Parallel Algorithms for the Traveling Salesperson Problem

Andrew Singh (andrewsi), Ria Pradeep (rpradeep)

May 6, 2020

Summary

We parallelized three different approaches to solving the Traveling Salesperson Problem (TSP) using the OpenMP platform. We implemented one exact algorithm and two approximation algorithms, and we analyzed each algorithm for accuracy, efficiency, and scalability. Our implementation is available at https://github.com/andrewsingh/parallel-tsp.

Background

The traveling salesperson problem is one of the most studied problems in optimization. The problem asks the question: given a list of cities, distances between the cities, and a starting city, what is the shortest path that visits every city and returns to the starting city? This problem can be modeled as an undirected graph problem, where the nodes are the cities and the weight of an edge indicates the distance between its two cities. The paths in consideration are called tours, and the objective is to find the tour of least cost.

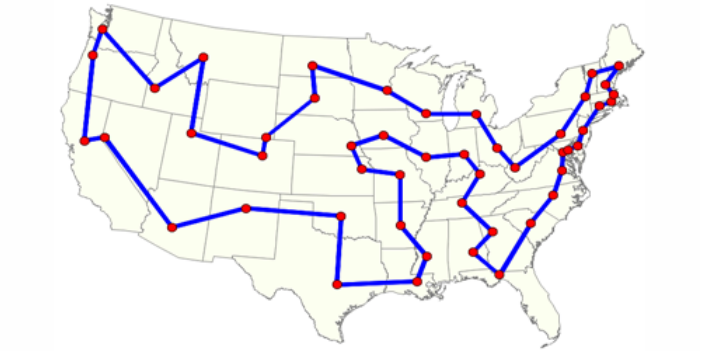

Figure 1: A tour of cities in the United States [5]

The traveling salesperson problem is of interest to us because it is computationally difficult and has many real-world applications. To date, the lower bound on exactly solving a TSP is $\Omega(2^n)$, where n is the number of cities. This exponential bound means that finding an optimal solution for large instances is intractable. This exponential limitation is what makes the problem interesting, as it opens up a variety of different approaches for computing approximate solutions, many of which are parallelizable.

In addition to an interesting solution space, the traveling salesperson problem has many real-world applications. For example, TSP can be applied to problems in transportation and logistics, such as delivery vehicle routing, school bus routing, and airline scheduling.

Specifically, we implemented one exact algorithm, the Held-Karp algorithm, and two approximation algorithms, the Lin-Kernighan heuristic and a genetic algorithm. For all algorithms, the input for an n-city TSP

instance is either an $n \times n$ matrix of distances or a list of n Euclidean coordinates identifying the locations of the n cities. The output is the cost of the minimum cost tour.

Held-Karp Algorithm

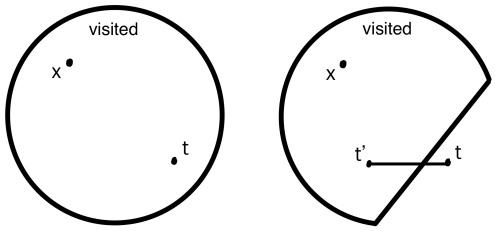

Figure 2: Visualization of the recurrence of the Held-Karp algorithm [6]

app

The Held-Karp algorithm is a dynamic programming approach that computes the optimal solution in $O(n^2 2^n)$ time. The key property that this algorithm utilizes is that every sub-path along a shortest path is itself a shortest path. This algorithm considers subproblems defined by a subset of visited vertices S and an ending vertex t, and calculates the shortest path that begins at the starting vertex x, visits every vertex in S, and ends at vertex t. This approach gives rise to the recurrence

$$C[S,t] = \min_{t' \in S, t' \neq t, t' \neq x} C[S-t,t'] + dist(t',t)$$

The algorithm allows for parallelism amongst the non-dependent subproblems. In a bottom-up approach, dependencies exist between levels of the computation: a subproblem considering a set of size s previously visited vertices is dependent on subproblems considering a set of size $s-1$ previously visited vertices. Within the iteration considering sets of s visited vertices, though, the computation can be parallelized, as none of the subproblems considering the same size subset are dependent on each other.

Lin-Kernighan Heuristic

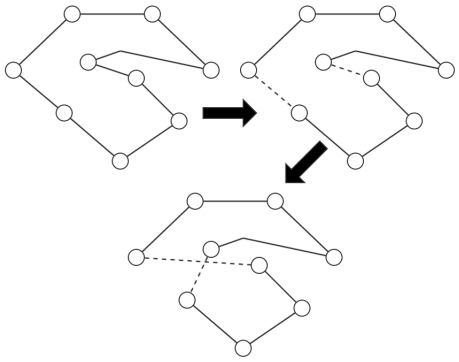

Figure 3: A 2-edge exchange in the Lin-Kernighan heuristic [7]

The Lin-Kernighan heuristic is an approximation method that has been experimentally shown to be one of the most successful approaches to solving the TSP [2]. It is a variation of the 2-opt technique, which involves finding an initial tour and then iteratively replacing 2 edges from the tour with 2 different edges to create a shorter tour. Lin-Kernighan generalizes this method to replace a variable number of edges on each iteration instead of just 2. While a single run of the algorithm is sequential due to the requirement

of sequential edge exchanges, the algorithm's accuracy can be boosted by running it multiple times because the result can be different for different initial tours. The full algorithm runs the Lin-Kernighan heuristic m times, randomizing the initial tour on each run, and chooses the minimum result of the m runs. Exploiting parallelism here is simple and effective: we can have each of the t threads execute $\frac{m}{t}$ runs of the heuristic and then perform a reduction to obtain the final answer.

