Performance Analysis and Tuning for Parallelization of Ant Colony Optimization by Using OpenMP

Ahmed A. Abouelfarag, Walid Mohamed Aly(✉), and Ashraf Gamal Elbialy

[1] College of Engineering and Technology, Arab Academy for Science and Technology, & Maritime Transport, Alexandria, Egypt
abouelfarag@aast.edu
[2] College of Computing and Information Technology, Arab Academy for Science, Technology & Maritime Transport, Alexandria, Egypt
walid.ali@aast.edu, ashraf2k@gmail.com

Abstract. Ant colony optimization algorithm (ACO) is a soft computing metaheuristic that belongs to swarm intelligence methods. ACO has proven a well performance in solving certain NP-hard problems in polynomial time. This paper proposes the analysis, design and implementation of ACO as a parallel metaheuristics using the OpenMP framework. To improve the efficiency of ACO parallelization, different related aspects are examined, including scheduling of threads, race hazards and efficient tuning of the effective number of threads. A case study of solving the traveling salesman problem (TSP) using different configurations is presented to evaluate the performance of the proposed approach. Experimental results show a significant speedup in execution time for more than 3 times over the sequential implementation.

Keywords: Parallel metaheuristic · Ant colony optimization · Shared memory model · Openmp · Parallel threads

1 Introduction

Some of the real-life optimization problems cannot be tackled by exact methods which would be implemented laboriously and in a time-consuming manner. For such optimization problems, metaheuristics are used with less computational effort to find good solution from a set of large feasible solutions. Although other algorithms may give the exact solution to some problems, metaheuristics provide a kind of near-optimal solution for a wide range of NP-hard problems [1].

Since introduced in 1992 by Marco Dorigo [2], ACO algorithms have been applied to many combinatorial optimization problems, ranging from Scheduling Problems [3] to routing vehicles [4] and a lot of derived methods have been adapted to dynamic problems in real variables, multi-targets and parallel implementations.

ACO was proposed as a solution when suffering from limited computation capacity and incomplete information [3]. ACO metaheuristic proved a significant performance improvement compared with other metaheuristic techniques in solving many NP-hard problems such as solving the traveling salesman problem [5].

K. Saeed and W. Homenda (Eds.): CISIM 2015, LNCS 9339, pp. 73–85, 2015.
DOI: 10.1007/978-3-319-24369-6_6

The multicore computation power encouraged the modification of the standard metaheuristic approaches to be applied in a parallel form.

In this paper, OpenMP is used on CPU with multi-cores to measure the performance speedup. To make the data accessible and shared for all parallel threads in global address space, a shared memory model is implemented in C++. OpenMP is implemented with its parallel regions, directives to control "*for*" loops. Scheduling clause for fine tuning. For eliminating race condition, omp critical sections have been also implemented.

The importance of TSP problem as a test case comes from its history of applications with many metaheuristics. TSP is also easy for mapping with real life problems.

The speedup gain in parallelization of a typical sequential TSP with ACO depends mainly on the proper analysis of where parallel regions should be placed in the algorithm. Theoretically, Amdahl's law [8] limits the expected speedup achieved to an algorithm by a relation between parts that could be parallel to the parts remain serial. One of the targets of the experiment is to assign the optimal number of parallel threads and tuning them dynamically with the available number of CPU cores to get effective speedup.

This paper is organized as the following: in Section 2, the related work to ACO and the research efforts towards its parallelization are presented. Section 3 presents the sequential ACO algorithm mapped to TSP. In section 4, the proposed ACO parallelization using OpenMP is presented where its sub-sections show the analysis of different elements of OpenMP and its effects on performance. In section 5, results and performance evaluation are investigated using the TSP problem as an implementation of parallel ACO algorithm. Finally, section 6 concludes the research and suggests the future work.

2 Related Work

Many strategies have been followed to implement ACO algorithm on different parallel platforms. In [9], Compute Unified Device Architecture (CUDA) is used to get the parallel throughput when executing more concurrent threads over GPUs. Results showed faster execution time with CUDA than OpenMP, but the main disadvantage of CUDA computing power is its dependence on GPU memory capacity related to problem size.

Marco Dorigo and Krzysztof Socha [10] addressed that the central component of ACO is the pheromone model. Based on the underlying model of the problem, parallelization of this component is the master point to the parallel ACO.

