Parallel and Distributed Systems: Paradigms and Models

Ant Colony Optimization Traveling Salesman Problem

Javad Khalili
Matricola : 546677

under the direction of
Prof. Marco Danelutto

Department of Computer Science

University of Pisa

April 17, 2019

1 Introduction

In this project Ant Colony Optimization (ACO) is used for solving Traveling Salesman problem. This algorithm is used to produce near-optimal solution for TSP.

The traveling salesman problem (TSP) is described as given a list of cities and their pairwise distance, find the shortest possible tour that visits each city exactly once and then returns to original city.A variety of heuristic algorithms have been devised to produce approximate solutions good enough for application in TSPs. ACO and it's variations is among the heuristic algorithms that have been widely used in solving various cases of TSPs.
ACO is a multi-agent system simulating the searching actions of some ant species, i.e., ants deposit pheromones on the paths in order to mark the favorable trail that can be followed by other members of the colony.The shortest route is eventually taken by all the ants to transport food from the source to the nest the most efficiently. ACO simulates such a process using a number of artificial ants to build-up the shortest route as the solution to a given optimization problem.

2 Traveling Salesman Problem

Traveling Salesman problem(TSP) is one of the well-known and extensively studied problems in discrete or combinational optimization and asks for the shortest roundtrip of minimal total cost visiting each given city(node) exactly once.
A complete weighted graph $G = (N, E)$ can be used to represent a TSP, where N is the set of n cities and E is the set of edges(paths) fully connected all cities. Each edge $(i, j) \in E$ is assigned a cost d_{ij}, which is the distance between cities i and j. d_{ij} can be defined in the Euclidean space and is given as follows:

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

3 Ant Colony Optimization

3.1 Ant System

Initially, each ant is randomly put on a city. During the construction of a feasible solution, ants selects the following city ot be visited through a *probabilistic decision rule*. When an ant k states in city i and construct the partial solution, the probability moving to the next city j neighboring on city i is given by

$$p_{ij}^k = \begin{cases} \frac{[\tau_{ij}(t)]^\alpha [\eta_{ij}]^\beta}{\sum_{u \in J_k(i)} [\tau_{iu}(t)]^\alpha [\eta_{iu}]^\beta} & , \quad if \quad j \in J_k(i) \\ 0 & , \quad otherwise \end{cases}$$

where τ_{ij} is the intensity of trails between edge (i, j) and η_{ij} is the heuristic visibility of edge (i, j), and $\eta_{ij} = \frac{1}{d_{ij}}$. $J_k(i)$ is a set of cities which remain to be visited when the ant is at city i. α and β are two adjustable positive parameters that control the relative weights of the pheromone trail and of the heuristic visibility. After each ant completes its tour, the pheromone amount on each path will be adjusted with following equation:

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

In this equation,

$$\Delta\tau_{ij}(t) = \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t)$$

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} \frac{Q}{L_k}, & if(i,j) \in \text{tour done by ant k} \\ 0 \quad , & otherwise \end{cases}$$

$(1-\rho)$ is the pheromone decay parameter $(0 < \rho < 1)$ where it represents the trail evaporation when the ant chooses a city and decide to move, L_k is the length of the tour performed by ant k and m is the number of ants.

3.2 Edge Selection

In this project Roulette Wheel Selection Algorithms has been used for selecting the next city to be visited. The algorithm is as follow:

Algorithm 1: Roulette Wheel Selection

```
r = random number , where 0 ≤ r < 1;
sum = 0 ;
foreach i do
    sum = sum + Prob(i);
    if r < sum then
        return i;
        break;
    end
end
```

4 Algorithms

In the project ACO algorithm is implemented in sequential and parallel using threads. In the following you can find both the sequential and parallel implementation psudocode.

