EC 527 - High-Performance Programming with Multicore and GPUs

Ant Colony Optimization for the Travelling Salesman Problem

Anirudh Sriram, Berk Gur, Lin Ma

December 20th 2018

Table of Contents

1 Description of the problem & the algorithm

1.1 The Travelling Salesman Problem

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one (e.g. the hometown) and return to the same original city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip.

We model the problem as a graph with n vertices that represent the cities. Let v0, v1, …, vn−1 be vertices that represent n cities, in a 2-dimensional space with given x and y coordinates. In this project, we assume that the distance between two cities is their Euclidean distance. Namely, each distance between cities i and j is
d(i, j)=d(j, i)= $\sqrt{(xi-xj)^2 + (yi-yj)^2}$.

To understand how the travelling salesman problem works, take the following example :

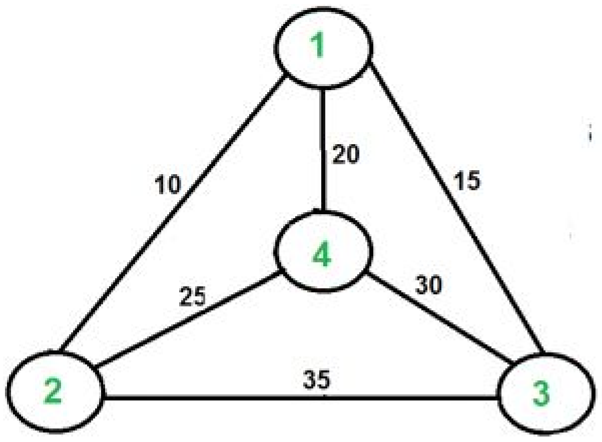


Figure 1.1 - Travelling Salesman Problem Example

The above graph has many possible paths that can be taken. If the Path 1-2-3-4-1 is taken, the distance adds up as 10+35+30+20 = 95. However, a more optimal path does exist - 1-2-4-3-1, which has a total cost of 80.

The Travelling Salesman Problem is an NP-Complete problem, which means algorithms that do find the optimal solution have exponential runtimes. However there also exist approximating algorithms, which find solutions with a very high probability of match to the optimal solution, for an extremely large number of cities, in a reasonable timeframe. The Ant Colony Optimization is one of these algorithms.

1.2 The Ant Colony Optimization

The Ant colony optimization (ACO) was introduced as a nature-inspired meta-heuristic for the solution of combinatorial optimization problems. This project aims to provide an approximating solution to the Travelling Salesman Problem using the Ant-Colony Optimization meta-heuristic. The idea of ACO is based on the behavior of real ants exploring a path between their colony and a food source. More specifically, when searching for food, ants initially explore the area surrounding their nest at random. Once an ant finds a food source, it evaluates the quantity and the quality of the food and carries some of it back to the nest. During the return trip, the ant deposits a chemical pheromone trail on the ground. The quantity of pheromone will guide other ants to the food source. Indirect communication between the ants via pheromone trails makes them possible to find shortest paths between their nest and food sources. In the ACO algorithm, these characteristics are implemented to solve optimization problems.

The ACO algorithm has the following two main steps:

1. Initialization

    i) Initialize the pheromone trail

2. Iteration

    i)For each ant repeat until stopping criteria

    a) Construct a solution using the pheromone trail
    b) Update the pheromone trail

The first step mainly consists in the initialization of the pheromone trail. In the iteration step, each ant constructs a complete solution for the problem according to a probabilistic state transition rule. The rule depends chiefly on the quantity of the pheromone. Once all ants construct solutions, the quantity of the pheromone is updated in two phases: an evaporation phase in which a fraction of the pheromone evaporates, and a deposit phase in which each ant deposits an amount of pheromone that is proportional to the fitness of its solution. This process is repeated until stopping criteria.

2. CPU Implementation

2.1 Serial Ant Colony Optimization

2.1.1 Initialization

In the initialization stage, a structure of cities and ants is created. The ants have the following properties:

- Current City - The integer value of the current city the ant is in.
- Next City - The integer value of the next city the ant needs to go to.
- Visited Cities - An integer vector containing all the cities that have already been visited
- Tour - An integer vector which will contain all the cities that the ant has visited in its entire tour

The structure for cities has the following properties:

- X-Coordinate - X-Coordinate of the given city in the graph
- Y-Coordinate - Y-Coordinate of the given city in the graph

Initialized Variables:

- Delta_Pheromones - This two-dimensional array will contain the change in the pheromone levels, which will be used to update pheromones at the end of each iteration. The row, and column elements in this two-dimensional array is the two cities between which the pheromones will be updated. Initialized to 0.
- Distance - Two-dimensional array - This two-dimensional array contains all the distances between all cities that are connected. Initialized using $\sqrt{(xi-xj)^2 + (yi-yj)^2}$.
- Pheromone - Two-dimensional array - Contains the current pheromone levels between cities. Initialized to 0.

Initialized Tour:

The tour for all the ants is also initialized by randomly assigning an ant a starting city, and setting all the other variables for each of the ants to zero.

2.1.2 Tour construction - Selecting the Next City

In each construction step, each ant moves, based on a probabilistic decision, to a city it has not yet visited. This probabilistic choice is based on the pheromone trail $\tau_{ij}(t)$ and by a locally available heuristic information η_{ij} , a variable for visibility. The latter is a function of the path length; For the TSP, $\eta_{ij}=1/d_{ij}$. Ants prefer cities which are close and connected by paths with a high pheromone trail and for any given ant *k* currently located at city *i,* the ant will choose to go to a city *j* with the following given probability:

$$p_{ij}^{k} = \begin{cases} \dfrac{[\tau_{ij}]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{s \in allowed_k} [\tau_{is}]^{\alpha} \cdot [\eta_{is}]^{\beta}} \\ 0 \end{cases}$$

Figure 2.12 - The probability for a given ant k to travel from city i to city j

The method used above by the ants to select the next city for its tour is known as the roulette-wheel selection. Here the symbols represent the following:

- τij is the intensity of the pheromone trail between cities.
- α the parameter to regulate the τij
- nij is the visibility of the city I = 1/dij, where dij is the distance between city i and j.
- β the parameter to regulate the influence of nij
- Allowed k is the set of cities that have not been visited

Here is an example of the probability decision making by the simulated ants:

Scenario 1:

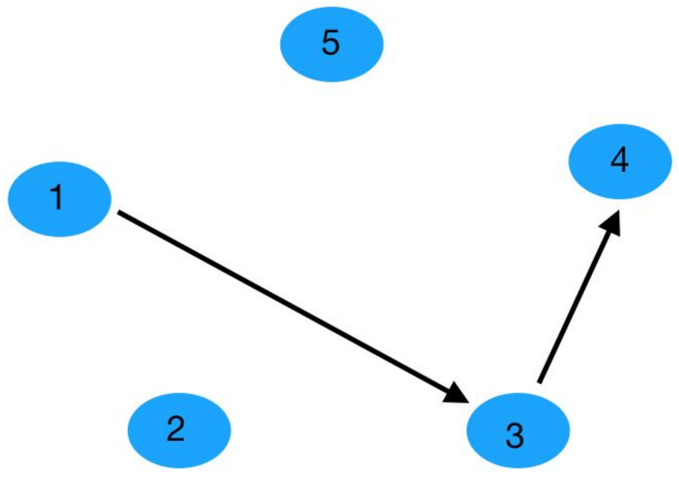


Figure 2.1 - A simulated ant has started from City 1, and then travelled to City3, and City 4, in that given order

In this given example, there is a higher probability of the ant choosing the shorter distance if there happens to be the same amount of pheromones between the different cities. At city 4, the ant will more likely choose to go to city 5 rather than to city 2 given that there is the same amount of pheromones between city 4 & city 5, and between city 4 & city 2.

