Assignment 2: Traveling Salesman

Adrian Kwok (#200136359)
Benjamin Saunders(#301111447)

CMPT 885 – Professor Shriraman
Due Date: July 8th, 2011

1. Introduction

For our assignment, we implemented two heuristics approaches to the Travelling Salesman Problem (TSP): a genetic algorithm, modeled after natural selection, and one exploiting the path finding behaviour of ants. We decided to forgo traditional graph-based MST approximation algorithms such as Kruskal's and Prim's since we felt that the open-endedness of the assignment invited exploration of the more interesting outside-of-the-box and less well understood algorithms in current research. Unlike MST-based algorithms, however, genetic and ant colony heuristics are unbounded in its solution quality; fortunately, they have been found to frequently deliver near-optimal solutions for the TSP, in very reasonable runtimes[2][7]. Furthermore, a notable number of recent papers have been published which greatly extend the capabilities and performance of both heuristics[3][4][7][8][9]. However, our primary emphasis for this assignment was to focus on their basic underlying frameworks – while the algorithms themselves are embarrassingly easy to parallelize, just understanding and implementing them correctly proved to be a sufficient challenge in itself.

2.1. Genetic Algorithm

The Genetic Algorithm (GA) for the TSP draws heavily from evolution: each possible TSP cycle is encoded as a numerical string (analogous to genes in the natural world) wherein each city is represented by an index, a series thereof denoting a path, with the paths defining the initial population selected at random. Through multiple *generations*, this population is 'evolved' towards an optimal solutions of the TSP. During a generation, the most fit half of the population is chosen to reproduce, analogous to survival of the fittest, with the resulting recombinations taking the place of the weaker half of the population in the next generation. The best TSP cycle in a given generation is simply the individual that is deemed to be the most fit among the whole population.

The complexity of the algorithm lies in how the most 'fit' half of the population is chosen, and how two cycles mate – defined as crossing over in the literature – to create children for the next generation. We will now describe our implementation in detail. Steps with an asterisk were parallelized:

1. ***Initialization:** we first need to seed the algorithm with an initial population – this is done by randomly choosing cycles in the TSP graph. Since the input is always a complete graph, generating such cycles can be trivially done by shuffling the sequence of numbers from 1 to *numCities*. This was parallelized with a simple *pragma omp for* directive across the population size.

   We also compute the total distance for these random cycles; since the distances between all adjacent cities have been pre computed into a distance matrix, calculating the actual distance for a cycle requires just iterating through all of its cities. Because this can be done independently across all cycles in the population, we again parallelized this with a simple *pragma omp for* directive.

2. **Choosing the most fit half of the population for reproduction:** we initially implemented the naive approach for this step as outlined in the assignment description page[5], where we would simply sort the population and consider only the top half of the sorted list, but we found that it often lead to premature solution stagnation – just because a cycle is deemed 'unfit' in one generation does not necessarily mean that its offspring will also be unfit. We decided on a more robust method, tournament selection[4], where a select number of candidates (tournament size) would be chosen from the population, and each assigned a probability based on its rank in the tournament. The winner of the tournament would be added to the mating pool, and we would repeat this until the mating pool is full. This allows for more diversity in the overall population, and is not much more inefficient than the naive approach.

At this step, we can also find the best cycle computed by the algorithm thus far by scanning through the entire population and retrieving the cycle with the lowest total distance, since by this point all distances will have been computed already.

3. ***Mating consecutive pairs of the chosen population:** for this step, we populate the second half of the next generation's population by mating every consecutive pair of cycles in the chosen population. We utilized Grefenstette's greedy crossover algorithm[2], which initializes the first city in a child's cycle to be one of the parents' first city, and then, for each city *x* in the child's cycle, we examine the next city connected to *x* in both parents' cycles, labeled *a* and *b*. We then do one of the following:

   a. We choose the closer of the two cities *a* and *b*, and if it has not been used yet in the child's cycle, we extend the cycle with it.
   b. Otherwise, if the closer of the two cities already exists in the child's cycle, we choose the other one that has not been used yet in the child's cycle, and we extend the cycle with it.
   c. Otherwise, if both cities already exist in the child's cycle, we randomly choose a city that has not been used yet and extend the cycle with it.

   Since the generation of each child is independent from one to the next, we can trivially parallelize this step by assigning the creation of each child to a one thread. This, again, was done with a simple *pragma omp for* directive across all consecutive parent pairs.

4. ***Mutating some children:** to ensure that the algorithm does not prematurely stagnate, we randomly mutate some of the newly created children. With a user-defined mutation likelihood *m*, five random city pairs in a child's cycle are swapped. Since the input is guaranteed to be a complete graph, this mutation step is a trivial operation, and we again parallelize it by using a *pragma omp for* directive across all newly created children.

5. ***Calculating the total distance for each child cycle:** this step is a repeat of the initialization step, but assigned only to the newly created children. This step, combined with the crossover step, proved to be the majority of the computation for the overall genetic algorithm.

