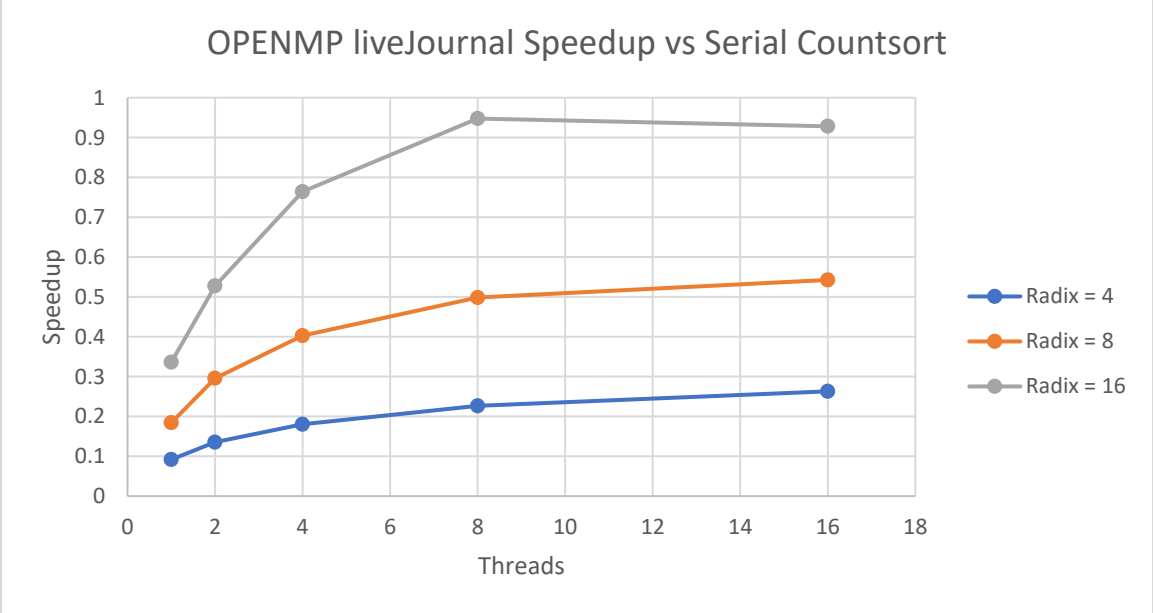
GRAPH #3



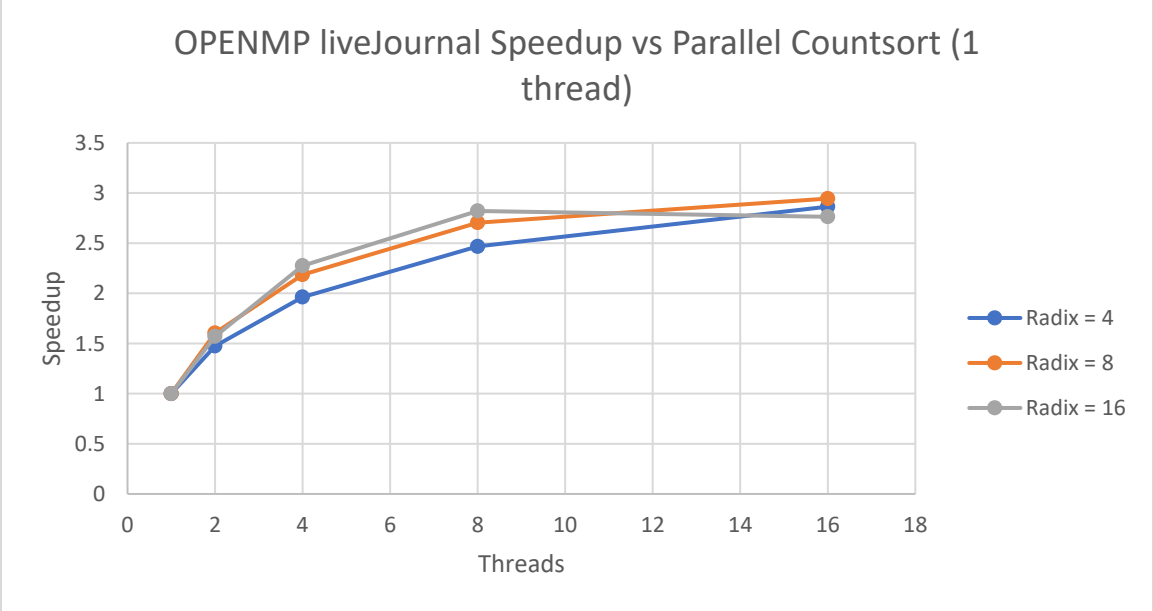
GRAPH #4



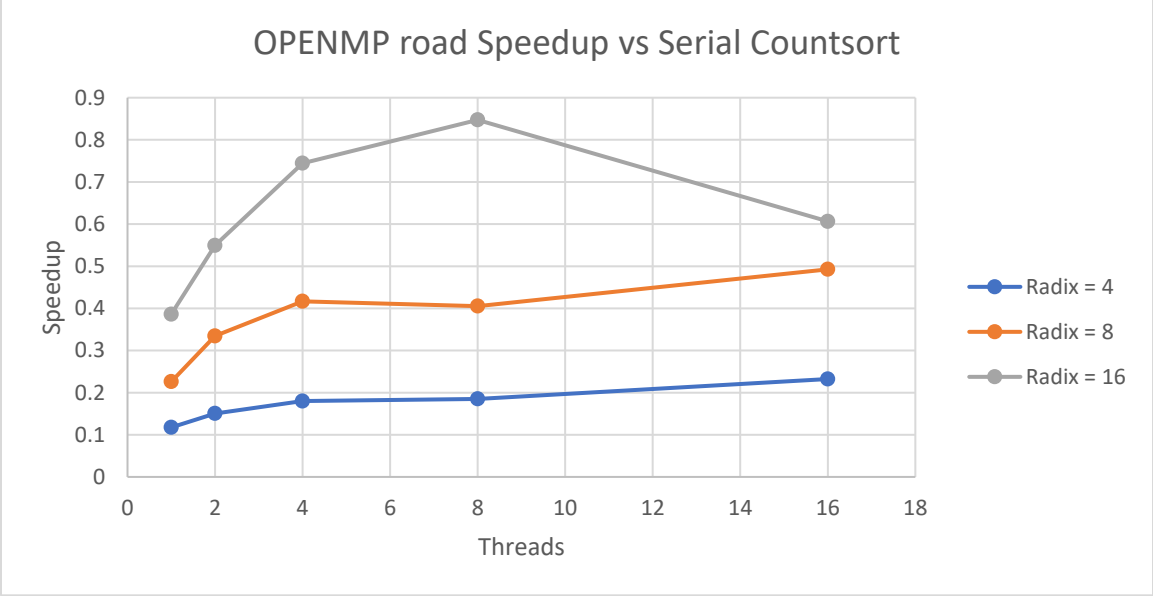
GRAPH #5



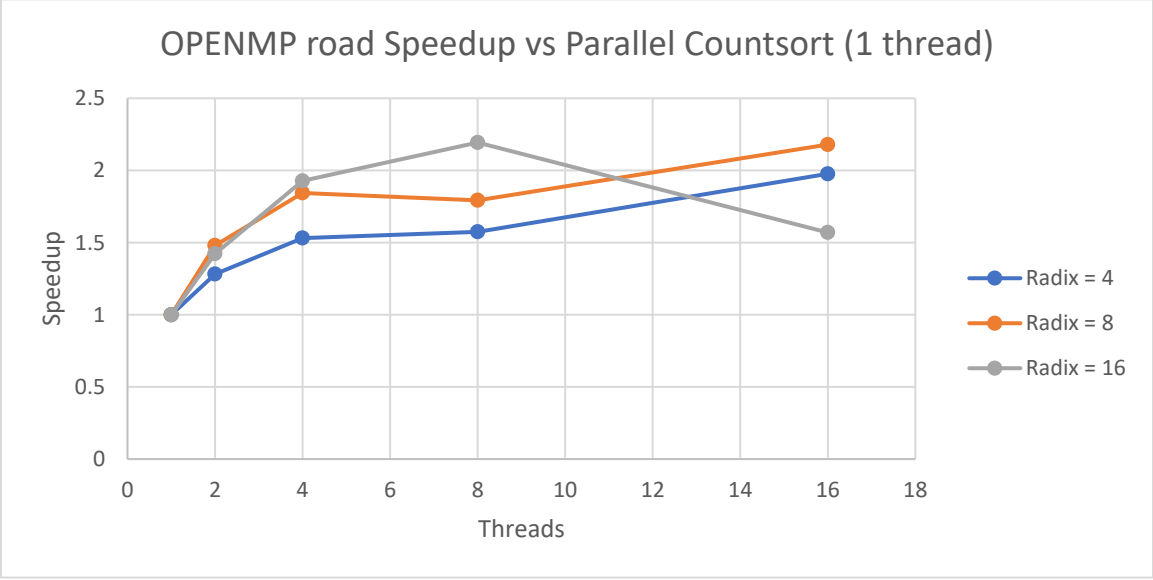
GRAPH #6



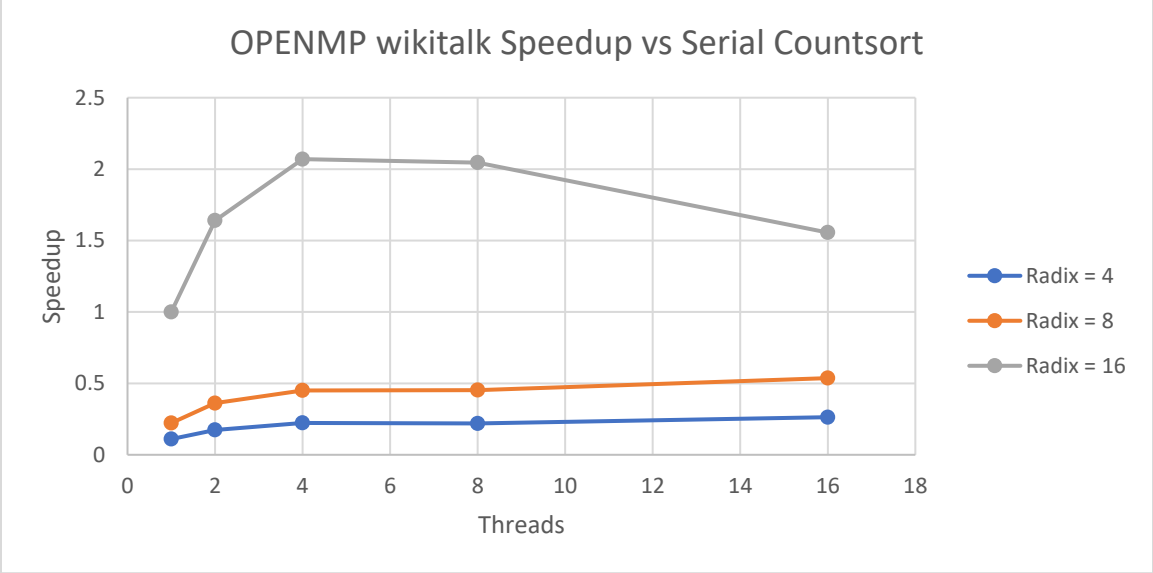
