A Parallel Ant Colony Optimization Algorithm for the Traveling Salesman Problem

Felipe Ramírez González

Departamento de Informática, Universidad Técnica Federico Santa María
Avenida España #1680, Valparaíso, Chile

Abstract

In this paper we propose a parallel ant colony optimization algorithm to solve the Traveling Salesman Problem in less CPU time than the original sequential version. The parallel scheme proposed allows ants to generate successively shorter asynchronous paths employing accumulated information in a synchronous common means. Computer simulations show that the proposed algorithm is able to find solutions as good as the sequential version to several instances of the TSP in significantly less CPU time.

Keywords: Traveling Salesman Problem, Ant Colony Optimization, Parallel Computing

1. Introduction

The Traveling Salesman Problem (TSP) is a well-known NP-hard problem [4], that consists in finding the shorter closed path among a set of cities arbitrarily distributed, such that each city is visited exactly once. Formally, given an undirected graph with (symmetric) non-negative costs associated to each vertex, it is required to find a hamiltonian cycle of minimum cost. Although a number of variants and generalizations of this problem have been proposed in the literature, such as the probabilistic TSP (pTSP) and the vehicle routing problem (VRP), in this paper we focus in solving the standard form of the TSP, as it is commonly used as a benchmark problem for assessing the performance of meta-heuristics.

Ant Colony Optimization (ACO) algorithms were originally proposed by Marco Dorigo in his Ph.D thesis [3], inspired in the cooperative behavior of these living organisms when searching for food sources. Ants lay a chemical called pheromone on good routes to suggest good paths for other members of the colony to follow, which in time converges to optimal solutions. The mapping of the ACO algorithm for solving the TSP is straightforward. Ants can be thought of agents that move from one city to another in the graph, choosing their next destination based on a probabilistic function. Initially m ants are randomly placed in different nodes of the graph to complete their routes independently. Once all ants finish their routes, the best one (the one that found the shorter path) is selected to lay pheromone on it. Several iterations of this procedure are performed in such a way that the best circuits are strengthen with chemical from each iteration to another.

Meta-heuristics such as ACO were originally proposed as a solution for combinatorial problems that are impossible to solve in polynomial time. In this sense, meta-heuristics are multi-objective approaches, as they aim to find good solutions (as close to local optima) in a reasonable amount of time, specially in real-world applications where response times need to be low. Recent development of low-cost parallel computer architectures have broaden the interest of the research community in designing multi-threaded versions of existing algorithms and heuristics in order to accelerate their convergence [1].

Although recent work addresses the parallelization of meta-heuristics such as ACO algorithms [7], none performs a comparative study of the effect of the parallelization over the consumption of CPU time. In this paper we propose a parallel ACO algorithm (PACO) for solving the TSP. We conduct a comparative study of the performance of the algorithm faced to its sequential version (SACO), in terms of CPU consumption in several well-known benchmark instances.

The remainder of this paper is as follows. In Section 2 we briefly state the original ACO algorithm and propose an asynchronous routing scheme. In Section 3 we describe the experimental setup and we report the results of the comparative study. In Section 4 we give a brief literature review on meta-heuristic parallelization trends. And finally in Section 5 we conclude and discuss our final remarks.

2. Methodology

The proposed method consists in a parallelization scheme for the original ACO algorithm with the aim of speeding it up preserving the consistency of the solutions found.

2.1. Ant Colony Optimization Algorithm

As stated earlier, the application of ACO algorithms for solving the TSP is straightforward. Let τ_{ij} and η_{ij} be, respectively, the *pheromone level* and *visibility* at vertex (i, j), α and β input parameters that weight the pheromone level and visibility, respectively, N the number of iterations to be performed, and m the number of ants to route. The ACO algorithm is given by Algorithm 1.

According to this setting the algorithm routes each ant sequentially, which in multi-core architectures does not take full advantage of the available resources. Our method aims to address this flaw introducing an asynchronous routing scheme.

