CMSC 502
Assignment 2
Paul Hudgins

Summary:
This implementation of TSP achieves consistent accuracy, and can process 1,000,000 cities in 533 seconds with 32 processes. Four tricks were used to maintain n*log(n) time complexity and full parallelization. These were: a limited inversion sweep, full parallelization of inversion sweeps, approximate partitioning, and only considering nearby points for TSP-merge. The algorithm distributes subproblems on a cartesian topology on MPI, but performs alternating x and y axis merges instead of performing all x-axis merges then ally y-axis merges. Asymptotic speedup was S = (n log n) / ((n/p) log n + n log p).

Implementation

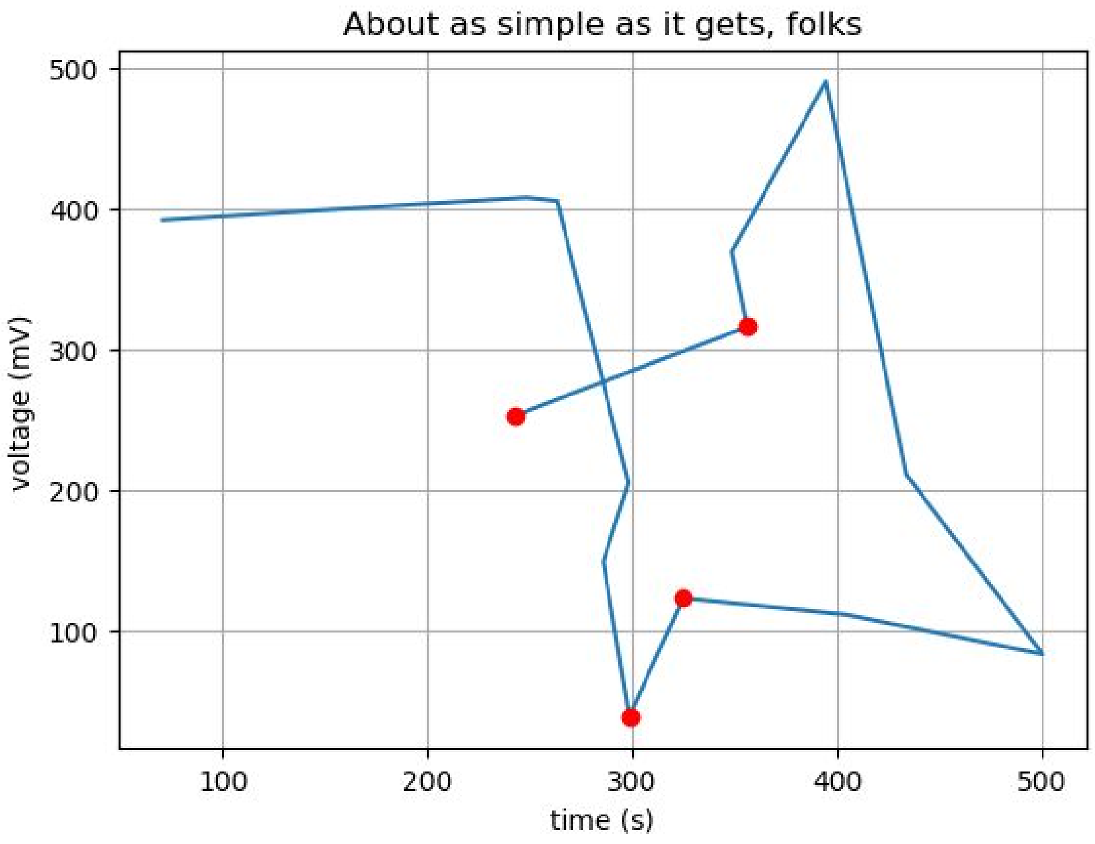
This assignment was implemented in python3 using the mpi4py module. Instructions for performing a user install for this module on the class server are included in the project's README.