GRAPH #7



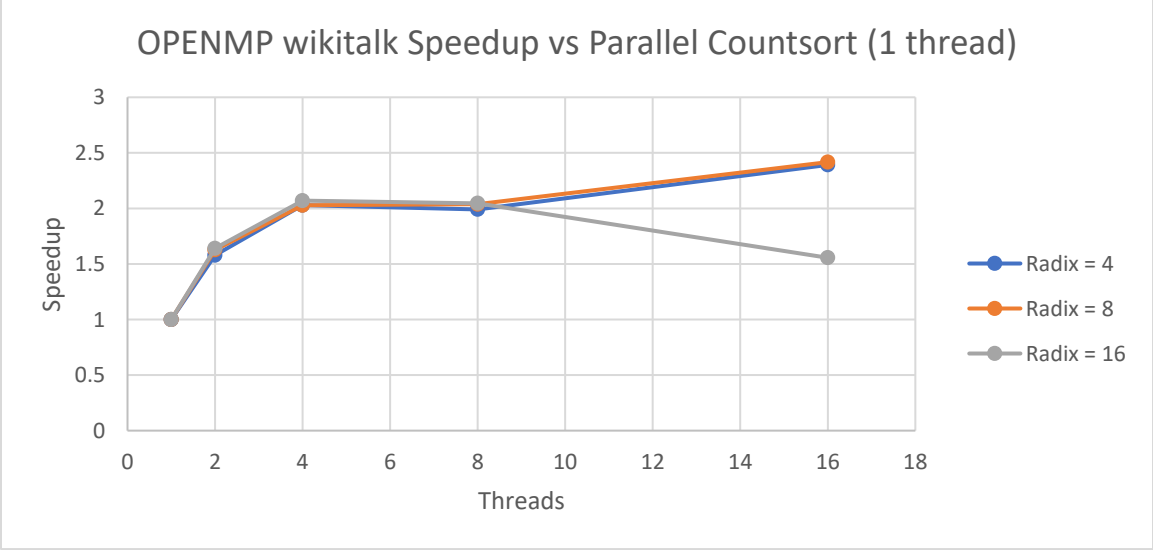
GRAPH #8



GRAPH #9



GRAPH #10



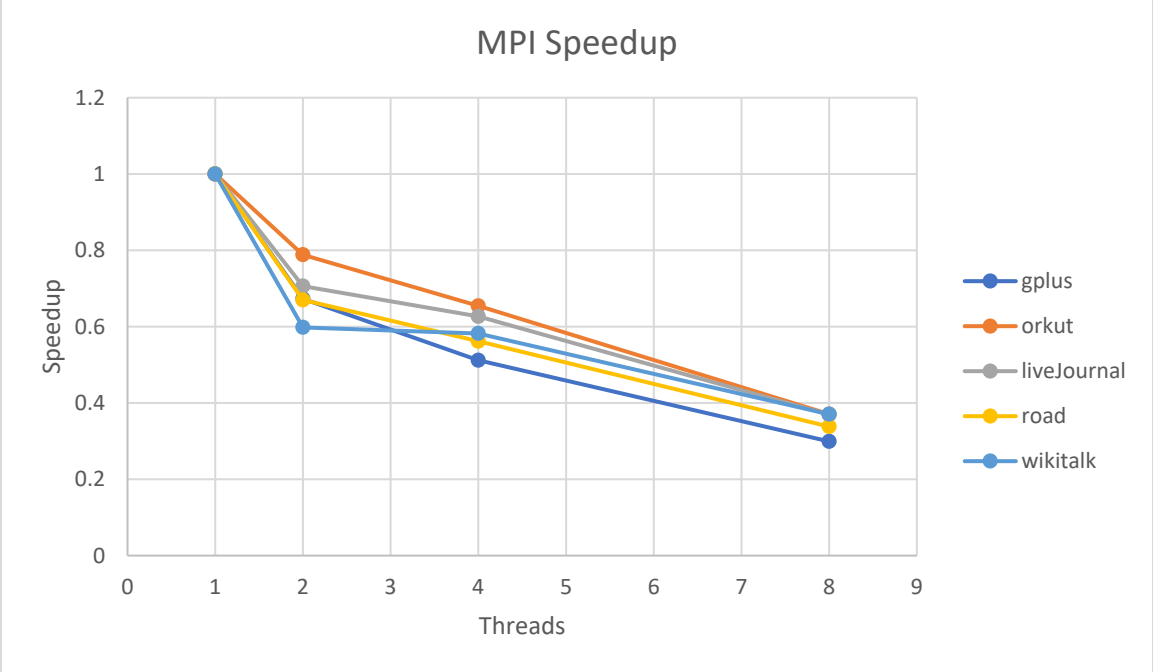
OpenMP Analysis:

I believe that radix sort is performing better than countsort due to the fact that we can limit the number of vertices we need to keep track of during each sort individual radix sort. For example, with an 8-bit radix, we need to keep track of 256 vertices. With the full 32-bit int we would potentially need to keep track 4,294,967,296 vertices. Having less vertices means we can more effectively cache our vertex_sort array without the need to refer to main memory as much as we would need to when traversing 4,294,967,296 vertices.

