Traveling Salesman: Heuristic Scaling Analysis

Garrett Folks, Quincy Mast, Zamua Nasrawt

April 27, 2017

1 Background

All code referenced is available at **https://github.com/mastqe/470-gputsp/tree/final**

1.1 Traveling Salesman

The traveling salesman problem (abbreviated as TSP) asks the question: "Given a list of cities and the distances between them, what is the shortest path that begins and ends in the same city while passing through every city exactly once?" It is classified as an NP-Hard problem, so there are no known exact solvers that can run in polynomial time. In fact, there is no known exact solver faster than $O(2^n)$, and an exhaustive search is $O(n!)$ time. Heuristics are used to find 'good enough' solutions for large problems in reasonable amounts of time. We had initially hoped to analyze several different algorithms in terms of each other; however, we had difficulty finding even serial implementations of many of the algorithms we found. We did find a couple of parallel heuristic implementations, but they had different termination conditions and were not comparable to each other. In the end, we decided on two common heuristics: k-opt search and ant colony optimization. We analyzed these two heuristics and how well they scale when parallelized. We also looked at one exact solver, but it was not implemented in parallel and serves more to contrast exact solvers and heuristics.

1.2 K-Opt Heuristic

The K-Opt heuristic takes k edges of a path, performs a swap of the edges, and checks if the new path length improves the current minimum. In order to keep improving, the initial path must be updated so that new swaps can be tried. The simplest version of this algorithm only involves two edges in each swap. Parallel implementations can take all possible edge pairs and test all swaps simultaneously. After all swaps are tested, a single swap can be chosen that will create the shortest tour. LOGO-Solver is an application we found that approximates a solution for the TSP using a 2-opt local search [RS13]. It uses the process described above to keep improving tours until it finds one that is within some percentage of optimal. This assumes that you have access to the optimal solution length, but you could also just stop if after a given runtime and take the current minimum tour. This implementation is heavily parallelized and was optimal for our analysis because we did not need to make any modifications to the package.

1.3 Ant Colony Optimization

The ant colony optimization (abbreviated as ACO) is a probabilistic technique for finding "good" paths through graphs. The premise behind the ACO is that initially a colony of ants will wander around randomly until finding food. To be able to find the food again after returning to the colony, the ant will lay down a pheromone trail so that it can find the food source again. Other ants in the colony will probabilistically choose to follow pheromone trails with the strongest concentration, thus reinforcing the existing pheromone trails. The key to reinforcing shorter trails is that over time, the pheromone trails evaporate, leaving the longer trails to fade while the shorter trails are reinforced much more frequently, giving the ants a relatively short path to their food source. We

can apply this behavior to produce relatively good solutions for paths through a weighted graph (i.e. the traveling salesman problem). The basic idea is to simulate ants traveling circuits through the cities and leaving pheromones along their paths. After simulating some ants the pheromone trails can be gathered and a global set can be updated. Future ants will use this set to determine what paths they should take. In order to parallelize this, you can simulate multiple ants simultaneously to get a better sample to update the pheromones. In our research we found an implementation created by the original authors who proposed the ACO [DS04]. This package implements most of the variations of ACO that exist and is quite robust; however, there is not a parallel version readily available. For this reason, part of this project included parallelizing this implementation so that its performance scaling could be analyzed.

2 LOGO-Solver

2.1 Experimentation

LOGO-Solver is the only implementation of 2-opt search that we found which includes both CPU and GPU parallel options. We used instances from TSPLIB standard problem library as inputs [Rei91]. This library includes various instances, many of which are modeled from real world problems such as circuit board layout. We selected several input sizes ranging from 52 to 33,810 nodes/cities. We ran each instance size on various thread counts and a GPU, measuring wall time as our metric of performance. All tests we ran were on an Nvidia Quadro K5200 GPU and dual quad-core hyper-threaded CPUs for a total of 16 cores. We did not limit our scaling tests to this core count though. We tested all the way up to 64 threads, and saw scaling even after the thread count exceeded physical cores. In order to keep run times reasonable, we set the termination condition to be five percent of optimal. This means that LOGO-Solver would run until it found a solution that was within five percent of the known optimal solution.