Scenario 2:

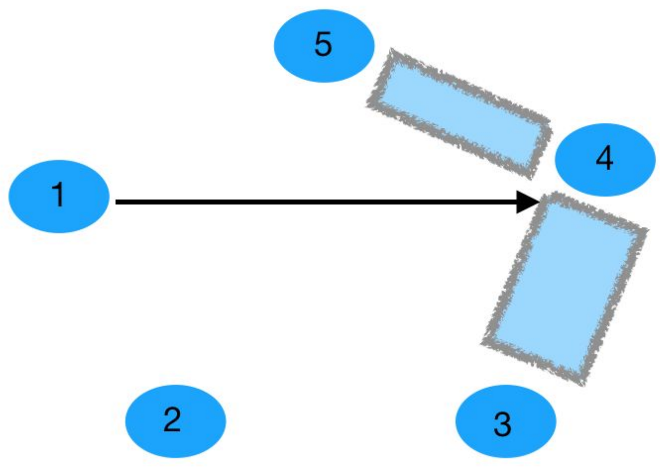


Figure 2.2- A simulated ant has started from City 1, and then travelled to City3, and City 4, in that given order.

In this example , there is a higher probability of choosing the path with more pheromones when the distances are the same between all cities. If the size of represents the quantity of pheromones, an ant here will have a higher probability of choosing city 3 than city 5.

2.1.2.1Calculating the Probability

To calculate the probability a PHI_numerator function is first used, which calculates the numerator of the probability function in Figure 2.12. It takes as input two cities, i and j, and calculates the pheromone levels between those two cities, and uses the distance from the two-dimensional distance array to calculate the visibility. The Alpha and Beta values are constants which are pre-defined to regulate both pheromone level and the visibility.

2.1.2.2 Returning the Next City

To calculate the next city for all the ants, the function nextCity is used. This function takes as input an ant k, and the number of cities.This function first sums up the PHI numerator for all the cities the given ant k has not visited. This acts as the denominator for the probability function. Next all the cities the ant has yet to visit, are iterated through. For the chosen city the probability of going from the current city to the particular city is calculated. Then a randomized number is generated between 0 and 1, which would act as a threshold. If the probability for the given city passes the threshold, the city gets added to the tour for the particular ant.

2.1.3 Tour construction - Creating a TSP Tour

2.1.3.1 Tour Construction

In the tour construction, m ants are initially positioned on n cities chosen randomly.
In the function, a for loop is used to iterate through all the cities in the map. Inside this for loop, another for loop is used to iterate through all the ants in the map. For each ant, using the nextCity function, the next city is calculated for its tour, and for each ant its attributes are updated. This continues until all ants have completely covered all the cities in the map.

2.1.3.2 Ending Tour

To end the tour for all the ants, since the Travelling Salesman Problem is considered in this case, each of the ants has to travel back from the last city back to its original city.In this function the total length covered by each ant is calculated, and the best tour is calculated based on whichever ant could cover the least distance. After this, each of the ants updates the Delta_Pheromone two-dimensional array. In this two-dimensional array, the row and column values represent the different cities, and hence for a given element Delta_Pheromone[i][j], the amount of pheromones between City i and City j is stored. The amount of pheromones stored in the two-dimensional array depends on Q, a predefined constant, and the total length of the tour taken by the given ant. The value of Q is divided by the tour length of the given ant, and this value is added to the particular Delta_Pheromones element. These Delta_Pheromone values are useful in the later pheromone update stages of the algorithm.

2.1.4 Pheromone Update

After all ants have completed the tour construction, the pheromone trails are updated. This is done first by lowering the pheromone trails by a constant factor (evaporation) and then by allowing the ants to deposit pheromones on the paths they have visited. In particular, the update follows this rule:

$$\tau_{ij}(t+1) = \rho \cdot \tau_{ij}(t) + \Delta\tau_{ij}$$

$$\Delta\tau_{ij} = \sum_{k=1}^{l} \Delta\tau_{ij}^{k}$$

$$\Delta\tau_{ij}^{k} = \begin{cases} Q/L_k & \text{if ant } k \text{ travels on edge } (i,j) \\ 0 & \text{otherwise} \end{cases}$$

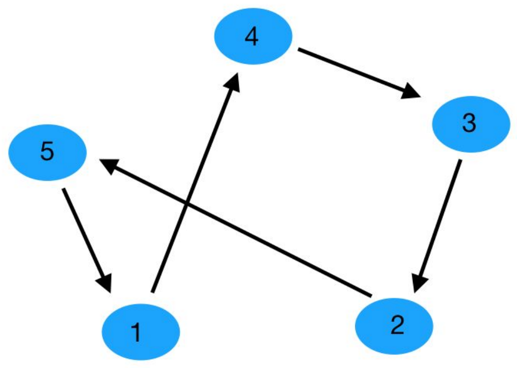
Here, the symbols represent the following:

- ρ: is the evaporation factor
- τij(t): pheromone at time t
- Δτij: the change of pheromones which equals to the sum off all the pheromones left by ants from l to k
- Δτkij: if k has to travel on path (i,j). Q/Lk is the amount it changes and 0 otherwise

Here is an example of update pheromone in action:

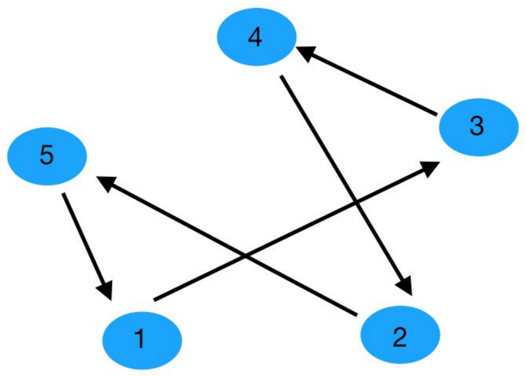
1st Iteration:

Ant 1



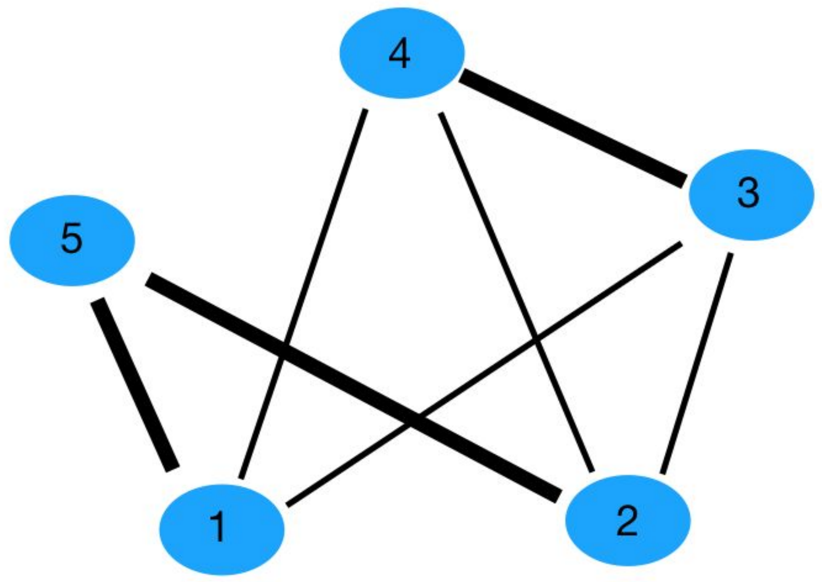
1 -> 4 -> 3 -> 2-> 5 Distance = 45

Ant 2



1 -> 3 -> 4 -> 2-> 5 Distance = 55

2nd Iteration:



- Given the starting city is 1,the new ant choosing to go to city 5 is more likely, due to the higher amount of pheromones as well as its closer distance compared to its existing neighbors.
- After picking city 5, city 4 can be selected due to its closer distance, city 2 can be selected due to a greater pheromone amount or city 3 can be selected due to pure chance.
- If the ant selects city 4, then it will have lesser distance to cover and as a result a higher value on the updated pheromone.
- After city 4, the ant will have the greatest probability to choose city 3 due to the short distance and the high amount of pheromones.

After N Iterations:

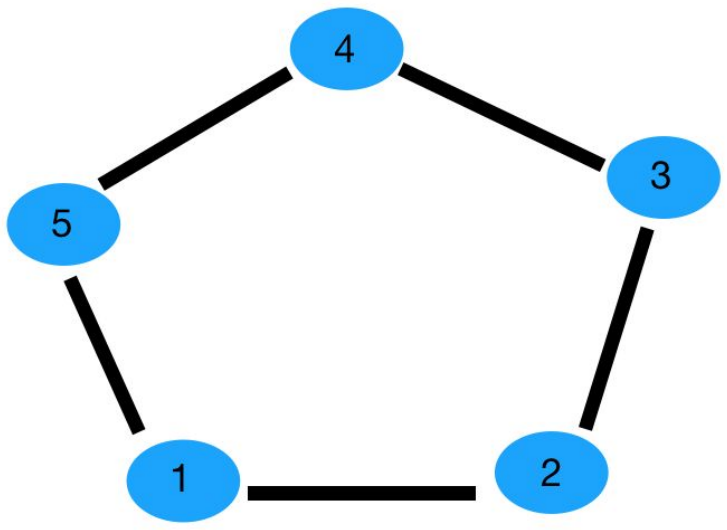


Figure 2.1.4 - Illustration of the convergence

After N iterations the program will find the optimal path for the given graph.

The pheromone update function implements the formula in Figure 2.14, using the Delta Pheromone values calculated in the tour construction phase, and uses the RHO evaporation constant, to update the two-dimensional array, pheromones. The values for the row and column in the pheromones two-dimensional array is representative of the cities, and the values in the matrix of the pheromone level.

3. OpenMP Implementation

The OpenMP version of the ACO algorithm leverages the powerful abstraction OpenMP has. During our implementation process we realized controlling the number of threads using the omp_set_num_threads() command was not a good method, as the program runtime consistently exceeded that of the serial implementation during the debugging phase. It was also found that only 1 thread was actually running. Omp_set_num_threads() overrides the value of the environment variable OMP_NUM_THREADS, however this environment variable only controls the upper limit of the size of the thread team OpenMP spawns for the parallel regions. The tweak that fixed the issue was disabling dynamic teams, as this was leading to potentially picking smaller number of threads if the run-time system decides its more appropriate. By changing this, we had absolute control over our parallelization parameter (number of threads), by setting omp_set_dynamic(0);

The safe parallelization opportunities arose in the following functions: initialize(), initializetour() ,tourconstruction() and endtour(). For these functions, it was possible to let the OpenMP run-time divide the iteration space to the selected number of threads without worry about race conditions.However as seen in figure 4.1.1 updatePheromone() function creates the critical section problem. The pheromone matrix is being accessed and updated by multiple threads and hence without the pragma omp critical envelope around its update, the update may not properly occur. The reason for choosing pragma omp critical over atomic is because atomic only protects one assignment and brings the constraint that only specific operations can be used. However the update loop has conditionals and one multiplication followed by an addition(on the condition that city index i and j are not same).

Additionally, as a side tweak to the critical section problem we enforce flushes on the shared memory. Although the flush operation is not performance friendly we needed to be sure that each thread had the same view on the pheromone and delta pheromone matrices.

Unfortunately the openmp code is slower than the serial code, one reason for this could be false sharing,if the pheromone matrix elements are sitting in the same cache line each update to these matrix indices could be potentially causing cache lines to slosh back and forth between threads. Also the next city calculation is not using the SIMD architecture as it updates scalar probabilities. Perhaps if we were to convert the probabilities into a vector and have individual threads update this vector independently we could have explored an improvement. The performance of cuda is very successful compared to this, as cuda involves fine tuning. We have utilized a 32x32 threads per block structure on a [(number of cities-1)/32]+1 by [(number of cities-1)/32]+1 grid, which in turn allowed us to utilize fine grain parallelism.Also the biggest bottleneck in openmp code was the tourConstruction() function. We used dynamic scheduling for the partition of this particular iteration space, since the iterations take varying amount of times(because the calculation of next city is based on a random thresh-hold number and if this number is not passed, we keep adding probabilities to pass it, so each iteration is a random process) however this might have caused a load imbalance and thus might have forced most of our threads to become idle.