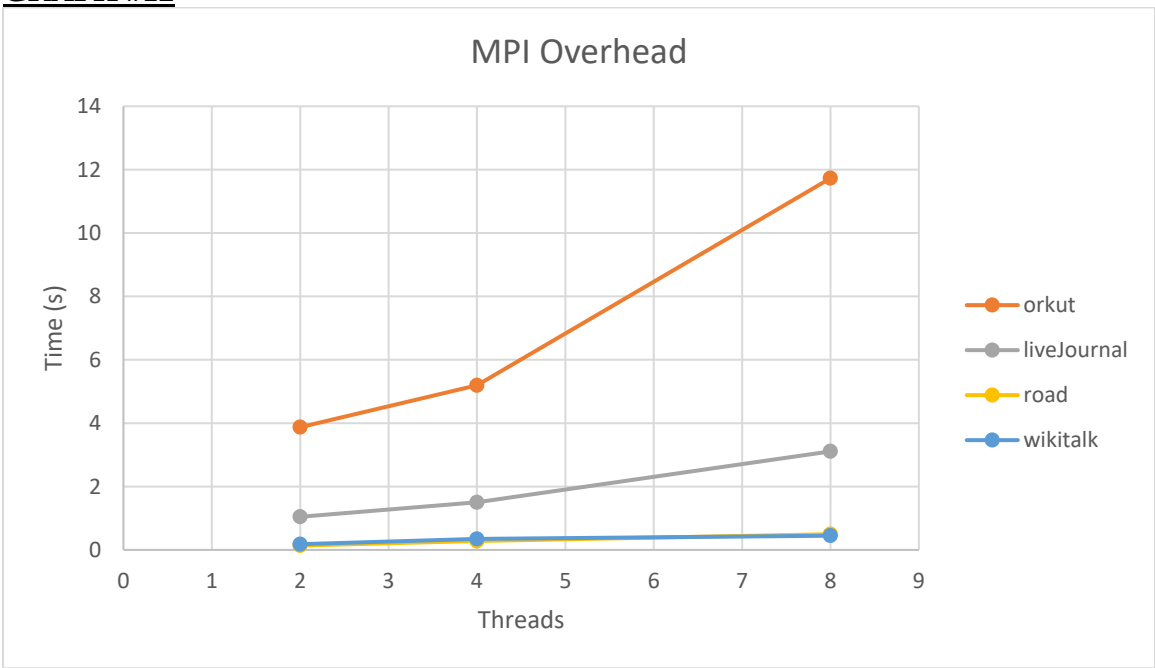
Regarding the size of the dataset, radix sort using OpenMP performs better for the larger datasets versus the smaller datasets. I think this is because the overhead of implementing radix sort and OpenMP dwindles in comparison to the actual time spent sorting the graph. Also, the larger datasets more vertices which would mean using a fixed-bit radix would lead to a larger reduction of vertices needed than smaller datasets. As the threads increased, the larger datasets saw the most improvement. This is because the added overhead of synchronizing the threads was negligible in comparison to the division of edges which need to be sorted. Some of the smaller datasets actually saw a decrease in performance when moving from 8 to 16 threads.

In all of the graphs, there are diminishing returns as the threads increase in relation to the speedup that you receiving from doubling the threads. This is due to the added overhead of adding the extra threads which keeps the performance from doubling.