2.2 Results

In Figure 1 we see the main results that we gathered from our LOGO-Solver tests. This graph demonstrates the speedup of each thread count as the problem size is increased. The dark black line represents 1x speedup, i.e. serial execution time. Points that are below this line are showing execution times that are slower than serial. Above the line, points are showing faster than serial execution. This immediately shows us that problem sizes smaller than about 1000 cities do not scale while larger instances begin to scale well.

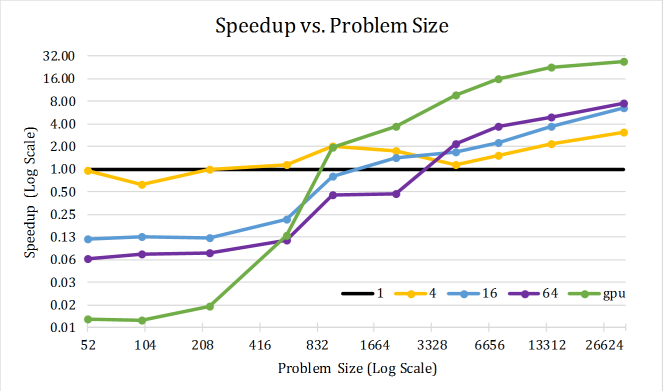

Figure 1: Logo-Solver scaling results

For large problem sizes we believe the time to compute all possible swaps masks the time needed to generate new initial tours. Small instances cannot mask the time it takes to compute initial tours serially, so they do not scale. Small input sizes also show the effects of

threading overhead which negate any gains from parallel computation. As thread count increases it actually begins to increase time to solution for these small instances. We do begin to see speedup after the number of cities reaches 1000, but the scaling of the speedup is nowhere near linear. We observed that speedup continued to grow even after exceeding the number of physical cores. This problem seems to be very computation heavy because the benefits of parallel execution mask the threading overhead, especially with large instances. The maximum speedup we saw from increasing the number of threads on CPU was about 7x, with larger problem sets having better speedup.

The speedup we saw from GPU acceleration is very interesting, and seems to show the strengths and weaknesses of the GPU architecture. After 1000 cities we start to see the GPU being saturated which allows us to get increasingly greater speedups compared to the CPU implementation. In GPU computations, memory management can become a concern since things need to be explicitly moved between the host and the device. Once the problem sizes are large enough the computation time begins to mask the time needed for memory management. For the largest input size (33,810 cities) the GPU achieved a 26x speedup, compared to the CPU's 7x. This strong scaling was significantly more impressive than the CPU and is what we wanted to see from GPU accelerated workloads. With that being said, when the problem size was less than 1000 cities, the CPU implementation was consistently faster because memory management becomes a significant value in the timing.

We ran the NVIDIA profiler to see how the GPU was executing the code and found that with small input sizes only 70-80 percent of the time was being used for computation. As the problem sizes increased though we began to see the percentage of time spent doing computation on the GPU approach 100 percent. These smaller input sizes show the inefficiency of an unsaturated GPU. The architecture requires heavy workloads to be efficient, and the small problem sizes aren't enough to get full utilization.

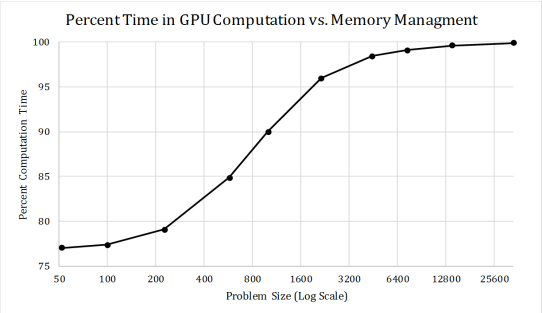

Figure 2: GPU computation percentage

When analyzing GPU code it is always good to determine what amount of CPU power is needed to achieve the same performance. In our case we decided to do some testing on the same thread counts but with a machine that has enough physical cores to match the threads. Using Amazon Web Services (AWS) we were able to run most of the tests on a machine with 64 physical cores. Using this machine we were able to determine that 64 physical cores are almost able to achieve the same speedup as the GPU. Overall the shape of the graph stays the same, but the scaling of large instances on higher thread counts is better. From our preliminary test it appears that a GPU is still faster than 64 CPU cores, but not by much. The power of the GPU still shines as it is a much more cost effective way to achieve this kind of performance.