4.1 Sequential Algorithm

After initialization of parameters, in each iteration a number of tours are constructed by the ants by calling the function $findig\_Paths()$ which create a vector holding the tours constructed by ants to be used for updating pheromones on the edges of the graph that is taken by ants in their tours and finding the best tour. Operations of updating pheromones of edges is done by calling $Phermones\_Update()$ function.
Next we find the best tour constructed by ants in each iteration and compare it with $Global\_Best$ tour that is the best found so far to see if the best solution of iteration is better that that so we can update it by new found best tour. $find\_best()$ function returns the best tour of the iterations and store it in $best$. and at the end of each iteration we update the pheromone matrix values(all of them) by the factor of $1 - \rho$. The Sequential Algorithm is as follow:

Algorithm 2: ACO for TSP Sequential Algorithm

```
1  Parameter Initialization;
2  first ← true
3  for i ← 0 to no_Of_Iterations do
4  |  finding_Paths();
5  |  Phermones_Update();
6  |  if first then
7  |  |  best ← find_best();
8  |  |  first ← false
9  |  end
10 |  else
11 |  |  if best.length < Global_Best.length then
12 |  |  |  Global_Best ← best
13 |  |  end
14 |  end
15 |  updateTauMatrix(1 - RO);
16 end
```

4.2 Parallel Algorithm

For making the algorithm parallelized first we have to observe which part of the sequential algorithm is taking the largest amount of time and identify sets of tasks that can be run concurrently and/or partitions of data that can be processed concurrently. To find this out we can profile the program to see time taken by different parts of the program. In this project *gprof* is used to profile the program which is a profiling tool among other tools available.
The observation made by this tool is as below:

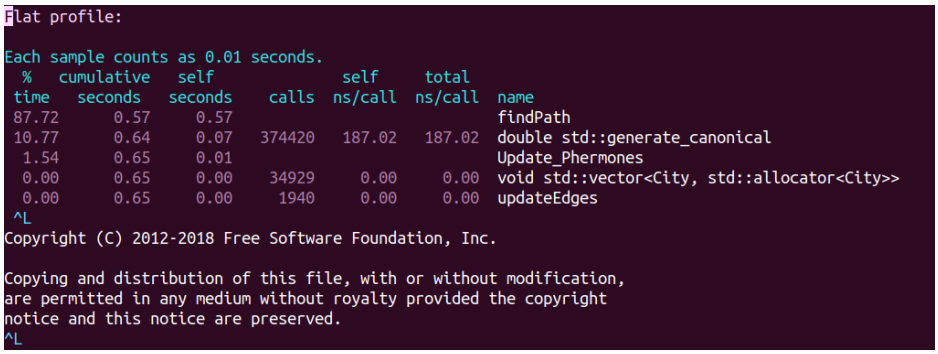

Figure 1: Profiling Result

As you can see in above figure $findPath$ function takes 87.72% of execution time and the rest of it taken by other functions like $Update\_Phermones()$ .Therefore, we try to parallelized the finding paths process such that a set of tasks (in our case finding path by a set of ants) can be done by number of processing units(workers/threads).
For the sake of parallelization MapReduce model and fastflow version of it is used. They are discussed in following.

4.2.1 MapReduce

A common situation is having a large amount of consistent data which must be processed. If the data can be decomposed into partitions, we can devise a parallel solutions. In our case we divide the ant set into number of worker available (parallelism degree) using $tiling\_jobs()$ function. and then we send its ant subset to Map the $findPath$ function on them and receive the result and store in $Tour$ vector (Mapping Phase).
After collecting all the results from all ants we have to find the best tour(solution) in Reduce Phase. since the Reduce phase is just a for loop and take very short time(less than one millisecond) we will do in sequential. we do the same thing for update pheromones which 1.54% of execution time which in compare to finding paths process is very small.
This model implements *static load balancing* which is commonly used if all

tasks are performing the almost same amount of work on identical machines.

The psudocode for parallel algorithm is as follow:

Algorithm 3: ACO for TSP Parallel Algorithm using Threads

```
1  Parameter Initialization;
2  first ← true
3  for i ← 0 to no_Of_Iterations do
4  |  for a ← 0 to ANTS do // do in Parallel
5  |  |  Tour[a] ← findPath();
6  |  end
7  |  Phermones_Update();
8  |  if first then
9  |  |  best ← find_best();
10 |  |  first ← false
11 |  end
12 |  else
13 |  |  if best.length < Global_Best.length then
14 |  |  |  Global_Best ← best
15 |  |  end
16 |  end
17 |  updateTauMatrix(1 - RO);
18 end
```

