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). Three memory accesses need to be synchronized, the priority queue, the minimum weight array, and the final minimum spanning tree structure. 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. All three of the synchronized memories are protected by read-write locks (multiple readers, one writer) instead of mutexes, to better parallelize them.

This is where a little discussion on Rust comes in.
Rust is a language that attempts to maintain the performance standards that are found in lower-level languages such as C or C++. However, what differentiates it is it's enforcement of it's ownership based memory model that ensures that there are no data races, or memory leaks. Rust can guarantee at compile time that these hold true based on its rules on how data is passed around. To steer the discussion towards the parallel algorithm, all memory that are shared amongst threads must be wrapped by an atomic reference counters (thread safe). Rust uses OS level threads instead of green threads.

When variables are wrapped in atomic reference counters, it forces the compiler to do away with many of the compiler optimizations possible as ordering becomes important. It is particularly even more significant because the loop that is being parallelized is actually not CPU bound as there aren't too much in terms of computation that is happening, but rather, there is a lot of contention. This means that the sequential algorithm in many scenarios, when the optimization level is set to the maximum, comes out significantly on top over the threaded one. To properly benchmark the logical conversion from the sequential algorithm to the parallel algorithm, the optimization levels were all set to 0. Significant experimentation/research went into finding a way around this, but unfortunately, we had to settle for this.

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 in the graph. This is due to the ratio of cost of computation vs the cost of creating/setting up synchronization 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).

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 0.

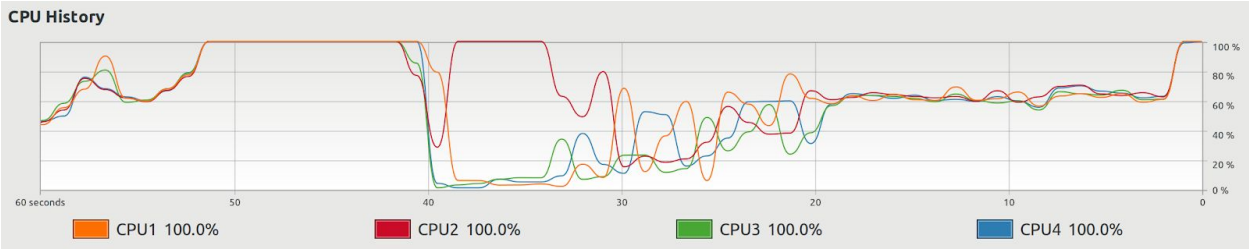
[Sequential] MST of 1000 nodes took 279 ms
[1 Thread(s)] Parallel MST of 1000 nodes took 428 ms
[2 Thread(s)] Parallel MST of 1000 nodes took 295 ms
[4 Thread(s)] Parallel MST of 1000 nodes took 272 ms
[8 Thread(s)] Parallel MST of 1000 nodes took 288 ms
[16 Thread(s)] Parallel MST of 1000 nodes took 324 ms

[Sequential] MST of 2000 nodes took 1049 ms
[1 Thread(s)] Parallel MST of 2000 nodes took 1521 ms
[2 Thread(s)] Parallel MST of 2000 nodes took 991 ms
[4 Thread(s)] Parallel MST of 2000 nodes took 861 ms
[8 Thread(s)] Parallel MST of 2000 nodes took 845 ms
[16 Thread(s)] Parallel MST of 2000 nodes took 902 ms

[Sequential] MST of 4000 nodes took 3994 ms
[1 Thread(s)] Parallel MST of 4000 nodes took 5820 ms
[2 Thread(s)] Parallel MST of 4000 nodes took 3531 ms
[4 Thread(s)] Parallel MST of 4000 nodes took 2975 ms
[8 Thread(s)] Parallel MST of 4000 nodes took 2885 ms
[16 Thread(s)] Parallel MST of 4000 nodes took 2999 ms

[Sequential] MST of 8000 nodes took 15514 ms
[1 Thread(s)] Parallel MST of 8000 nodes took 22073 ms
[2 Thread(s)] Parallel MST of 8000 nodes took 12815 ms
[4 Thread(s)] Parallel MST of 8000 nodes took 10655 ms
[8 Thread(s)] Parallel MST of 8000 nodes took 9991 ms
[16 Thread(s)] Parallel MST of 8000 nodes took 10122 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.

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.