Parallelizing Genetic Algorithm for Traveling Salesman Problem

15618 – Parallel Computer Architecture
Haoxuan Zhu (haoxuanz), Ziyi Liu (ziyiliu)

Summary

This project focuses on exploring and analyzing the performance gain of different methods to parallelize the genetic algorithm for solving the traveling salesman problem. We implemented a vanilla sequential version (seqv) in C++ and parallelized it using CUDA (gpuv) for GPU acceleration and using OpenMP (ompv) for multi-processor acceleration. We further compared and reasoned the performance difference of the three implementations.

Background

Traveling Salesman Problem

The traveling salesperson problem (TSP) is to find the shortest possible route that visits a given set of cities and returns to the starting city. It is NP-hard, meaning that the time required to find an optimal solution grows exponentially with the number of cities. As a result, heuristic algorithms like GAs are often used to find approximate solutions to the TSP. Originally, the TSP dataset provides a set of city coordinates as input and requires the program to generate a minimum cost (Integer/Float) and its corresponding route. Since our primary interest lies in paralleling the genetic Algorithm, we modified the TSP as follows. Implementation Details and Reasoning is included in the **Approach** section.

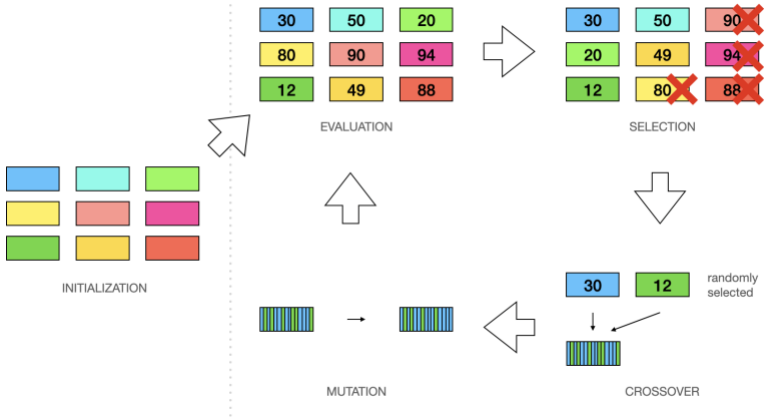
- PREASSUMPTION: the starting city is fixed (CityID=0)
- INPUT: distance matrix (2d integer array with dimension of NUM_CITY x NUM_CITY)
- OUTPUT: minimum cost (integer) and its corresponding route (string)

Genetic Algorithm

Genetic Algorithm (GA) is a search-based algorithm that is widely used to solve NP-complete problems. Inspired by genetics principles, the algorithm converges to solutions by reflecting the process of natural selection where the fittest chromosomes are chosen for reproduction to produce better offspring of the next generation until an optimized solution is generated.

Genetic Algorithm is an iterative process. After randomly initializing a population, the algorithm executes the following 4 steps recursively, scoring and selecting elite parents from the population for mating (S ELECTION), applying crossover to generate next generation chromosomes (CROSSOVER), conducting

mutation to introduce data augmentation (MUTATION), and updating existing population by removing low-score chromosomes (EVALUATION), until a predefined criteria is met. For a N-city TSP problem, brute force requires examination on N! candidates to get the optimal solution. Genetic Algorithm avoids this by multiple iterations until convergence.



Key data structure and operations

The basic element in GA is called Chromosome, which contains a gnome and its fitness score. In our use case, the gnome is an integer array representing the route, where each element is the CityID, and the fitness score is the cost of current route (genome). Another important data is the 2d distance matrix where Matrix[x][y] represents the cost of getting to city y from city x.

The distance matrix only suffers from concurrent read operations. The Gnome suffers most operations including indexing, addition, element swapping, etc. Details will be discussed in the **Approach** section.

Workload analysis

Kernel for Population Initialization: This is the first step of the algorithm and is only executed once. The kernel needs to generate POPULATION_SIZE chromosomes using random number generator (RNG) and then use the distance matrix to calculate the fitness score (cost.) Each chromosome's generation and evaluation are independent and requires no dependency. Hence, it is data-parallel and can be easily parallelized.

Kernel for Selection: This is the second step of the algorithm and is executed iteratively. We used tournament selection approach to pick the best chromosomes and store them in mating_population array. Each tournament will randomly select TOUR_SIZE candidates and pick the one with the lowest cost score. Since we want to avoid picking a same chromosome multiple times, it introduces dependency on random number generation. One

needs to make sure that a same id cannot be used in a same tournament, and winners from each tournament should be unique.