4.2.2 FastFlow

FastFlow is a parallel programming framework written in C/C++ promoting pattern based parallel programming. In this project we use *parallel_for/map* of FastFlow we gives us to parallelized the program the same way we in previous subsection but in case of FastFlow we do not need to tile the jobs, FastFlow itself will take care of it by considering the parallelism degree that we give inside the program.
The class interface is defined in the file *parallel_for.hpp*.
The syntax is as follow:

```
#include<ff/parallel_for.hpp>
using namespace ff;

ParallelFor pf(max_no_of_workers); //defining the object
pf.parallel_for( first , last , step , tile , body_of_function , no_of_workers );
```

The modification that we have to make in our parallel program is shown below:

Algorithm 4: FastFlow

```
1 #include<ff.parallel_for.hpp>
2 using namesapce ff;
3 ParallelFor pf(ANTS);
4 pf.parallel_for( 0 , ANTS , 1 , 0,[&Tour , &A , &Tau_Matrix,
    &cities](const long j){
5 Tour[j] = findPath(Tau_Matrix , A , cities);
6 } , p_d);
```

5 Performance Analysis

In this section we study the performance of parallel program such as speedup , efficiency , scalability and so on in compare to sequential program.
In the following table, there are parameters that we need to set for both the sequential and parallel program.

Parameter	Value
ANTS	256
cities	194
no_of_iterations	10
α	2
β	5
ρ	0.7
Q	1

Table 1: Parameters Used in ACO for TSP

cities and *no_of_iterations* are command line argument that have been considered for this observation for the sake of study and they can be change by giving different input file and iterations number.The execution time of sequential algorithm with above parameters is 32416 milliseconds.
In the following tables observations of parallel programs are shown:

Parallelism Degree	Exe. time(ms)	SpeedUp	Ideal SpeedUp	Efficiency	Scalability	Overhead
1	32416	1	1	1	-	3
2	17012	1.8	2	0.92	1.9	4
4	8584	3.7	4	0.91	3.7	6
8	4433	7.2	8	0.90	4.3	13
16	2334	13.8	16	0.86	13.8	20
32	1243	25.9	32	0.8	26	38
64	774	41.6	64	0.62	41.8	73
128	696	46.3	128	0.36	46.5	155
256	893	36.1	256	0.14	36.3	332

Table 2: Parallel Program Observations using threads

Parallelism Degree	Exe. time(ms)	SpeedUp	Ideal SpeedUp	Efficiency	Scalability	Overhead
1	32394	1	1	1	-	0
2	17045	1.8	2	0.92	1.9	2
4	8620	3.7	4	0.91	3.7	2
8	4437	7.2	8	0.90	7.3	2
16	2513	12.8	16	0.8	12.8	2
32	1523	21.1	32	0.65	21.2	2
64	1030	32.3	64	0.5	31.4	6
128	608	53	128	0.41	53.2	9
256	514	62.7	256	0.24	63.1	19

Table 3: Parallel Program Observations using FastFlow

5.1 Speedup

The speedup is defined as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve same problem on n processor.It can be computed by following formula.

$$speedup(n) = \frac{T_{seq}}{T_{par}(n)}$$

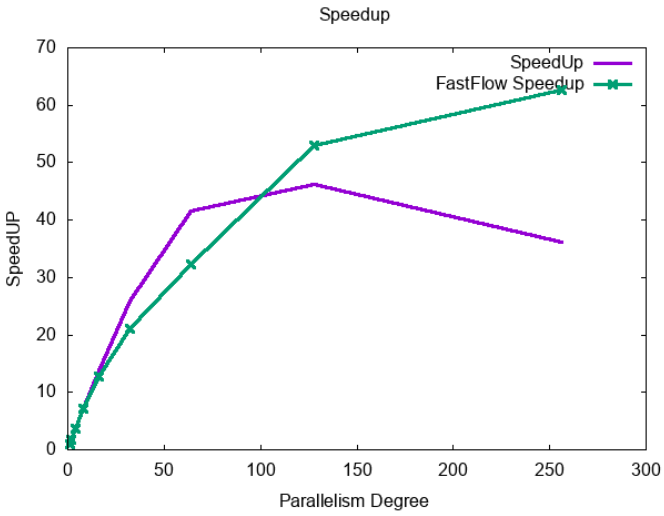

Figure 2: Speedup

5.2 Efficiency

The efficiency is defined as the ratio of speedup to the number of processors. Efficiency measures the fraction of time for which a processor is usefully utilized. Efficiency can be calculated using following formula.

