TRAVELING SALESMAN: A HEURISTIC SCALING ANALYSIS[*]

Quincy E. Mast, Zamua O. Nasrawt, Garrett L. Folks, Michael O. Lam
Department of Computer Science
James Madison University
Harrisonburg, VA 22807
540-568-3335
{mastqe, nasrawzo, folksgl}@dukes.jmu.edu, lam2mo@jmu.edu

ABSTRACT

In this paper, we analyze two heuristics that approximate the Traveling Salesman Problem: K-Opt search and ant colony optimization. Our goal was to explore how these heuristics perform when run in parallel on multiple CPU cores as well as using GPU computing. We found that the K-Opt search heuristic showed impressive performance scaling results, especially when executed on a GPU. We also parallelized portions of the ant colony optimization and found good scaling. We conjecture that the ant colony optimization could be greatly improved with the use of GPU computing.

BACKGROUND

Traveling Salesman

The traveling salesman problem (TSP) poses the following question: given a complete weighted graph of n cities (vertices) and the distances (edge weights) between them, what is the shortest Hamiltonian cycle (a path that begins and ends in the same vertex and passes through every other vertex exactly once)? TSP is classified as an NP-Hard problem [1]. An exhaustive search is O(n!) time, and there is no known exact solver faster than $O(2^n)$, where n is the number of cities to be visited. In this paper, we analyze the parallel performance and scaling behavior of two common TSP heuristics (K-Opt search and ant colony optimization). We also contrast these results with a robust modern exact solver running serially.

K-Opt Heuristic

The K-Opt heuristic continually permutes k edges of a path, attempting to improve the current shortest path. We focus on 2-opt, the simplest version of this algorithm that involves only two edges in each swap. Parallel implementations can test many swaps simultaneously and then choose the single swap that will create the shortest tour. LOGO-Solver is an implementation of this heuristic that approximates a solution for TSP using a 2-opt local search [5]. It uses the process described above to keep improving tours until it finds one that is within some percentage of the optimal solution. This assumes the optimal solution is known, but the process can be terminated prematurely, returning the current minimum tour.

Ant Colony Optimization

The ant colony optimization (ACO) is a probabilistic technique for finding "good" paths through graphs. The premise behind ACO is that an ant will wander around randomly until it finds a food source, after which it will lay down a pheromone trail so that it can find the food again. Other ants in the colony will probabilistically choose to follow stronger pheromone trails. Over time, the pheromone trails evaporate, leaving longer trails to fade while shorter trails are continually reinforced, resulting in an approximation of the overall shortest path.

This behavior produces relatively good solutions for paths through a weighted graph. This can be applied to TSP by simulating ants traveling Hamiltonian cycles through the cities. Parallelization of this process is done by simulating multiple ants simultaneously and aggregating their pheromone trails. The implementation we used was created by the original authors of ACO [2]. This package implements most of the variations of ACO that exist and is quite robust; however, there is no parallel version readily available. For this reason, we parallelized a portion of this implementation for our analysis.

Outline

In the first section of this paper we analyze an existing implementation of the K-Opt heuristic, presenting results of our experiments exploring the parallel scalability of this implementation. Next, we discuss our parallelization of ACO and its performance analysis results. Finally, we present experiments using a serial implementation of an exact TSP solver, comparing it to the heuristics.

K-OPT HEURISTIC WITH LOGO-SOLVER

Methods

LOGO-Solver was the only available implementation of 2-opt search that included both CPU- and GPU-parallel implementations [5]. We used this because it was already heavily parallelized and didn't require any modification for our analysis. We designed our own scaling tests and analyzed the results.

Experimentation

We ran the program using inputs from TSPLIB, a standard TSP dataset library [4]. The library is large and includes instances modeled after real world problems such as circuit board layout. Graph sizes range from 52 to 33,810 nodes, and the tests were executed using various thread counts as well as a GPU. All tests were run on a single server with dual Intel Xeon E5-2623v3 3.0 GHz CPUs with a total of 8 physical (16 hyper-threaded) cores, as well as an Nvidia Quadro K5200 GPU with 2304 CUDA cores. We ran several trials for each experiment and took the minimum wall time results for each test to minimize variation in the results due to the hardware and operating system. The termination condition was set to be within five percent of the known optimal solution for reasonable run times.