Genetic Algorithm

A genetic algorithm is a technique inspired by natural selection, and is a popular metaheuristic for solving hard optimization problems. This method involves representing each solution as a potential individual and simulating selecting the "fittest" solution through multiple rounds of selection. A genetic algorithm for solving TSP would involve enumerating different tours and using a "natural selection" process to choose which individuals cross-over to create the next generation, thus slowly eliminating the less fit (longer) tours. For our purposes, the individuals are crossed over based on a Greedy Crossover method [4], where a random starting city is selected, and the shortest path from that city to some other city in each individual is followed. If neither individual has a valid next city to visit, one is randomly selected, and this process is repeated until a full tour is complete. Finally, the individuals are mutated independently randomly. This process repeats until convergence, and returns the fittest individual in the remaining population.

There are dependencies between generations, and within a generation there is a dependency between selecting parents before crossing them over. But a large amount of the computation within a generation can be parallelized, including selecting parents and crossing them over. The generation of the initial population can also be done in parallel. We can assign individuals to threads, so for an instance with n vertices and running with t threads, each thread is responsible for the computation of $\frac{n}{t}$ individuals or parent pairs.

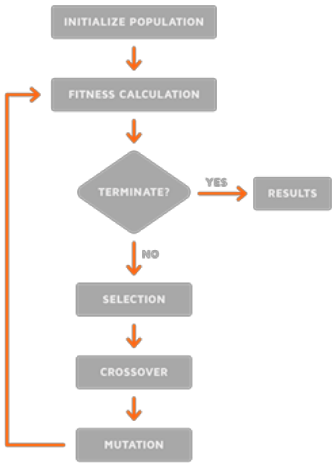

Figure 4: Flowchart of a generic Genetic Algorithm [8]

Approach and Implementation

We implemented these three algorithms in C++ and used the OpenMP platform for parallelization. We initially did our testing on the GHC machines, but when it came time to conduct our benchmarks we noticed that the GHC machines were noticeably inconsistent. We implemented the functionality to run our jobs on the latedays machines, allowing us to have consistent benchmarks and additionally evaluate our algorithms on more threads (12 vs. 8). We maintained the same language and parallelization platform for all three algorithms to evaluate them in the same context.

Held-Karp Algorithm

The Held-Karp algorithm is a dynamic programming approach to computing an exact solution to a TSP instance. We first implemented a sequential version from scratch, and then parallelized it with OpenMP. The parallelization presented several algorithmic challenges. One of the parameters to the memoized DP array of subproblems is a subset of vertices. This parameter is what gives Held-Karp its exponential runtime, as there are 2^n possible subsets of n vertices. In our implementation, the subset is encoded as an n-bit integer where bit $i = 1$ if vertex i is included in the subset and $i = 0$ otherwise. The algorithm involves looping through each possible subset size from 2 to $n - 1$ and for each size s, looping through all possible subsets of size s. When parallelizing the algorithm, we cannot parallelize across the outer loop of subset sizes because the algorithm is bottom-up, using the computed solutions for the previous sizes to compute solutions for the current size. Therefore we needed to parallelize the inner loop that iterates through all subsets of size s. This is equivalent to parallelizing every level of the dependency graph.

For the sequential version, we used Gosper's Hack to update the subset encoding at the end of each iteration, but this introduces a dependency across the iterations of the subset loop: the subset encoding in the current iteration is used to compute the encoding in the next iteration. To get around this dependency for the parallel version, we precomputed all of the set encodings before the main DP loop. This removed the dependency on the set encoding in the inner DP loop and allowed us to statically parallelize it. We were even able to parallelize the precomputation step by parallelizing the outer loop across different set sizes; we used dynamic scheduling here because the i^{th} iteration runs in time proportional to $\binom{n}{i}$, making for a very unbalanced workload across iterations.

Lin-Kernighan Heuristic

The Lin-Kernighan heuristic involves iteratively modifying the current tour of the graph such that each iteration yields a shorter tour. Since different initial tours can yield different results, we boosted the accuracy of this algorithm by running it multiple times with random initial tours and outputting the minimum cost tour among the runs. This random restart method is a common technique in optimization algorithms where the result depends on the starting state.