Bullnheimer et al. [11], introduced the parallel execution of the ants construction phase in a single colony. This research target was decreasing computations time by distributing ants to computing elements. They suggested two strategies for implementing ACO for parallelization: the synchronous parallel algorithm and the partially asynchronous parallel algorithm. Through their experiment, they used TSP and evaluated the speedup and efficiency. In the synchronous parallel algorithm, the speedup is poor for the small problem size and resulting to "slowdown" the efficiency close to

zero. While in large problem size, the speedup is improved by increasing the number of workers (slaves). Communication and idle time have a great effect on limiting the overall performance. The authors conclude that the second approach, partially asynchronous parallel algorithm, implemented the concept of parallelism with better speedup and efficiency. The disadvantage of this model was the communication overhead for the master ant waiting for the workers to finish their task.

Stützle [12], introduced the execution of multiple ant colonies, where the ant colonies are distributed to processors in order to increase the speed of computations and to improve solution quality by introducing cooperation between colonies. This method would be implemented through distributed memory model which would require a huge communication that caused high overhead affecting the overall performance.

Xiong Jie et al. [13] used message passing interface MPI with C language to present a new parallel ACO interacting multi ant colonies. The main drawback of this approach is the coarse-granularity where the master node have to wait for all slave nodes to finish their work and then updates with the new low cost solution.

This paper proposes a solution with OpenMP to get the performance gain of parallel regions. These parallel regions provide parallelizing to the ACO algorithm by controlling the time-consuming loops, avoiding race hazards and maintain load balance.

3 The ACO Algorithm

In ACO as a metaheuristic, cooperation is a key design component of ACO algorithms [14]. The artificial cooperating ants build a solution for a combinatorial optimization problem by traversing a fully connected graph. Solution is built in a constructive method. The solution component is denoted by c_{ij}, c is representing a set of all possible solution components. When combining c components with graph vertices V or with set of edges E the result would be the graph $G_C(V,E)$.

3.1 ACO Solution Steps

ACO algorithm consists of three main procedures which are:

- **ConstructAntsSolutions(edge selection)** phase: the ants traversed through adjacent neighbor nodes of the graph is made by a stochastic local decision according to two main factors, pheromone trails and heuristic information. The solution construction phase starts with a partial solution $s^p=\phi$. From the adjacent neighbors a feasible solution component $N(s^p) \subseteq C$ is added to the partial solution. The partial built solution made by an ant is evaluated for the purpose of using it later in the UpdatePheromones procedure. Dorigo [14] formed an equation for the probability of selecting solution component:

$$p(\boldsymbol{C}_{ij} \mid \boldsymbol{S}^{p}) = \frac{\boldsymbol{\tau}_{ij}^{\alpha} \cdot \boldsymbol{\eta}_{ij}^{\beta}}{\sum_{\boldsymbol{C}_{il} \in N(\boldsymbol{S}^{p})} \boldsymbol{\tau}_{ij}^{\alpha} \cdot \boldsymbol{\eta}_{ij}^{\beta}}, \forall \boldsymbol{C}_{ij} \in N(\boldsymbol{S}^{p}) \quad (1)$$

Where τ_{ij} is the deposited pheromone value in the transition from state *i* to state *j*, and η_{ij} is the heuristic value between *i, j*. Both τ_{ij} , η_{ij} associated with the component c_{ij}. Where α and β are two parameters which controls the parameters of τ_{ij} and ηij respectively, where $\alpha \geq 0, \beta \geq 1$

- **LocalSearch** phase: This step is started after solution construction phase and before pheromone update. The result of this step are locally optimized solutions. This is required - as a centralized action - to improve the solution construction phase.
- **UpdatePheromones** phase: is the most important phase where a procedure of pheromone level is increased or decreased. After all ants completed the solution, the following rule controls the pheromone update:

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij} + \sum_{k} \Delta\tau_{ij}^{k} \tag{2}$$

Where ρ is pheromone evaporation coefficient and $\Delta\tau_{ij}^{k}$ is the amount of pheromone released by *k*-th ants on the trip finished between i, j

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{Q}{L_k} & if\ ant\ k\ traverse\ edge(i,j) \\ 0 & else \end{cases} \tag{3}$$

Where Q is a constant, L_k is tour length traversed by the ant k.
Continues increase in pheromone levels in each iteration would produce an attractive path to the following iterations. This leads to the trap of local optima ants discarding the exploration of other connections. To explore new areas, pheromone evaporation rate is activated to participate in lowering pheromone levels in each tour.