4. GPU CUDA Implementation

4.1 Ant Colony Optimization CUDA Version

4.1.1 Overview

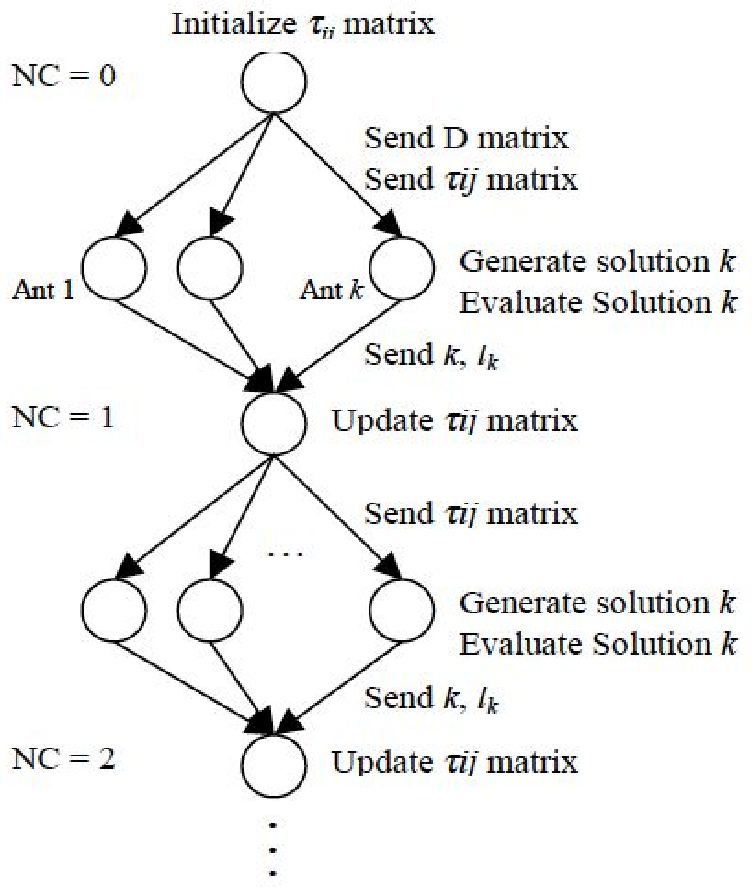


Figure 4.1.1 - Showcases overview of CUDA Implementation

The idea of the CUDA implementation is that there are multiple ants, which calculate their tours independentantly, and update the pheromones independently in each iteration. All of the ants then combine and update the final pheromone two-dimensional array. This is repeated for each iteration. Our implementation consists of three CUDA parts, initialization, tour construction, and pheromone update. We describe the details of them as follows.

4.1.2 Initialization

The initialize phase runs on global memory. The distance, update pheromone, and the pheromone matrices are initialized in parallel. Each column and row for the matrices is assigned an individual thread. Initialized random seeds for CURAND are used in processes performed later in the program. CURAND is a library that provides a pseudorandom number generator on the GPU by NVIDIA.

To initialize the tour, each ant is assigned a thread, which then initializes all of its parameters.

4.1.3 Tour Construction

Recall again that in the tour construction, m ants are initially positioned on n cities chosen randomly. Each ant makes a tour with roulette-wheel selection independently. Hence a thread is assigned to each ant again for the tour construction. Whenever each ant visits a city, it determines which city to visit with roulette-wheel selection. Let us consider the case when ant k is in city i. In advance, the numerator for the probability function is calculated for all the possible cities and stored to the 2-dimensional array in global memory. Also, the elements related to city i are stored in the same row so that the access to the elements can be performed with coalesced access. In the tour construction, each ant k makes a tour index array tk such that element tk(i) stores the index of the next city from city i. The following image shows what this looks like:

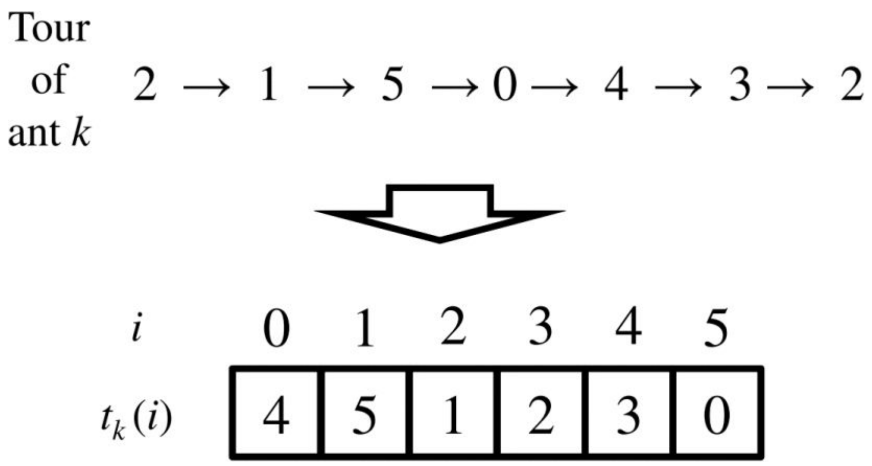

Tour of ant k
$2 \rightarrow 1 \rightarrow 5 \rightarrow 0 \rightarrow 4 \rightarrow 3 \rightarrow 2$
i
0 1 2 3 4 5
$t_k(i)$
4 5 1 2 3 0

Figure 4.1.3 - Tour Construction

The method of selecting the next city works in the following way:

Each ant iterates through all possible cities, and if a given city has not been visited, it advances to find the probability of selecting that city.

Next, to perform the roulette-wheel selection, when ant k is in city i. we compute this as follows;

1. Calculate the probability sums (denominator of probability function)
    a. Calculate the denominator sum for adjacent cities using the below formula:

$$\sum_{s \in allowed_k} [\tau_{is}]^{\alpha} \cdot [\eta_{is}]^{\beta}$$

b. Divide the numerator value calculated previously for a given city j from city i, by the sum in step a to get the probability.

c. Generate a random threshold value x in [0,1] using CURAND

d. Find the city j that surpasses the threshold

This roulette-wheel selection takes place through the nextCity function. Each ant is assigned a thread for the tour creation. For all the cities, each thread calculates the next city, and then adds it to the ants tour. To complete the tour the endTour function, again assigns each ant a thread, which then adds its corresponding original city into its tour. The ant with the minimum tour length is found using the atomicMin() function. The endTour function also updates the delta pheromones two-dimensional array, by adding the pheromones deposited by each of the ants.

4.1.4 Pheromone Update

Recall that pheromone update consists of factors involving pheromone evaporation and pheromone deposit. In our implementation, the values of pheromone $\tau(i, j)(0 \leq i \leq j \leq n-1)$ are stored in a 2-dimensional array, which is a symmetric array, that is, $\tau(i, j)=\tau(j, i)$, in the global memory and are updated by the results of the tour construction. Also, each total tour length of each ant $L_0, L_1, \ldots, L_{m-1}$ stored in the global memory is read. To update the pheromones, the updatePheromone() function is again used. In this function each row of the delta pheromones two-dimensional array is assigned to a thread. Each thread then adds all the elements in its corresponding row to the pheromones two-dimensional array.