We referred to the code in [3] for our sequential implementation of the heuristic, adding the random restart functionality ourselves. We additionally had to decide how many restarts the algorithm should perform: more restarts gave a solution closer to the optimal, but there were diminishing returns and eventually a hard bound that was sometimes at the optimal but often near the optimal. We tested the algorithm on each instance in our test suite at different amounts of random restarts, analyzing the tradeoff between number of restarts and accuracy as the size of instances increased. We devised the following formula for number of random restarts r as a function of the instance size n: $r = 1721n^{-0.74}$. This formula strikes a good balance between running time and accuracy, and the algorithm scales well to larger instances as shown in the results.

The parallelization itself was quite simple: since each of the m runs was roughly the same amount of work, we statically assigned each of the t threads to perform $\frac{m}{t}$ runs of the heuristic, and then perform a min reduction on each thread's result. One implementation issue that arose was how to handle randomization in a way that was thread-safe but still deterministic such that if we were to benchmark the overall algorithm many times with the exact same parameters (TSP instance, number of threads), we would get the same result each time. This condition was necessary to ensure consistency when evaluating the algorithm for accuracy. We observed that the i^{th} run of the heuristic is only run by a single thread, so we could use the iteration number as the seed for our generator. This gives us the benefits of randomness by varying the initialization of different runs within a single benchmark, while still ensuring that the results are consistent across multiple benchmarks. Furthermore, since each random generator is local to an individual run of the heuristic, which is local to a single thread, this method is thread-safe as well.

Genetic Algorithm

The genetic algorithm involves generating an initial population and repeatedly, and randomly, crossing over individuals to form a "fitter" next generation. For the TSP problem, an individual is a tour, represented as an array of cities to visit in order, as well as the cost of that tour. The population is represented as an array of individuals, from which pairs of individuals are randomly selected as parents, which are stored in another array. Because the population and parents are represented as arrays, we can modify and update individuals in parallel. These were initially represented as vectors, but changed to arrays to enable more parallelism and avoid the need for atomic updates.

The sequential algorithm first generates the initial population, creating n tours by randomly selecting unvisited cities until a tour is complete. Then, each individual is ranked by its tour length, and given a probability of being selected for crossover based on its rank, such that the shorter tours are more likely to be selected. $n - 1$ pairs of individuals are selected to be parents using roulette selection based on the probabilities assigned, so our population size decreases by 1 each iteration. Then the parents are crossed over and mutated to form the next generation. This process repeats until convergence, which is defined to be when all the individuals have the same tour or there is one individual left. The algorithm, in the worst case, converges after a linear number of iterations. We could also decrease our population size by a factor of 2 with each generation to converge in a logarithmic number of iterations, but chose to go with a linear decrease to get a

better approximation of the optimal tour.

Due to dependencies between generations, there is no parallelization across iterations. But, the dependincies within an iteration is limited to selecting the parents before crossing them over, so the computation before and after this dependency can be parallelized. Each individual or pair of parents is operated on by a single thread. When generating the initial population, each thread generates an individual and stores it in the corresponding position in the array. The threads are dynamically assigned to individuals to generate as this yields the best results. The work of computing the unvisited vertices depends on what vertices are randomly visited, which varies across threads. Similarly, when selecting parents, threads are statically assigned to generate a pair of parents and store it in the array, since the computation takes approximately the same amount of time for each pair. Finally, when crossing over the selected parents, each thread computes the new individual's tour and stores it in the array. The work here is dynamically assigned to threads. Depending on the parent tours, it may take longer to compute the new tour, specifically in the case of finding a random, unvisited vertex. We experimented with different scheduling techniques before settling on this.

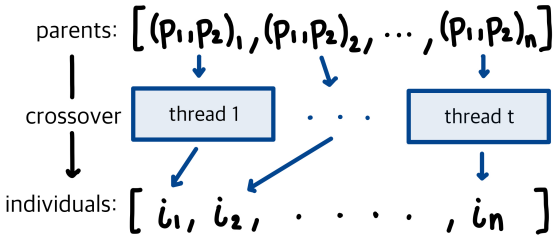

Figure 5: An example of parallelization in the crossover step (not necesserily this mapping of work to threads)

The genetic algorithm, like the Lin-Kernighan heuristic, is based on randomness, which brings the same consistency issue when benchmarking our results. To make the results repeatable, we implemented a similar approach as we did for the Lin-Kernighan heuristic: varying the initialization seeds such that the randomness is repeatable, but using enough randomness to boost accuracy. Each of the three parallelized steps has a unique seed, and each thread creates its own random number generator based on the seed for the computation for that specific individual or pair of parents. The seeds are combinations of the thread id and the individual or parent index. We experimented with different combinations for the seed, but decided on this since it yielded the best results on average.

Results

We conducted experiments to evaluate each algorithm for accuracy, efficiency, and scalability. Our test suite was a set of 9 TSP instances obtained from TSPLIB [9], a repository of TSP instances widely used in research for benchmarking TSP algorithms. Each n-city TSP instance in our test suite is represented as either an $n \times n$ matrix of distances or a list of n Euclidean coordinates identifying the locations of the n cities. In the plots below, each instance is given by its name from the TSPLIB website, which consists of several letters followed by a number. This number is the size of the instance (number of vertices).

