Parallel Traveling Salesman Problem.

albert_s@sfu.ca

The Traveling Salesman Problem.

The Traveling Salesman Problem (TSP) is an NP-complete problem for which a polynomial time algorithm is not known to exist. The aim of a TSP is to find a minimum-distance tour between a number of cities that a salesperson may visit. This is equivalent to finding a Hamiltonian Cycle (HC) in a graph that models the cities. Since for this assignment we are dealing with a graph of cities distributed in 2-D Euclidean space, we assume that the triangle inequality holds for purposes of TSP. The triangle inequality does not alter the NP-completeness of the TSP, therefore we are not aware of a polynomial-time algorithm to solve this problem exactly. A brute force search is bounded by O(N!), where N is the number of cities. Therefore we will use a heuristic approach here.
A well know approximation algorithm for the TSP is to find a minimum spanning tree (MST) for the graph representing the cities and perform a pre-order walk on the tree. This will return a Hamiltonian cycle of the graph that's within factor of 2 of the optimal tour. There are several MST algorithms, most famous being the Prim algorithm, and the Kruskal algorithm. We will use Kruskal's algorithm in approximating the tour for TSP. Here is the approximation algorithm for TSP.

```
TSP-APPROX(G)
1 find an MST (call it T) using Kruskal's algorithm.
2 perform a pre-order walk of T, let L be the list of that walk
3 return the HC by listing the vertices in the order of L
```

The heart of the above algorithm is of course Kruskal's MST algorithm on which we will concentrate here. The pseudo-code follows:

```
MST_KRUSKAL(G,w)
1 set T to Ø
2 for each vertex v Ɛ V[G]
3    do Make-Set(v)
4 sort the edges of E by nondecreasing weight w
5 for each edge (u,v) Ɛ E[G], in order by nondecreasing weight
6    do if Find-Set(u) ≠ Find-Set(v)
7         then T ←T U {(u,v)}
8              Union(u,v)
9 return T
```

The algorithm uses a disjoint-set data structure to maintain a set of disjoint forests as we grow our MST. We will call this data structure DSF for disjoint set forests. The operations `Make-Set, Find-Set, and Union` belong to DSF. `Find-Set(u)` returns a representative element from that set that contains u. `Union` combines two trees. The `Make-Set` procedure creates a tree out of just one vertex.
In order to improve the running time we will use two heuristics in implementing the methods for DSF. One, called union-by-rank helps with combining trees, the idea is to make the smaller tree point to the bigger one when combining the two. We do not keep track of the size of the trees, instead we calculate a rank that closely approximates the height of the tree.
Our second heuristic is called path compression, during the `Find-Set` operations we point each node on our find path to the root point directly to the root. Note that path compression does not affect rank. The running time of the operations on DSF is O(m lg n) where m is the number of edges and n is the number of vertices. This is the fastest implementation of DSF known.
Using DSF in the implementation of MST_KRUSKAL yields a running time of Kruskal's algorithm of O (E lg E), where E is the number of edges in our input graph G. A pre-order walk of a tree runs in time O(V) since we are simply listing vertices of T in an order. Since our input graph is a complete graph we can say that E > V since (E = V * (V-1) / 2). Thus the total running time of TSP-

APPROX is O (E lg E) .