Kernel for Crossover: This is the third step of the algorithm and is executed iteratively. Crossover is for generating new population based on the selected mating population. We used RNG to randomly pick two parents from the mating population and generate an offspring. The step is repeated until POPULATION_SIZE offspring are generated. For each new offspring, the gene in the ith position is chosen from one of the cities in both parents that follow the (i-1)th city in the offspring's genome. Whichever city from the parent has a closer distance will be chosen. If a city is already chosen, the algorithm will pick the first CityID seen that does not exist in the offspring's genome. For each gene in the genome of a new offspring, it has dependency on all its ancestors, which brings difficulty in parallelizing.

Kernel for Mutation: This is the fourth step of the algorithm and is executed iteratively. Mutation happens randomly determined by RNG. When a mutation happens, it will randomly generate two indices i, j and reverse sub-genome genome[i] to genome[j] (eg; given a genome 1,2,3,4,5 with two indices 1, 3, the output is 1, 4,3,2,5). This step is essential in the genetic algorithm to provide new population to avoid converging to a local optima. When NUM_CITY is large, this swapping process can repeat for a certain number of times in a single mutation step, since swapping only two elements might not lead to the desired degree of variation. Mutation is independent in terms of chromosomes, but for each gene, it has dependencies as it involves element swapping.

Kernel for Evaluation: This the fifth step of the algorithm and is executed iteratively. With the new-generation population, each will be evaluated by calculating the fitness score (calculate the total cost based on the route each chromosome represents.) This operation is independent of chromosome and can be parallelized easily using data parallelism scheme.

Benefits from parallelization

To reach correct result, a larger initial population is required for faster convergence to the optimal solution. However, with a large population and many iterations, the computation required for fitness score evaluation, crossovers, and mutation will increase, resulting in severe performance reduction. Different parallelization strategies can be applied to different phases based on their computation and data dependency nature and may ideally accelerate the process. Data parallelism can be performed during chromosome evaluation by exploiting independencies of chromosomes, and ad-hoc parallelism strategy for chromosome evolution (crossover and

mutation phases) can be designed to accelerate the process. The huge population and computation requirements allow parallel programming to scale well. The locality can also be leveraged by wise grouping chromosome and mating population reuse.

Approach

Honoring the project proposal, we implemented all sequential and parallelized versions on our own. All implementations only rely on the standard libraries so that we can compare and analyze the performance gain from different parallelization strategies. Based on the workload analyze, we abandon SIMD since all kernels contain random picking and swapping, which is not suitable for vectorized computation. Hence, our primary focus lies on GPU and shared-memory multiprocessing programming. Since both parallelism methods (CUDA and OpenMP) supports C++ very well. We decided to use C++ for all implementations.

OpenMP

This section discusses various OpenMP approaches we attempted to parallelize the sequential version using OpenMP. The performance of multiprocessing programming depends on the actual hardware we use. Hence, we first targeted the 8-core GHC machines. When implementing the sequential version, we used class to represent chromosome, and all crossover, mutation, evaluation operations are performed by the class' private functions. The use of class objects does not provide a good foundation for parallelization. Hence, we use a 2-d vector to hold all gnomes where each row represents a chromosome and another 1-d vector to store each chromosome's fitness score. However, from perf-tool results, we found that vector operations take most of the time due to its background resizing and data movement. Hence, we constructed a struct Chro with two attributes, an integer for fitness score, and a pointer to an integer array to store genomes. The genome array will be allocated using malloc when only requested. After all these changes, we started experimenting adding OpenMP to accelerate the revised sequential version.

We started with naïve approach by using "#pragma omp parallel for schedule(dynamic)" to parallelize all the for-loops. This step tests whether the Makefile is correctly written and if OpenMP can be correctively called. The results were inevitable terrible due to terrible load balancing, synchronization and overhead of creating new threads. Hence, we fine-tuned all the kernels mentioned above to get the best combination.

Kernel for Population Initialization:

Population initialization mallocs an integer array for each chromosome and assign random sequence to it. The pseudo code is provided below,

```
for (int i = 0; i < POPULATION_SIZE; ++i) {

   chro->gnome = (int*) malloc(num_city * sizeof(int));

   for (int i = 1; i < num_city; ++i) {

      chro->gnome[i] = randomly_select_unused_cityID();

   }

   calculate_fitness(chro, dist_matrix);

}
```

Since generation of each chromosome is independent, we tested different scheduling policies, and the results is shown in the table below

Initialization	Dynamic	Static	Auto	Guided	Sequential
2000	340838	341254	340444	340704	1623774
20000	3390938	3398390	3392503	3393643	15994781
200000	33909022	35263824	35237569	35025923	159450390

Scheduling policy performance (microseconds) on different population size for population initialization

Since the work done for each chromosome is the same, we assumed that the dynamic policy will be the best choice, and the experiment verified our idea. When the population size is small, all policies show similar performance. The reason behind is that when the workload is small, the overhead is dominating. As the population size gets better, dynamic policy shows the best performance overall. Our reasoning is that since the dynamic scheduling policy works by dividing the iterations into chunks and assigning one chunk to each thread as it becomes available, it is suitable for this case where the workload of the iterations is uniform, and the number of available threads is fixed. Since the population size greatly exceeds the number of available threads, dynamic assignment shall have better load-balancing performance as it provides great flexibility to distribute chromosomes evenly.