Recursive Bisection Algorithm

As we discussed in class, this implementation is not restricted to an equal number of cities per block. Instead, the problem is recursively bisected until the desired number of cities per block is reached. If the number of cities is less than or equal to 10, a modified Held-Karp algorithm is run sqrt(n) times. If the number of cities is more than 10, the set of cities is bisected along the x or y axis, alternating, and subsolutions are computed. This repeats recursively.

Merging and Inversions

Blocks are merged using the TSP-merge algorithm described in the handout. When checking both possible swaps, an inversion will not be directly produced by the swap. However secondary inversions can be produced by the geometry of other points. This occurs rather frequently, and occurs whether-or-not merges are conducted row-wise. An example is shown below.



Ignore axes. The two open ends are joined in the path. The red points are the optimal swap points. The inversion is not from the swap itself, but produced by the geometry of points not included in the swap.

A “sweep” for inversions detects and corrects all inversions after merging. Removing some inversions can create new ones, so the sweep must be performed multiple times. In all experiments, all inversions were removed by the 3rd sweep.

MPI Implementation

The problem is fully distributed using a Cartesian topology. There is no master-slave relationship, but generation of cities is handled in one process and output is handled by that same process, to facilitate the possibility of input and output. Data is distributed n*log(p) time using a binary tree structure. Merging is performed on alternating x and y axes, instead of performing all x-axis merges first followed by all y-axis merges. This approach still takes advantages of the proximity induced by the cartesian topology.

For partitions less than a specified depth, half of the problem is sent to another process. For partitions less than the split depth, data is subdivided within a single process. Therefore the number of processes must be a power of two.

Trade-Offs and Accuracy

Four trade-offs were found that have limited or no impact on accuracy but an enormous impact on time complexity.

Approximate Bisections:

To perform a perfect bisection would require sorting the points along an axis, which takes O(n log n) time resulting in a total time complexity of O(n log^2 n). Instead, a point halfway between the minimum and maximum points of the block is used as a divider. This takes O(n) time for an O(n*log(n)) total time. This may result in some unevenness in block division, but was not found to affect accuracy.

Considering only nearby points for TSP-merge:

For merging problems with more than 200 points per path, only the 200 points nearest to the boundary of the two paths are considered. A geometry could be constructed where this is a suboptimal solution, but this almost never happens. Any effect on path length was found to be less than the variability of path length. This reduces the time complexity of TSP-merge from O(n^2) to O(n) per merge.

