An Efficient CPU Implementation of a Genetic Algorithm for the Travelling Salesman Problem

Alex Fillmore
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
afillmore@oakland.edu

***Abstract*—This project explores using pthreads as an optimization technique for parallelizing a CPU implementation of a genetic algorithm that solves the travelling salesman problem. Pthreads is a successful method for optimizing the algortihm, speeding up the processing time by sixty percent over the purely sequential implementation. It is concluded that a parallization strategy that considers the target processor is a an ideal way to implement pthreads for algorithm speedup.**

***Index Terms*—Genetic Algorithm, Travelling Salesman, Embedded, Optimization, Multi-thread**

I. Introduction

The Travelling Salesman Problem (TSP) is a historical problem in the world of computer algorithm optimization, with exploration of the problem dating back to 1954 [1]. The problem is a simple one to understand and goes like this: A salesman wants to travel to a number of cities and then return home to his starting city. Given the distances between each pairing of the cities is known, what is the shortest possible route the salesman can take to travel to each of the cities exactly once and then return to his home city? While being a simple problem to understand, finding the best solution to the problem is significantly more complex.

In fact, the TSP has been shown to be a NP-Hard problem, meaning that the problem is at least as difficult to solve as the hardest nondeterministic polynomial time problem [1]. A naive approach to solving this problem might be to brute force it by calculating the distance travelled in every possible subset of the population and checking for the minimum total distance. However, if we consider that the starting city will always be the same then for a set of N cities there will be (N-1)! possible permutations of the order of cities to visit. While this may be feasible for small data sets, the brute force method quickly becomes untenable for larger data sets.

Considering the O((N-1)!) time complexity of the brute force method, it is clear that a more efficient methodology for finding an optimal solution to the TSP is desirable. This is where genetic algorithms (GA) may be able to help us. A genetic algorithm is a evolutionary algorithm that takes inspiration from biological processes such as evolution and natural selection to search the solution space of problems for an optimal solution [2]. Evolutionary algorithms are one subclass of metaheuristic search algorithms, with other examples being Particle Swarm Optimization (PSO), Gravitational Search Algorithm (GSA), and Ant Colony Organization (ACO) [3]. Metaheuristic algorithms in general are meant to explore the solution space of problems to find good solutions despite having incomplete information.

Genetic algorithms work by creating a population of potential solutions to the problem and having the population undergo change through methods that emulate evolution and natural selection. The general flow of a genetic algorithm can be seen in Fig. 1.

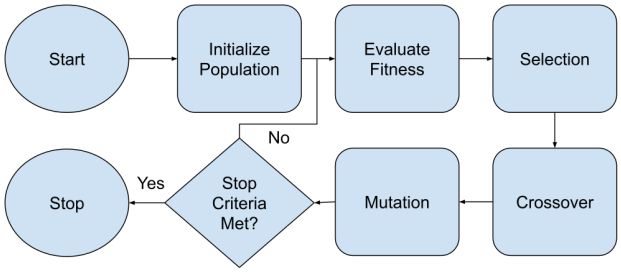

Fig. 1. Genetic Algorithm Methodology

To begin, we must initialize an arbitrarily sized population of potential solutions, which we call members of the population. Typically this population is large to allow for greater variety in the solutions. A important determiner in whether a problem can be solved using a genetic algorithm is whether or not we can formulate a solution to the problem as what we call a chromosome. For the TSP, a chromosome of a member might look like the following:

C[1 5 4 3 2]

This chromosome outlines the solution to pictured in Fig. 2. In this case, the salesman begins in city one, then travels to cities two, four, five, three and then back to city one in that order. This completes a valid solution to the TSP. In order to initialize the population, we can just generate random permutations of the list of all cities.

Once we have initialized a population, we now want to evaluate the fitness of each of the individual solutions to quantitatively determine their quality. For the travelling salesman problem, instead of looking at fitness it is simpler

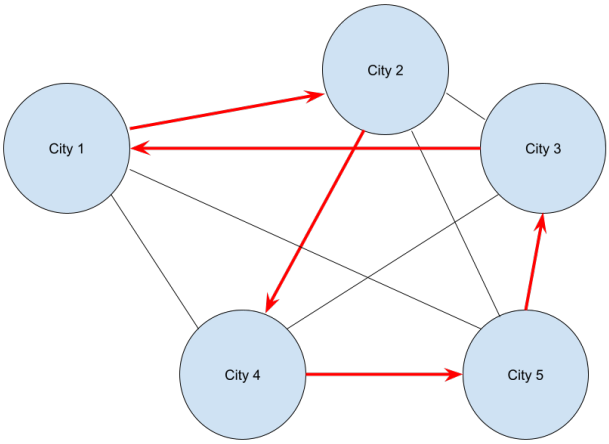

Fig. 2. A Solution to the TSP

to look at cost. Where a greater fitness is better, a lower cost is considered. Fitness and cost are the inverse of one another. To calculate the cost of the TSP, we simply sum the distance travelled between each of the cities in the solution.

