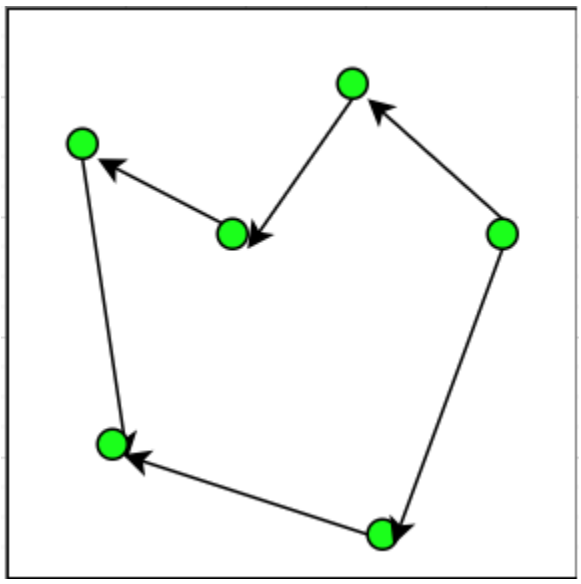
Parallel approach to Traveling Salesman

By: Kevin McCole
Github: https://github.com/Stickydarp/Traveling-Salesman-project

Abstract: The traveling salesman problem has been a staple of the field of computer science for many years. This is because on the surface it seems like a simple problem, all you do is find the shortest path between points on a graph. The challenge is that the computational cost grows exponentially as the number of cities increases. Because of its popularity the traveling salesman problem has many solutions, many of them better than Is feasible in this paper.

Example of a solution to a traveling salesman map



Proposal:

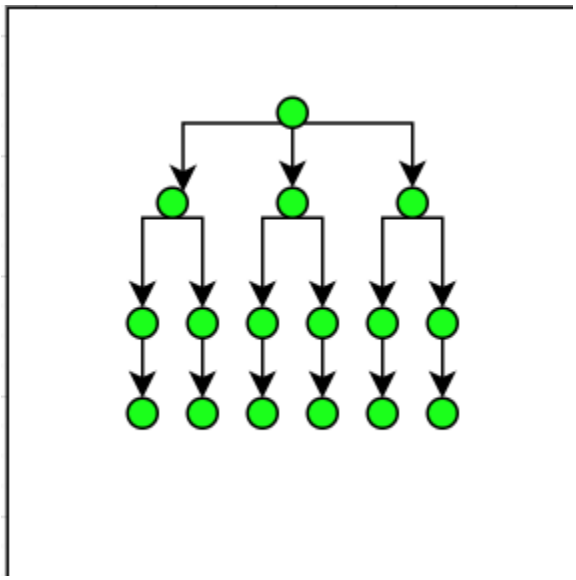
For this problem I propose two solutions. The first will be a sequential recursive depth first search algorithm. This method will not have any optimization tricks, it will just be plain depth first search which will allow me to find the true solution to the graphs and prove the effectiveness of my other algorithm.

The second solution is a direct parallelization of depth-first search: we fix city 0 as the starting point and assign each of its immediate neighbors (the second-level nodes) to a separate MPI process for traversal. The leaves will be assigned as follows

((city - 1) % mpi_size) == mpi_rank

Once each thread traverses and finds the best path within its tree the branches will be compared to find the best path using MPI.

Visual example of what the tree looks like



Sequential RDFS:

The sequential depth-first search approach to solving the Traveling Salesman Problem is straightforward. Start by representing the cities as indices in a list. Build an implicit search tree with root at index 0. From each node in the tree (representing the current partial path), branch out to every unvisited city by adding it as the next step in the path.

Recursively explore each branch as deeply as possible: continue adding unvisited cities until the path includes all cities (reaching a leaf node). At each complete path (visiting all cities), add the return distance back to city 0, calculate the total path length using Euclidean (Pythagorean) distances, and track the minimum length found at that point.

After evaluating a complete path, backtrack by removing the last city and try the next alternative branch. Continue this process until all possible paths have been explored. The shortest path saved will be the optimal solution.