We then attempted parallelizing the inner loop that assigns cityID to each array element. It backfires as the work of assigning gene is dependent due to the constraint that every cityID should be unique in the array. The overhead incurred by threads and synchronization outweighs its advantage. Hence, we did not parallelize the inner loop.

Kernel for Selection

Since we are using tournament selection and the mating population size is much smaller than the population size, the number of operations per chromosome is limited and not worth using parallelism. Our experiment verifies that. With the tournament size = 50 and mating population = population / 2, sequential version takes 6590 microseconds for selection, but OpenMP version takes more than 50000 microseconds for selection. Here, the overhead of using multiple threads and synchronization is more expensive.

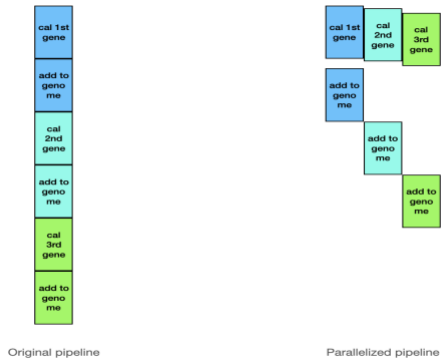
Kernel for Crossover

The process of generating offspring can be parallelized as each offspring is independent to each other. We tested different policies for the outer loop (chromosome level) and the result shows that dynamic scheduling is the best thanks to the similar workload of each task.

Crossover	Dynamic	Static	Auto	Guided	Sequential
2000	90609	96243	96660	94741	725922
20000	953726	978870	1004776	964975	7694947
200000	11195810	11431799	11485984	11248466	76933312

Scheduling policy performance (microseconds) on different population size for crossover

However, the difficulty lies in the dependency of genes of each chromosome. Since the ith gene is determined by (i-1)th gene, it has to be performed sequentially. Hence, we proposed the idea of parallelizing the process of getting the next gene candidate given a gene. Instead of generating the genome one by one, we can parallelize the process of looking for the next gene candidates from both parents given a gene, and then stitch them together based on the old rule. The reason why we cannot add gene to the genome on the fly is because each gene cannot be added more than once. Hence, each gene must check if it is already used before being added. If it is already used, then it will sequentially check from 0 to NUM_CITY and pick the first value that has not been added to the route.



Kernel for Mutation

In the original genetic algorithm, the mutation stage happens after crossover is finished. But we don't think it is efficient for parallel programming given that mutation requires few calculations. Hence, we merged the mutation to the crossover stage. Each crossover task needs to handle mutation as well. In this way, we can avoid any overhead incurred by OpenMP's thread and synchronization requirements.

Kernel for Evaluation

In the evaluation stage, we can solely leverage data parallelism to calculate the fitness score of each new chromosome.

Memory Optimization

We noticed that in the original design, the mutation operations for chromosomes will not start until all crossover is finished. It is bad for cache performance as each chromosome will need to be reloaded again during the mutation stage. Hence, combining the mutation operations with crossover into a single function, we could reduce the overhead and potentially leverage the memory locality. With the similar idea to reduce the overhead brought by the function calls, the evaluation functions are also rearranged to explore the possible improvement in memory locality.

CUDA (GPU)

For CUDA parallelization, the different stages of the Genetic Algorithm naturally map to different kernel functions. Each GPU thread is responsible for the calculation of one chromosome in the population, which divides the population into concurrently running thread blocks. The mapping works well because although there are dependencies between stages, the computation of each chromosome is rather independent within the stage. Therefore, running one thread per chromosome achieves the maximum parallelism.

