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	92
Breeding	1793
Crossover	1
Mutation	0.1
Adding	0.2
Ranking	3294

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.

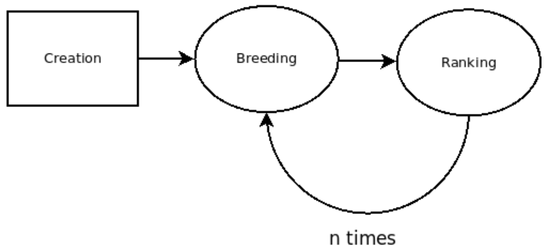

Figure 1: WorkFlow of the Algorithm (the squared shape represents serial operation and the rounded shape represents the non serial operation)

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 more time (1.8 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.

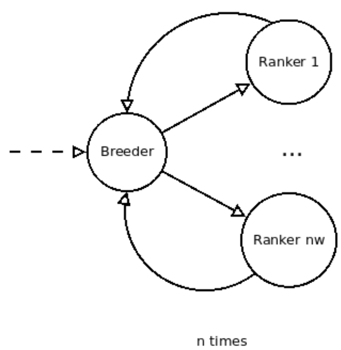

Figure 2: Master-Worker template with Breeder as Emitter and Ranker as Worker

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 discards 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, that executes the two operations n times (number of generations), 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(1793)
rplsh> rank = seq(3294)
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
363.357143    16    [4]  : farm(comp(breed,rank)) with [ nw: 14]
363.357143    16    [16] : comp(farm(breed) with [ nw: 14],farm(rank) with [ nw: 14])
448.250000    16    [9]  : pipe(farm(breed) with [ nw: 4],farm(rank) with [ nw: 8])
470.571429    16    [7]  : farm(pipe(breed,rank)) with [ nw: 7]
549.000000    16    [13] : farm(farm(pipe(breed,rank)) with [ nw: 6]) with [ nw: 1]
```

The final Workflow is shown in Figure 3.

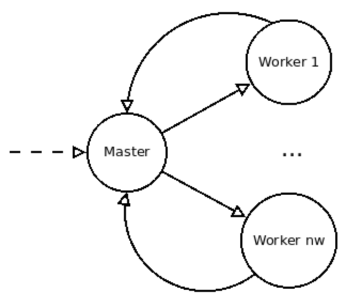

Figure 3: Final Workflow with a Master node that divides and collects the tasks, and Worker nodes that computes breeding and ranking operations n (number of generations) times

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

3 Speedup analysis

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

$$f = \frac{t_{serial}}{t_{serial} + t_{non\,serial}} = \frac{t_{creation}}{t_{creation} + n * (t_{breeding} + t_{ranking})} = \frac{92}{92 + n * 5087}$$

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	0.851073	0.856575	0.897296	1
2	1.12543	1.18475	1.52724	1.96509
4	1.71029	1.77534	1.67833	3.79762

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

Par. degree	**C++ Threads**	**C++ Threads v2**	**FastFlow**	**Ideal**
1	0.811867	0.869933	0.883269	1
2	1.26028	1.27939	1.57553	1.96509
4	1.61313	1.71275	1.59749	3.79762

Table 3: Speedups on machine #1 with population size=2000 and num. of generations=1)

Par. degree	**C++ Threads**	**C++ Threads v2**	**FastFlow**	**Ideal**
1	0.739558	0.899451	0.882169	1
2	1.04453	1.27052	1.61805	1.96509
4	1.5548	1.69965	1.58161	3.79762

Table 4: Speedups on machine #1 with population size=5000 and num. of generations=1)

Par. degree	**C++ Threads**	**C++ Threads v2**	**FastFlow**	**Ideal**
1	0.918435	0.941349	0.925606	1
2	1.66002	1.68985	1.12219	1.9964
4	1.6709	2.14626	2.09373	3.97845

Table 5: Speedups on machine #1 with population size=1000 and num. of generations=10)

Par. degree	C++ Threads	C++ Threads v2	FastFlow	Ideal
1	0.813205	0.794346	0.801019	1
2	1.39108	1.39954	1.69082	1.99819
4	1.72489	1.80825	1.74284	3.98919

Table 6: Speedups on machine #1 with population size=1000 and num. of generations=20)

Par. degree	C++ Threads	C++ Threads v2	FastFlow	Ideal
1	0.757022	0.789678	0.821264	1
2	1.48496	1.55376	1.64263	1.96509
4	3.23743	3.35165	3.23665	3.79762
8	5.58628	6.18385	5.91526	7.11523
16	8.15313	8.4203	6.9353	12.6336
