Travelling Salesman Problem with Genetic Algorithms

Project Report of Parallel and Distributed Systems: Paradigms and Models Course

University of Pisa

Antonio Di Mauro
599785

Contents

1 Introduction

Travelling Salesman Problem (TSP) is a problem of finding the shortest path, visiting once the cities/nodes in the graph, from starting city to destination city.

The Genetic Algorithm (GA) is used as heuristic that mimics genetic evolution of species. At the beginning there is a population of chromosomes and at each generation the new population is made by crossover and mutation procedures. Crossover takes two parents and made up a new chromosome/children. Mutation swap randomly two "genes" of a chromosome. The individuals that survives to the next generation is decided by a fitness function that select the best ones.

In the TSP setting a chromosome is a list of cities (identified by a unique integer) that represents a path from a start city to a destination city.

2 Workflow analysis

The problem of TSP with GA could be represented by the pseudocode in Algorithm 1.

Algorithm 1 Pseudocode of TSP with GA

```
create population
foreach generation:
    breeding population
    ranking population
```

At each generation the population size has keeped constant. The breeding operation computes crossover and mutation and the ranking operation sorts the population by fitness.

In Table 1 are collected the times of the single operations obtained running several times the sequential implementation of the algorithm.

Operation	**Latency (μsec)**
Creation	46
Breeding	86
Ranking	1628

Table 1: Times of the operations

In a graphical way the workflow of the algorithm, with serial and non serial operations, is represented as in Figure 1.

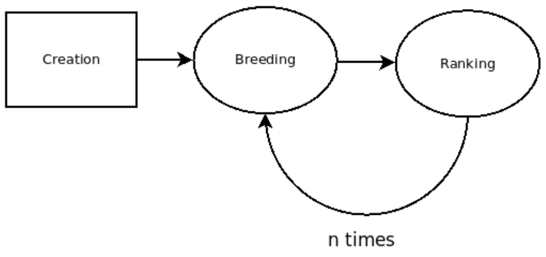

Figure 1: WorkFlow of the Algorithm (the squared shape represents serial operation and the rounded shape represents the non serial operation)

In the parallel view of the algorithm we could leave out the Creation operation because done just once time at the beginning, and focus on the other two.

In a first approach we could think that it's not convenient to parallelize the Breeding operation, because the Ranking operation takes much more time (18 times) than the one said before. So we could think to parallelize only the Ranking operation.

The thought said before is represented using the Master-Worker template of Figure 2.

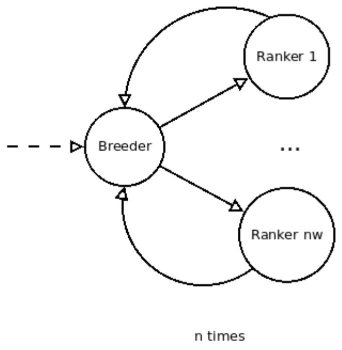

Figure 2: Master-Worker template with Breeder as Emitter and Ranker as Worker

The Breeder receives the initial population and then computes the breeding operation. After the population is divided in nw (number of workers) sub-populations, these are passed to each Ranker node that computes the ranking operation and returns the sub-population to the Breeder. Finally, the Breeder computes the breeding operation on the top-k (where k is the size of initial population) of the population and discard the others.

The breeding and ranking operations are repeated n times, where n is the number of generations.

The second approach uses RPLSH with optimization/refactoring rules. The result is that the breeding and the ranking operations are combined in a single node in a configuration with the minimum service time and resources used.

The RPLSH analysis is in Listing 1.

Listing 1: RPLSH analysis

```
rplsh> breed = seq(86)
rplsh> rank = seq(1628)
rplsh> main = pipe(breed,rank)
rplsh> rewrite main with allrules, allrules
rplsh> optimize main with farmopt, pipeopt, maxresources
rplsh> show main by servicetime, resources +5
122.428571      16       [4] : farm(comp(breed,rank)) with [ nw: 14]
122.428571      16      [16] : comp(farm(breed) with [ nw: 14],farm(rank) with [ nw: 14])
125.230769      16       [5] : pipe(breed,farm(rank) with [ nw: 13])
148.000000      16       [1] : pipe(breed,farm(farm(rank) with [ nw: 11]) with [ nw: 1])
148.000000      16       [9] : pipe(farm(breed) with [ nw: 1],farm(rank) with [ nw: 11])
```

The final Workflow is shown in Figure 3.

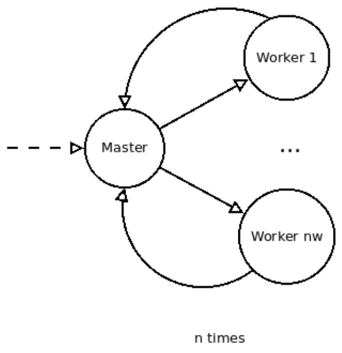

Figure 3: Final Workflow with a Master node that divides and collects the tasks, and Worker nodes that computes breeding and ranking operations

The final workflow is used in the C++ Threads, C++ Threads v2 and FastFlow implementations.