6. **Generating the next population:** this step joins the parent and child sub-pools together and sets it as the next population. We used two `copy()` methods to accomplish this; while it could have been parallelized with a simple *pragma omp for*, the amount of total work done is minimal and does not justify the possible overhead introduced by parallelization.

We then repeat steps 2 to 6 until the user specifies that the returned solution is adequate, or if a timeout threshold has been reached.

2.2. Ant Colony Optimization

The Ant Colony Optimization (ACO) heuristic for the TSP utilizes a swarm of 'ants', each of which independently and greedily attempts to find the best cycle in the graph – best in terms of minimal distance and maximum pheromone. After finding such a cycle, an ant then retraces its steps and deposits pheromone ("scent trails") at each edge in the path inversely proportional to the total length of the path. The best global cycle is then defined to be one with the highest amount of pheromone deposited along the path, easily found via a simple greedy algorithm that only considers edges with the strongest pheromone. The ants repeat this process, refining the best path during every iteration by strengthening the pheromone in both edges that were visited frequently by other ants as well as overall paths with short distances.

Due to the nature of the heuristic, the algorithm is very easily parallelizable since each ant works independently of one another. We will now describe our implementation in detail – steps with an asterisk were parallelized:

1. ***Ant-to-city assignment:** we set the number of ants to be equal to the number of cities, and assign one ant to each city. We experimented with using a fixed amount of ants, randomly and uniformly distributed across all cities, but this lead to path stagnation in the basic heuristic. We parallelized this step using a simple *#pragma omp for* directive across all ants. This is a superficial parallelization though, since for most inputs this is a menial task.

2. ***Ant movement:** each ant is then told to greedily find the best possible path which visits each city exactly once. This is done by telling each ant to move to the next 'best' city, *numCities-1* times. When deciding which city to go to from one city to the next, it computes a probability for each neighboring city, which is determined by both the pheromone in a connecting edge as well as the "visibility" of the city from the ant's current location ("visibility" is defined to be the distance to the city normalized by the distance to all connected cities). More formally, the probability that an ant will move from city x to a candidate city y is defined in [6]:

$$prob(x,y) = \frac{pherom(x,y)^{\alpha} \cdot dist(x,y)^{\beta}}{\sum_{neighbors\ k} pherom(x,k)^{\alpha} \cdot dist(x,k)^{\beta}}$$

   Note that all of the distances between cities have been pre-computed into a distance matrix, so "calculating" the distance between two cities is an inexpensive lookup. An ant then chooses its next city at random based on these computed probability distributions.

   As each ant is able to compute such paths independently of one another, we easily parallelized this by using a *pragma omp for* directive, assigning each ant to a thread and having it move one step. Ants wait until all other ants have moved their one step before proceeding again. When an ant has conducted *numCities-1* steps, it will have found a cycle leading back to its original city – so when all ants have moved *numCities-1* steps, they will have completed their work.

3. ***Depositing and evaporating ant pheromone:** now that each ant has found a cycle that it has deemed to be the 'best', it calculates the total distance of the path and retraces it, depositing pheromone along all edges in the path. We utilized a pheromone matrix to store these edges, which is similar to the distance matrix described earlier but with pheromone amounts recorded between cities instead of distances. The amount of pheromone an ant deposits at a given edge between cities x and y in its path is calculated via the following equation:

$$pherom(x,y) = pherom(x,y) + p \cdot \frac{Q}{path\_length}$$

In general terms, an ant deposits a globally constant amount Q divided by the total length of the path – this ensures that edges in paths that are short receive large deposits of pheromones, while edges in paths that are long receive small deposits. The deposited amount is multiplied by p, a user-specified pheromone intensification rate.

We parallelized this step by assigning one ant to each thread, again using a *pragma omp for* directive. However, this step was more involved than in previous cases: if two ants deposit its pheromone on the same edge in parallel, there is a high possibility that there will be lost updates (i.e. both ant A and ant B read the pheromone on the edge, ant A updates the amount of pheromone, and ant B updates the amount of pheromone afterwards, causing ant A's update to be lost). This was resolved by placing an ant's pheromone matrix update (read, add, and write) in an *pragma omp critical* section.

It is necessary to note, however, that this solution reduces parallelism more so than necessary as it prevents two ants from updating the pheromone in *any* two edges simultaneously, and *not* just updates the *same* edge. OpenMP allows for the *pragma omp atomic* directive, which would work well for serializing the write to the matrix, but still results in race conditions when reading and adding to the edge. A custom lock-based matrix data structure would have solved this problem effectively, but given that the *omp critical* solution's reduction in parallelism is negligible compared to the overall algorithm's runtime, we decided to forgo this route.

As a final step, since we have completed an iteration in the algorithm, we evaporate the pheromone in all edges of the graph by a user-defined amount. This reduces the likelihood that an ant will consider poor paths that not many other ants have taken. Since evaporating each cell in the pheromone matrix can be done independently, it was easily parallelized in OpenMP using an *omp for*.