While implementing the kernel function for each stage, we experimented with different strategies for launching the kernels. Specifically, for CROSSOVER, we tried two different strategies. The first one launches one kernel for each chromosome in the population, which is similar to the sequential version, and the second one launches one kernel for each element in each chromosome. The motivation for strategy 2 was that we noticed that with strategy 1, as NUM_CITIES increases, the number of threads launched per block needs to decrease due to resource limit and increased computation complexity. For example, the GPU on the GHC machines supports at most 1024 threads per block, and with ≤40 cities, all 1024 threads can be launched, but with 194 cities, only 512 threads can be launched. Therefore, more blocks are needed, and it is likely that not all blocks can be run concurrently with a large population size. Therefore, we decided to try strategy 2 - assign less work to each thread, to launch more threads per block. Indeed, with this strategy, all 1024 threads can still be launched even with 194 cities, but it turned out to be much slower than strategy 1, especially with more cities. This is likely because although there are more threads per block, we are also launching much more blocks as NUM_CITIES increases, so many of them still have to wait. Moreover, there is also significant overhead of launching this many threads, which can greatly hurt the performance. We also tried these two strategies on the MUTATION stage and discovered the same result.

```
//strategy 1

crossover<<<num_blocks, BLOCK_SIZE>>>(cudaPop, cudaParents, cudaStates);

//strategy 2

for (int c = 1; c < NUM_CITIES; c++){

    crossover_single<<<num_blocks, BLOCK_SIZE>>>(cudaPop, cudaParents, c, cudaStates);

}
```

For the SELECTION stage, our original implementation in the sequential version is based on sorting everyone's fitness score and pick the chromosomes with the best scores. However, we quickly realized that this is not the most efficient solution on GPUs since sorting large arrays that are in global memory is slow and hard to parallelize. Therefore, we changed to tournament selection as described above.

Another challenging aspect with the CUDA implementation is generating random numbers. The Genetic Algorithm largely depends on the random function as SELECTION, CROSSOVER, and MUTATION stages all have to use it. We used the cuRAND library for its simplicity and efficiency. We set up POPULATION_SIZE generator states prior to the start of the iterations, so that each thread can have its own state for better memory access patterns. However, the generators can only give pseudorandom numbers, meaning that the sequence of random numbers for each thread is the same for each iteration. This poses a serious problem to our algorithm, as for each thread, winner will be selected from the same pool, the same parents will be chosen for crossover, and the same mutation pattern will happen. This leads to the program to converge very quickly, but most of the time not to the optimal solution. To solve this issue, we tried many different ways to add more randomness to the generator. A few of them are outlined below:

1. We tried to use the fitness score of the chromosome in the current iteration as an offset to choose a different random generator each time, but the fitness scores will eventually converge, leading to even less randomness
2. We tried to use the current time to compute a seed in CPU and pass to the kernel function. This works but since a seed has to be computed for every kernel function in every iteration, the total time increased.
3. Our final solution is to use the current iteration number as an offset to choose a different generator each iteration, but only for the SELECTION and MUTATION kernel. This is because as long as a different parent is selected in each iteration, even CROSSOVER picks the parents at the same indices, it will be different parents from last iteration, and thus generating a different child. On the other hand, if CROSSOVER also uses the same offset, it will just be picking the same parents that the thread before it was picking in the last iteration, which won't solve the problem. Using this method, the algorithm might converge in more iterations, but closer to the optimal solution.

The last aspect that differs from the sequential version is that in the sequential version, the best chromosome is evaluated in every iteration, but in CUDA, this requires copy the entire population from GPU back to CPU. Therefore, we defined a variable called CHECK_INTERVAL, and only perform this operation every CHECK_INTERVAL iterations.

Results

To measure the performance of our implementations, we timed the execution time of each stage by inserting clocks around the function calls. Then, we averaged the total execution time over the number of iterations that the program takes to converge. This is because the stages might have different execution time before and after convergence, where convergence is determined if the best fitness score doesn't improve for a predefined number of iterations. These times are used to calculate speedup of the parallelized implementations in the following experiments. In the analysis below, we will refer to 4 different implementations

1. *sequential_vec*: the first single-threaded sequential implementation using vectors to represent the chromosomes.
2. *sequential_arr*: the improved single-threaded sequential implementation using arrays, since we realized that while vectors are easier to implement, they tend to be slower and can not be easily parallelized. For example, CUDA doesn't even support vectors. In most of the experiments, this implementation is our baseline as sequential_vec is too slow to converge in most cases.
3. *OpenMP*: the parallelized implementation using OpenMP. All experiments with OpenMP are run on GHC machines with 8 threads, except one that explicitly mentioned to be run on PSC machines.
4. *CUDA*: the parallelized implementation using GPU. All experiments with GPU are run on GHC machines with 512 threads per block for the reasons mentioned in Approach.