32	8.88415	8.18004	5.96233	20.6361
64	6.27635	5.93378	4.02413	30.201
128	3.95809	3.85283	2.23667	39.3117
256	2.11561	2.08413	1.16201	46.2945

Table 7: Speedups on machine #2 with population size=1000 and num. of generations = 1

Par. degree	C++ Threads	C++ Threads v2	FastFlow	Ideal
1	0.753454	0.798881	0.814576	1
2	1.48747	1.61704	1.60474	1.96509
4	3.0453	3.33055	3.19023	3.79762
8	5.68714	6.2468	5.92191	7.11523
16	9.25257	10.5072	9.6265	12.6336
32	11.9741	10.7051	9.67435	20.6361
64	11.4896	10.0574	7.71396	30.201
128	7.37519	6.95957	4.53211	39.3117
256	4.31657	4.1597	2.38415	46.2945

Table 8: Speedups on machine #2 with population size=2000 and num. of generations = 1

Par. degree	C++ Threads	C++ Threads v2	FastFlow	Ideal
1	0.719794	0.793558	0.793813	1
2	1.41493	1.58475	1.55758	1.96509
4	2.83275	3.22042	3.15464	3.79762
8	5.4739	6.24574	5.90533	7.11523
16	9.02825	10.6509	9.71962	12.6336
32	13.5466	14.3158	13.5786	20.6361
64	17.3014	13.8417	13.793	30.201
128	16.2987	13.1153	10.4463	39.3117
256	10.2289	9.22219	6.07049	46.2945

Table 9: Speedups on machine #2 with population size=5000 and num. of generations = 1

Par. degree	C++ Threads	C++ Threads v2	FastFlow	Ideal
1	0.803561	0.810559	0.814955	1
2	1.6548	1.66809	1.67186	1.9964
4	3.59791	3.67	3.63829	3.97845
8	7.47846	8.00092	7.86416	7.90017
16	14.2217	15.0491	16.1903	15.5782
32	23.4193	23.2696	24.7212	30.3041
64	14.4274	14.3771	26.8707	57.4645
128	7.51412	7.55097	20.2154	104.127
256	3.66139	3.67091	11.2603	175.301

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

Par. degree	C++ Threads	C++ Threads v2	FastFlow	Ideal
1	0.804585	0.806463	0.811388	1
2	1.66047	1.66301	1.67032	1.99819
4	3.63413	3.65031	3.64861	3.98919
8	7.65118	8.03507	7.96522	7.94972
16	14.8984	15.4823	17.0902	15.7861
32	25.603	25.3169	28.891	31.1282
64	14.8072	14.7705	30.7292	60.5535
128	7.5413	7.56014	31.8849	114.825
256	3.798	3.83627	21.0397	208.066

Table 11: Speedups on machine #2 with population size=1000 and 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.

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 of the version 2 aren't much better compared to the previous one.

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 has a slightly better speedup than the other versions.

Looking the results seems that the program scales well before 16/32 parallel degree and after the performances drops down. Intel VTune Profiler has been used, on machine #1, to monitor the activity of the threads of the C++ Threads and C++ Threads v2 implementations to find the reasons of this dropping of performances.

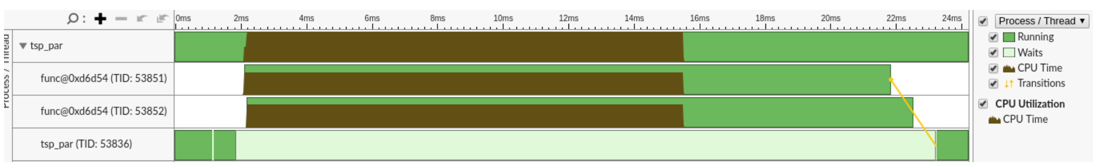

Figure 4: VTune Profiler Threading analysis of C++ Threads implementation with 2 workers

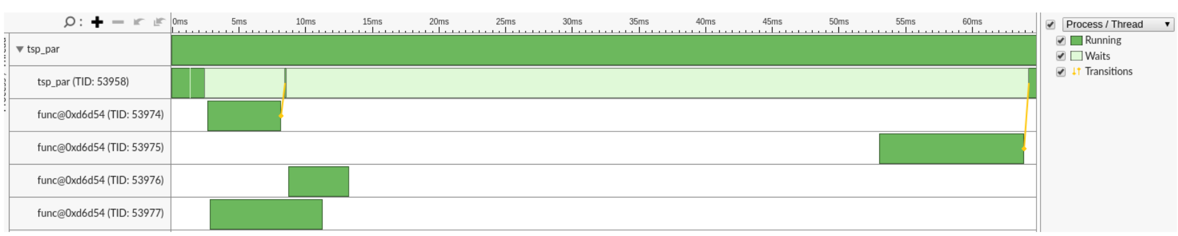

Figure 5: VTune Profiler Threading analysis of C++ Threads implementation with 4 workers

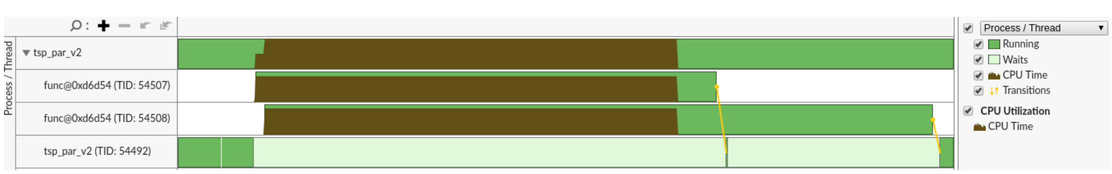

Figure 6: VTune Profiler Threading analysis of C++ Threads v2 implementation with 2 workers

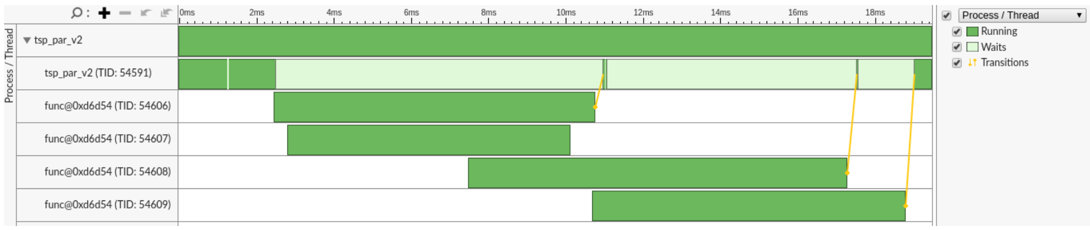

Figure 7: VTune Profiler Threading analysis of C++ Threads implementation v2 with 4 workers

In Figure 5 we could find a reason of the dropping in performances, the lack of physical core available that causes delays in threads executions.

Another problem could be the load balancing because, like in Figure 6, one thread takes more time than the others.