To cut down the real time of executing TSP-APPROX we will parallelize MST_KRUSKAL algorithm. Since Kruskal is looking at each edge in nondecreasing order it is inherently serial. However, there is a way to introduce parallelism. As the minimum spanning forest is being build, there are edges that have not yet been examined, that will form a cycle within the MSF. Such edges can be discarded and Kruskal's algorithm doesn't need to look at them.
We divide the array that holds the edges into equal parts. We distinguish between the main thread and helper threads. The main thread starts at the beginning of the array and starts building MSF. The helper threads work on their part of the array, until the main thread reaches that portion. The helper threads discard any edges that are found to form a cycle with the current MSF. With complete graphs (as in our current assignment) our helper threads eliminate a large percentage (> 95% in our experiments) of edges within their assigned portion of edges. (i.e. running just one helper thread eliminates almost half of all edges that Kruskal has to look at.) Our helper threads use a read only version of `Find-Set,` a version with no path-compression.
The overhead of our parallelization strategy consists of 2 arrays that will keep track of edges marked by the main thread (main_array) and the helper threads (helper_array), and the overhead due to creating multiple threads by our library of choice: openMP.
The upside of this particular parallelization strategy is that the helper threads operate on different parts of the helper_array, so there is no need for synchronization among the helper threads. The only point of synchronization is where the main thread reads an entry in the helper_array while the helper thread has not written to it yet. In this case the main thread will proceed and examine this edge itself. So the correctness of the algorithm is maintained even without synchronization.

If we were examining graphs that are connected but not complete we could improve our scheme further. We can make the helper threads continuously check their edges (until the main thread enters that region). This of course will require synchronization on the DSF structure as the main thread modifies it while helper threads are reading it, which can cause problems if the DSF is changing while it is being read by a helper thread.

Transactional memory vs. locks

Shared data structures that are not lock-free sometimes need to be acquired exclusively by a process in order to write to the structure. A traditional approach has been a lock. A lock is quite expensive to obtain. Furthermore in large systems there can be many locks, this places a burden upon the programmer to remember them all and to make sure that races and deadlocks are avoided.
Transactional memory offers an arguably better approach. A series of statements marked as a transaction are executed tentative changes are made to memory. When the transaction is finished the result is either committed or aborted causing any changes to be discarded. Transactional memory seems to perform faster than many (if not all) types of locks, and it frees the programmer from the burden of keeping track of locks. Of course a transaction is inherently serial and furthermore it appears that only one transaction can be executed at a time.

Implementation.

The TSP-APPROX algorithm was implemented in c++, using OpenMP for parallelization and transactional memory of c++11 standard. It was compiled using GCC 4.7

Results:

MST of usa.in:

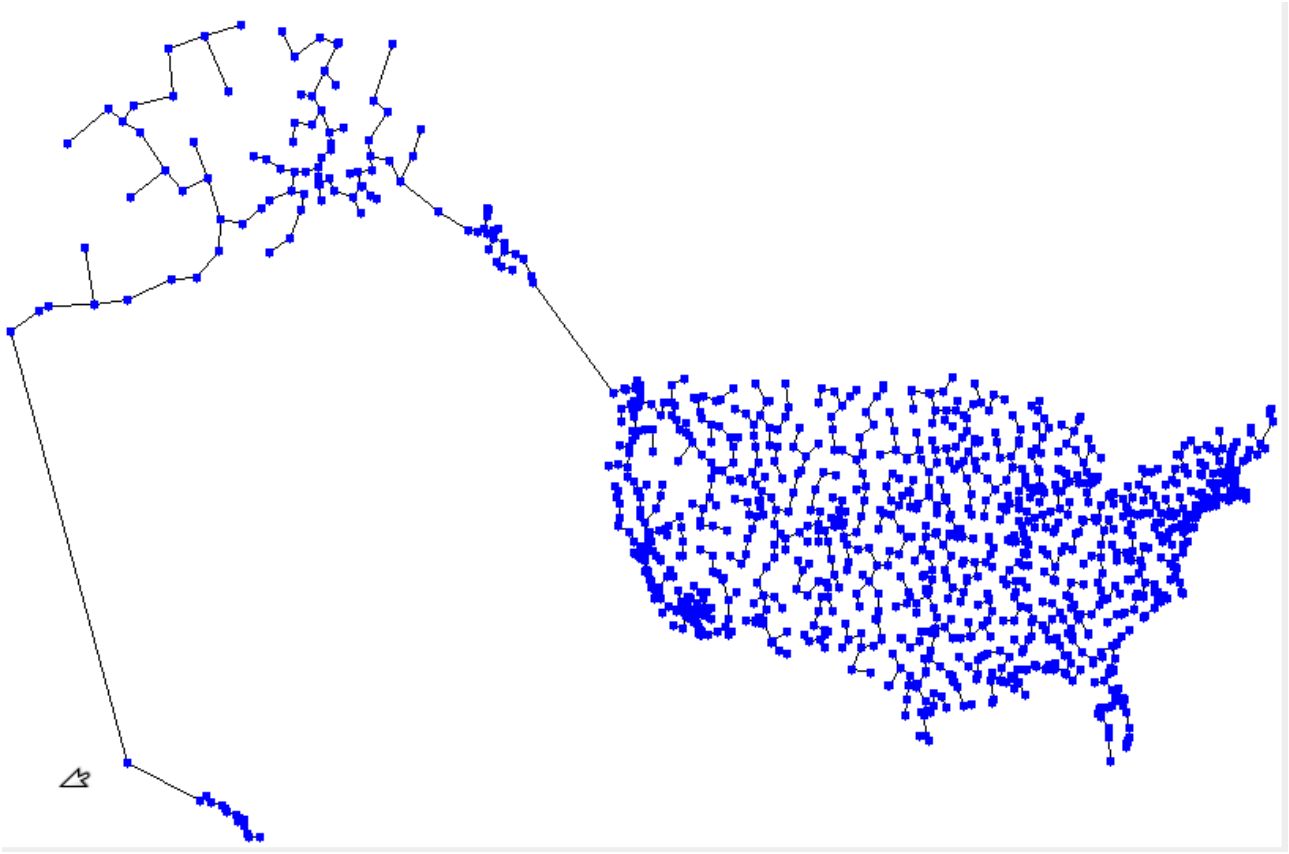

Figure 1: map of a usa MST (please note the map is not to scale)