4. **Finding the best path in the current iteration:** after completing all of the previous steps, we now have a graph with differing pheromone deposits on all edges. We can find the "best" cycle in the graph by starting at any city, and then following edges with the largest deposit of pheromone to the next unvisited city. This can only be done in a serial manner, from one city to the next, and thus this step was not parallelized – in any case, the amount of work required is minimal and linear in the number of cities.

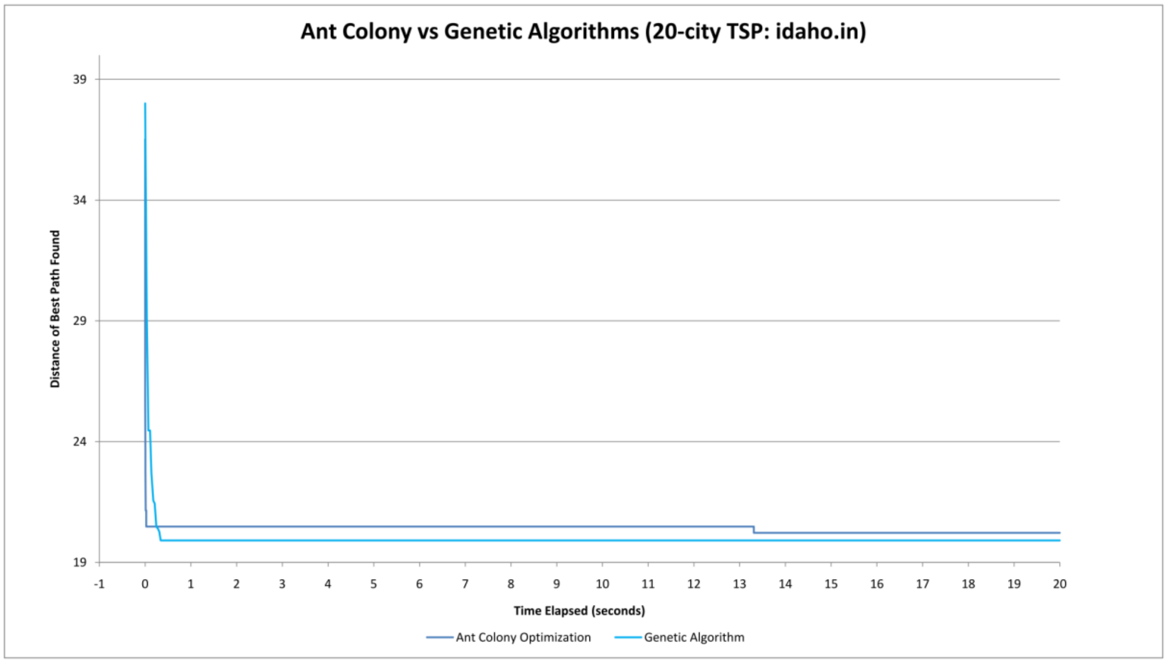
We then repeat these steps until the user specifies that the returned solution is adequate, or if a timeout threshold has been reached.

2.3. Genetic & Ant Colony Algorithm Comparison

To compare our two algorithms, we ran them on *miga.cs.sfu.ca*, a 12-core AMD Opteron-based server. Both programs were implemented using OpenMP and C++, and were run with 12 concurrent threads while gathering results.

For the genetic algorithm, the initial population was set as 50,000, the mutation likelihood as 10%, the tournament size 5, and the tournament probability as 60%. These parameters were set according to findings from [2] and [4], as well as through our own testing and experimentation on a variety of input sizes. For the ant colony optimization algorithm, we assigned one ant to each city, and utilized similar parameters as from [6] and [9]: α=0.6 (pheromone weight when choosing next city), β=1 (distance weight when choosing next city), p=0.6 (pheromone intensification/evaporation rate), Q=10 (global pheromone deposit constant), and base and max pheromone as 10 and 1000 respectively. The alpha parameter of 0.6 was chosen through self experimentation as it seemed to lessen the likelihood of premature path stagnation due to pheromone strength, while not jeopardizing the *quality* of the results.