3.2 The ACO Algorithm for TSP

As the exemplary task, Traveling Salesman Problem TSP is considered to verify the efficiency of the proposed parallel approach as well as some related aspects like the scheduling of threads, the race hazards and tuning of the effective number of threads.
In the algorithm of TSP, the weighted graph G = (N, A) where N is the number of nodes representing cities. The connection between cities (i,j) □ A and d_{ij} is the distance between (i,j). The τ_{ij} representing the desirability of visiting city j directly after visiting city i according to pheromone trails, η_{ij} depicts the heuristic information where $\eta_{ij} = 1/ d_{ij}$ and there will be a matrix of τ_{ij} which includes pheromone trails.
The value of pheromone at initial state for TSP is:

$$\tau_{ij}(0) = m / C_{min} \tag{4}$$

Where m is the number of ants, C_{min} is the minimum distance between any i, j. When ants planning to construct its path ant k determines the probability P of visiting the next city according to formula in (1).

The j is the city not visited yet by ant k, both α and β are two parameters which control the relative importance of pheromone (τ_{ij}) against heuristic information ($\eta_{ij} = 1/d_{ij}$), $tabu_k$ is the list of already visited cities by k-th ants. The update pheromone process starts after all ants have finished their tours construction. At first, pheromone values are lowered by a constant factor for all connections between cities. Then, pheromone levels are increased only for the visited connections by ants, pheromone evaporation determined by:

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij} \tag{5}$$

Consider ρ as pheromone evaporation rate, where $0 < \rho \leq 1$. After number of iterations, the ants release pheromone in all visited connections during their tour formula in (2) Where $\Delta\tau_{ij}^{k}(t)$ denotes the amount of pheromone deposited by ant k on the trip finished between nodes i and j defined as in formula (3).

4 Proposed ACO Parallelization by Using OpenMP

ACO is a potential candidate for parallelization for different reasons, including:
The individual independent behavior of ants.

- The large number of iterations required in updating pheromone trails.
- The computations needed for the single ant to construct a solution in the graph.
  Parallel ACO could be implemented with two different strategies [6]:
- Coarse-grained: single CPU is being used by many ants or even the whole colony with rarely information exchange between CPUs
- Fine-grained: few numbers of ants are to be assigned with each core of CPU with more communication and information exchange between them.

The main difference between previous two approaches is the amount of information exchange between the CPUs. Fine-grain model needs more communication which causes an overhead consuming most of the execution time. Coarse-grain parallelization model is most suitable for multiple colonies of ACO implementation [7]. Fine-grain parallelization strategy has been adopted in this paper to study the behavior of multithreading with relation to the multicores available in CPU with a single colony.

An improvement in ACO algorithm could be achieved mainly by using multithread programming with multi-core processors. This section introduces an implementation for parallel ACO using OpenMP platform. A shared memory model has been chosen to get the benefit of creating a common space sharing pheromone matrix without the overhead of communication, especially when applying both "ConstructAntSolutions" and "UpdatePheromones" processes. OpenMP is implemented to reduce the execution time and not altering the ACO algorithm with major change.

The main effort here is to analyze and select the places which consume most execution time in the sequential ACO and to overcome the problem of communication overhead by using the OpenMP directives. Larger and in-place OpenMP parallel

regions are used, because fragmented parallel regions would increase the overhead of creating and terminating threads.

4.1 Tuning Optimal Number of Threads

One of the major questions here when implementing parallel regions is: what is the optimal number of threads to execute through *for* loops? To answer this question, a hypothesis have been adopted. The optimal number of threads would depend on both parallel implementation of ACO and the number of multi-cores available in the CPU. This is according to two factors.

- Amdahl's law [8], which means that adding more threads would be neglected with no significant speedup because of sequential part.
- The number of threads can be chosen to be more than the number of cores. This is the case when a thread is in waiting/blocking condition. Hyperthreading availability in modern CPUs provides management for many threads per core.