We will also refer to the following parameters in the analysis:

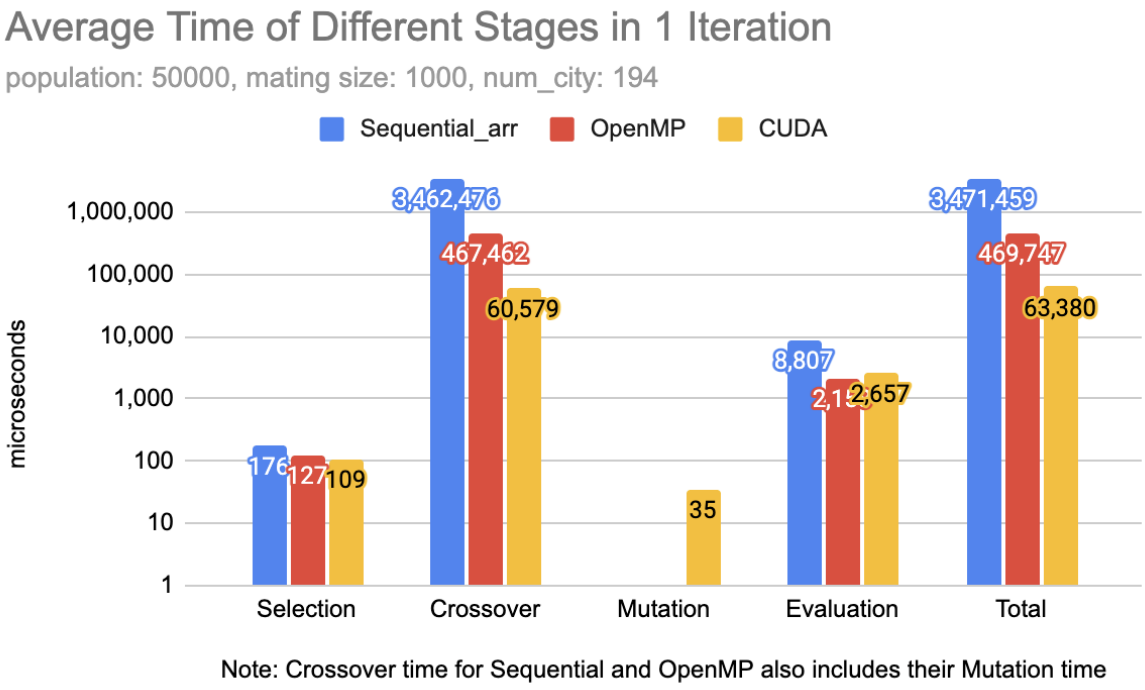
1. *Population size*: the number of chromosomes in the population
2. *Mating size*: the number of parents to be selected by the SELECTION stage which are used to produce the next generation of population.
3. *Num_city*: the number of cities in the input file.

To understand the impact of the workload to our implementations, we conducted 4 sets of experiments detailed in the following sections. The mutation probability for all experiments is 40%.

Average execution time of each stage

As the first step to understand the problem workload, we compared the execution time of the stages for each implementation. We used population size 50,000, mating size 1,000, and num_city 194, as this configuration is neither too easy or too hard for all implementations to compute and converge. As an implementation choice,

sequential_arr and OpenMP combined CROSSOVER and MUTATION into one function. Nonetheless, the chart shows that CUDA performs significantly better than OpenMP in most stages. This is because our problem heavily relies on data parallelism and compute power. GPUs are specifically designed for highly parallel computations. They have a higher degree of parallelism and with the many thread blocks, can perform more calculations in parallel than CPU, and are thus more suitable for this task.



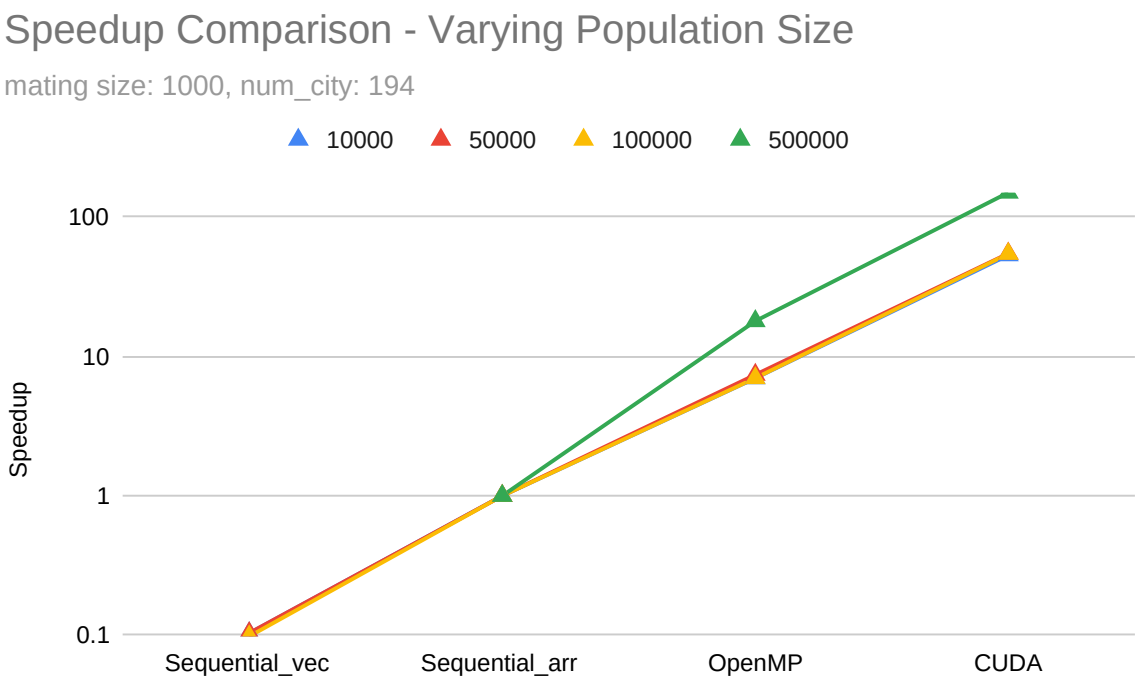
The only exception is for EVALUATION, which is different from what we expected, as each CUDA thread will be performing completely independent work. Upon further investigation, we found that one reason could be we used constant memory to store distance matrix, which will be fast if threads in the same warp accesses only a few distinct locations. When threads in a warp access different locations the reads are serialized, and thus the cost scales linearly with the number of unique addresses read by these threads. Our case falls exactly to the second situation, especially when the distance matrix is large. This is further confirmed in a later experiment of running with different num_city. Thus, one possible improvement would be to store the distance matrix in global or shared memory.