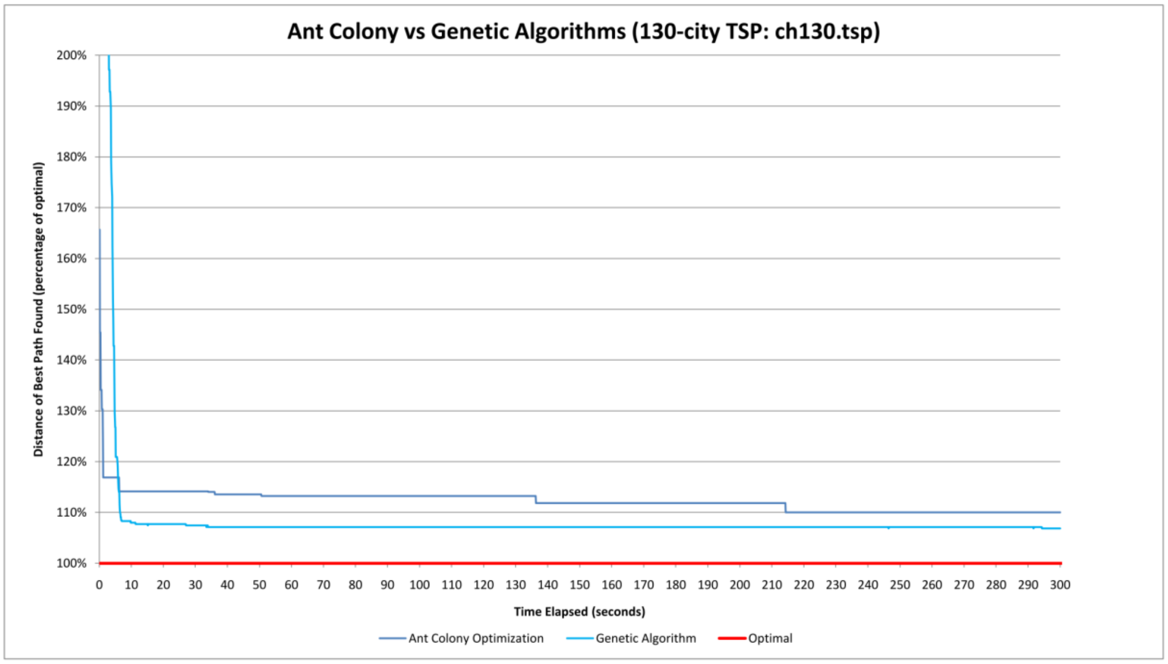
We ran a multitude of tests across varying input sizes, and the most interesting results are shown in figures 1-3. We attempted to equally distribute the type of inputs used, varying from the 20-city TSP given to us in the assignment description, to Churritz's 130-city and Padberg & Rinaldi's 1002-city problems from TSPLib[9]. Optimal results for the latter two were provided, so we were able to normalise our heuristics' results in accordance with the optimal solution.



[Figure 1]: Algorithm comparison with a 20-city TSP input

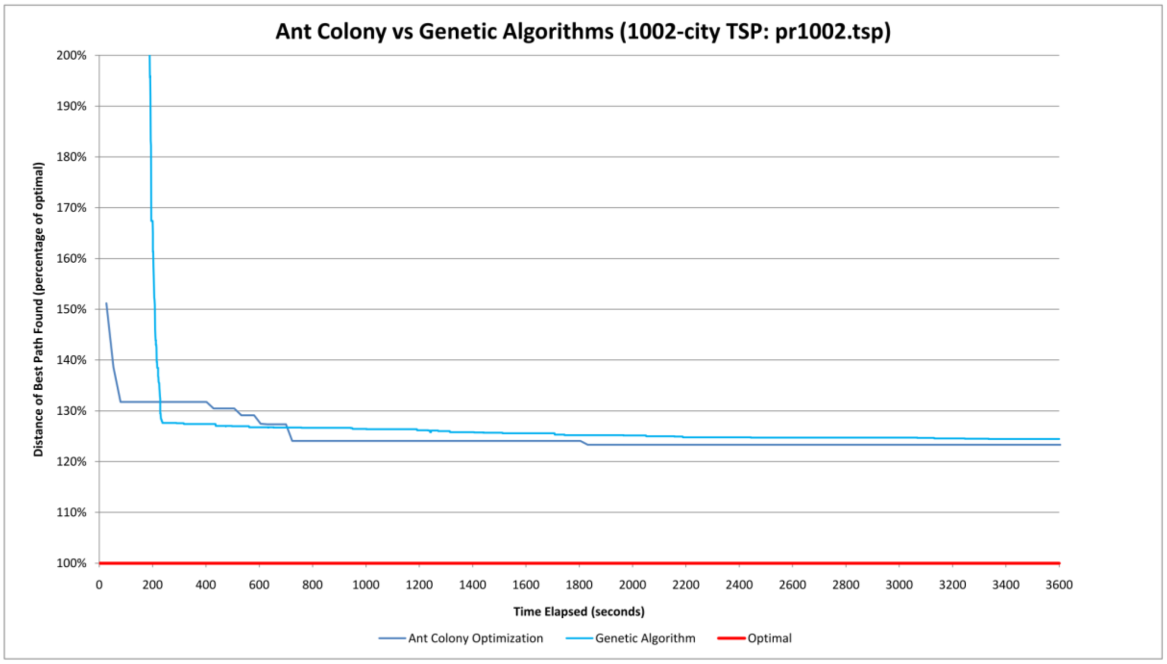
At first, the results from the 20-city TSP are a bit surprising – while the ant colony algorithm was able to quickly find a 'decent' answer, it took much longer for the genetic algorithm to do the same. This is likely due to the completely opposite approaches utilized in each algorithm: the ant colony algorithm relies on a decent starting path (while taking a bit more time to do so), and slowly refines its answer over time, while the genetic algorithm starts with random paths and attempts to refine it quickly. However, unlike in larger input sizes, the time it takes to get to a decent solution is minimal in both cases – less than 1 second – and the difference in solution quality is negligible.