Accuracy

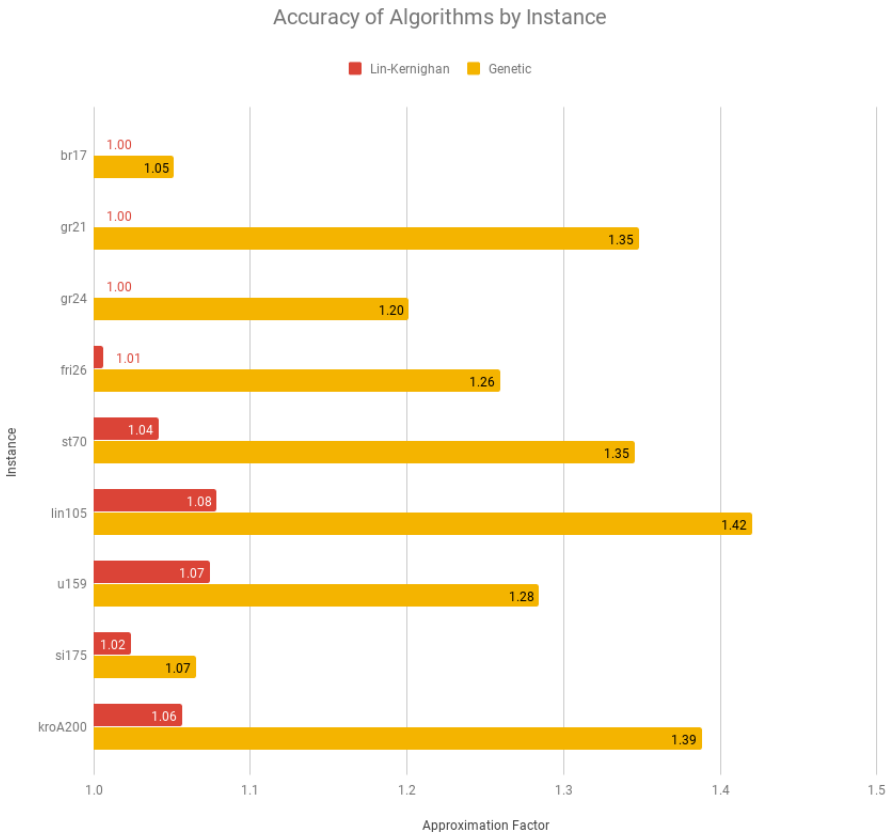

Figure 6: Approximation factor of the two approximation algorithms for each instance

To evaluate accuracy, we recorded each algorithm's output for each of the instances in our test suite, and then computed the approximation factor to determine how close each result was to the optimal. The benchmarks were run on the latedays machines using 12 threads. The approximation factor is simply the ratio of the algorithm's tour length to the optimal tour length. Because the Held-Karp algorithm is an exact algorithm, it always outputs the optimal solution. Therefore we restrict our analysis of accuracy to the two approximation algorithms. Our results are shown in Figure 6. We see that the Lin-Kernighan heuristic manages to find the optimal solution for the smaller instances, but even as the instances get larger, the algorithm stays near the optimal. Among all the instances we tested, up to 200 vertices, Lin-Kernighan outputs a result within 10% of the optimal. The genetic algorithm doesn't find the optimal solution for these instances, but all of its solutions are within 50% of the optimal.

Efficiency

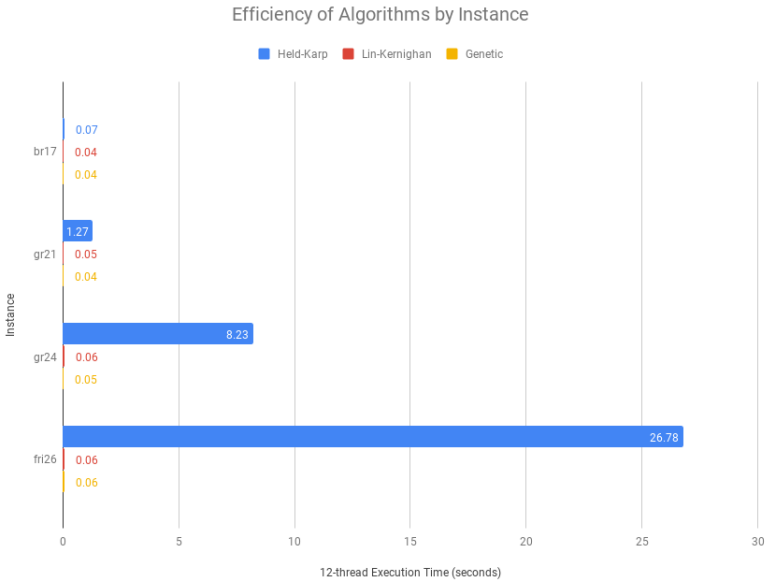

Figure 7: Execution time of all three algorithms running at 12 threads for each small instance