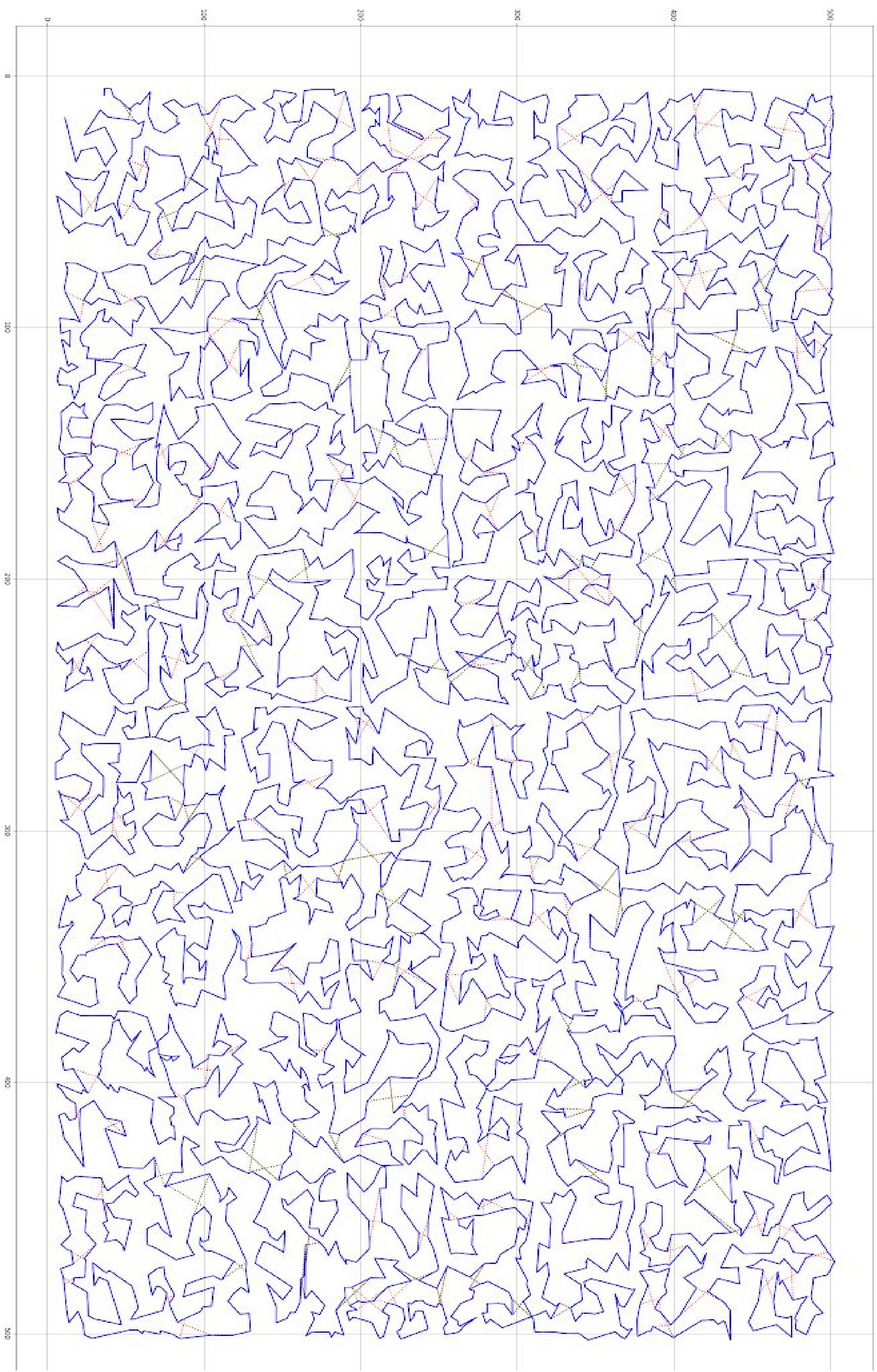
Parallelization of Inversion Sweeps:

Instead of performing inversion sweeps at the end, each process performs inversion sweeps at the “split depth,” the recursion depth just before results start being merged with other processes. This allows the time-intensive inversion sweeps to be completely distributed, but results in a negligible 2*p unchecked swaps being performed after the sweeps.

Limited Inversion Sweep:

Sweeping for all inversions takes O(n^2) time. However most inversions only induce small “loops” on one side of the inversion, so the large majority of inversions can be detected by limiting the search to pairs of points less than 100 edges apart. This reduces the time complexity of the sweep to O(n).

An example is shown below. The red path is the original path, with all inversions. The green path shows inversions remaining after a limited inversion sweep. The blue path has all inversions removed. The green path is the solution used by my implementation.