Although not shown in [Figure 1] due to the constraints of the graph size, the genetic algorithm starts with an extremely bad initial solution at the first time step – more than ten times worse than the best solution, and as well the genetic algorithm generates solutions at a significantly faster rate than the ant colony algorithm.



[Figure 2]: Algorithm comparison with a 130-city TSP input

As discussed previously, as the input size increases to a 130-city problem, the differences in algorithm approach becomes more apparent. While the ant colony algorithm was able to find a decent solution in its first iteration (17% within optimal), the genetic algorithm struggles and takes many, many iterations to arrive at a comparable solution. However, once it does, it quickly reaches a steady state, while the ant colony algorithm stagnates for long periods of time and requires over 2500 iterations (5 minutes) until it reaches its best solution, which is still noticeably worse than the solution the genetic algorithm was able to achieve in 8 seconds.



[Figure 3]: Algorithm comparison with a 1002-city TSP input

With the largest input size tested[Fig. 3], the trends from the 130-city problem become magnified: again, the ant colony algorithm finds a decent solution in its first iteration (132% of optimal in 80 seconds), and it takes the genetic algorithm 3 times as long to reach a comparable solution. However, both algorithms end up with reasonable solutions after an hour each.

From these figures and discussion, there is no "best" algorithm – if the intent is to quickly find a decent solution, the ant colony optimization method will be ideal; however, if one wishes to find the best solution possible without resorting to an exact algorithm, the genetic algorithm is preferable. It is, however, necessary to note that there are many more extensions available to the ant algorithm to provide better results (i.e. lessen premature stagnation) – improvements can be made to the ant-city assignment policy, an ant's next-city decisions[9], pheromone deposits[7], and cooperation between ants[8]. It is likely that with these improvements that we can still maintain the algorithm's success in achieving a decent solution on its first iteration, while significantly speeding up the refinement process. On the other hand, while the genetic algorithm can still be improved such that its initial population is a bit better than random[3], the improvements to its crossover algorithm, its greatest determinant of success, seem a bit less optimistic.

If given more time, it would have been exciting to implement some of the extensions to the ant colony algorithm in the literature, just to see how truly efficient a TSP solver this unorthodox method can be.

3.0. CONCURRENCY PLATFORMS: PTHREADS & OPENMP

To complete the extra credits available for this assignment, we implemented the genetic algorithm in both pthreads and OpenMP, and compared the relative performance of these parallel platforms.

We found that implementing the genetic heuristic concurrently with pthreads to be significantly more challenging than using OpenMP due to the much lower level primitives available through it, and the need to manually handle inter-thread communication and synchronization. In fact, we found it necessary to build a higher-level intermediary interface to handle parallel work execution through pthreads efficiently: a simple threadpool. The work involved in this ultimately formed a large part of the additional effort necessary for the pthreads-oriented implementation. Additionally, reimplementation of such a common concurrency primitive is inevitably prone to sub-optimality compared to implementations available in existing, more thoroughly reviewed and refined, higher-level toolkits.

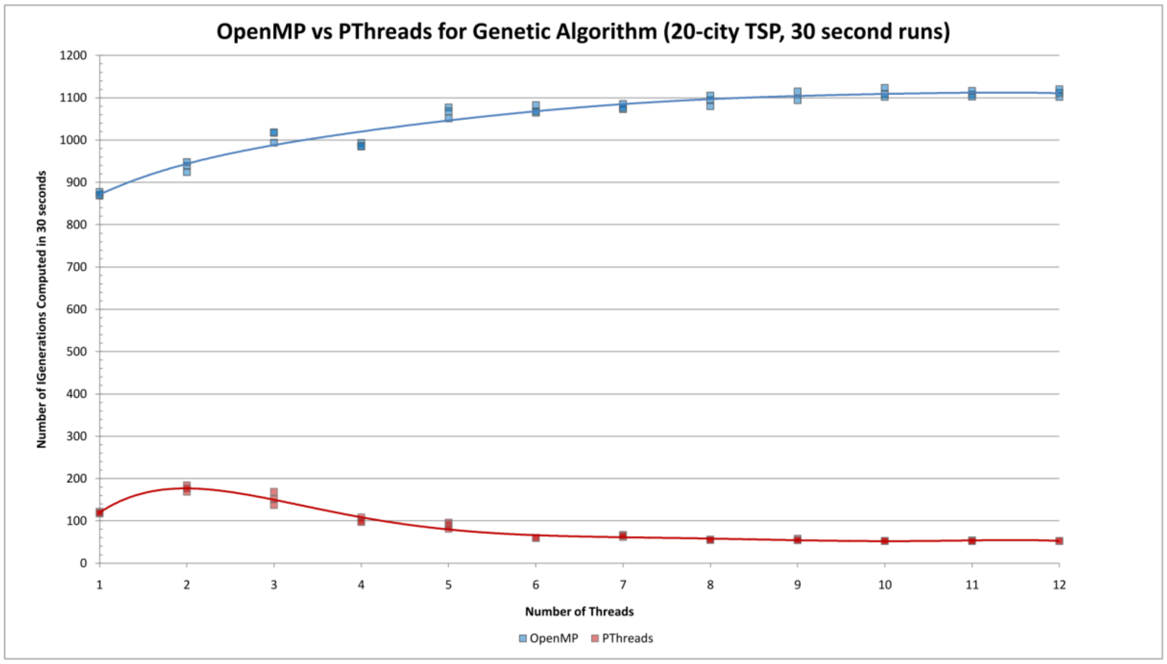
However, working directly with the low-level primitives as pthreads requires is not without benefit: at least in theory, it permits a more specialized implementation, ultimately offering the potential for better performance from code more tuned to the task—so long as the tendency for reinvented wheels to be less optimal than reference implementations does not overwhelm such potential gains. In our case, we found our pthreads implementation to introduce additional overhead sufficient to be significantly less efficient for small input sizes, but despite this, to handle larger inputs much more efficiently. We believe this additional overhead to have been incurred by a suboptimal threadpool implementation as implied above; specifically, the ring buffer used to queue tasks is prone to spurious wakeups when full or empty, waking up every single listening thread when a space is consumed or becomes available, rather than just the threads able to make use of it. The allocation necessary to instantiate tasks is also clearly suboptimal, in part due to the absence of a native optimized closure mechanism in C++; the trivial extent of the tasks used does not justify the heap allocation that is performed.