Going deeper in the analysis of the load balancing problem, in Table 12 are collected the times of the main parts of the C++ Threads implementation (the times of C++ Threads v2 are not collected because the two have similar performances).

			times (usecs)						
	nw	**pop**	**creation**	**split pop**	**worker**	**join**	**merging**	**total**	
Machine #1	2	1000	892	420	45477	45578	2813	50221	
	4	1000	864	445	26666	41291	1804	45206	
					1.705430136	**1.103824078**	**1.559312639**	**1.110936601**	
	2	2000	2467	1398	179116	179278	11879	196085	
	4	2000	2736	1456	86129	85936	4603	95838	
					2.079624749	**2.086180413**	**2.580708234**	**2.046004716**	
	2	3000	2881	1521	168599	168659	23894	198349	
	4	2000	2744	1426	138339	139768	8852	153757	
					1.218738028	**1.206706828**	**2.699277**	**1.290016064**	
Machine #2	2	1000	1375	663	81080	81380	3763	88337	
	4	1000	1400	609	43290	43714	2120	51878	
					1.872949873	**1.861646155**	**1.775**	**1.702783453**	
	8	1000	1399	644	18989	18632	1315	26034	
					2.279740903	**2.346178617**	**1.6121673**	**1.992701851**	
	16	1000	1352	603	21435	7482	787	36720	min worker time 6696
					0.8858875671	**2.49024325**	**1.67090216**	**0.7089869281**	
	32	1000	1324	635	7857	4630	638	25796	
					2.728140512	**1.615982721**	**1.23354232**	**1.423476508**	
	64	1000	1330	629	6863	7271	533	29437	min worker time 955
					1.14483462	**0.6367762344**	**1.196998124**	**0.8763121242**	
	128	1000	1337	679	1757	13813	614	46264	min worker time 302
					3.906089926	**0.526388185**	**0.8680781759**	**0.6362830711**	
	256	1000	1254	627	531	30487	1129	79320	min worker time 59
					3.308851224	**0.4530783613**	**0.5438441098**	**0.5832576904**	
Machine #2	2	2000	2679	1240	160172	160497	14001	179941	
	4	2000	2626	1291	83474	82459	7756	96960	
					1.918825023	**1.946385476**	**1.805183084**	**1.855827145**	
	8	2000	2665	1237	39984	40197	3277	50926	
					2.087685074	**2.051371993**	**2.366798901**	**1.903939049**	
	16	2000	2628	1197	18967	16862	2090	31154	
					2.108082459	**2.383880916**	**1.567942584**	**1.634653656**	
	32	2000	2592	1124	12357	9783	1530	33944	
					1.534919479	**1.723602167**	**1.366013072**	**0.9178057978**	
	64	2000	2650	1278	7868	8160	1179	39095	min worker time 2609
					1.570538892	**1.198897059**	**1.297709924**	**0.868244021**	
	128	2000	2603	1190	11974	12401	1042	50747	min worker time 949
					0.6570903625	**0.6580114507**	**1.131477927**	**0.7703903679**	
	256	2000	1921	996	27243	27404	1542	82458	min worker time 276
					0.4395257497	**0.4525251788**	**0.6757457847**	**0.6154284605**	

Table 12: C++ Threads implementation times on machine #1 and #2

As we can see in Table 12, in bold cells there are the ratio of the last two measures of times. The ratio should be 2 every time in an optimal situation, but this is not. In this case, the ratio, highlights an instability in the execution of the workers. Concerning the worker times, this is the greatest one and in some cases the gap between the maximum and the minimum one is high. This gap support the unbalancing problem issue.

In conclusion the problem of dropping in performances after 16/32 parallel degree is related to the load balancing issue among workers that is correlated also with increasing of times in joining threads.