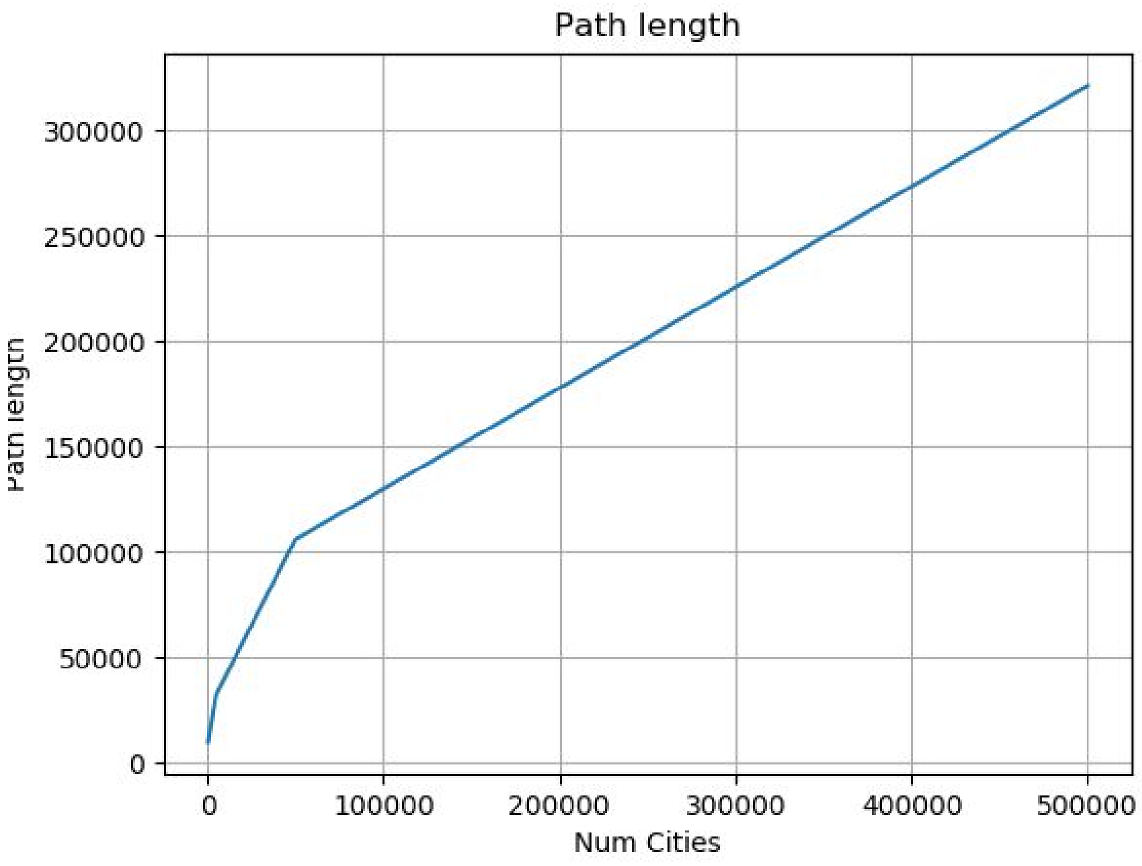


0
100
200
300
400
500
0
100
200
300
400
500

For an example run of 5000 cities (all produced very similar results), the original length without removing inversions was 34410. Removing inversions with loop sizes of less than 100 reduced the path length to 32659. Removing all inversions further reduced the length to 31729. Therefore, performing a limited inversion sweep resulted in a 2.9% accuracy penalty but an asymptotic reduction in time complexity.

Accuracy

For a 5000 city benchmark, my solution from assignment one produced a path length of 43188, or 32% longer. A graph of path length to number of cities is provided below. The increase is perfectly quadratic, as expected. Cities are randomly distributed in a 500x500 grid. Accuracy was not affected by parallelization because forking the recursive algorithm does not change its result.