Results

Figure 1 displays scaling results for LOGO-Solver. The graph shows speedup for several thread counts as we increased the problem size. The dark black line represents serial execution, so points below this line are slower-than-serial execution times. This immediately shows that there is no improvement in parallelizing until the problem size reaches approximately 1000.

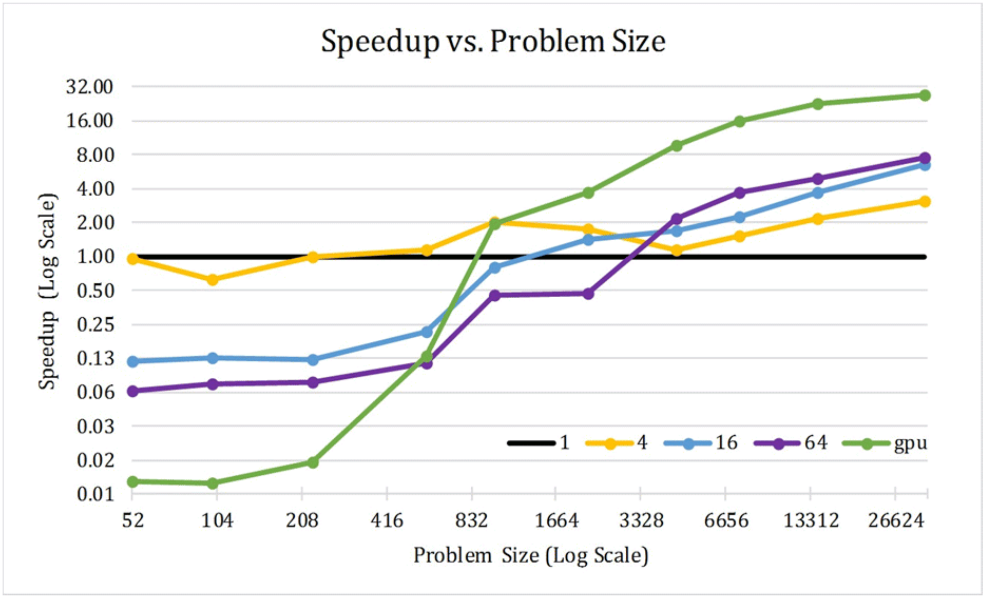

Figure 1: LOGO-Solver scaling results

For small problem sizes, it is likely that the time needed to generate initial tours will overwhelm the time spent optimizing those tours. Small input sizes are also more significantly affected by threading overhead, and so higher thread counts actually increases the time to solution. Even with larger inputs, the maximum speedup on a CPU was about 7.5x.

The observed speedup from GPU acceleration highlights the strengths and weaknesses of the GPU architecture, where data must be copied explicitly between the host and the GPU. Until the problem sizes are large enough, the data movement time overshadows the computation time. After 1000 cities the GPU has enough computational load to maximize utilization of its cores, allowing for increasingly greater speedups compared to the CPU implementation. For the largest input size (33,810 cities) the GPU achieved a 26x speedup compared to the CPU's 7.5x.

We used the Nvidia profiler to verify that only 70-80 percent of execution time was utilized for computation with small problem sizes while large problem sizes approached 100 percent. This trend is shown in Figure 2. Smaller input sizes show the inefficiencies of a GPU. The architecture requires heavy workloads to be efficient, and small problem sizes don't provide enough computational load for maximal GPU efficiency.