3.1. PTHREADS: THREADPOOL IMPLEMENTATION

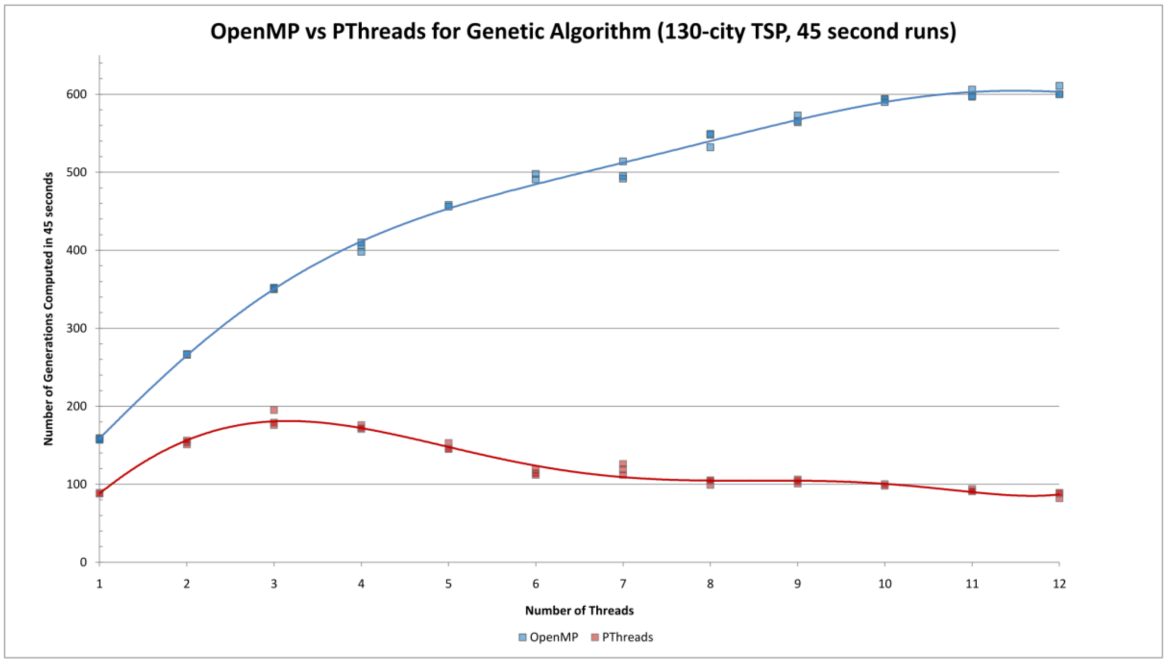
The threadpool used to negotiate between parallel tasks and the low-level pthreads API was implemented as a C++ class whose instances allocate a given number of threads once at instantiation which then consume task objects (effectively manually-implemented closures) from a shared queue (repurposed from a previous project) until termination. This queue is then filled by the controlling thread, which then (although is not strictly required to) blocks on the semaphore which tracks the number of completed tasks until that number reaches the number assigned to the pool, emulating the effect a pthread join would have had each task been given its own dedicated thread.

This method avoids the overhead of thread creation/deletion that a more naive implementation might incur, and ensures that unnecessary scheduler overhead may be avoided by doing work in only as many threads as there are cores, ultimately providing functionality and behavior similar to that found in OpenMP, allowing for a minimum of divergence from the natural flow of the algorithm code. We suspect that this code may in fact be an reimplementation of a relevant portion of OpenMP, as a threadpool serves the need of executing discrete jobs, such as independent iterations of appropriately tagged for loops, in parallel over the duration of the program's execution very well.