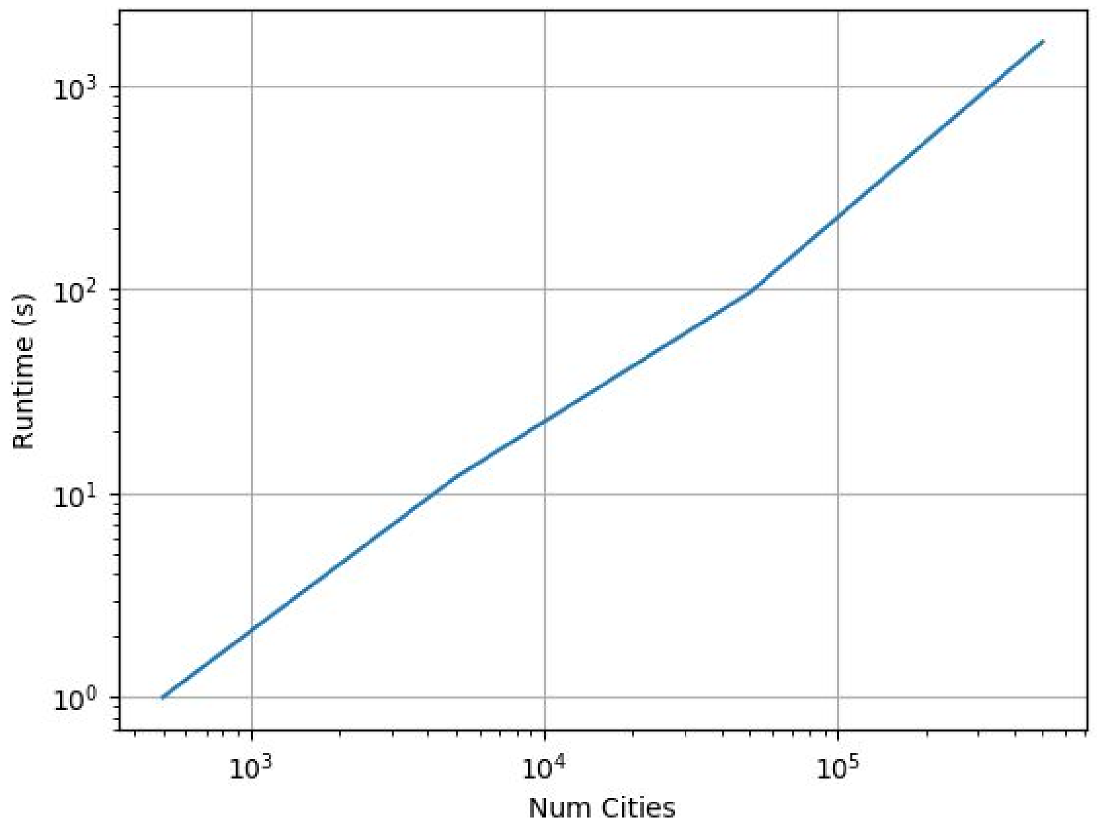
While not included in the above graphs, for one million cities and 32 processes, the algorithm took 533 seconds and produced a path length of 467049.

Complexity

Serial Time Complexity:

- Generating cities: O(n)
- Partitioning: O(n log n)
- Merging: O(n log n)
- Inversion sweeps: O(n) but with a significant constant factor. About 50% of the runtime for 50K cities, but only 25% of the runtime for 500K cities.
- Total runtime: O(n log n) with a significant O(n) component.

Serial runtimes are shown below:



Overheads

Overheads were examined analytically and by inspection using “splits” of wall clock time

- Interprocess interactions: Interprocess interactions consist of transmitting cities and returning paths between processes, which are distributed in a tree structure with O(n) total communications and O(log n) steps when parallelized. The time to transmit cities is significantly greater than to return paths, so only that time is considered. For 50K cities, the time of a single communication is 0.05 seconds on average, and so is negligible. Total contribution of communication to T_p is O(n log n) with small constants.
- Idling: Because of the lack of barriers, idling will only occur at the beginning and the end of a processes run. For 50K cities and p=16, the worst time idling is 1.5 seconds at the beginning, waiting for partitioning, and 1.5 seconds at the end, waiting for merges, out of a runtime of 16 seconds. Idling is the worst source of overhead by far. Idling time is O(n * log p), for both the beginning partitioning and final merges.
- Excess computation: Because of the recursive structure of the algorithm, excess computation is extremely minimal. It is constant with respect to n and linear with respect to p, with an extremely small constant.