The elements related to city i, i.e., $\tau(i, 0), \tau(i, 1), \ldots, \tau(i, n-1)$, are stored in the same row so that the access to the elements can be performed in a coalesced fashion. After the addition, the values of pheromones are stored back to the global memory.

All of the above stages, and their corresponding kernels are repeated for N iterations. Each of the functions has a gridDim of (n-1)/32, and BlockDim of 32, where n is the number of cities in the graph.

5. Results

For this project we used the NVIDIA Tesla K40m. We have used the Intel(R) Core(TM) i7-2635QM CPU running at 2.00GHz with 16GB memory to run the sequential implementation of the ACO algorithm. We have evaluated our implementation using a set of benchmark instances from the TSPLIB library. In the following evaluation, we utilize 9 instances: lin8, bays29, att48, kroC100, bier127, kroB200, lin318, pr439, pr1002 from TSBLIB. Each name consists of the name of the instance and the number of cities. The parameters of ACO, α, β, and ρ in Figure 2.12 and Figure 2.14, are set to 0.5, 0.8, and 0.5, respectively. The value of Q, which is divided by the ant tour length & then added to the Update Pheromones matrix, is set to 80. The number of iterations the program runs is set to 30. The MAX_CITIES & MAX_ANTS, are changed for each of the instances. For consistency, the MAX_CITIES is set to the input number of cities, and the MAX_ANTS is set to floor(MAX_CITIES/2).

5.1 Evaluation of Overall Performance (CUDA)

Number of Cities	**Time(s) (Serial)**	**Time(s) (CUDA)**	**Speedup**
8	0.00198	0.2	0.0099
29	0.098958	0.61	0.16222623
48	0.439157	0.7	0.62736714
100	3.98089	1.35	2.94880741
127	7.83249	1.82	4.30356593

200	32.466	3.77	8.61167109
318	64.0243	8.96	7.1455692
439	172.622	16.77	10.2935003
1002	1990.04	102.81	19.3564828

Figure 5.1 - Evaluating Overall Performance (CUDA)

5.2 Evaluation of Overall Performance (OpenMP)

Number of Cities	**Time(s) (OpenMP)**	**Time(s) (Serial)**	**SpeedUp**
8	0.05	0.00198	0.0396
29	0.24	0.098958	0.412325
48	0.62	0.439157	0.70831774
100	4.36	3.98089	0.91304817
127	10.43	7.83249	0.75095781
200	59.66	32.466	0.54418371
318	307.24	64.0243	0.2083853
439	880.11	172.622	0.19613685
1002	7922.74	1990.04	0.25118078

Figure 5.2 - Evaluating Overall Performance (CUDA)

5.3 Execution Time Comparison CUDA vs Serial

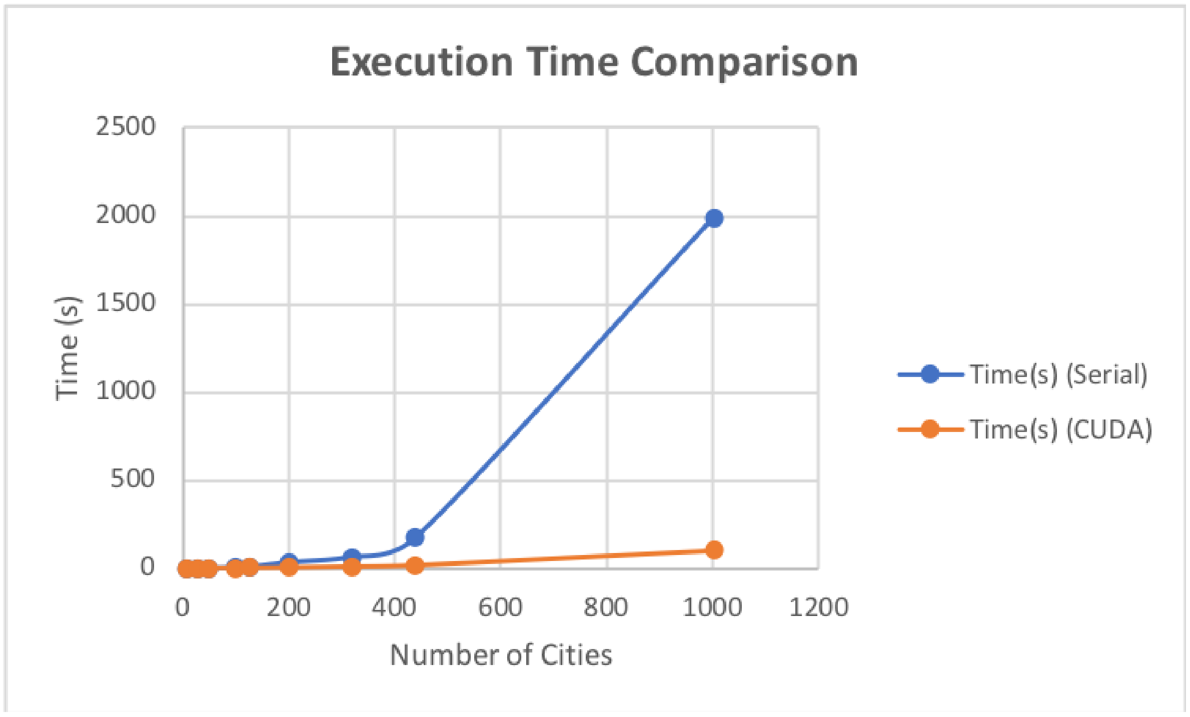


Figure 5.3 - Evaluating Execution Time (CUDA)

For the execution time, it can be clearly seen that after the number of cities exceed about 100, the CUDA implementation starts to have a clear advantage on runtime over the CPU implementation. The maximum speedup we achieve over the CPU implementation is roughly 19.4. This is very close to the theoretical maximum speedup of 20, based on Amdahl's Law!