3.2. PTHREADS & OPENMP COMPARISON

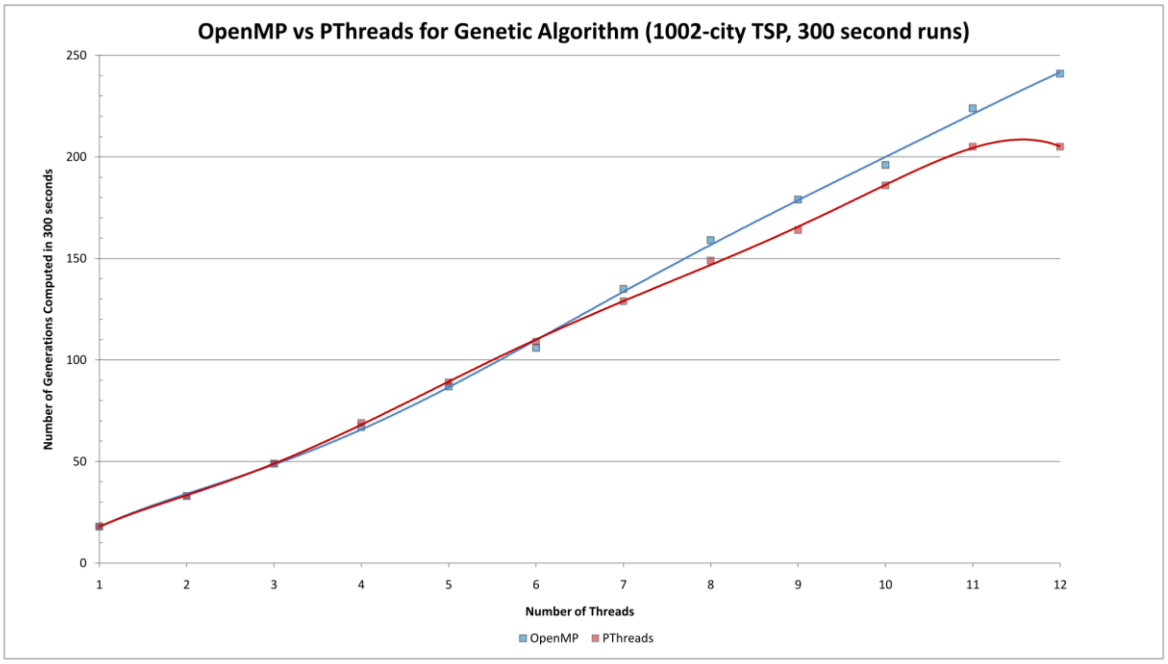


[Figure 4]: Platform comparison using a 20-city TSP input, and 30 second runs.



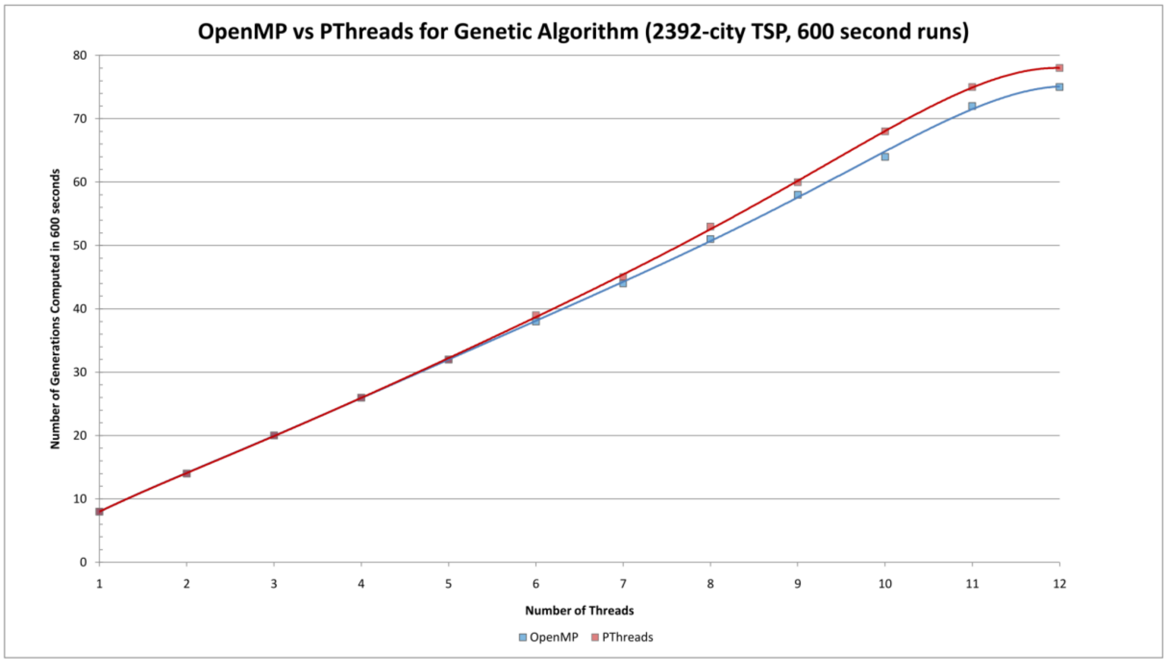
[Figure 5]: Platform comparison using a 130-city TSP input, and 45 second runs.