Once the quality of each member of the population is known, we must select pairs of solutions to produce offspring. This is the part of the process that emulates natural selection. To mimic this process, the selection methods of genetic algorithms need to be biased towards the most fit/least cost individuals of the population.

Once we have selected parents for the next generation of solutions, we need to be able to breed them in some way that the parents can pass on their qualities to the children. This process is known as crossover. Through crossover, every member of the population is replaced with a new child solution.

Finally mutation is performed on select children. Each child has a random percent chance to mutate, randomly altering their solution in some way. This is done to introduce variety into the gene pool, preventing the solutions from converging to a local minima of the solution space. Without mutation, it is more likely that the population will converge to a good answer while never finding its way to the best answer.

Together, these processes make up the genetic algorithm. By performing this sequence repetitively, the population will converge to an optimal solution to the problem. However, now it must be determined when to stop producing new generations. This is known as stopping criteria. Common stopping criteria might be after a certain number of generations have passed, once the fitness has passed a given threshold, or once a certain percentage of the population has converged to the same answer.

For optimization, it is important to recognize that at each stage of the genetic algorithm each of the members of the population are being operated on individually and don't have interdependence. This means that genetic algorithms are a good target for speedups from parallelization, where in theory each member of the population could be operated on in total isolation. To accomplish this sort of optimization in this project, pthreads are used. Pthreads—also known as POSIX threads—are a parallel execution model for Unix based systems that enable parallel processing of data on a CPU.

The intent of this project was to develop a sequential implementation of a genetic algorithm for solving the travelling salesman problem, then explore how pthreads could be used to optimize the sequential implementation to save execution time without negatively the quality of results.

II. Methodology

In this section I will cover the specifics of each of the stages of the genetic algorithm, how those stages were implemented in the sequential implementation, and then how those stages benefited from a parallel implementation with pthreads.

A. Evaluate Fitness

As mentioned previously, for the TSP cost is typically used instead of fitness. The cost can easily be calculated for a given solution by looping through the cities indexes and summing the distance between successive locations. The distance between two cities is pre-calculated during initialization and stored in a 2D array lookup table of size `NUMCITIES * NUMCITIES` where `NUMCITIES` is the number of cities in the problem. This loopup table can be indexed by inputting the two city numbers as the two indexes to the array to find the distance between the two cities. To find the total cost of a given solution, we must execute through a for loop summing the distances a total of `NUMCITIES` times. Then we must loop through every member of the population, `POPULATIONSIZE`, to get the cost of each individual.

Since the cost of each individual is independent of the other members of the population, we can parallelize the fitness evaluation using pthreads by splitting the population equally into smaller subsets. Then we can spawn threads using pthreads, with each thread responsible for one of the subsets of the population. For the case where the number of threads spawned, a defined constant `NUMTHREADS`, is four each thread would operate on a fourth of the population. So instead of a single for loop executing on the range zero to `POPULATIONSIZE-1`, each thread would instead operation on a fourth of `POPULATIONSIZE`. This parallelization strategy is visualized in Fig. 3.

In order for each thread to know which fourth of the set to operate on, we must pass it some parameters. In general, the method for passing parameters to a pthread instantiation is to pass it a pointer to a structure that contains all of the information you want to pass. For this project implementation, I opted to create a single struct definition that would contain all of the information any of the thread calls would need. This structure contains pointers to arrays for storing all pertinent information, as well as start and end indexes for the threads to determine the ranges of the population to operate on.