We looked briefly at using the GPU's more efficient computation to achieve higher quality solutions in the serial execution time. Since the GPU is so much faster on large problem sizes it is possible to let it run longer and get a better solution. Using the CPU serial execution time as a basis we explored how much better the GPU could do in that time. In the end, we found that only a one to two percent improvement is achieved. As the heuristic approaches the optimal solution, it takes more time for each new improvement

to be found since it relies on randomness. Unless this small improvement is strictly necessary, it seems to be more efficient to keep the worse solution that is found in the much quicker time.

3 Ant Colony Optimization

3.1 Methods

The implementation of ACO that we found only includes a serial run mode. Thus, our goal was to parallelize this implementation so that we could analyze its performance scaling. Initial profiling of the program showed that the majority of runtime is spent doing tour constructions. As we analyzed this further, we found that during this phase each ant is working independently. The main function for constructing tours is just a series of for loops that perform operations for each ant in the set. The pheromone updates after each set of tour constructions contributed minimally to the overall execution, so we parallelized only the tour construction phase.

The section of code that we parallelized was a perfect candidate for using OpenMP since it was just a series of for loops with no dependencies. There were a few problems that we needed to resolve before the parallelization worked, but in the end we were successful. Initially the program was simulating a single step for each ant before moving on to the next step. We found that simulating a full tour for a single ant before moving on to the next ant lent itself to being parallelized more easily. To do this we had to reverse one nested loop so that the inner loop became the outer loop and vice versa. The second hurdle was random number generation for initially placing the ants. We had to rework how seeds were allocated so that each thread was given an independent seed to maintain better randomness. After this was completed we had a parallel implementation in which each ant (or a block of ants) could be simulated in a parallel thread.

3.2 Experimentation

We only tested various CPU core counts since we were not able to implement a GPU parallelization although we believe it is possible. Inputs were the same ones tested with LOGO-Solver from the TSPLIB standard problem library [Rei91]. Each problem size was again run on various thread counts up to 64 threads. Our results are from running this implementation on an AWS instance with 64 physical cores.

The termination condition for this implementation is either an optimal tour length or a time limit. In order to simulate LOGO-Solver's termination condition we multiplied the optimal tour length by 1.05 to increase the necessary length by five percent. We also had to increase the time limit so that we knew the program was terminating based on tour length not time. We could not achieve reasonable times on the large problem sizes so we only went up to 1000 cities. On larger instances the heuristic seems to stagnate and it takes large stretches of time for very small improvements.

When we ran this implementation with thread counts beyond the number of physical cores we did not get good scaling. We did some initial testing on a 16 core machine and we did not see speedup beyond 2x. This implementation is not doing as much heavy computation and the threading overhead is not masked by larger problem sizes. These results were very underwhelming so we ran our tests on an AWS instance with 64 physical cores.

3.3 Results

Using this machine our results showed strong scaling. Since we don't have to deal with threading overhead we are able to approach the theoretical maximum speedup. Amdahl's law limits the maximum speedup to around 4x because we parallelized about 75% of the program's execution. We were not able to run tests on very large instances but we do not believe that the implementation would get more efficient. Unlike LOGO-Solver we do not see inefficiencies with small problem sizes, and good scaling starts on much smaller instances.

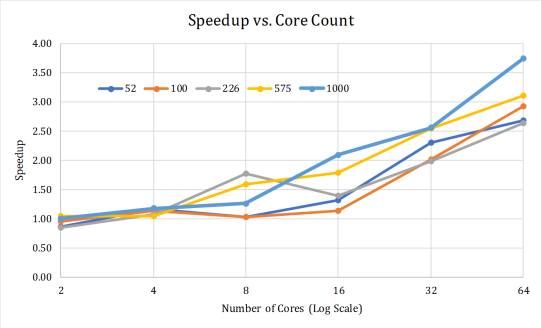

Figure 3: ACO scaling results

4 Comparison to Concorde Exact Solver

Concorde solver is a serial CPU implementation that is "a state of the art implementation" and as of 2001, "is widely regarded as the fastest TSP solver, for large instances, currently in existence" [Wik17]. This implementation uses a branch-and-bound algorithm (for large instances, but will choose a more optimal algorithm for smaller instances) that guarantees the optimal solution. Concorde is faster for solving instances to an exact solution since the heuristics are relying on randomization to improve their solutions. For example, LOGO-solver took 30 seconds to get within 0.2 percent error for 575 cities, while Concorde took only 16 seconds to solve it exactly. From reading about Concorde it appears that there should be some way to run it in parallel, possibly distributed, but the documentation is sparse and we were unable to get anything working beyond the serial implementation. It is not really a fair metric to compare heuristics to exact solvers; however, it is interesting to explore a problem where the problem space is split in terms of best solution. Certain problems may require an exact solution while others may just need something that is close enough. The optimal algorithm really depends on the needs of the user.