4.2 Tuning Parallel Regions

The pseudocode of ACO is shown in Fig. 1, which simplifies the three main components of the algorithm. The "ConstructAntSolutions" is the function of asynchronous concurrent ants while visiting neighbor nodes of the graph. Ants progressively build their path towards the optimal solution with the help of "UpdatePheromones" function. In the function of "UpdatePheromones" the pheromone trails are updated with increased levels of pheromones by releasing more pheromone on connections between nodes, or the pheromone decreased by the effect of evaporation. Increasing pheromone levels means increasing the probability of successive future ants in their way to find the shortest path allowing only specific ants to release pheromone.

```
procedure ACOMetaheuristic
  Begin
  Set parameters, initialize pheromone trails
    while (termination condition not met) do
      ConstructAntSolutions
      ApplyLocalSearch  % optional
      UpdatePheromones
    end while
  end
```

Fig. 1. The pseudocode of ACO

The main experimental objective here is to apply a *pragma omp parallel* region to the main parts of ACO, first on *ConstructAntSolutions* only and then measure the performance. After that, the parallel region will be applied to *updatePheromone procedure*, where a parallel "for" applied with "n" number of threads. At the end of each

parallel region there will be an implicit automatic barrier, its mission is to synchronize with the main thread before starting new parallel region.

4.3 Tuning OpenMP Scheduling Clause

Three types of OpenMP schedule clause could be experimented to control the granularity of thread execution: static (which is the default), dynamic, and guided schedule. The default scheduling used in *parallel for* is static, which distributes the work and iterations between threads. This is not the case of different jobs assigned to different ants. The proposed solution adds the schedule dynamic clause to the "*parallel for*" loops to give a full control for the distribution of iterations to the available threads. The iteration granularity is determined by the chunk size. The main benefit of dynamic scheduling is its flexibility in assigning more chunks to threads that can finish their chunks earlier. The rule is, the fastest thread shouldn't wait for the slowest.

4.4 Eliminating Race Condition Hazards

The race condition would occur when many threads update the same memory location at the same time. ACO algorithm may suffer from this problem, especially when two or more ants are trying to update the pheromone matrix at the same time. To avoid data race condition in the process of increasing/decreasing pheromone levels, critical sections are applied.

However, in our proposed parallelization, each thread will be responsible for updating pheromone level of each edge. Thus, the value of pheromone level update is the sole responsibility of a single thread. Accordingly, race hazards can be eliminated.

5 Results and Performance Analysis

In this paper, Travel Salesman Problem (TSP) NP-hard problem has been chosen as a well-known application of the generic ACO algorithm. In this paper, TSP parameters were initially set, and OpenMP was applied as a parallelization API. After that, results were gathered from the experiment. Finally, the performance of ACO algorithm with OpenMP was finally analyzed.

5.1 ACO Parallelization Environment

In the conducted experiment of this paper, OpenMP 4.0 and Visual Studio Ultimate 2013, ACO algorithm was implemented in C++. Computer configuration is Intel® Core™ i5-460M 2.53GHz, CPU– L3 cache 3MB, 4GB RAM.

The parallel regions of OpenMP with number of threads n=2, 4, 8, 16, 32, 64 are applied, utilizing 1000 ants. Different sizes for TSP problem with 40, 80, 130 cities are used to test the scalability of the parallelization. The test and the analysis would measure the speedup to gauge the parallelization impact on execution time and effi-

ciency. The performance is measured by using speedup which shows the performance to determine the optimal solution in a specific computing time:

$$speedup = t_s / t_p \tag{6}$$

In equation (6), t_s is the time required to solve the problem with the sequential version of code on a specific computer, t_p is the time to solve the same problem with the parallel version of code using p threads on the same computer. And the efficiency of the parallel implementation is calculated through the equation:

$$efficiency = speedup / P \tag{7}$$

The strategy of implementation described before has been put under experiment by starting from an existing sequential implementation. Then, the appropriate OpenMP directives were added, the necessary changes were made as discussed before.