Tour of usa.in

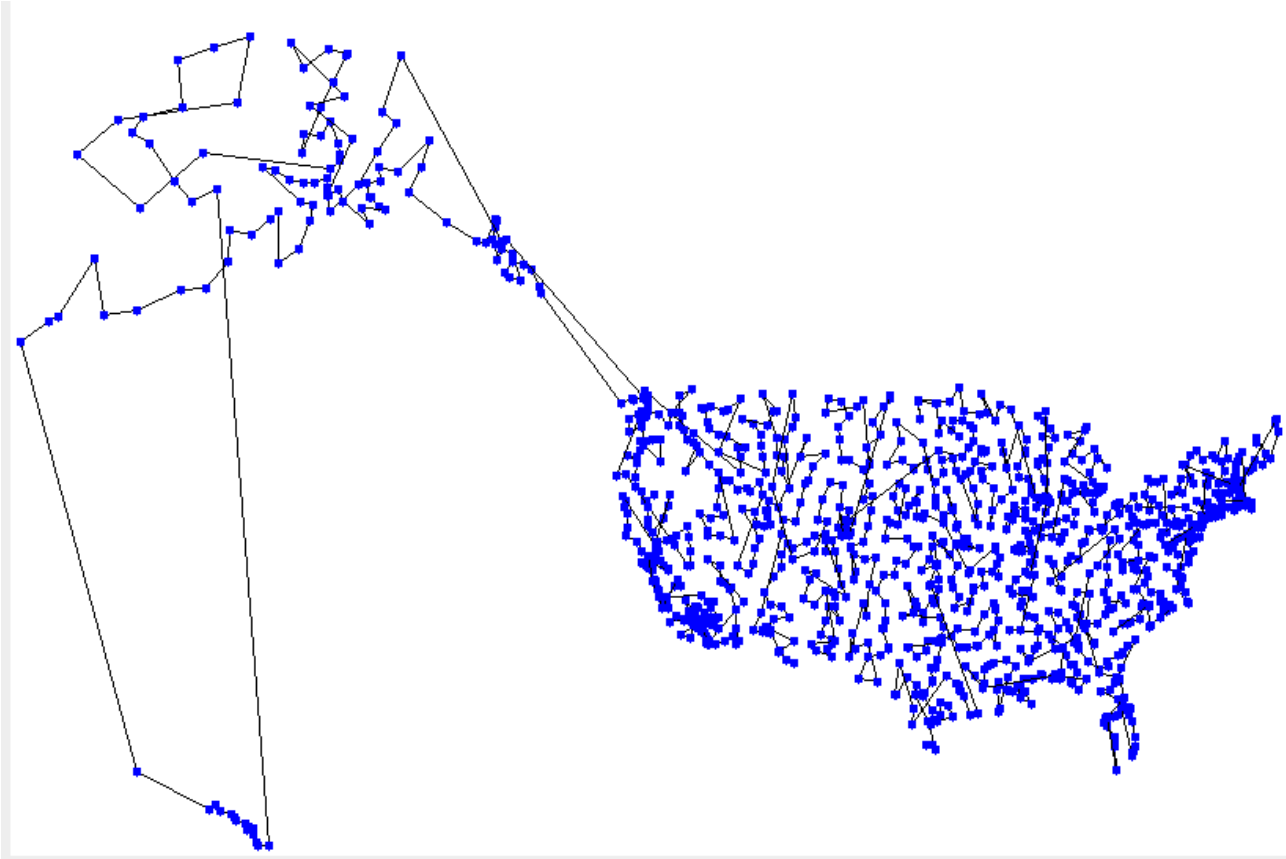

Figure 2: map of a usa tour as calculated by tsp-approx. (please note the map is not to scale)

Performance.

The MST-Kruskal implementation is the asymptotically fastest version known to exist.
When parallelizing it we have to take into account the overhead of openMP and transactional memory. For small graphs (10,000 vertices) the overhead as measured in running time is about 10%. For even smaller graphs the penalty can be up to 20%. The parallelization scheme allows Kruskal to look at a smaller number of edges thereby saving some time. The parallelized Kruskal runs about 15% faster in a 4-thread mode on a 4-core machine. The speed-up does depend on the data. Randomly generated vertices in the plane behave differently than graphs with vertices ordered (generated by 2 for-loops, where the vertex coordinates are simply the two indicies). I would like to try the algorithm on really large graphs but due to hardware constraints I was not able to.