5.4 Execution Time Comparison OpenMP vs Serial

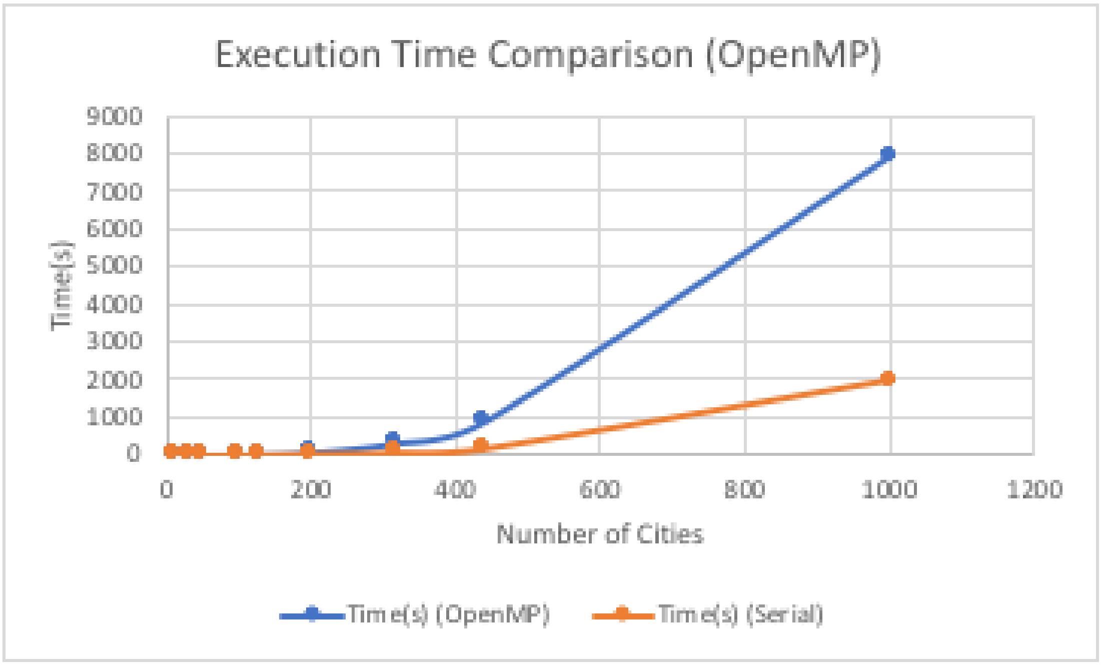


Figure 5.3 - Evaluating Execution Time (OpenMP)

From the implementation of the OpenMP program it was discovered that the runtime for the OpenMP version is slower than the serial implementation. One reason for this could be false sharing. If the pheromone matrix elements are sitting in the same cache line, each update to the matrix indices could be potentially causing cache lines to slosh back and forth between threads. Also the next city calculation is not using the SIMD architecture as it updates scalar probabilities. Perhaps if the probabilities were to be converted into a vector and have individual threads update this vector independently we could have explored an improvement. The performance of the CUDA implementation is more successful comparatively, as CUDA involves fine tuning. We have utilized a 32x32 threads per block structure on a [(number of cities-1)/32]+1 by [(number of cities-1)/32]+1 grid, which in turn allowed us to utilize fine grain parallelism.

Another big bottleneck in the openmp implementation was the tourConstruction() function. We used dynamic scheduling for the partition of this particular iteration space, since the iterations take varying amount of times(because the calculation of next city is based on a random threshold number and if this number is not passed, we keep adding probabilities to pass it, so each iteration is a random process) however this might have caused a load imbalance and thus might have forced most of our threads to become idle.

5.5 Best Tour Length Comparison (CUDA)

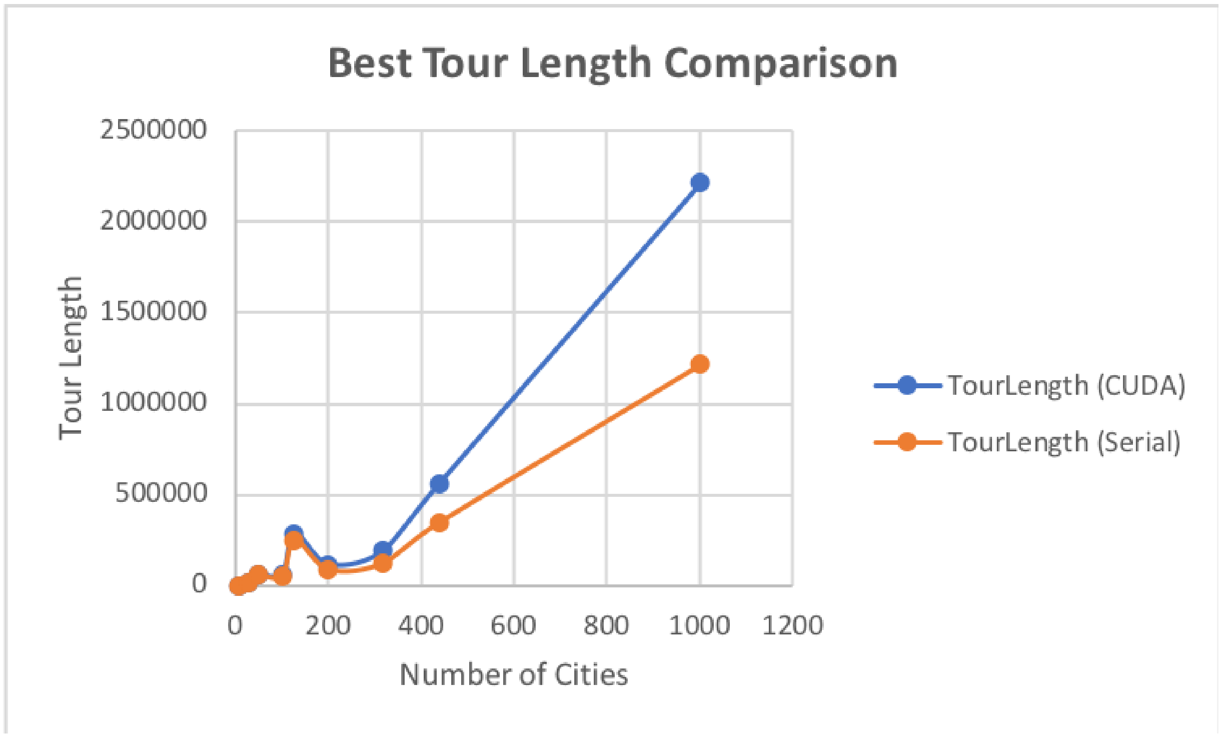


Figure 5.5 - Evaluating Best Tour Length (CUDA)

Although it would be expected that the optimal tour length be approximately the same for both the CPU and GPU versions of the code, there could be certain factors affecting the results which have caused the optimal length to be a lot longer for the CUDA version of the code. One of the most probable reasons for the discrepancy in the results could be owing to the fact that CPUs tend to do floating point calculations in an 80-bit 'extended' mode and keep the results in this intermediate format. As such, subsequent calculations are also using the 80 bit value. On the GPU the single precision is 32 bit and double precision is 64 bit. Hence doing multiple floating point operations with these discrepancies would lead to disparate results.