To achieve accurate results representing real execution time, code running was repeated ten times for every change in thread numbers. In this experiment, Sequential code was applied first to measure the difference between parallel and sequential versions of code. Tables 1, 2, 3 show the results of average execution time, speedup and efficiency when default schedule static was initially applied, then the application of dynamic schedule with n number of threads was compared showing the difference. By using k=1000 as number of ants, the experiment was sequentially executed with problem size of 40 cities of the ACO and the execution time was marked. Parallelization started with 2, 4, 8, 16, 32, and 64 threads respectively. Then, the same experiment was repeated with different problem sizes 80 and 130 cities. The speedup and efficiency are measured by equations (6) and (7).

Table 1. Ant colony size, 40 cities 1000 ants

Number of threads	Default Schedule Exec. time(sec)	Dynamic Schedule Execution time(sec)	Speedup (sequential to dynamic)	Efficiency
Sequential	1.5855	1.5855	-	-
2	1.2543	1.1264	1.41	0.70
4	1.0347	0.9427	1.68	0.42
8	1.0494	0.9338	1.70	0.21
16	1.0764	0.9430	1.68	0.11
32	1.0603	0.9454	1.68	0.05
64	1.0761	0.9650	1.64	0.02

Analyzing the results of execution times in table 2 has proved a better performance by using 4 and 8 threads, then no significant speedup was noticed on adding more threads. The colony size increased to 80 cities. A better performance took place with a leap in execution time especially after applying dynamic scheduling clause. The same could be addressed by increasing the TSP problem size to 130 cities as shown in Table 3. A fine tuning was done using schedule dynamic clause which caused a

noticed performance speedup. This is due to the dynamically generated chunks at runtime which control the thread execution over iterations.

Table 2. Ant colony size, 80 cities, 1000 ants

Number of threads	Dynamic Exec. time(sec)	speedup	efficiency
Sequential	7.0755	-	-
2	4.0492	1.75	0.87
4	2.7932	2.53	0.63
8	2.7204	2.60	0.33
16	2.7889	2.54	0.16
32	2.8113	2.52	0.08
64	2.8151	2.51	0.04

Table 3. Ant colony size, 130 cities 1000 ants

Number of threads	Default Schedule Execution time(sec)	Dynamic Schedule Execution time(sec)	speedup (sequential to dynamic)	efficiency
Sequential	25.9013	25.9013	-	-
2	17.764	10.6557	2.43	1.22
4	9.57293	7.3100	3.54	0.89
8	8.1691	7.2090	3.59	0.45
16	7.90743	7.2510	3.57	0.22
32	7.79117	7.3096	3.54	0.11
64	7.80114	7.3259	3.54	0.06

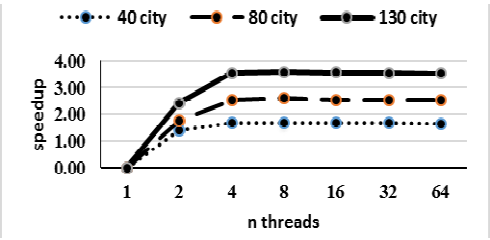

Fig. 2. The speedup with *n* number of threads applied on different ant colony sizes

After combining the results from the three tables 1, 2, and 3 in Fig. 2, a relative speedup for parallelization over sequential implementation was observed especially on increasing the TSP problem size 40, 80 and then 130 cities.

As shown in table 4, parallel regions of OpenMP wraps the most time consuming parts of ACO algorithm. When execution time was measured for each region, Updatepheromone was found to be the most time-consuming part. A speedup was achieved after applying OpenMP parallel. This is clearly illustrated in Fig. 3 which shows a significant time-consuming UpdatePheromone function and

AntSolutionConstruction is the second most time-consuming part. They both gain significant speedup after applying parallel regions of OpenMP.

Table 4. Execution time of Parallel regions against different *n* threads

number of threads	initialize PheromoneTrail	AntSolution Construction	update Pheromone	Overall execution time
1	0.000135	2.414022	9.531944	12.29551
2	0.000078	1.298681	4.116056	5.677514
4	0.000072	0.836327	3.056812	4.125318
8	0.000098	0.807538	3.000157	4.054615
16	0.000086	0.828095	3.060481	4.188573
32	0.000139	0.832196	3.0479	4.137231
64	0.000217	0.869248	3.024268	4.185221