	SACO		PACO			
Instance	**Time**	**Solution**	**Time**	**Solution**	**Speedup**	**OPT**
a280	166.80	2622.47	58.15	2599.82	2.86	2579
eil51	56.00	656.59	27.01	646.54	2.07	426
eil76	27.35	628.98	22.88	628.98	1.19	538
eil101	41.02	747.76	24.70	750.08	1.66	629
pr1002	921.57	333538.43	343.91	334446.50	2.67	259045

Table 1: Results of each algorithm routing 100 ants over five well-known benchmark instances. All instances were tested using the same set of parameters.

Algorithm 1 ACO-TSP Algorithm

```
1:  for each vertex $(i, j)$ do
2:      $\tau_{ij}(0) = \tau_0$
3:  end for
4:  for $k = 1 \to m$ do
5:      Place ant $k$ at random node.
6:  end for
7:  for $t = 1 \to N$ do
8:      for $k = 1 \to m$ do
9:          repeat
10:             Choose next node according to
11:             $P^k_{ij}(t) = \frac{[\tau_{ij}(t)]^\alpha [\eta_{ij}]^\beta}{\sum_{h \in J^k_i} [\tau_{ih}(t)]^\alpha [\eta_{ih}]^\beta}$
12:         until Ant $k$ completes its route
13:         Store route length of ant $k$, $L_k$
14:     end for
15:     Store best route $T^+$ of length $L^+$
16:     for each vertex $(i, j)$ do
17:         $\tau_{ij}(t) = (1 - \rho)\tau_{ij}(t) + \Delta\tau_{ij}(t) + e\Delta\tau^e_{ij}(t)$
18:     end for
19: end for
```

2.2. *Proposed Parallelization*

The proposed parallelization consists in executing the routing procedure of each ant (lines 9 through 12 in Algorithm 1) in separate independent threads, *i.e.*, one thread per ant. Since according to the original setting, all ants must find their routes before updating the pheromone levels, we placed a thread barrier before line 15 to prevent ants that finished their routes going further in the updating procedure in order to preserve the consistency of the algorithm. Once all threads reach the barrier (all ants complete their routes) the algorithm continues normally updating the pheromone levels and iterating. Figure 1 depicts a schematic representation of both versions (sequential and parallel) of the algorithm. We call this parallelization scheme *asynchronous routing*.

Note that, according to this setting, the number of threads is restricted to the number of ants, hence, the complexity of the algorithm increases with the number of threads. This means that even though adding threads may result in more parallelism, at the same time more ants need to be routed, hence it is expected that the CPU time consumed by PACO will increase with the number of threads used. Nevertheless, we expect that this increase will be lower than that of SACO.

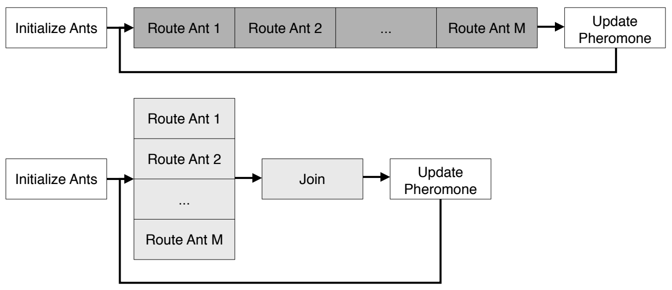

Figure 1: Schematic representation of the sequential and parallelized versions of the ACS-TSP Algorithm. Basically, the parallel version routes ants in independent threads, whereas in the sequential version ants are routed one after the latter.

3. Results

We conducted experiments with both the sequential and parallel versions of the ACO-TSP Algorithm, in order to assess their performance in terms of both solution accuracy and CPU time. Both algorithms were coded in C and we used the POSIX interface to implement threads in the parallel version. All simulations were performed in a 2,93 GHz Hyper-Threading Quad Core Intel Core i7 with 4 GB of RAM running Mac OS X 10.7.2.

We properly tuned the parameters of the algorithm to optimize the accuracy of solutions, and used this set of parameters to run both versions. The optimum number of ants resulted in near 100 (hence, 100 threads were used in the parallel version according to the scheme proposed in Section 2).