Initial small-scale testing confirmed suspicions that our pthreads implementation introduced significant additional overhead compared to the OpenMP implementation, as discussed above, to the extent that even additional threads over which to distribute the load not only failed to accelerate processing, but in some cases slowed it further. This suggested that a large part of the additional overhead might be constant per-thread cost, which gave some initial hope for relatively improved performance in larger data sets.



[Figure 6]: Platform comparison using a 1002-city TSP input, and 5 minute runs.

With nearly an order of magnitude more cities**[Fig. 6]**, the gap closes, and although pthreads continued to fail to demonstrate the expected returns from specialization, this data verified that the approach used was, though inefficient when faced with smaller/fewer tasks, not fundamentally flawed.



[Figure 7]: Platform comparison using a 2392-city TSP input, and 10 minute runs.

With roughly twice as large an input as the previous test, the predicted gains from a specialized implementation are finally clearly manifest, with the pthreads implementation having significantly more success taking advantage of increased concurrency. It is noteworthy that even such an unrefined threadpool as was used was able to ultimately outperform OpenMP, as if we presume that OpenMP is largely optimal, this suggests large gains indeed are to be had from more carefully refined use of pthreads in performance-sensitive contexts, especially when large workloads exist. Despite the significant time investment and debugging challenges necessary to successful optimal work at the level provided by the pthread API, it seems clear that existing higher-level toolkits are not yet advanced enough to overcome the cost of abstraction.

4. Difficulties Encountered

One issue that confounded us thoroughly for some time was what we first concluded to be an optimizer issue: With Microsoft Visual C++ on an Intel i7, our algorithms all behaved as expected, while on the AMD Opteron test machines using GCC, maximum performance was obtained with a single thread, with additional threads rapidly slowing the already perplexingly slow binaries. We ultimately stumbled upon the realization that we had mistakenly used the non-reentrant standard C function `rand()` throughout the parallel components of our implementations, and that contrary to expectation (although not in violation of the specification) both libc implementations tested implement a threadsafe `rand()`. This alone, of course, does not explain our situation. The difference lies in how this undocumented safety is implemented: in MSVC, `rand()` state appears to reside in thread-local storage, allowing for lockless use, while programs linked against glibc appear to instead use a locking `rand()` associated with global state. The resultant behavior was that MSVC quietly ignored our slip-up and behaved as we had intended, while glibc negated the performance benefit of concurrency entirely, leaving our speed to be whittled away with each additional thread by the remaining overhead. This was resolved by replacing all `rand()` calls with `drand48_r()` and `lrand48_r()` calls, which not only allow state pointers to be explicitly created, passed, and maintained, but provide much higher-quality randomness in a wider variety of useful forms, such as an even distribution between doubles 0.0 and 1.0, than the more commonly used `rand_r()` could provide.

5. References

1. Zaritsky, Assaf, "Introduction to Genetic Algorithms", *lecture slides*. Ben-Gurion University, Israel.
2. Grefenstette, J., et. al, "Genetic algorithm for the traveling salesman problem", *Proceedings of the 1st International Conference of Genetic Algorithms, pp.160-168. 1985.*
3. Kureichick, V. M., Miagkikh, V. V., Topchy, E. P., "Genetic Algorithm for Solution of the Travelling Salesman Problem with New Features against Premature Convergence", ECDC-96. 1996.
4. Miller, B. L., Goldberg, D.E., "Genetic Algorithms, Tournament Selection, and the Effects of Noise", *Complex Systems*, Volume 9, pp. 193-212. 1995.
5. Fukuda, M. (2011). "Program 1: Parallelizing Traveling Salesman Problem with OpenMP". *Available at:* http://courses.washington.edu/css534/prog/prog1.pdf
6. Jones, M. T., "Artificial Intelligence: A Systems Approach", Infinity Science Press, Hingham, MA 02043. ISBN 978-0-9778582-3-1, pp.423-429.
7. Stutzle, T., Hoos, H., "MAX-MIN Ant System", *Future Generation Computer Systems*, Vol. 16, Issue 9, pp.889-914. June 2000.
8. Dorigo, M., Gambardella, L. M. , "Ant colony system: a cooperative learning approach to the traveling salesman problem," *Evolutionary Computation, IEEE Transactions on* , vol.1, no.1, pp.53-66, Apr 1997
9. Gaertner, D., Clark, K., "On Optimal Parameters for Ant Colony Optimization algorithms," *Proceedings of the International Conference on Artificial Intelligence*, pp.83-89. 2005.
10. TSPLib, Universität Heidelberg (2008). Available at: http://www2.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/
11. Forum post, "openmp slower than single threaded" (2009). Available at: http://secure-software.intel.com/en-us/forums/showthread.php?t=66808