For SELECTION, since we chose a relatively small mating size, parallelization doesn't lead to significant improvement. We also experimented with different mating size but found that larger mating size doesn't

necessarily help with better convergence, but only increased SELECTION time, so we decided to use this size for all other experiments.

Another point is that CROSSOVER takes up the most time in an iteration, which is what we expected as this is the most computational heavy stage. Indeed, parallelization leads to the greatest performance improvement in this stage.

Speedup Comparison against varying population size

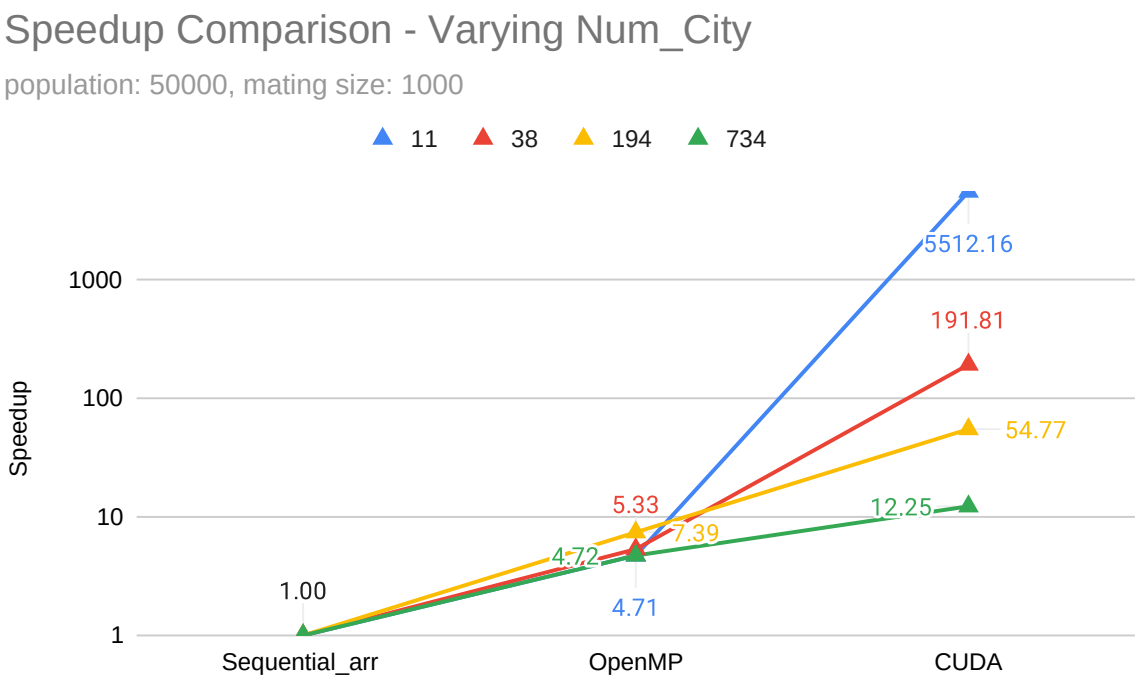


In this experiment, we investigated the effect of population size. We used single-threaded Sequential_arr run on GHC machines as our baseline, and the speedup is computed using the averaged total time of one iteration. We varied population size using 10,000, 50,000, 100,000, and 500,000. On average, OpenMP achieved around 7x speedup, and CUDA achieved around 54x. This is similar to what we expected as we used 8 threads for OpenMP. The code has few sequential sections, and all tasks within a stage have relatively balanced workload. However, the scheduling overhead still exists as the number of tasks is much larger than the number of threads. Thus, it has close, but not quite linear speedup. The significant CUDA speedup is also expected as this problem is amenable to parallelization.