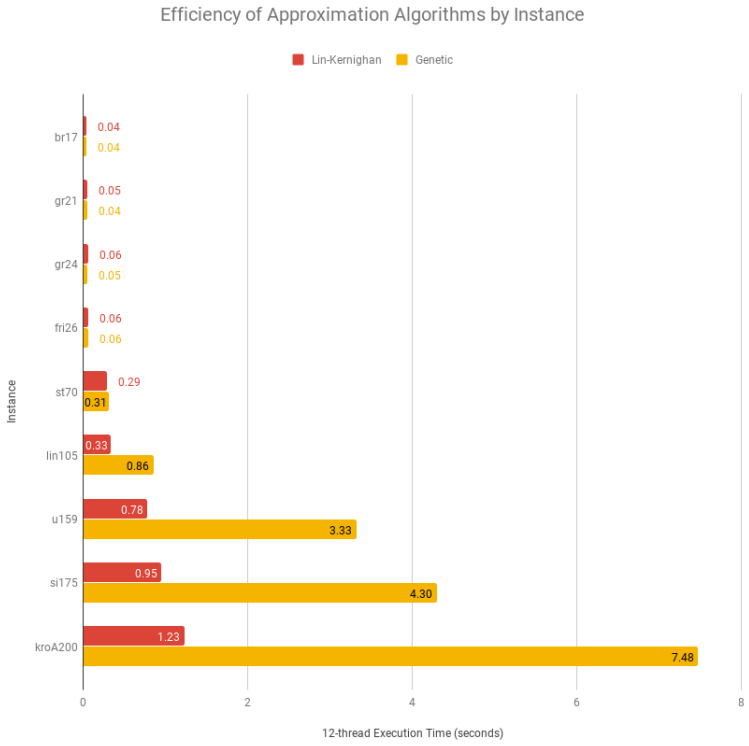

Figure 8: Execution time of the two approximation algorithms running at 12 threads for each instance

To compare the efficiency of our three algorithms, we recorded the execution time (measured as wall-clock time) of each algorithm on each instance, run with 12 threads. We ran these benchmarks on the latedays

machines, repeating each entry of (instance, algorithm, thread count) 5 times and taking the average execution time of the runs. Due to the exponential time and space complexity of the Held-Karp algorithm, it could only be evaluated on the four small instances in our test suite. Figure 7 compares the runtime of all three algorithms, and therefore is just of these four small instances. As expected, Held-Karp takes by far the longest time on all four instances, as its runtime is exponential in the instance size. The Lin-Kernighan heuristic and genetic algorithm have approximately the same runtime for these small instances. Figure 8 compares the runtime of the two approximation algorithms, and is able to use all instances in our test suite. For small instances, the two algorithms are roughly the same in efficiency, but as the input size increases, we start to observe the Lin-Kernighan heuristic's better scaling compared to the genetic algorithm. Observing this difference, we created an additional plot, shown in Figure 12, to explicitly evaluate how our algorithms scale with instance size.

Scalability

To evaluate the scalability of our three algorithms, we measured the execution time (measured as wall-clock time) of each algorithm on each instance in our test suite at t threads for $t \in [1, 2, 4, 8, 12]$. We ran these benchmarks on the latedays machines, repeating each entry of (instance, algorithm, thread count) 5 times and taking the average execution time of the runs. We then computed the corresponding speedup for each entry and generated the following speedup plots. We measured speedup as the ratio of execution time of our baseline sequential algorithm to the execution time of the algorithm with t threads.

Note that since the parallel implementation of the Held-Karp algorithm differs non-trivially from the sequential version (precomputing and storing the set encodings rather than updating them on-the-fly), we use the sequential version for running the algorithm at 1 thread. Indeed, we benchmarked both the sequential version and the parallel version at 1 thread, and observed the parallel at 1 thread was slightly slower. For the two approximation algorithms, the parallel version was virtually the same as the ideal sequential version, which we confirmed by comparing the runtimes of the two. For these two algorithms, we maintain a single implementation that we run for 1 thread as well as multiple threads.

