Authors: Khalid Akash, Brandon Smith, Suva Shahria, Ryan Morey
Problem 2 Write Up - Parallel MST

Build instructions can be found on readme.md on the problem_2 root directory.

Note: This problem was implemented in **Rust**, a new and rapidly growing systems level programming language, with the permission of the instructor. The problem is discussed with the reference point and limitations of our limited mastery and understanding of the language.

The version of the minimum spanning tree algorithm that was implemented was the Prim's algorithm.

Graph Generation

mod.rs - generate_graph()

Our program first generates a graph using an adjacency list to store the neighbors. To simplify the book keeping part of our algorithm, we assigned the index that the node was kept in as the id of the node. For example, a node at the index 2 would have the id of 2. The graph was ensured to be connected by making sure that it had connected to at least one unique neighbor. The amount of neighbors that were assigned to each node was random, but scaled with the size of the graph. The edges were non-unique, which meant that the MST's were non-unique as well; this was done to speed up graph generation time for large number of nodes. Generating a graph with node counts of over 2000 took significant amount of time, with 8000 nodes taking approximately a minute. We decided to add multithreading to the generation, with conditional variables coordinating the array placements of the generating graph.

.

Sequential Algorithm

mst.rs - compute_sequential_mst()

The purely sequential version of the algorithm was implemented with a custom made priority queue (min_heap.rs) that allowed for the decrease key operation. The priority queue is also supplemented with a hash set that allows for quick look up to see if a certain node exists, this is crucial for fast computation on the inner loop of the algorithm. The algorithm used the priority queue to have quick retrieval of the node with the minimum recorded edge and continues to run the main loop as long as the priority queue is not empty. The node that is retrieved (popped from the min heap priority queue). An inner loop is run to query every single neighbor to update the minimum spanning tree. It also keeps the minimum edges that are connected to a particular node in an array and is updated on every inner loop.

Parallel Algorithm

mst.rs - compute_parallel_mst()

Prim's algorithm can be very hard to parallelize. Doing significant research, we found that the best way to parallelize the algorithm was to multithread the inner loop where the neighbors are queried for their weight. The outer loop that takes the minimum weight from the priority queue cannot be parallelized as it needs to be sequential to find the least weighted node. Based on the number of requested threads, the neighbors of the particular nodes are divided evenly (with remainders on the last thread). One memory access need to be synchronized, the priority queue, and in particular the decrease key operation. Even though the other variables, such as the minimum spanning tree, or the weights are accessed, the individual memory locations that the threads access within the arrays are different, so they need not be synchronized. In Rust, instead of protecting critical regions in code, like you would with pthreads, the variables must be protected. When the variable goes out of scope, either from a loop, a function, or an artificially created scope, the lock is released and the mutex is freed from memory.

Due to Rust's static enforcement of synchronization of variables even when synchronization isn't necessary. To not follow some of Rust's enforcements, we had to mark part of the code as "unsafe", even though logically, we feel that it is perfectly fine. We also wrapped the shared variables in an atomic reference counter, similar to Shared Pointers in C++, to ensure that once the references go to 0, the memory is freed as well. To parallelize the neighbors, we evenly partitioned the neighbors among the threads and processed them. Any bottleneck within the threads would likely be in the decrease key operation as that can take Log(N) time complexity, and contention for them during that time can block many of the threads.

Analysis

As can be seen by our benchmark results, the performance benefits of the threaded algorithm grows as a function of the numbers of nodes/edges in the graph. This is the cost of launching the threads, synchronization, and the partition of the neighbors able to be computed upon for each threads increases for every additional neighbor being processed per thread (even though the number of neighbors per node is random, the amount of neighbors a node can have grows based on the number of nodes available). As mentioned earlier, the performance gains aren't much compared to other types of problems as a lot of the memory is being contented for rather than fully maximizing the computation power of the CPU. The use of atomic reference counters/synchronization primitives also increase the cost as atomic instructions are very expensive (memory access, cpu blocking).

The results are much more convincing on a debug build on smaller sample sizes.

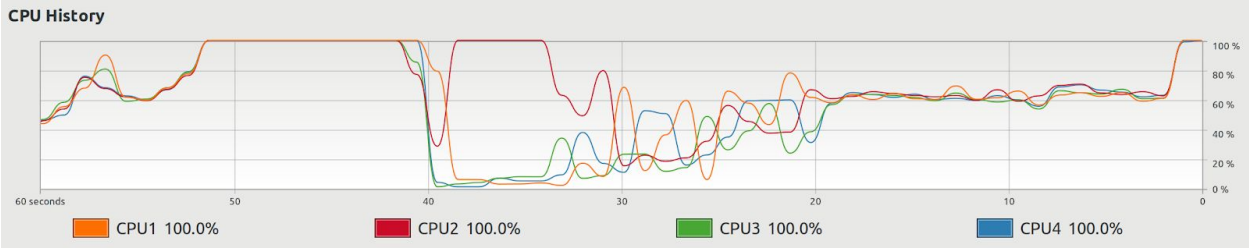
Note: The difference between the sequential algorithm vs 1 threaded algorithm is that the

sequential algorithm creates no additional threads and sets no synchronization primitives/protection whilst the 1 threaded one does.