Sequential_vec takes too long with 500,000 for even one iteration, which is why it doesn't have a data point. Based on the chart, it seems that OpenMP and CUDA have much better speedup with 500,000 population. In fact, however, Sequential_vec fails to converge within a reasonable amount of time. As mentioned above, the execution time of the stages tends to be longer before convergence, especially CROSSOVER. Therefore, the green line is comparing converged OpenMP and CUDA with non-converged Sequential_arr, which result in the seemingly better speedup line.

In general, the increased population size doesn't seem to affect speedup, which shows that the speedup the most likely limited by the fact that the stages and iterations must be run in sequential order (Amdahl's law). Moreover, 50,000 does tend to converge faster than 10,000, but larger population size won't make too much of a difference than 50,000. This indicates that for 194 cities, having 50,000 population might be enough for the search space of 194!. However, this number is only based on experiments and will certainly change with different number of cities and different implementations of the genetic algorithm.

Speedup comparison against varying num_city



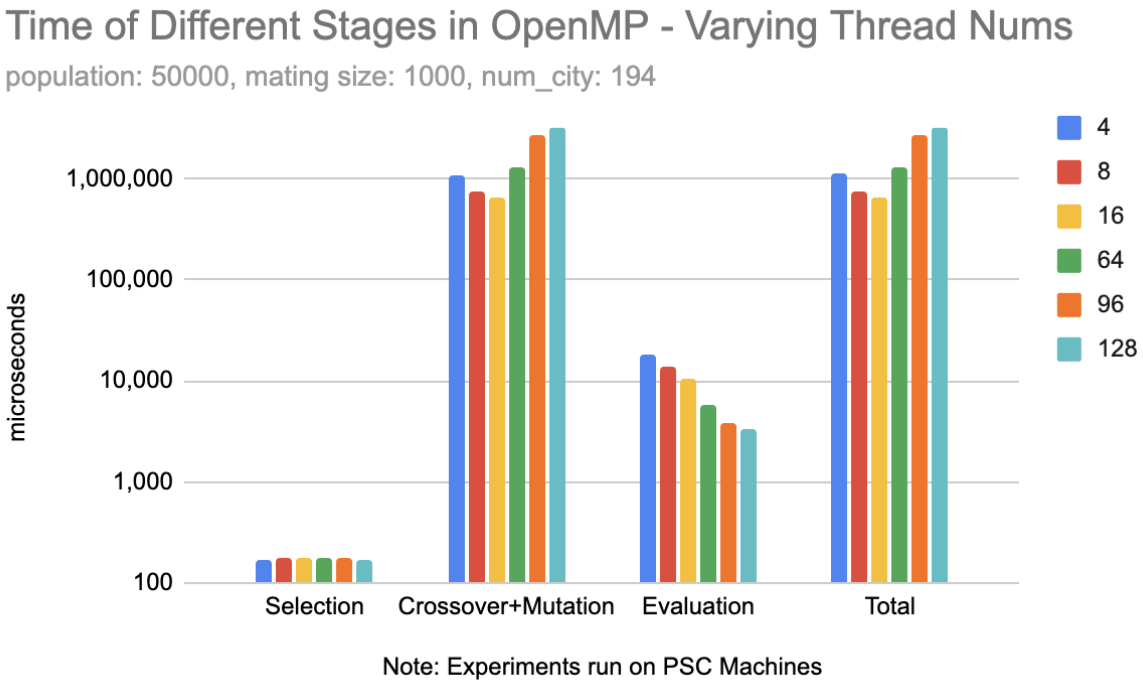
In this experiment, we investigated the effect of num_city. The speedup is computed similarly as in the previous experiment. We used 11, 38, 194, and 734 cities, 50,000 population size, and 1000 mating size.

For CUDA, the speedup decreased as the number of cities increased. One reason is that more cities lead to more work per thread, and also more contention to shared resources, such as the population and parents stored in global memory. The execution break-down shows that CROSSOVER increased the most (from 96 microseconds with 11 cities to 3,187,760 microseconds with 734 cities) while the increases in other stages remain in a relatively small range. On one hand, the calculation is much more complex within the thread, and on the other hand, each thread has to access three long arrays (two from parents, and one of its own), which will be copied from global memory and likely to be placed on local memory, leading to expensive memory access. Moreover, as mentioned above, each element in the array has to access twice the distance matrix, which are likely to be of different value before convergence. This could also lead to the slowdown, especially given that CROSSOVER becomes much faster after convergence. Therefore, to improve performance, other than using a different memory space to store the distance matrix, we could also try utilizing shared memory to store chunks of data accessed by the same thread block and write back to global population only after all threads within the block finished computation.