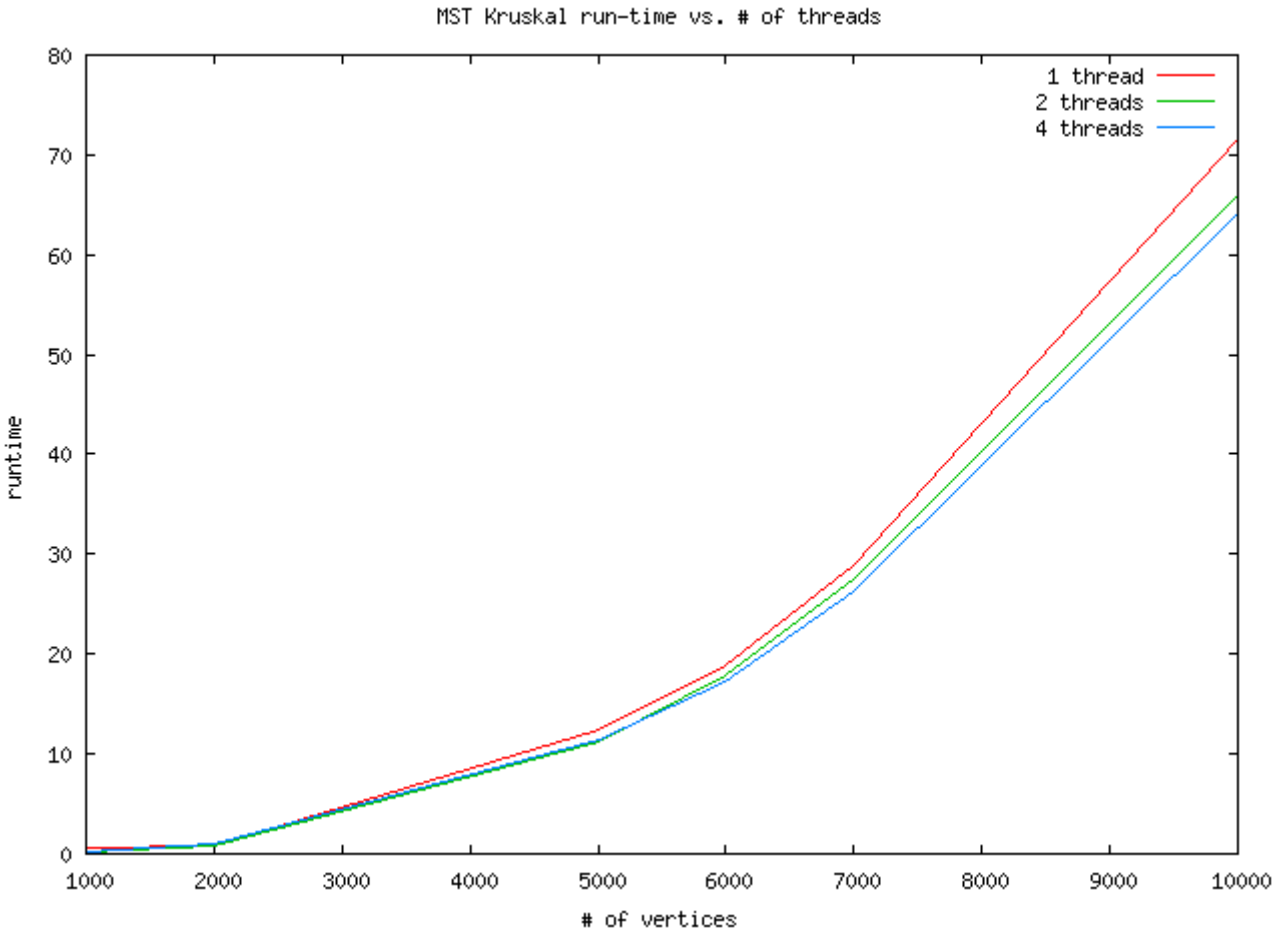

Figure 3: the runtime of MST-Kruskal with different graph sizes and varying thread numbers.

Dissatisfied with the lack of performance of the parallelized Kruskal's I set out to investigate the bottleneck. After a number of tests, I determined that it was the sort routine in the algorithm that took up the most time. I have parallelized that routine and ran more tests. Now the performance was satisfactory as the figure below shows (Figure 4).