5 Conclusion

Parallelization of different implementations of TSP showed scaling on both CPU and GPU architectures. We were able to see speedups of about 7x when executing a parallel implementation of LOGO-solver on CPU, and an exceptional 25x speedup on the GPU when given large instances. We also found that it takes 64 physical CPU cores to achieve performance close to that of the GPU. So even though threading overhead can be overcome in this case, it is still a factor and greatly inhibits scaling. However, we found that the GPU implementation only excels in this manner when given an instance size over 1000 cities, which was likely due to the GPU being saturated with work. It also demonstrates the overhead of memory management on GPUs that can cause poor performance.

In looking for other implementations we did not find any that had CPU and GPU run options. Instead we took a serial implementation of the ant colony optimization and parallelized it for analysis. ACO is a novel approach to solve the traveling sales-

man problem and seems to have some inherently parallel elements. Using OpenMP we were able to parallelize the code which was taking the majority of serial execution time. After parallelization we observed speedups approaching the theoretical maximum based on Amdahl's Law. This implementation responded well to parallelization and has the potential to achieve even higher speedups.

6 Future Work

The majority of future work related to this project comes from the ant colony optimization. 2-opt search demonstrates some interesting results, but LOGO-Solver is a very good parallel implantation and quite comprehensive. The implementation of ACO that we worked with is also quite comprehensive; however, it is strictly serial, so there is a lot of room for improvement in terms of parallelization. We made some basic improvements with OpenMP, but not for the entire algorithm. Future work could include more analysis of the algorithm and its variations to determine if more of the implementation could be parallelized beyond the tour construction phase. There is a lot of research available on different versions of the ant colony optimization, but very little available code implementing them. It may also be more appropriate to build a new implementation from scratch with parallelism in mind from the start.

The most concrete future work we can envision is porting our parallel ACO implementation to run on a GPU with CUDA. We did some preliminary work on this; however, we were unsuccessful. In theory this is quite an achievable task, but it is beyond our expertise at this time. The basic structure of the program is conducive to having each ant be simulated on a single thread of the GPU. The big trick is getting all of the data structures that store the ant and city information to the GPU. As it is the memory would need to be copied back and forth to the device between the tour construction and pheromone update phases. It is possible that if more of the program was parallelized that this would be easier and possibly only need an initial memory copy. We believe that with a little more GPU expertise or time to learn, this would be achievable.

An interesting inclusion in the ACO implementation we used is a local search phase. We disabled this for our testing, but it uses a k-opt search to improve on the tours the ants construct. Between the tour construction and pheromone updates a local search is run to find even better tours to update the pheromones with. Using this stage the speed to optimal solutions and the overall quality of produced tours is greatly improved. It may be possible that the parallel implementations of 2-opt search used in LOGO-Solver could be incorporated into this code. When running ACO with this local search phase enabled the search took the majority of execution time, rather than tour construction. If this could be parallelized then we believe that quality solutions could be found very quickly.

Concorde solver could also be a stepping stone for future work. This program is extremely comprehensive and quite efficient for an exact solver. There are some references to this being used in a distributed fashion, and this could be an interesting avenue to explore. Much of the code is also implemented as an API so that a new program could be built on the main functions but with parallelism in mind. The limit to this could be the need for a linear programming solver, but more research is needed.

References

[DS04] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004.

[Rei91] Gerhard Reinelt. TSPLIB— a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[RS13] Kamil Rocki and Reiji Suda. High performance gpu accelerated local optimization in tsp. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1788–1796, Washington, DC, USA, 2013. IEEE Computer Society.

[Wik17] Wikipedia. Concorde TSP Solver — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Concorde%20TSP%20Solver&oldid=725437558, 2017. [Online; accessed 23-March-2017].