5.6 Best Tour Length Comparison (OpenMP)

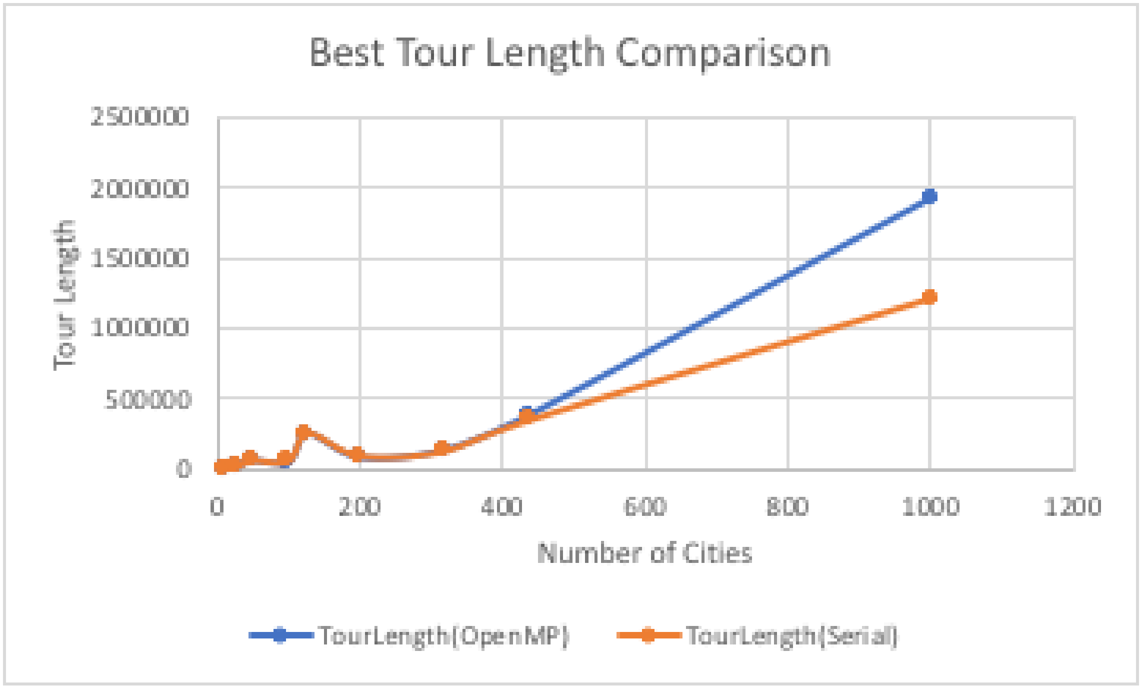


Figure 5.6 - Evaluating Best Tour Length (OpenMP)

In this case, the best tour length can be seen to be very comparable for both the OpenMP and the serial code up until the number of cities in the map is 1002. However the optimal length is a greater value for the OpenMP version, compared to the serial version. This value could be an outlier, since all of the previous values calculated before were fairly similar.

5.7 Speedup Comparison CUDA Over Serial

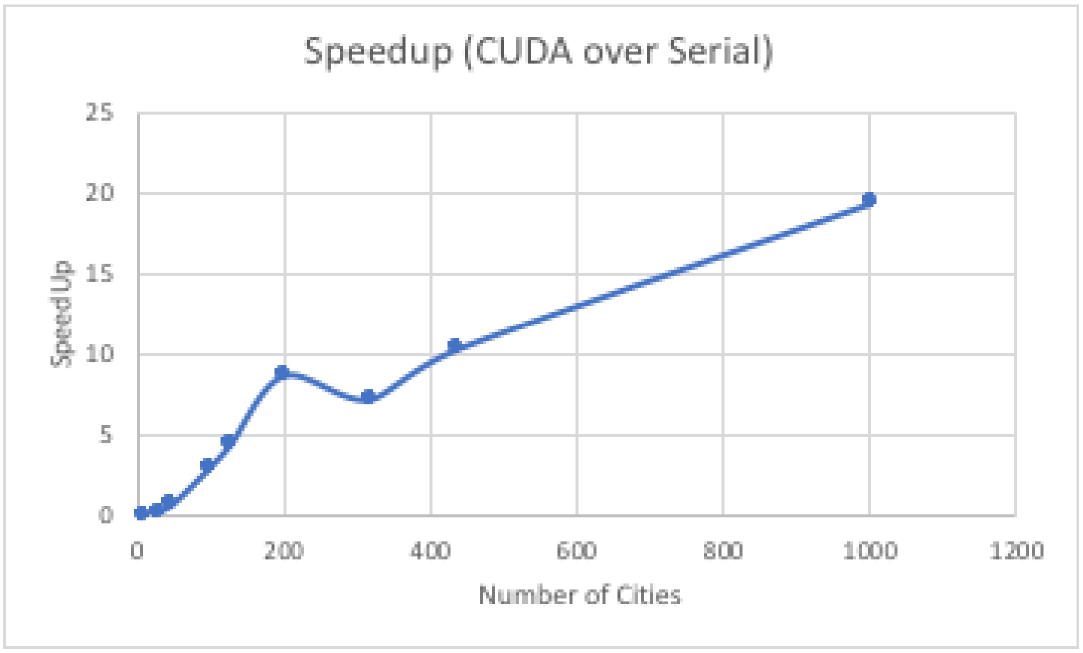


Figure 5.4 Speedup CUDA Implementation Over CPU

As seen in the above figure, the SpeedUp for the CUDA implementation reaches a maximum of roughly 19.4 over the CPU implementation, as the number of cities in the graph increases to 1002.

5.8 Speedup Comparison OpenMP Over Serial

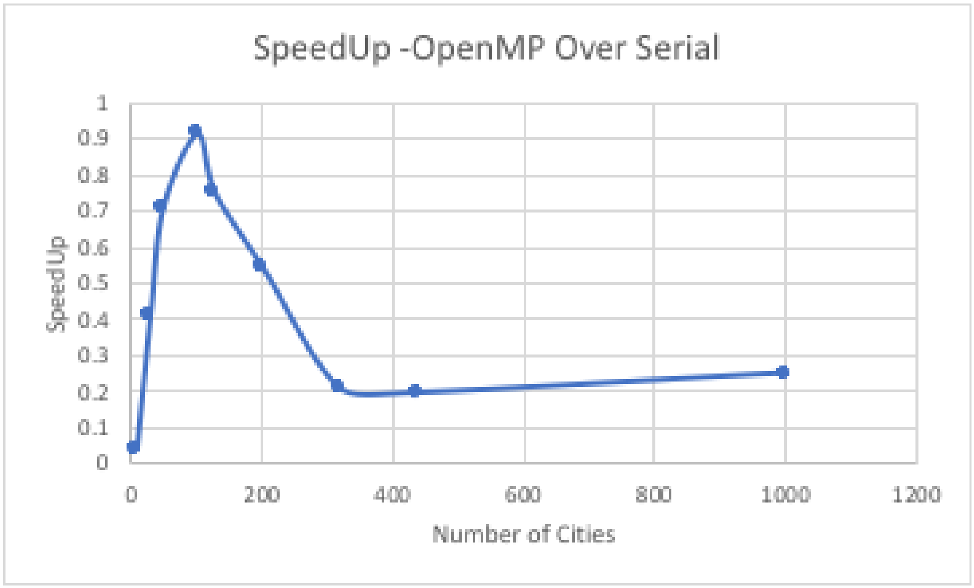


Figure 5.5 Speedup OpenMP Implementation Over Serial

As can be seen from the above figure, the SpeedUp for the OpenMP version, is below 1 for all the number of cities used as input.