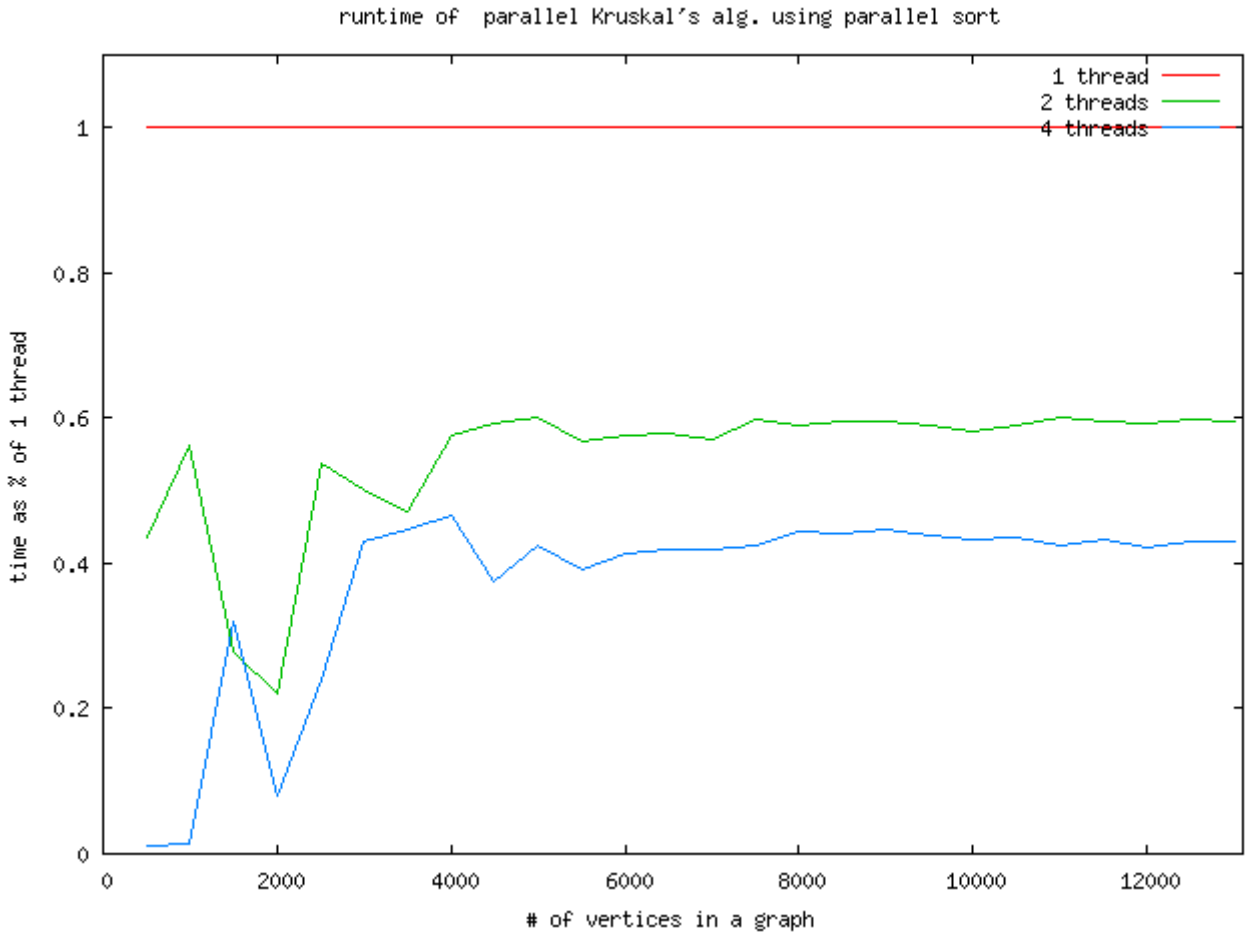

Figure 4: the runtime of parallel MST-Kruskal with a parallel sort function. Al almost 2.5 x speed-up was achieved.

Further Improvements.

There is a number of further possible improvements in the algorithm.

1. Kruskal – stop Kruskal when the edges found is n-1 (a tree of n vertices has n-1 edges), there is no need to examine further edges.
2. Data structures – it is possible to rethink the way we store data in the program and therefore be more space efficient. This will speed up the algorithm and free up space to try even bigger graphs.
3. Redesign the DSF data structure so that we can allow helper threads to run continuously in there assigned areas of the edge array.

References:

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2001.

Herlihy and Moss 1993 M. Herlihy and J.E.B. Moss, “Transactional Memory: Architectural Support for Lock-Free Data Structures,” in Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA ’93), 1993, pp. 289–300.

Katsigiannis, Anastasios, Nikos Anastopoulos, Konstantinos Nikas, and Nectarios Koziris. "An Approach to Parallelize Kruskal's Algorithm Using Helper Threads." In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pp. 1601-1610. IEEE, 2012.