Once the individual cost of each member is known, it is

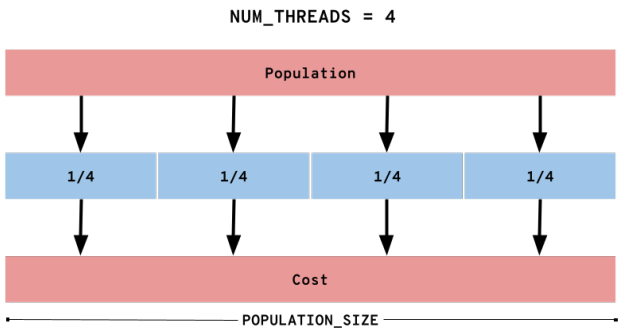

Fig. 3. Fitness Evaluation Parallel Strategy

pertinent to find the minimum cost of the entire population in order to determine whether we have reached a satisfactory solution yet. Similar to fitness evaluation, this process loops across the entire population. However, in this case members of the population must be compared and can't be operated on in pure isolation. Therefore, we must use a slightly different parallelization strategy. We will used what is known as the reduction approach. The population is still broken into equal chunks for each thread, but instead of each thread returning an output for every member of the population each thread returns the minimum cost of that subset. Then the outputs of each thread are compared in the main thread to find the global minimum. This strategy is pictured in Fig. 4.

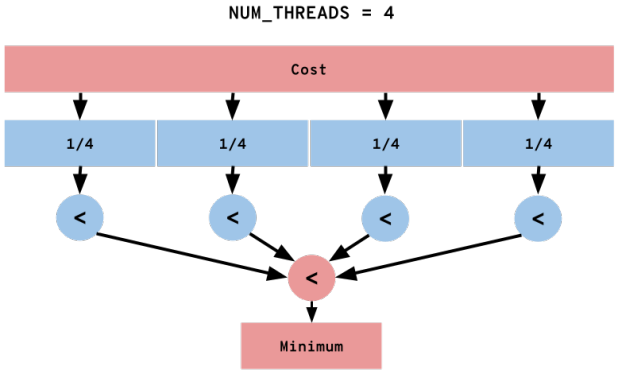

Fig. 4. Minimum Cost Parallel Strategy

B. Selection

To perform selection, I chose to go with what is known as a tournament selection. To execute a tournament selection, I select a random subset of the population with a size of `TOURNAMENTSIZE`-a global defined parameter. For a given subset, the member which has the least cost is chosen to be a parent. This occurs `2*POPULATIONSIZE` times since we want to replace every member of the population with children and each child requires two parents. To parallelize this stage, a similar strategy of divide and conquer from the fitness evaluation is used. The required number of parents to produce is evenly split across each thread.

C. Crossover

Once parents are chosen, we need to combine their information in some way to produce a child. This process is known as crossover. To begin with, each child is initialized with their first city as 1. The salesman always starts in the same city, so this much of the solution is given. From their, the rest of the solution is inherited from the child's parents. Look at Fig. 5 and suppose that we have two parent solutions P1 and P2. To find the second city of the child solution, we will pick the city in position two from the parents that is more optimal. Because of this step, this implementation is known as greedy crossover. In the case of Fig. 5, city three is closer to city one than city five is so the child inherits city three from P2. In the third position, city four is closer to city three than city two is so the child inherits from P1. This process continues until the entire child solution has been filled. In the case where the city in the parent has already been used in the child, the next valid city in the parent solution is considered. If there are no more valid cities left in the parent, the next unused city in the child is considered.

This process again operates on a single member of the population at a time, so we use the same parallelization strategy outlined in Fig. 3 for the fitness evaluation. It is important to note that one member of the population can serve as a parent for more than one child, so it is entirely possible that multiple threads will be accessing the same location of the parent array at the same time. However, since each thread is acting only as a reader to the parent array and not a writer this does not introduce any race conditions to be concerned with.

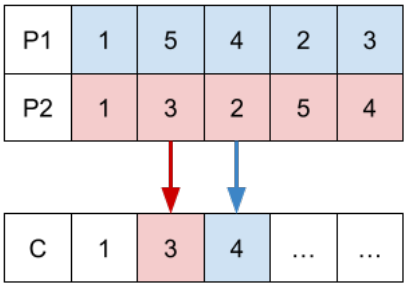

Fig. 5. Crossover Process

D. Mutation

For mutation, the execution is relatively simple. A global define, `MUTATIONCHANCE`, determines the percent chance for any individual child to mutate. In the sequential implementation, a random number in the range [0,100) is generated with the library function `rand()` and if the number is less than `MUTATIONCHANCE` then the child mutates. If a child mutates, then two random indexes are selected and

the city indexes in those positions are swapped. This follows the same divide and conquer parallelization strategy outlined for evaluate fitness. An important note for the parallelized implementation is that `rand()` is not a multi-thread safe function and when used in multiple threads will cause race conditions that hinder performance. Instead, each thread is passed a unique integer seed and the address of that seed is used to seed the multithread safe `randr(&seed)` function.