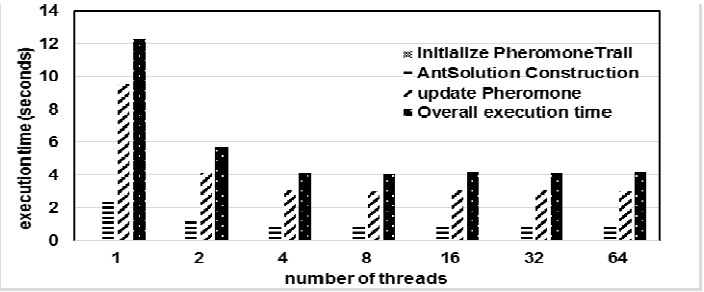

Fig. 3. Execution time of Parallel regions against different *n* threads with same problem size

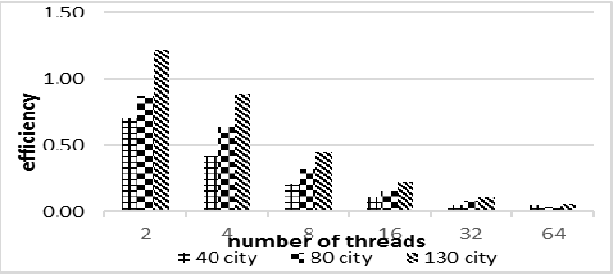

Fig. 4. Efficiency values when using 40, 80, and 130 city size

The experiment repeated with different numbers of threads 2, 4, 8, 16, 32, and 64 shown in Fig. 4 indicates an improvement in efficiency which occurred as a result of increasing problem size regarding the number of threads, since efficiency = speedup/ number of threads.

As the main goal is to provide better performance through parallelism, the experiments in this research would investigate the optimal number of threads needed. For this purpose, a tool of thread visualizing and monitoring the interaction and relation between threads and cores has been used. One selected tool is Microsoft concurrency

visualizer which is a plug-in tool for Visual studio 2013. Different numbers of threads were implemented in each run and results have been collected and analyzed in the results section.

In the current experiment, 1, 4, and 8 threads have been selected to be analyzed by Concurrency Visualizer on a machine with 4 logical cores for the following reasons:

- Finding the optimal number of threads related to the available number of cores.
- Visualizing and analysis of concurrently executing 4 threads that's equal to the number of logical cores.
- Visualizing and analysis of concurrently executing 8 threads that's more than the number of cores.
- Visualizing and analysis of the behavior of multithreads and how they execute, block, and synchronize.

Executing the ACO with a bigger number of threads than the number of cores, an overhead of context switching, synchronization, and preemption of the threads is detected. In the meanwhile, OpenMP gives a better utilization of the multicore environment. Fig. 5, shows a detailed view of 4 and 8 threads on 2 cores CPU with hyper-threading which are logically equivalent to 4 cores. When the number of threads is equal to the number of cores, threads are distributed among the available cores. The advantage of this is less synchronization and preemption time. Most of this saved time is assigned to execution causing the parallel threads to achieve better speedup. Whereas, if the number of threads largely exceeds the number of available cores, an overhead and time wasting is detected. This is because of thread blocking, synchronization, and context switching. This experiment shows the fact that the optimal number of threads should not exceed the available number of cores. Consequently, if the possibility of thread blocking does not exist, the number of threads should be optimized according to the available number of cores, as each thread will utilize each CPU core.

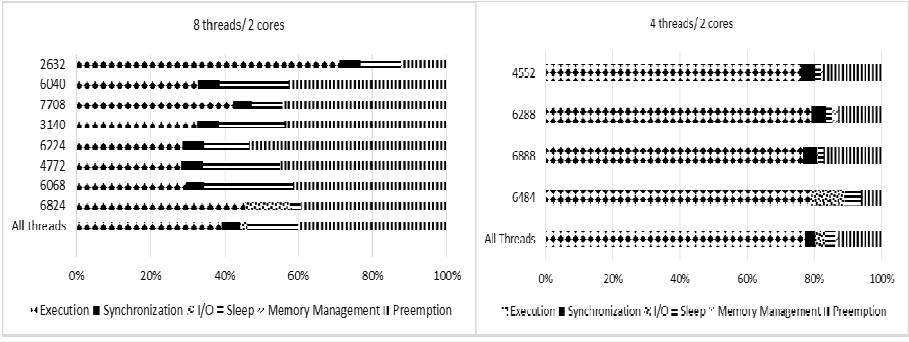

Fig. 5. Thread distribution when executing 4 and 8 threads/2 cores CPU with hyper-threading