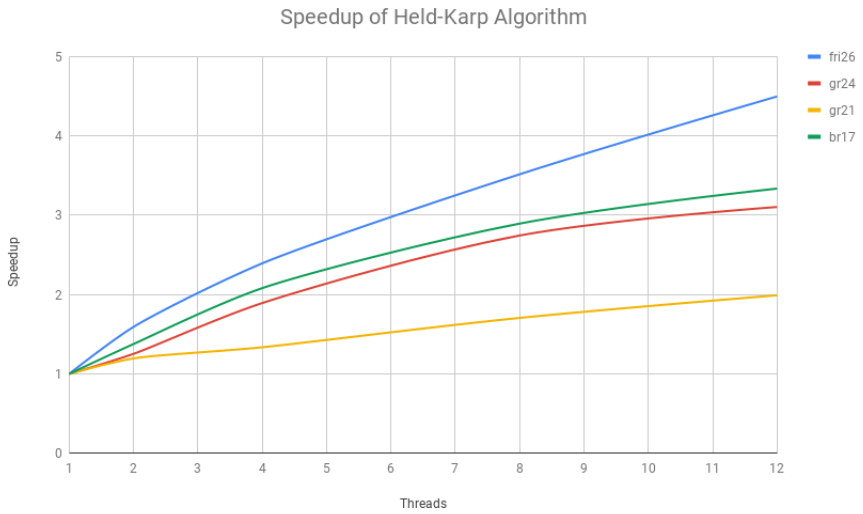

Figure 9: Scaling of the Held-Karp algorithm from 1 to 12 threads

Because the Held-Karp algorithm is exponential in runtime and space, we can evaluate it for only the four small instances in our test suite. Figure 9 plots the resulting speedup for these four instances as the algorithm scales from 1 to 12 threads. When selecting a large instance for Held-Karp, the space complexity was actually the limiting factor we ran into rather than the time complexity: the 26-city instance requires nearly 8 GB of memory to run. Almost all of this space is used for storing the subproblems, since the algorithm computes $O(n2^n)$ of them. In the parallel implementation, each thread accesses this array, and the access patterns

are not spatially or temporally local. Due to OpenMP's shared memory model, however, the large memory requirement does not appear to hinder speedup. In fact, we see that the 26-city instance, which involves much more computation and memory than the other instances, has significantly better speedup. Because the smaller instances involve exponentially less computation than the 26-city instance, the overhead involved with parallelism is a larger fraction of the total work. Specifically, this overhead is in the precomputation when dynamically assigning each thread to compute the set encodings for a size $2 \leq s < n$, and in the main loop when statically assigning each thread a portion of the size s subsets.

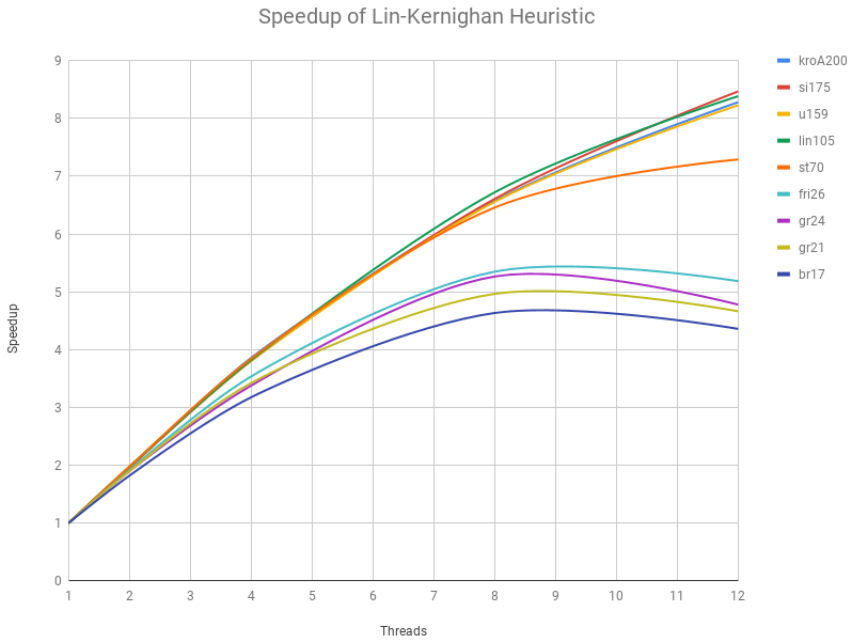

Figure 10: Scaling of the Lin-Kernighan heuristic from 1 to 12 threads

Since our parallel implementation of the Lin-Kernighan heuristic simply parallelizes over independent runs of the algorithm and then performs a reduction across threads, we expected to observe a high degree of parallelism. Figure 10 plots the speedup for each instance in our test suite as we scale from 1 to 12 threads. We do in fact get near-linear speedup on the larger instances from 1 to 8 threads, and then slightly less from 8 to 12 threads. On the four small instances, the speedup curves are noticeably shallower. The speedup levels off at around 8 threads and then actually starts to decrease. This sharp difference between the speedup curves of the smaller instances and the larger instances is expected. Because there is much less total computation to be done with the smaller instances, the overhead involved with parallelizing the algorithm is a much larger fraction of the total work. Specifically, this overhead consists of statically assigning each thread its chunk of $\frac{m}{t}$ runs to execute, and once all threads have completed their chunk of runs, performing the reduction of each thread's result. Beyond 8 threads, the overhead is so great that it even starts to hinder performance on the small instances.