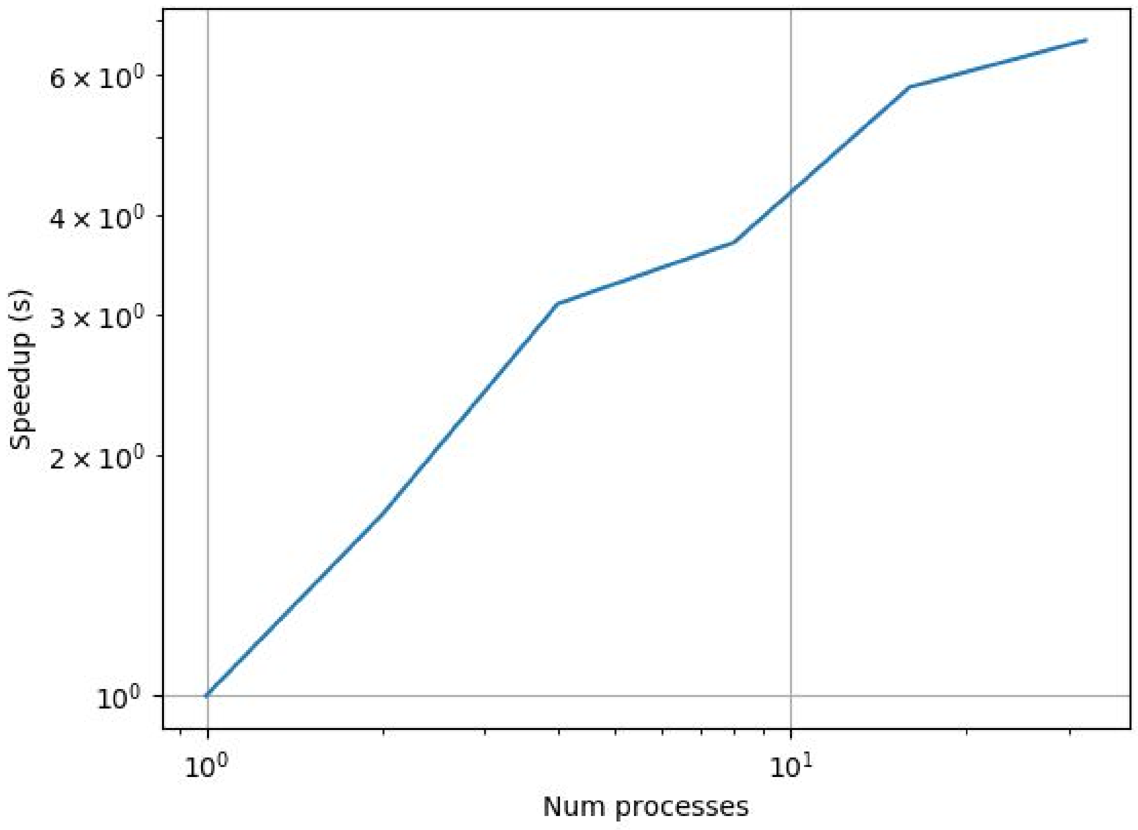
Speed analysis:

Block size is not a factor in these calculations, because it is fixed at 10.