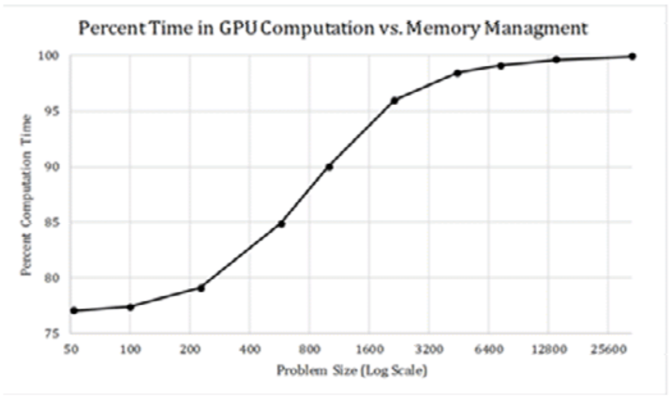

Figure 2: GPU computation percentage

Finally, we used the Amazon Web Services (AWS) cloud-computing platform to determine the amount of CPU power necessary to match the performance of the GPU on a single machine. We re-ran our tests on an AWS EC2 instance with 64 physical cores and determined that this machine was almost able to achieve the same speedup as the GPU. Overall the shape of the graph was the same as in Figure 1, but the scaling of large instances on higher CPU thread counts improved. It still appears that a GPU is still marginally faster than 64 CPU cores and is thus a much more cost-effective solution in this domain ($0.90/hr vs. $3.20/hr on AWS).

ANT COLONY OPTIMIZATION

Methods

The referenced implementation of ACO is serial-only, and our goal was to parallelize it and analyze its performance. Initial profiling showed that the majority of runtime was spent doing tour constructions. The pheromone updates after each set of tour constructions contributed minimally to the overall execution, so we focused on the tour construction phase.

The tour construction phase consisted of a series of for loops without dependencies, performing operations independently for each ant. As such, it was an ideal candidate for parallelization using OpenMP. However, there were a few obstacles. Originally, the program simulated a single step for a single ant in each iteration, but simulating a complete tour for a single ant each iteration is far more conducive to parallelization as it reduces the need for synchronization between steps. To do this, one nested loop needed to be inverted so that the inner loop became the outer loop and vice versa. The second hurdle was random number generation for the initial placement of the ants. We had to modify how the seeds were allocated so that each thread was given a different seed to ensure that random number generation is independent between threads. After this, each ant (or block of ants) could be simulated in a separate parallel thread.

Experimentation

The implementation of ACO that we worked with was CPU-only, although we conjecture that it would be feasible to create a GPU implementation. We used the same test inputs we used with LOGO-Solver. Each problem size was run on various thread counts up to 64 threads on another AWS instance with 64 physical cores. We ran each experiment for several trials and again took the minimum wall time results for each experiment.

The termination condition for this implementation is either an optimal tour length or a time limit. For a fair comparison, we multiplied the optimal tour length by 1.05 to simulate the termination condition used with LOGO-Solver. The time limit was also increased so that execution would not time out. Reasonable times were unachievable with large problem sizes, so input was capped at 1000 cities. On larger instances the heuristic seems to stagnate because more time is required for each incremental improvement.