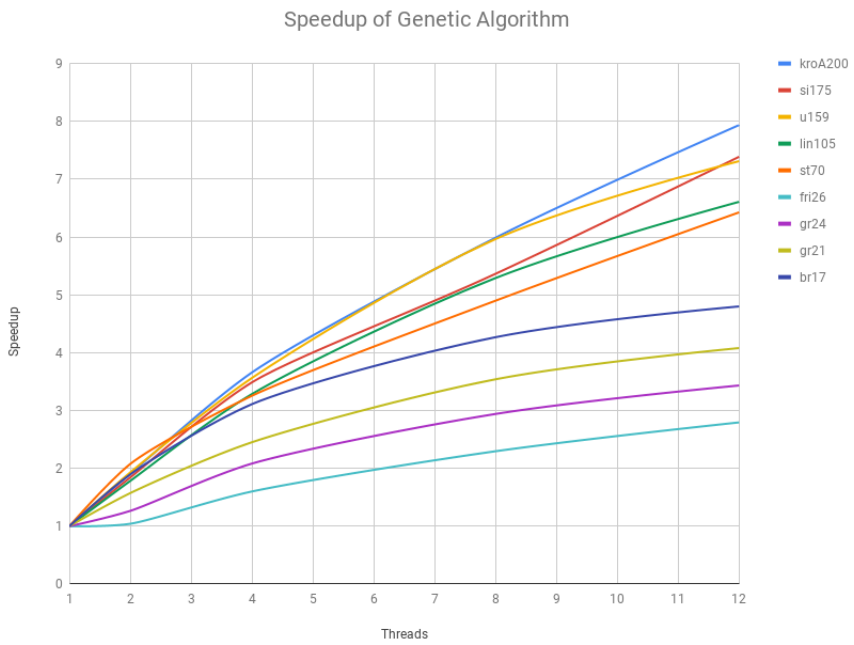

Figure 11: Scaling of the Genetic algorithm from 1 to 12 threads

The parallel implementation of the genetic algorithm parallelizes all the computation within a iteration, so we also expected to observe a significant amount of parallelism. Figure 11 plots the speedup of running each instance in the test suite as we scale from 1 to 12 threads. For larger input sizes, we observe near-linear speedup relative to the number of threads, due to the parallelism in computation at every step between dependencies. The speedup also seems to vary more between each instance, compared to other algorithms, implying the parallelism of this algorithm is more dependent on input size (which it is, as population size is bounded by input size). The performance for the smallest four instances does not appear to scale as well as the other, larger instances, due to the overhead of creating multiple threads and statically assigning them to work (where applicable) compared to the small problem size.

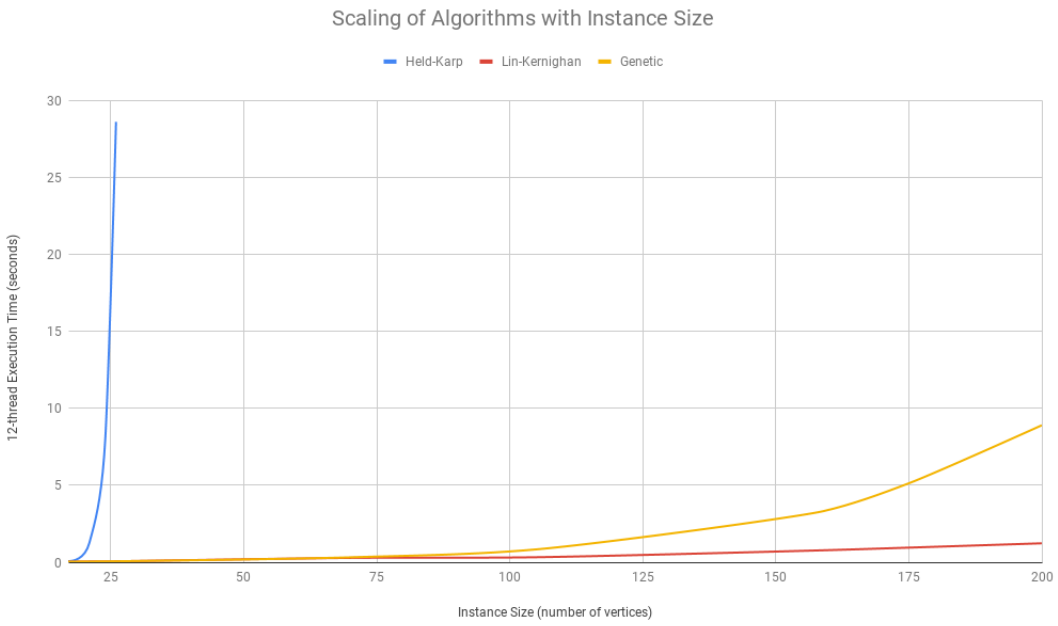

Figure 12: Scaling of each algorithm's 12-thread execution time with the instance size

We also plotted how the runtime of our algorithms scales with the instance size. We used the execution times of the algorithms running at 12 threads from our earlier benchmarks. From Figure 12 we clearly see the exponential increase in execution time as input size increases for the Held-Karp algorithm. The approximation algorithms both have an polynomial running time, so the scaling is much less extreme. Observing the instances larger than our 70-city instance, we can see that the Lin-Kernighan heuristic scales with the instance size better than the genetic algorithm.