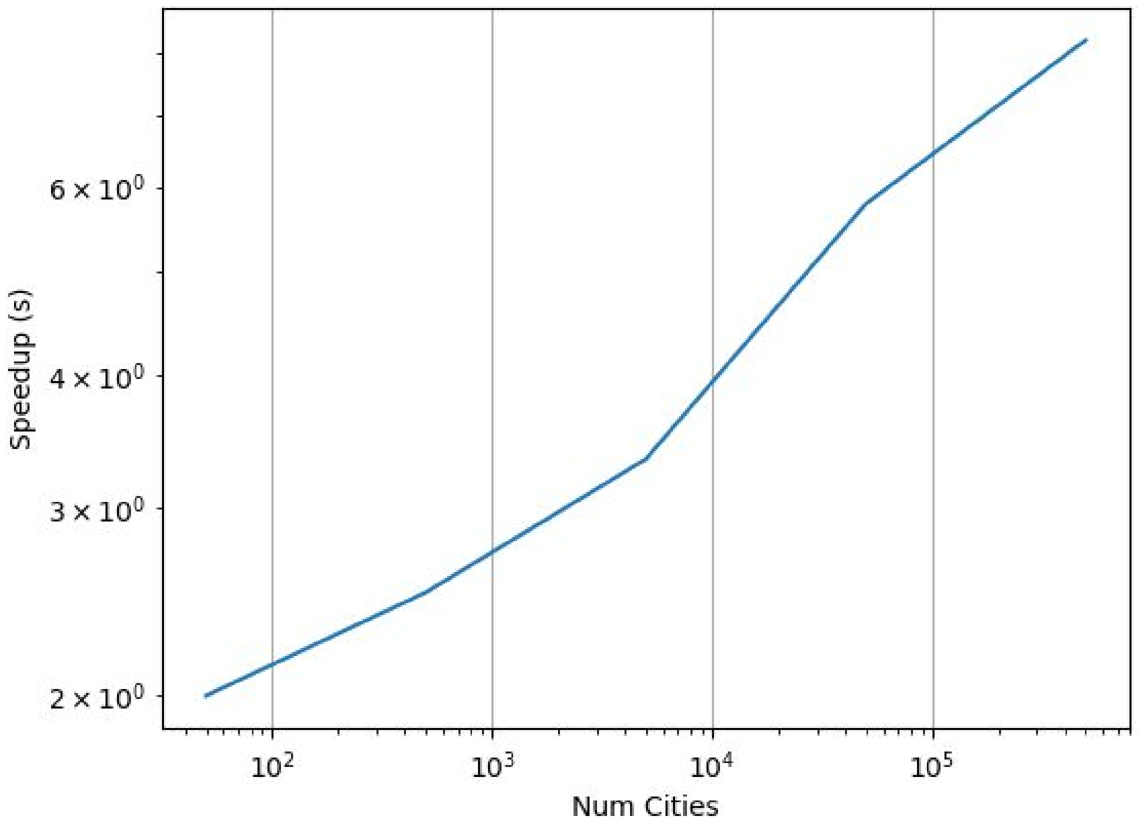
Serial time complexity is:
T_s = O(n log n) with a significant O(n) factor
T_p = O(n log n)/p + O(n * log(p)), where the second factor is the overhead from idling
T_o = p*T_p - T_s = O(p * n * log(p))

For a fixed number of cities, we expect S = c / (c/p + logp), for some constant c, which corresponds to the graph below:

For a fixed number of processes, we expect S = n log n / (n log n + cn), which corresponds to the second graph.



Speedups for 50,000 cities above.



Speedups for 32 processes

Other metrics:

Parallel efficiency:
E = S/p = (n log n) / (n log n + np log p)
Efficiency for 16 processes and 50,000 cities was (6.1/16) = 38%, but efficiency for 4 processes was 75%

Note: Serial portions of the parallel algorithm, as shown by “split times” slowed down with increased numbers of processes, so limited resources on the server may have been a factor.

Memory efficiency:
Theoretical memory efficiency was poor by design, because the total memory required was limited. Total memory required is O(n*p) because all cities are transferred between processes

for indexing purposes for dealing with inversions. This was not a limiting factor, but if it became one the total memory required could be reduced to O(n*p) by adjusting the way indexes are handled by the inversion sweep component.

Cost optimality:

T_p = (n log n)/p + n * log(p)
Cost = n log n + p * n * log(p)
Therefore this algorithm is cost optimal if log(n) ~= p log p, or
N = Big_Theta(2^(p log p))
P_opt = O(log n)

Isoefficiency:

Overhead for adding n numbers on p processing elementis is approximately n log p = log W * log p, therefore the asymptotic isoefficiency is Big_Theta(p).