6 Conclusion and Future Work

In this research, parallel implementation of ACO using OpenMP API directives effectively solves the common TSP problem. Results were evaluated, and comparison between sequential and parallel multithread were also analyzed. OpenMP parallel regions achieved a speedup more than 3X of sequential execution. The optimal number of threads was found to be equal to the number of processors available. With TSP sizes of 40, 80, and 130 cities, better speedup was detected with a larger number of cities. Moreover, tuning was added to the implementation of parallel ACO using OpenMP with different schedules clauses. Dynamic schedule was found to achieve better performance with average speedup 8-25% than default schedule clause especially on increasing the number of cities. This paper shows an upper border of speedup related to the available number of cores.

The future work would be oriented towards using this kind of parallel implementation using OpenMP for different newly metaheuristics such as Cuckoo search (CS) and to compare results to parallel ACO and measures which one positively affected more by the parallelization of OpenMP platform.

References

1. Alba, E.: Parallel Metaheuristics: A New Class of Algorithms. Wiley. ISBN 0-471-67806-6, July 2005
2. Dorigo, M.: Optimization, Learning and Natural Algorithms, PhD thesis, Politecnico di Milano, Italy (1992)
3. Chen, W.N., Zhang, J.: Ant Colony Optimization Approach to Grid Workflow Scheduling Problem with Various QoS Requirements. IEEE Transactions on Systems, Cybernetics-Part C: Applications and Reviews **31**(1), 29–43 (2009)
4. Donati, A.V., et al.: Time dependent vehicle routing problem with a multi ant colony system. European Journal of Operational Research **185**(3), 1174–1191 (2008)
5. Dumitrescu, I., Stützle, T.: Combinations of local search and exact algorithms. In: Raidl, G.R., Cagnoni, S., Cardalda, J.J., Corne, D.W., Gottlieb, J., Guillot, A., Hart, E., Johnson, C.G., Marchiori, E., Meyer, J.-A., Middendorf, M. (eds.) EvoIASP 2003, EvoWorkshops 2003, EvoSTIM 2003, EvoROB/EvoRobot 2003, EvoCOP 2003, EvoBIO 2003, and EvoMUSART 2003. LNCS, vol. 2611, pp. 211–223. Springer, Heidelberg (2003)
6. Xue-dong, X., et al.: The basic principle and application of ant colony optimization algorithm. In: Artificial Intelligence and Education (ICAIE) conference, Hangzhou, China (2010)
7. Dorigo, M., Gambardella, L.M.: Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Transactions on Evolutionary Computation (2002)
8. Amdahl, G.: Validity of the single processor approach to achieving large Scale computing capabilities. In: AFIPS Conference Proceedings, vol. 30, pp. 483–485. Thompson Book, Washington, D.C., April 1967
9. Arnautovic, M., et al.: Parallelization of the ant colony optimization for the shortest path problem using OpenMP and CUDA (MIPRO), Opatija, Croatia (2013)

10. Dorigo, M., Socha, K.: An Introduction to Ant Colony Optimization. Universit de Libre de Bruxelles, CP 194/6, Brussels (2007). http://iridia.ulb.ac.be
11. Bullnheimer, B., Kotsis, G., Strauss, C.: Parallelization strategies for the ant system. In: De Leone, R., Murli, A., Pardalos, P., Toraldo, G. (eds.) High Performance Algorithms and Software in Nonlinear Optimization. Applied Optimization, vol. 24, Dordrecht (1997)
12. Stützle, T.: Parallelization strategies for ant colony optimization. In: Eiben, A.E., Bäck, T., Schoenauer, M., Schwefel, H.-P. (eds.) PPSN 1998. LNCS, vol. 1498, pp. 722–731. Springer, Heidelberg (1998)
13. Xiong, J., Liu, C., Chen, Z.: A new parallel ant colony optimization algorithm based on message passing interface. In: Computational Intelligence and Industrial Application, PACIIA 2008. Pacific-Asia Workshop, Wuhan (2008)
14. Dorigo, M., Stutzle, T.: "Ant colony optimization" A Bradford Book. The MIT Press, Cambridge (2004)