3 Speedup analysis

Thanks to Gustafson law we consider the serial fraction that represents the fraction of time spent to executes the Creation operation.

$$f = \frac{t_{serial}}{t_{serial} + t_{non\,serial}} = \frac{t_{creation}}{t_{creation} + n * (t_{breeding} + t_{ranking})} = \frac{46}{46 + n * 1714}$$

the maximum reachable (ideal) speedup is the following one:

$$sp(nw) = \frac{t_{seq}}{f * t_{seq} + [(1 - f) * t_{seq}] \,/\, nw}$$

where n is the number of generations and nw is the number of workers.

4 Experimental results

The experiments were performed on two different machines:

1. 2-core with 2-way hyperthreading Dell XPS 13 9360 with Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz $\times$ 4

2. 64-core with 4-way hyperthreading Dell PowerEdge C6320p with Intel(R) Genuine Intel(R) CPU @ 1.30GHz $\times$ 256

Par. degree	**C++ Threads**	**C++ Threads v2**	**FastFlow**	**Ideal**
1	0.788872	0.914004	0.855597	1
2	0.723985	1.33097	1.65902	1.94906
4	0.626807	1.59838	1.36463	3.70917

Table 2: Speedups on machine #1 (population size=1000, num. of generations=1)

Par. degree	**C++ Threads**	**C++ Threads v2**	**FastFlow**	**Ideal**
1	0.781819	0.898461	0.841908	1
2	0.879274	1.42597	1.5055	1.99466
4	0.713392	1.74287	1.53337	3.96814

Table 3: Speedups on machine #1 (population size=1000, num. of generations=10)

Par. degree	**C++ Threads**	**C++ Threads v2**	**FastFlow**	**Ideal**
1	0.770463	0.902733	0.824271	1
2	0.879337	1.50633	1.46838	1.99732
4	0.699986	1.67319	1.49424	3.98398

Table 4: Speedups on machine #1 (population size=1000, num. of generations=20)

Par. degree	**C++ Threads**	**C++ Threads v2**	**FastFlow**	**Ideal**
1	0.710359	0.797314	0.842233	1
2	0.804872	1.57555	1.65019	1.94906
4	0.928813	3.3406	3.27235	3.70917
8	1.05624	6.21164	6.05411	6.76273
16	1.06268	8.27313	7.19564	11.4939
32	0.904287	7.90988	6.30063	17.6773
64	0.861444	5.79087	4.20327	24.182
128	0.767355	3.87189	2.3106	29.6343
256	0.725084	2.09641	1.18676	33.3995

Table 5: Speedups on machine #2 (population size=1000 and num. of generations = 1

Par. degree	**C++ Threads**	**C++ Threads v2**	**FastFlow**	**Ideal**
1	0.669405	0.788996	0.740546	1
2	0.82544	1.64205	1.45528	1.99466
4	0.931376	3.38095	2.99267	3.96814
8	1.04797	6.42371	5.68394	7.85287
16	1.16049	9.9026	8.9028	15.3824
32	0.85561	9.04004	7.33621	29.5482
64	0.812814	6.41966	4.64282	54.7652
128	0.770689	3.95854	2.52114	95.5275
256	0.790025	2.05764	1.29699	152.152

Table 6: Speedups on machine #2 with population size = 1000, num. of generations=10)

Par. degree	**C++ Threads**	**C++ Threads v2**	**FastFlow**	**Ideal**
1	0.665929	0.786586	0.734954	1
2	0.816739	1.63734	1.44902	1.99732
4	0.912549	3.31885	3.03699	3.98398
8	1.02159	6.23443	5.58708	7.92565
16	1.12949	9.55975	8.81808	15.6847
32	0.815517	8.68736	7.15742	30.7237
64	0.777131	6.14829	4.48287	59.0174
128	0.754491	3.79671	2.42672	109.384
256	0.755463	1.95742	1.23955	190.799

Table 7: Speedups on machine #2 (population size=1000, num. of generations=20)

5 Conclusions

The C++ Threads implementation faces the problem with the map data parallel pattern, trying to reduce the service time/latency. Its performances are very bad due to the fact that it is not able to ammortize the overhead.

The C++ Threads v2 implementation tries to solve the problem of synchronization among threads. The problem is overcome removing mutexes and using asyncs that operates on own data, encouraging data locality to reduce overhead. The results are similar to the FastFlow implementation.

The FastFlow implementation, instead, faces the problem with the farm stream parallel pattern, trying to increase the throughput. In all the results (on machine #1 and #2) it's obvious that the FastFlow and C++ Threads v2 solutions perform better than the C++ Threads solution, due to a better speedup.

Nevertheless the performances, excepts for C++ Threads implementation, are better on machine #2 because of a greater availability of physical cores with respect to machine #1 (at least up to parallel degree of 8).

Looking the results the problem seems to be difficult to scale because of shorter operations in times and the overhead does not ammortize well with increasing number of generations.