3.1. *Speedup with 100 threads*

We computed the response times and route lengths (solution accuracy) for five well-known benchmark instances. We measure and report the `real` CPU time given by `/usr/bin/time`, as `user` and `sys` times represent the sum of the time spent by the process among all cores, which in general is greater with PACO, due to thread management and synchronizations. Table 1 summarizes our results.

Note that, in general, the quality of solutions found by SACO don't differ by a significant amount from that of those found by PACO, this is due to the fact that, other than parallelization, the algorithms are equivalent. The slight differences may be a consequence of the random initialization of ants (see Algorithm 1, line 5). Comparing with the known optima, the quality of the solutions found is overall acceptable.

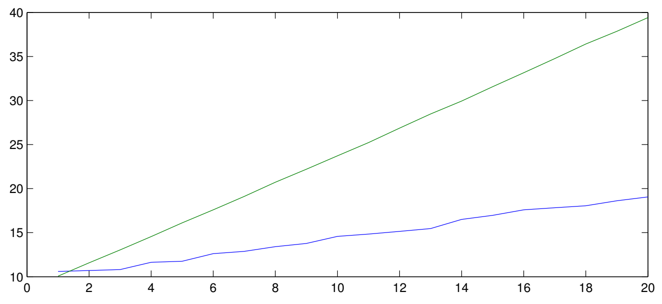

Figure 2: Response time of both versions of the algorithm as a function of the number of threads used each time. The upper line corresponds to SACO and the lower one to PACO.

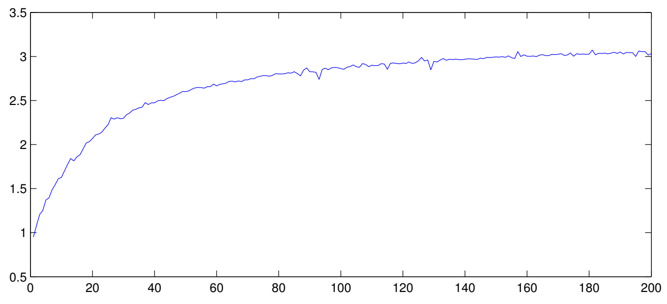

Figure 3: Speedup as a function of the number of ants used each time. The speedup increases logarithmically and stabilizes with over 10 threads.

In terms of CPU time, PACO clearly outperforms SACO. The speedup column shows factors of around 2, which means that PACO is in general as twice as fast as SACO depending on the instance at 100 threads. Variations of speedup among instances when using the same number of threads may be due to significant variations of the internal structure of the graphs, such as the number of nodes.

3.2. Speedup as a function of threads

We also evaluated the speedup between PACO and SACO as a function of the number of threads using instance *a280*. Figure 2 shows the CPU time of both versions of the algorithm with the number of threads varying from 1 to 20. With one thread the CPU time used by PACO is greater than that of SACO, which is expected since the parallel version spends additional time managing the single thread and synchronizing. That obvious case aside, we can see that both lines increase monotonically with the number of threads, this is because adding threads turns in additional computational burden for the algorithm (see Section 2). It is very interesting that the increase of time consumed by PACO is slower than that of SACO, which indicates that the speedup anyway increases with the number of threads.

Figure 3 confirms the aforementioned hypothesis. With the number of threads ranging from 1 to 200, we can see that the speedup increases logarithmically from 1 and stabilizes at around 3 with over 100 threads. This result also agrees with the number of ants selected for the previous experiment, as there is no significant increase of speedup beyond 100 threads, so the tuning satisfies both objectives simultaneously; achieve good solutions with optimal speedup.

4. Related Work

Several works have addressed the parallelization of meta-heuristics due to the recent availability of multi-core computer architectures, and the need of swifter solutions in real-world applications. In the following we briefly review the literature on meta-heuristic parallelization trends.

In [2] the authors survey strategies for parallelizing meta-heuristics in general. The survey identifies a set of useful principles and proposes research directions. Examples of parallel meta-heuristics can be seen in [5], a parallel simulated annealing (SA) algorithm for solving the vehicle routing problem and in [10], a parallel particle swarm optimization (PSO) algorithm.