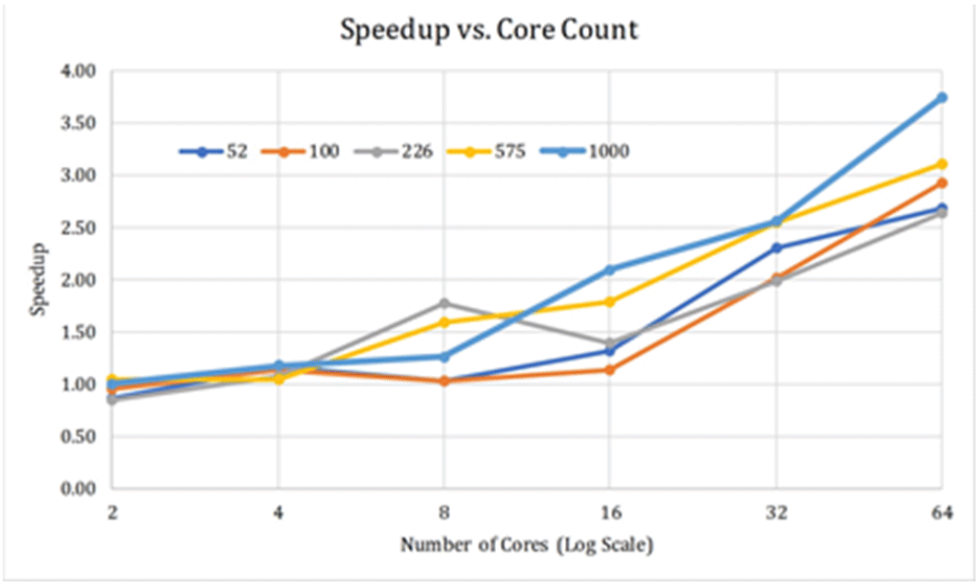

Figure 3: ACO scaling results

Results

Based on our profiling analysis, we parallelized about 75% of the program's execution, so Amdahl's Law estimates the maximum speedup to be around 4x. Initial testing was performed on the previously mentioned 8-core (16-hyperthread) server and speedup seemed to cap at 2x. This implementation is not doing as much heavy computation, so the threading overhead is not masked by larger problem sizes. These results were very underwhelming, so we re-ran the tests on another AWS instance with 64 physical cores. Using the more well-provisioned AWS machine, the results showed strong scaling (a decrease in wall time corresponding to an increase in thread count), and performance approached the theoretical maximum speedup (see Figure 3). Unlike with LOGO-Solver, we observed good scaling with much smaller problem sizes.

COMPARISON TO CONCORDE EXACT SOLVER

The Concorde solver is a modern serial CPU implementation that is currently regarded as one of the best TSP exact solvers available [3]. This implementation uses a branch-and-bound algorithm for large instances, but will choose other, faster algorithms for smaller instances; this guarantees an optimal solution while performing better than a brute-force approach. We found that Concorde is faster for finding exact solutions since the heuristics are relying on randomization for improvement. For example, LOGO-solver takes 30 seconds to get within a tight 0.2 percent error for 575 cities, while Concorde takes 16 seconds to solve it exactly. It appears that Concorde does support parallel execution, but the documentation is sparse and we were able to test only the serial implementation.

It is not really fair 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 approach. Certain problems may require an exact solution while others may just need something that is "close enough." If the user does not need the optimal solution then it is possible to find a satisfactory solution in a fraction of the time with a heuristic approach. The optimal approach depends on the needs of the user, based on how willing they are to trade computation time for solution quality.

CONCLUSIONS

These different parallel implementations of TSP showed good scaling on both CPU and GPU architectures. A 7.5x speedup was observed when executing a parallel implementation of LOGO-Solver on CPU, and an exceptional 26x speedup on the GPU when given large instances. However, the GPU implementation only excels when given instance sizes over 1000 cities because the GPU must have enough computational load to justify the cost of data movement.

We were unable to find other algorithms or heuristics with both CPU and GPU implementations. Instead, we parallelized a serial implementation of the ant colony optimization, a novel approach to solving the traveling salesman problem with some inherently parallel elements. We observed speedups approaching the theoretical maximum based on Amdahl's Law. This implementation has the potential to achieve even higher speedups using a GPU.

Source code is available on Github: `github.com/mastqe/470-gputsp`

REFERENCES

[1] Cormen, T., Leiserson, C., Rivest, R., Stein, C., *Introduction to Algorithms, Third Edition*, Cambridge, MA: The MIT Press, 2009.

[2] Dorigo, M., Stützle, T., *Ant Colony Optimization*, Scituate, MA: Bradford Company, 2004.

[3] Hahsler, M., Hornik, K., TSP infrastructure for the traveling salesperson problem, *Journal of Statistical Software*, 23 (2), 1-21, 2007.

[4] Reinelt, G., TSPLIB - a traveling salesman problem library, *ORSA Journal on Computing*, 3 (4), 376-384, 1991.

[5] Rocki, K., Suda, R., High performance GPU accelerated local optimization in TSP, Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing - Workshops and PhD Forum, 1 (1), 1788-1796, 2013.