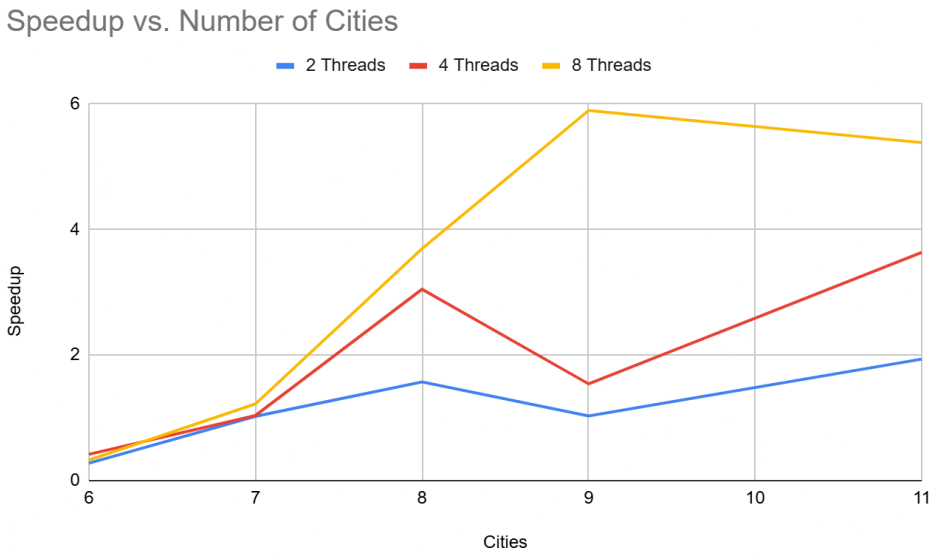
Parallel RDFS:

The parallel solution to this problem using mpi is exactly the same as the sequential solution but we are taking the second layer of the tree and splitting branches among tasks. In the case of a task having to traverse multiple branches we use a local minimum path that we find by comparing all the branches that the task has traversed. That local minimum is then sent to T0 and compared to the minimum paths of all the other Tasks and the best one is found.

Results:

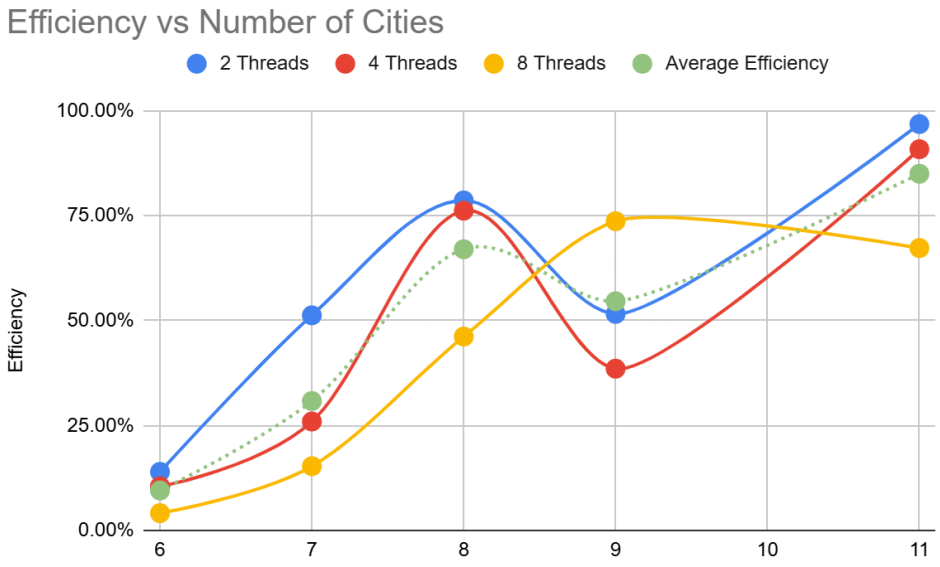
With my parallel implementation I achieved an average of about 70% efficiency through city sizes greater than 7(anything less and the computation spent creating the threads is a large factor).

Chart based on gathered data



Speedup: My implementation achieved a speedup as high as 5.8x with 8 threads at 9 cities. The parallel implementation performed worse than the sequential approach in city counts less than 7 because of the overhead from handling passing data between tasks and creating the task instances. However, for more than seven cities, the speedup trended toward the number of tasks used, reflecting the high efficiency of the parallelization.

Chart based on gathered data



Efficiency:
With city sizes larger than ten I achieved efficiency percentages as high as 90%. Due to computational limitations I was not able to reliably get results for sample sizes larger than 12. So I was not able to evaluate my implementation on those sizes. I predict that the efficiency

percentage would trend up and down in a wave pattern with peaks at numbers that are divisible by the number of threads used on the particular attempt. I believe the success of this implementation is due to the inherent parallelizability of tree traversal because each path is independent and the only point at which results need to come together and be compared is the end.

Conclusion:
In conclusion I am satisfied with my results and I feel that implementing it posed some interesting problems that I feel I solved in an efficient manner. If I were to continue with this further I would try to add some type of pruning to reduce computational costs. Though I fear that that would greatly reduce the memory efficiency of the current implementation.