The below is some sample data of one of our benchmarks on a 6 core machine (12 SMT), with optimization level set to 3.

========Generating graph of size 4000 with 12 threads========
[Sequential] MST of 4000 nodes and 5911030 edges took 114 ms
[1 Thread(s)] Parallel MST of 4000 nodes and 5911030 edges took 213 ms
[2 Thread(s)] Parallel MST of 4000 nodes and 5911030 edges took 188 ms
[4 Thread(s)] Parallel MST of 4000 nodes and 5911030 edges took 216 ms
[8 Thread(s)] Parallel MST of 4000 nodes and 5911030 edges took 341 ms
[16 Thread(s)] Parallel MST of 4000 nodes and 5911030 edges took 608 ms
========Generating graph of size 8000 with 12 threads========
[Sequential] MST of 8000 nodes and 23440304 edges took 470 ms
[1 Thread(s)] Parallel MST of 8000 nodes and 23440304 edges took 753 ms
[2 Thread(s)] Parallel MST of 8000 nodes and 23440304 edges took 525 ms
[4 Thread(s)] Parallel MST of 8000 nodes and 23440304 edges took 500 ms
[8 Thread(s)] Parallel MST of 8000 nodes and 23440304 edges took 737 ms
[16 Thread(s)] Parallel MST of 8000 nodes and 23440304 edges took 1269 ms
========Generating graph of size 16000 with 12 threads========
[Sequential] MST of 16000 nodes and 94360173 edges took 1960 ms
[1 Thread(s)] Parallel MST of 16000 nodes and 94360173 edges took 2797 ms
[2 Thread(s)] Parallel MST of 16000 nodes and 94360173 edges took 1694 ms
[4 Thread(s)] Parallel MST of 16000 nodes and 94360173 edges took 1291 ms
[8 Thread(s)] Parallel MST of 16000 nodes and 94360173 edges took 1629 ms
[16 Thread(s)] Parallel MST of 16000 nodes and 94360173 edges took 2661 ms
========Generating graph of size 32000 with 12 threads========
[Sequential] MST of 32000 nodes and 375528384 edges took 8313 ms
[1 Thread(s)] Parallel MST of 32000 nodes and 375528384 edges took 10562 ms
[2 Thread(s)] Parallel MST of 32000 nodes and 375528384 edges took 6121 ms
[4 Thread(s)] Parallel MST of 32000 nodes and 375528384 edges took 3804 ms
[8 Thread(s)] Parallel MST of 32000 nodes and 375528384 edges took 4143 ms
[16 Thread(s)] Parallel MST of 32000 nodes and 375528384 edges took 6002 ms
========Generating graph of size 48000 with 12 threads========
[Sequential] MST of 48000 nodes and 847480440 edges took 20785 ms
[1 Thread(s)] Parallel MST of 48000 nodes and 847480440 edges took 24867 ms
[2 Thread(s)] Parallel MST of 48000 nodes and 847480440 edges took 13944 ms
[4 Thread(s)] Parallel MST of 48000 nodes and 847480440 edges took 8059 ms
[8 Thread(s)] Parallel MST of 48000 nodes and 847480440 edges took 8135 ms
[16 Thread(s)] Parallel MST of 48000 nodes and 847480440 edges took 10605 ms

CPU Usage on a two core system (4 SMT), read from right to left, the time is read from time from now (most right):



The computation for the MST via threading starts at 30 seconds (middle of the graph), and as shown, the computation is not CPU bound; whilst the computation at 32 to 40 seconds maxes out 1 CPU, which is the sequential computation. The computation from 40 to 53 seconds is the graph generation, which maxes out all the cores.

Complexity

Analysis of sequential code complexity with a binary heap (min heap, priority queue)
For each node, V, there must be a delete min operation done on the priority queue (min heap). The delete min operation takes a log(V) time complexity as the priority queue is populated with V nodes as well. For each neighbor (edge), there must also be a decrease key operation done which takes a log(V) time complexity. So the algorithm overall takes (V+E)LogV, or ELogV. There are additional constant time computations done, but they are ignored for big O analysis.

Based on our execution times -
For every 2x increase in our sample size (nodes and edges), the execution time increases by 4x. This shows a strong correlation with an O(N) time complexity.

Analysis of parallel code is very similar to the sequential code. The delete min operation is done per node V, so there is a VLogV time complexity. The speedup comes from partitioning the neighbors (edges) via the number of threads (T); this would allow for a speed up of {V + (E/T)}LogV time complexity, however, this isn't quite a linear speed up as we would think it to be due to dependency checks done in the code. The results are quite a lot slower and depends greatly on the sample size of the graph.

Based on our execution times -
For every 2x increase in our sample size with 4-8 threads (which was optimal on our testing system of 6 cores), we achieved on average 2-3x increase in execution time. Which is a much better execution time seen than in the sequential execution, especially on a large sample size.