6. Possible Further Improvements

6.1 GPU Implementation

Greater usage of Shared Memory- In the Update pheromones kernel, threads in block i which read τ(i, 0), τ(i, 1), …, τ(i, n−1) in the i-th row could've been stored to shared memory.

Avoiding Branch Instructions - To avoid the branch instruction in the tour construction, the implementation suggested in Figure 5.1 could have been used. In our case we individually check if each city has been visited or not. The implementation here is more efficient. Here the fitness values are equivalent to the probability numerators in our case, and is represented by f(i,j). Here the numerator values are multiplied by a visited vector, which contains elements which are either 1 or 0. If a city has been visited by a particular ant, the element of the corresponding index would be set to 1. After the multiplication, a new vector Prefix Sums would contain a sum of all the values below and equal to its index in the products vector. After this the random value is generated using CURAND. This value is compared with the sums vector to see the first index where the threshold is surpassed. Here a parallel search method based on the parallel *K*-ary search is used . The idea of the parallel *K*-ary search is that a search space in each iteration is divided into K partitions and the search space is reduced to one of the partitions. This index is the next city added to a given ant's tour. This method uses an in-place parallel prefix sum algorithm with the shared memory on the GPU.It can also avoid most bank conflicts by adding a variable amount of padding to each shared memory array index. This implementation would have looked like the following:

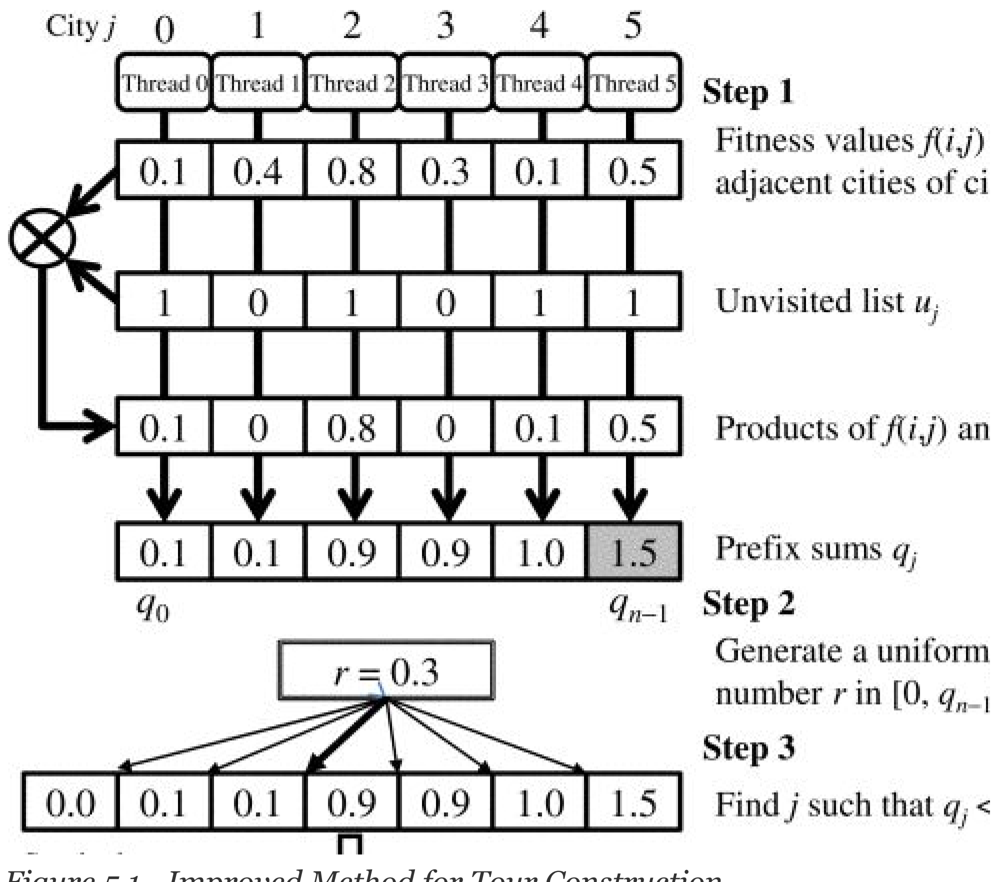


Figure 5.1 - Improved Method for Tour Construction

The feature of using the above method is that the fitness values can be read from the global memory with coalesced access.

7. Conclusion

In this paper, we have proposed an implementation of the ant colony optimization algorithm for the traveling salesman problem on the GPU using CUDA, as well as an OpenMP implementation.

Although through the CUDA version of the algorithm we achieve very significant speedups over the regular serial implementation, there is an opportunity cost. With the increased speedup, we also lose the accuracy in delivering optimal solutions for the Travelling Salesman Problem. The reduction in accuracy is owing to the discrepancies between floating point calculations in GPUs and CPUs.

For the OpenMP Implementation it was found that although the Optimal Routes were comparable for most of the inputs, the runtime for the serial reference code was much lower.

8. Future Works

ACO algorithms have proven to be successful in solving many academic and industrial combinatorial optimization problems. However, faced with extremely large and hard problems, they need a considerable amount of computing time and memory space to be effective in their exploration of the search space. Hence our plans for future works include GPU implementations for improved versions of the ant colony optimization such as the Max-Min Ant System (MMAS). The MMAS is generally recognized as one of the most effective versions of the ACO algorithms in this age. The MMAS differs from the regular ACO algorithm in that the MMAS algorithm achieves a strong exploitation of the search history by allowing only the best solutions to add pheromone during the pheromone trail update. MMAS is currently one of the best performing ACO algorithms for the TSP.

9. List of Files

- serial.cpp - Contains the serial implementation of the ACO Algorithm
- serial_omp.cpp - Contains the OpenMP implementation of the ACO Algorithm
- parallel.cu - Contains the CUDA implementation of ACO Algorithm
- lin8, bays29, att48, kroC100, bier127, kroB200, lin318, pr439, pr1002 - tsp files

10. Running the Files

10.1 CPU Code

To run the Serial code use the following command:

```
>g++ serial.cpp
>./a.out lin8.tsp (As an example)
```

Before you compile the program, make sure to change the NUM_CITIES = 8 (in this case), and NUM_ANTS = 4 (floor(NUM_CITIES/2)).

To run the OpenMP code use the following command:

```
> g++ serial_omp.cpp -fopenmp
>./a.out lin8.tsp (As an example)
```

Again make sure to change NUM_CITIES and NUM_ANTS.

10.2 GPU Code

To run the CUDA code use the following command:

```
> nvcc parallel.cu
> ./a.out lin8.tsp (As an example)
```

Again make sure to change NUM_CITIES and NUM_ANTS.