For OpenMP, the speedup remains generally stable for different number of cities. This suggests that it suffers less from the memory access problem in CROSSOVER as CUDA. Indeed, the distance matrix in OpenMP is stored in shared memory that is accessible by all threads. Thus, the increased execution time in CROSSOVER is largely due to increased computational complexity.

Thus, varying number of cities has a larger effect on the performance than other parameters. Small numbers converge quickly and generally have less memory access problems. Large numbers will expose those problems, which can quickly become the bottleneck of the speedup.

Analysis on OpenMP performance on Pittsburgh Supercomputing Center Clusters



When using GHC machines, the speedup of OpenMP improves as we increase the number of threads to the number of cores (8 in term of GHC.) Hence, we further experimented it on PSC machines that have 128 cores. However, the result is not as expected. Based on our tests, using 128 threads does not lead to best speedup. The above figure illustrates the time spent on different stages against different number of threads. Overall, when the thread number is set to 8, the program has the best performance.

The figure shows several interesting aspects here. First, the time taken for the EVALUATION stage does decrease as the number of threads increase. This is because in the evaluation stage, each chromosome needs to re-calculate its fitness score based on the new route (genome). This calculation is purely independent from other chromosomes, and there are way more chromosomes than threads. Hence, adding more threads does help reduce the time for evaluation.

Secondly, we noticed that the total time bar graph resembles the crossover + mutation bar graph. This stage is dominating the overall performance. Crossover is the stage to produce new generation based on two randomly picked parent chromosomes from the mating population. Since the mating population size is way smaller than the total population, to generate a population-size number of chromosomes concurrently, chromosomes from the mating population will be accessed simultaneously by multiple threads. This would cause contention for shared resources. If multiple threads are accessing shared resources, in our case, shared memory, this can lead to

contention and reduced performance. Also, the workload of each crossover may vary greatly due to randomness, a greater number of threads will also lead to poor load balancing. More threads may be idle while others are heavily loaded, leading to inefficiencies and reduced performance. Hence, we observe a U shape in this graph.

Conclusion and Reflection

In this project, we implemented and analyzed two different approaches to parallelize the genetic algorithm for solving the travelling salesman problem. Genetic algorithm itself is a perfect candidate for data parallelism from chromosome's perspective since most operations on chromosomes are independent to other chromosomes. However, the most complex operations are highly dependent from the gene's perspective. There are many different constraints that disqualify parallelism, such as the requirement that each gene can only appear once in one chromosome (following genes can only be determined after previous genes are determined.) After experiments, we find out that to fully exploit parallelism, we need to change the structure of the original genetic algorithm to expose parallelism opportunities. We managed to write our own genetic algorithm that is mostly data parallel. Hence, Our CUDA version has a dramatic performance gain compared to the other approaches in most cases. It also matches our knowledge that GPU is good for highly parallelized algorithms whose task are uniform as it can perform many operations simultaneously.

We also understand that to gain better performance, modifying the code for more parallelism is essential. Our OpenMP version had several major differences compared with the original design, such as combining crossover and mutation, using tournament selection instead of sorting. Hence, in some our cases, the shared-memory approach outweighs CUDA version thanks to fewer data movement, overhead and memory optimization.

Not all algorithms are born with parallelizable nature, it is of great importance to identify potential parallelizing patterns, modify the code structure to fully exploit parallelism for speedup.

Work distribution

Individual's contribution

Haoxuan Zhu (haoxuanz): Implemented TSP loader in Python; implemented sequential_vec version, sequential_arr version and parallelized the algorithm using OpenMP.

Ziyi Liu (ziyiliu): Adapted sequential_arr version and parallelized the algorithm using CUDA; Led performance and optimization analysis.

Together: Designed benchmark test sets, drafting the final report and poster

Distribution of total credit

Haoxuan Zhu (haoxuanz) 50% - Ziyi Liu (ziyiliu) 50%

Reference

[1] Abbasi, M., Rafiee, M., Khosravi, M.R. et al. An efficient parallel genetic algorithm solution for vehicle routing problem in cloud implementation of the intelligent transportation systems. J Cloud Comp 9, 6 (2020). https://doi.org/10.1186/s13677-020-0157-4

[2] Munroe S, Sandoval K, Martens DE, Sipkema D, Pomponi SA (2019) Genetic algorithm as an optimization tool for the development of sponge cell culture media. In Vitro Cell Dev Biol Anim 55(3):149–158

[3] TSP real city dataset https://www.math.uwaterloo.ca/tsp/world/countries.html