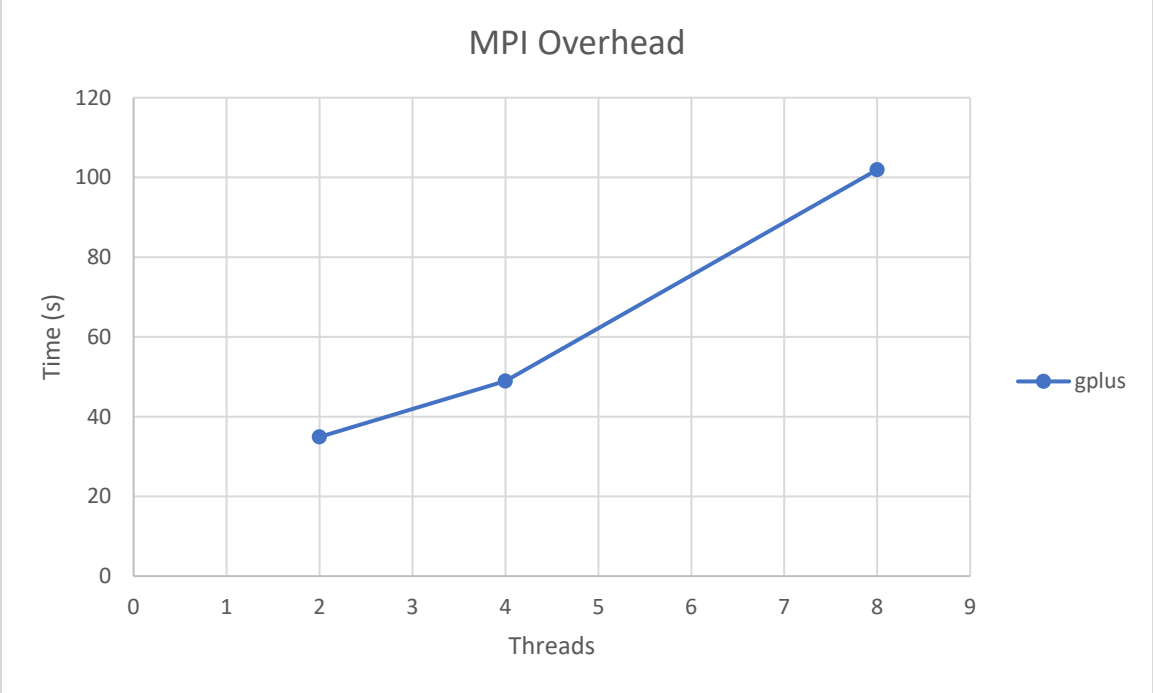
GRAPH #11



GRAPH #12



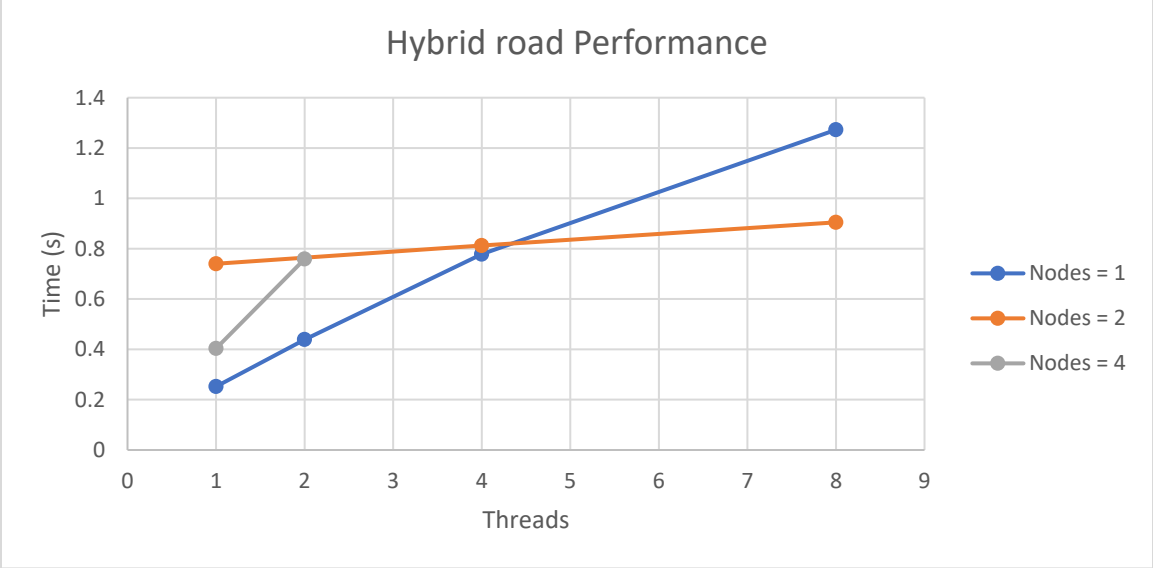
GRAPH #13



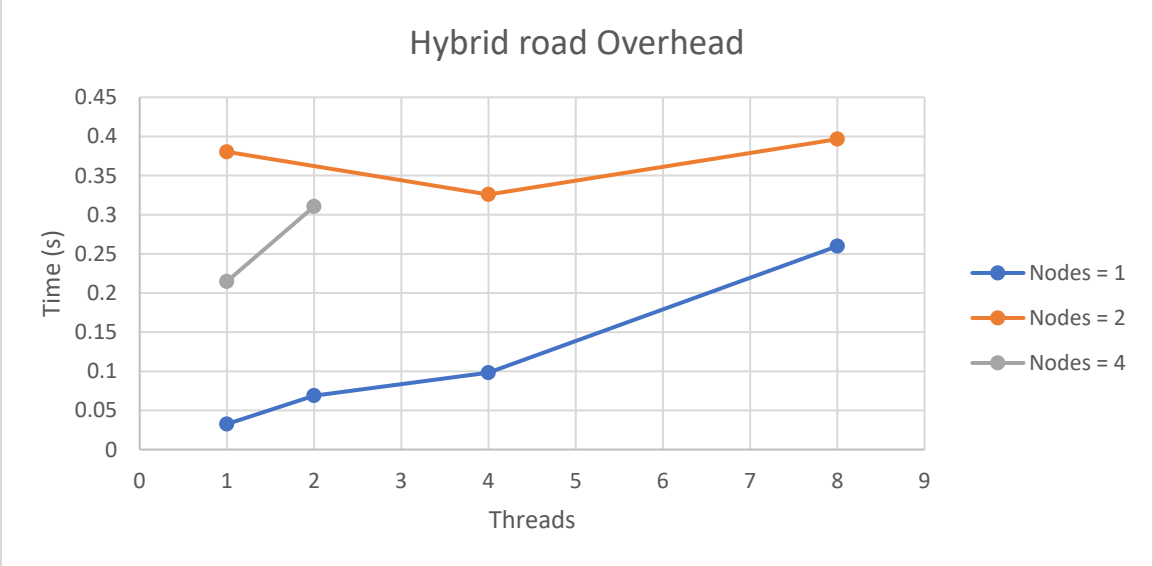
MPI Analysis:

For my MPI implementation, the performance got worse as more nodes were added to the system. This is due to the increasing overhead of the reduction on sorted edge arrays. These reduction operations accounted for the vast majority of the communication overhead of the system. I separated the gplus overhead from the others since the gplus overhead dominated the others when places on the same graph.

GRAPH #14



GRAPH #15



Hybrid Analysis:

For my Hybrid implementation, the performance got worse as more nodes and threads were added to the system. This is due to the increasing overhead of the reduction on sorted edge arrays and synchronization of threads. The reduction operations accounted for the majority of the communication overhead of the system. Also, the functionality of the Hybrid implementation was inconsistent. Sometimes there would be errors and sometimes there wouldn't be even with the same configuration and dataset being used. These issues weren't resolved. I chose the road dataset to test different configurations as this dataset was the most reliable for me.

Conclusions:

Of all the strategies utilized above, the best speedup was achieved using OpenMP. The MPI and Hybrid approaches resulted in performance which was worse than the baseline serial countsort and got worse as more nodes/threads were added. Different methods of communication were attempted in the MPI and Hybrid implementations (scatter vs send) but the performance was poor in both scenarios. Synchronization and communication overheads were crucial to the performance of all the implementations.