III. Experimental Setup

All experiments were conducted on a Terasix DE2i-FPGA Development Kit board, which includes an Intel Atom N2600 processor. This is a two core processor which can handle up to four hyperthreads. The code was executed in a Ubuntu Linux 12.04.4 operating system environment on the command line. Timing reports were build into the codebase, and reported the average time of each function across ten generations as well as the average time for the total execution of a single generation. The parameters in Fig. 6 were kept constant across testing while the number of threads was varied.

Fixed Parameters	
TOURNAMENT_SIZE	128
MUTATION_CHANCE	10%
NUM_GENERATIONS	10
POPULATION_SIZE	100000

Fig. 6. Fixed Experiment Parameters

IV. Results

The results of the experimental conditions were as reported in Fig. 7. It is clear that the largest time saving over the sequential implementation came from moving up to having two threads, with a 54% speedup in execution time. After the first additional thread however, the performance benefits from adding more threads are marginal. The best performance came at twenty threads with a 60% speedup. I suspect there weren't much performance gains to be had after the first additional thread considering the target environment. The target CPU only has two cores, and therefore is likely to only be able to truly run two threads in concurrency. Considering these threads are very compute intensive with little to no waiting time during which another thread could execute, I am unsuprised by these results.

Notably, the functions that began with faster execution times stood to benefit the least from parallelization. Both mutation and minimum cost started with relatively fast execution times compared to the other functions, and had worse performance than the sequential implementation at higher thread counts. This is likely due to the additional overhead of adding additional threads while not having much time to save in the first place. In fact, all multithreaded runs of the minimum cost function performed worse than its sequential counterpart. This is likely due to the fact that it was the quickest function by far to begin with.

V. Conclusions

In conclusion, I have found pthreads to be a powerful tool for optimizing CPU bound applications to minimize runtime. However, when multithreading an application there are important factors to consider. What processor is this program going to run on? It is worthwhile to consider this when determining the number of threads to launch, as the processors core count and architecture will directly affect the number of threads that can compute concurrently. Additionally, just because you can multithread a function doesn't mean you should. Multithreading introduces additional overhead in order to launch threads, precious time that may not be worth it as was the case for the minimum cost function. Finally, the algorithm itself is important to consider when deciding whether or not to multithread. Genetic algorithms happen to be highly parallelizable, but that is not the case in other algorithms where data and functions may be more interdependent on one another.

References

[1] Jünger, M., Reinelt, G., & Rinaldi, G. (1995). The traveling salesman problem. Handbooks in operations research and management science, 7, 225-330.

[2] A. Lambora, K. Gupta and K. Chopra, "Genetic Algorithm- A Literature Review," 2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon), Faridabad, India, 2019, pp. 380-384, doi: 10.1109/COMITCon.2019.8862255.

[3] M. Abdel-Basset, L. Abdel-Fatah, and A. K. Sangaiah, "Metaheuristic Algorithms: A Comprehensive Review," Computational Intelligence for Multimedia Big Data on the Cloud with Engineering Applications, pp. 185–231, 2018, doi: https://doi.org/10.1016/b978-0-12-813314-9.00010-4.

Function	NUM_THREADS (Processing time in us, averaged across 10 generations)									
	Sequential	2	4	6	8	10	20	50	100	200
Selection	4169605	1605586	1402526	1296798	1315613	1324419	1288261	1406318	1382768	1371027
Crossover	1791997	1160138	1134223	1104464	1078718	1040005	996470	1042226	1043087	1111122
Mutation	16431	12886	12939	14440	15554	18645	18987	16613	28138	42932
Cost Update	284874	177117	162433	158543	159155	156719	167272	169064	174868	199817
Minimum Cost	1773	2777	4825	3704	5463	4356	8761	18059	23014	38039
Generation Total	6301773	2902777	2704825	2603704	2605463	2604356	2508761	2718059	2623014	2738039
Percent Speedup	N/A	54%	57%	59%	59%	59%	60%	57%	58%	57%

Fig. 7. Timing Results for Varied Thread Counts