$$Efficiency = \frac{s(n)}{n}$$

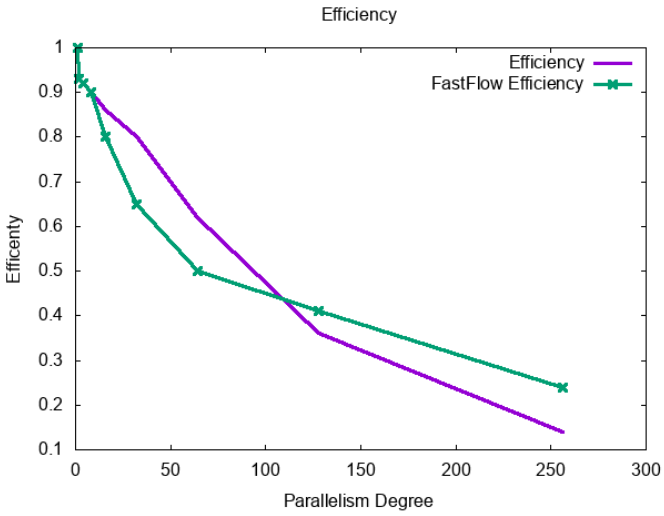

Figure 3: Efficiency

5.3 Scalability

Scalability is a measure of a parallel system's capacity to increase speedup in proportion to the number of processor. The following formula is for computing scalability.

$$scalability(n) = \frac{T_{par}(1)}{T_{par}(n)}$$

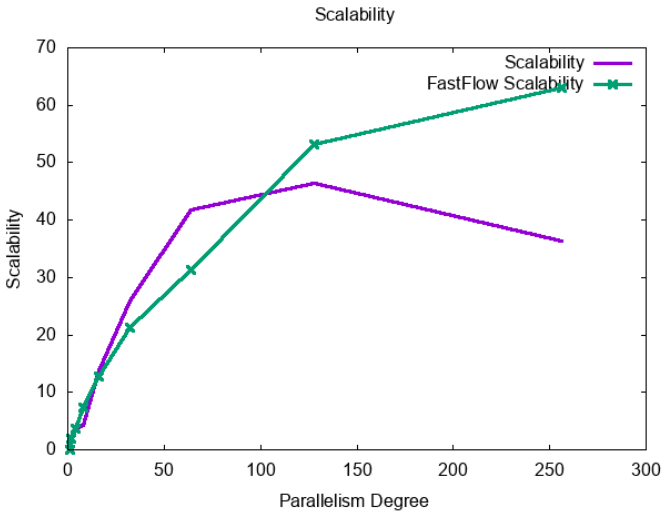

Figure 4: Scalability

5.4 Overhead

Time taken by non-functional part of parallel program that is not incurred by the serial algorithm is overhead. The source of parallel overhead could be one or more of the reasons such as interprocessor communication , load imbalance or extra computations.

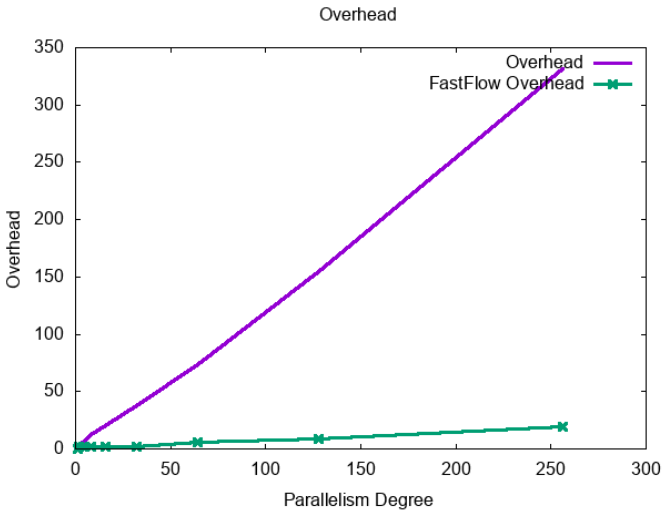

Figure 5: Overhead