Conclusions and Further Improvements

From our analysis, we can determine that beyond small ($n \leq 26$) instances, solving the traveling salesperson problem exactly quickly becomes infeasible, both in time and space requirements. We showed that while parallelism can make an exact algorithm run faster, the algorithm is still exponential and cannot scale to large instances. We explored several approximation algorithms to overcome this limitation, and showed that there are approximation algorithms that can get within 10% of the optimal on much larger ($n \leq 200$) instances and that scale well with the instance size. Futhermore, these approximation algorithms can take advantage of parallel hardware to further speed up their execution. Moving beyond this project, there are some further areas that would be interesting to explore for these algorithms.

Held-Karp Algorithm

The implementation we analyzed was a bottom-up dynamic programming approach, which starts from the smallest possible subproblems, computes the results to all possible subproblems, then finds the result for our specific problem. We also attempted to implement a version of this algorithm with a top-down dynamic programming approach, which involves starting from the problem we want to solve then only solving its dependencies. The implementation involves a recursive function that calls the subproblems the current problem is dependent on, and memoizing the results of subproblems. We included parallelism within the recursive function by attempting to parallelize calls to subproblems. Testing this implementation yielded no speedup, and after more consideration we realized a general top-down dynamic programming approach is inherently sequential, because there is an implicit ordering to the recursive calls. Additionally, multiple threads could be working on the same subproblem at the same time, since there is no coordination between

threads until the solution is stored in the memo table. An idea to resolve this, which we did not have time to implement, is to use synchronization variables for each subproblem to indicate if a thread is working on that problem. This would ensure each subproblem is computed once.

Typically, a benefit of top-down dynamic programming is doing less work overall, since we only compute subproblems we need to. For this algorithm, however, we end up considering all possible subproblems anyways, so we don't get this benefit. Additionally a top-down approach is more difficult to implement, involving more synchronization between threads. So overall, parallelizing a bottom-up dynamic programming approach for this algorithm is more feasible and yields reasonably decent speedup.

Lin-Kernighan Heuristic

The starting tour of the heuristic affects its result, which is why we implemented random restarts to boost accuracy. Our implementation chooses the starting tour uniformly at random; it would be interesting to determine if a certain subset of initial tours yielded better results than other tours. If this was the case, the accuracy could potentially be improved by randomizing over this subset rather than over all possible tours.

In addition, there are multiple variants of the Lin-Kernighan algorithm, each using some form of the k-opt technique of iteratively exchanging sets of k edges in the tour. It would be interesting to explore other implementations of this heuristic and determine if there are ways to make the core algorithm highly parallel, as opposed to parallelizing over multiple runs of the algorithm.

Genetic Algorithm

The data structures of the genetic algorithm is entirely array based, but we used a general parallelization method to improve the algorithm. It would be interesting to also explore using a domain-specific language for lists, such as Listz or a MapReduce model, such as Hadoop, since population is essentially a list of individuals. Operations for generating individuals or parents can be done with a map operation and testing for convergence can be done with a map and a reduce. The approach taken in the Er and Erdogan paper [4], uses Hadoop to parallelize a genetic algorithm. They note and improvement in runtime and accuracy from a sequential baseline when running a parallelized version. Comparing a MapReduce style parallelization to an OpenMP parallelization could produce interesting results, since using a domain-specific language could yield even better performance than the current speedup.

Additionally, as the input size increases, there are more individuals that could be evaluated or parents that could be crossed over in parallel. Mapping this computation onto a GPU would enable more of these individuals to be evaluated at once by mapping each thread to a single individual. This would lead to more parallelism overall, and therefore a faster runtime, and might be a better machine choice for this algorithm, when considering a genetic algorithm independently. Comparing the speedup on a CPU versus a GPU for genetic algorithms is another direction to explore.

Division of Work

Item	**Completed By**
Held-Karp Algorithm - Bottom-Up Approach	Andrew
Held-Karp Algorihtm - Top-Down Approach	Ria
Lin-Kernighan Heuristic	Andrew
Genetic Algorithm	Ria
Testing, Benchmarking, and Analysis	Andrew & Ria

References

[1] Wikipedia contributors. (2020, February 12). Held–Karp algorithm. In Wikipedia, The Free Encyclopedia. Retrieved 19:01, April 12, 2020, from https://en.wikipedia.org/w/index.php?title=Held%E2%80%93Karp_algorithm&oldid=940373060

[2] Helsgaun, Keld. (2000). An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic. European Journal of Operational Research. 126. 106-130. 10.1016/S0377-2217(99)00284-2.

[3] https://github.com/lingz/LK-Heuristic

[4] Er, Harun Raşit & Erdogan, Nadia. (2014). Parallel Genetic Algorithm to Solve Traveling Salesman Problem on MapReduce Framework using Hadoop Cluster. JSCSE. 10.7321/jscse.v3.n3.57.

[5] https://physics.aps.org/assets/a38de7c6-00ac-45fb-9bcf-3b3e14a72b41/es32_1.png

[6] https://www.cs.cmu.edu/~15451-s20/lectures/lec09-dp2.pdf

[7] Lady-shirakawa at English Wikipedia / CC BY-SA (https://creativecommons.org/licenses/by-sa/3.0)

[8] https://www.gridgain.com/docs/latest/developers-guide/machine-learning/genetic-alg

[9] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/