The parallelization of ant colony systems such as the described in this paper have also been addressed by many authors, in [8] a survey is presented. Moreover, parallel ant colonies have been successfully mapped to suit problems in other domains, such as in [9], where the ACO algorithm was adapted to solve the quadratic assignment problem (QAP).

Other strategies not explored in this work have been proposed, for example parallelizing other components of the algorithm, such as the migration controller [6]. Other strategies simply suggest running standard sequential versions of the algorithm in parallel [7] and select the best results.

5. Conclusions and future work

In this paper we have proposed a parallelization scheme for the ACO algorithm and evaluated its performance using benchmark instances of the TSP in terms of CPU time consumption and solution accuracy. We confirmed the relative consistency between the sequential and the parallel versions of the algorithm, in other words, we verified that their outputs do not differ by a significant amount, and the slight differences are due to random initialization of parameters, which can not be controlled. We observed that PACO significantly outperforms SACO in CPU time when using 100 threads. We conclude that PACO outperforms SACO in CPU time at comparable solution quality.

We also observed the behavior of the speedup between the two versions when varying the number of threads. Although the computer burden increases with the number of threads, we found that using 100 threads a good speedup is achieved, and at the same time, good solutions are found.

Future work may consider applying asynchronous schemes for further elements of the algorithm, such as pheromone updating, or even the complete asynchronous operation of ants.

References

[1] Enrique Alba. *Parallel Metaheuristics: A New Class of Algorithms*. Wiley-Interscience, 2005.

[2] Teodor Gabriel Crainic and Michel Toulouse. Parallel meta-heuristics. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research Management Science*, pages 497–541. Springer US, 2010. 10.1007/978-1-4419-1665-5_17.

[3] M. Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, 1979.

[5] Jian-Ming Li, Hong-Song Tan, Xu Li, and Lin-Lin Liu. A parallel simulated annealing solution for vrptw based on gpu acceleration. In Gloria Phillips-Wren, Lakhmi C. Jain, Kazumi Nakamatsu, Robert J. Howlett, Robert J. Howlett, and Lakhmi C. Jain, editors, *Advances in Intelligent Decision Technologies*, volume 4 of *Smart Innovation, Systems and Technologies*, pages 201–208. Springer Berlin Heidelberg, 2010. 10.1007/978-3-642-14616-9_19.

[6] Ying Lin, Jun Zhang, and Jing Xiao. A pseudo parallel ant algorithm with an adaptive migration controller. *Applied Mathematics and Computation*, 205(2):677 – 687, 2008. Special Issue on Advanced Intelligent Computing Theory and Methodology in Applied Mathematics and Computation.

[7] Max Manfrin, Mauro Birattari, Thomas Sttzle, and Marco Dorigo. Parallel ant colony optimization for the traveling salesman problem. In Marco Dorigo, Luca Gambardella, Mauro Birattari, Alcherio Martinoli, Riccardo Poli, and Thomas Sttzle, editors, *Ant Colony Optimization and Swarm Intelligence*, volume 4150 of *Lecture Notes in Computer Science*, pages 224–234. Springer Berlin / Heidelberg, 2006. 10.1007/11839088_20.

[8] Martn Pedemonte, Sergio Nesmachnow, and Hctor Cancela. A survey on parallel ant colony optimization. *Applied Soft Computing*, 11(8):5181 – 5197, 2011.

[9] E.-G. Talbi, O. Roux, C. Fonlupt, and D. Robillard. Parallel ant colonies for the quadratic assignment problem. *Future Generation Computer Systems*, 17(4):441 – 449, 2001. ¡ce:title¿Workshop on Bio-inspired Solutions to Parallel Computing problems¡/ce:title¿.

[10] Zhi-hui Zhan and Jun Zhang. Parallel particle swarm optimization with adaptive asynchronous migration strategy. In Arrems Hua and Shih-Liang Chang, editors, *Algorithms and Architectures for Parallel Processing*, volume 5574 of *Lecture Notes in Computer Science*, pages 490–501. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-03095-6_47.