International Journal of Swarm Intelligence and Evolutionary Computation

Abouelfarag et al., Int J Swarm Intel Evol Comput 2014, 4:1
DOI: 10.4172/2090-4908.1000117

Research Article Open Access

Performance Analysis and Tuning for Parallelization of Ant Colony Optimization Using Open MP

Ahmed A Abouelfarag[1], **Walid Mohamed Aly**[2*], **and Ashraf G Elbialy**[2]

[1]*College of Engineering and Technology, Arab Academy for Science and Technology, and Maritime Transport, Egypt*
[2]*College of Computing and Information Technology, Arab Academy for Science and Technology, Maritime Transport, Alexandria, Egypt*

Abstract

Ant colony optimization algorithm (ACO) is a soft computing met heuristic that belongs to swarm intelligence methods. ACO has proven a well performance in solving certain NP-hard problems in polynomial time. This paper presents the analysis, design and implementation of ACO as a Parallel Me-heuristics using the Open MP 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 con-figurations 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; Open MP; Parallel threads

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, met heuristics provide a kind of near-optimal solution for a wide range of NP-hard problems [1].

Ant Colony Optimization (ACO) algorithm – was first introduced in 1992 by Marco Dorigo [2] is a metaheuristic swarm optimization technique which typically searches for an optimal path in a connected graph. ACO is based on the behaviour of ants seeking the shortest path between their colony and a food source. ACO was pro-posed as a solution when suffering from limited computation capacity and incomplete Information [3]. Ant colony optimization meta-heuristic proved a significant performance improvement compared with other meta-heuristic techniques in solving many NP-hard problems such as solving the traveling salesman problem [4].

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

In this paper, Open MP 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++. Open MP is implemented with its parallel regions, directives to control loop flow. Scheduling clause for fine tuning. For eliminating race condition, omp critical sections have been also implemented.

Traveling salesman problem TSP is selected as a test case. The importance of the TSP problem comes from its history of applications with many metaheuristics. On the other hand TSP is easy to be mapped with similar real life problems.

The behaviour of single ant found to be similar to the salesman in TSP. The parallelization of many ants would significantly increase the possibility of achieving a satisfactory solution in a reasonable time.

The speedup gain in parallelization of a typical sequential TSP with ACO depends mainly on the proper and accurate analysis of where parallelization should be placed in the algorithm. Theoretically, Amdahl's law [5-7] limits the expected speedup achieved to an algorithm by a relation between parts that could be parallel to the parts remain serial. Therefore, a detailed analysis of the algorithm is done in this paper to place the proper parallel directives of Open MP in the most promising places. One target of the experiment is to assigning the optimal number of parallel threads and tuning them dynamically with the available number of CPU cores to get effective speed up.

This paper is organized as follows: 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 Open MP is presented where its sub-sections show the analysis of different elements of Open MP 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.

Related Work

Many strategies have been followed to implement ACO algorithm on different par-allel platforms. In [8], Compute Unified Device Architecture (CUDA) is used to get the parallel throughput when executing more concurrent threads over GPUs. The concept of master-slave ants has been adapted. Results showed faster execution time with CUDA than Open MP, but the main disadvantage of CUDA computing power is its dependence on GPU memory capacity related to problem size. Threading Building Blocks (TBB) library created by Intel also combined with ACO to form a solution to the TSP problem [9]. This

***Corresponding author:** Walid Mohamed Aly, College of Computing and Information Technology, Arab Academy for Science and Technology, Maritime Transport, Alexandria, Egypt, E-mail: walidmaly@yahoo.com

Received June 21, 2014; **Accepted** July 20, 2015; **Published** July 25, 2015

Citation: Abouelfarag AA, Mohamed Aly W, Elbialy AG (2015) Performance Analysis and Tuning for Parallelization of Ant Colony Optimization Using Open MP Int J Swarm Intel Evol Comput 4: 117. doi: 10.4172/2090-4908.1000117

Copyright: © 2015 Abouelfarag AA et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

requires special hardware to be added to the system, while a solution over the multi-core processors is not introduced.

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.

Some of ACO implementation efforts were oriented towards the granularity, which is the relation between computation and communication. Randall and Lewis [11] classified some general parallelization strategies of ACO implementation with application of these strategies to travelling salesman problem TSP. They concluded that the parallel solution is dependent on the nature of the problem being solved. As for TSP, the authors described five possible parallelization strategies for ACO metaheuristics based on master/slave approach. They proposed some rules as a guide to the parallelization of ACO met heuristic and showing a reasonable speedup when implementing ant tour construction in parallel using MPI model on MIMD architecture. Parallel ants assigned to each processor and placed randomly, and reasonable communication overhead is achieved for this technique.

Bullnheimer et al. [12] 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 main concept in the partially asynchronous parallel algorithm is to reduce the communication overhead. Because of communication reduction, improvement with this approach takes place in the speedup and efficiency even in small size problems. 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 over-head and the time consumed for the master ant waiting for the workers to finish their task. For the current research Open MP is used to overcome this drawback and evaluate the performance gained from applying parallel regions.

Stützle [13], 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.

Ling Chen et al. [14] similarly focused on reducing the communication overhead but with controlling the time interval of communication between processors and ex-change information adaptively according to the diversity of the solutions. To adaptively calculate the time interval each communication cycle, an expected overhead would occur, especially with huge populations.

Xiong Jie et al. [15] 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 has 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 Open MP, to get the better performance gain of parallel regions by controlling the time-consuming loops. Automatic barriers found in Open MP are implicit way of distributing and collecting threads. A fine-grain has been selected as time update of best-solution. Side effects of race condition are managed by critical sections approach. Synchronization is done in harmony because of dynamic scheduling, where the thread execution is divided to a number of small chunks which are created at runtime and relative to the iterations and work load.

The ACO Algorithm

ACO "is a met heuristic in which a colony of artificial ants cooperates in finding good solutions to difficult discrete optimization problems. Cooperation is a key design component of ACO algorithms" [16]. The enforcement of the shortest path is achieved by adding more pheromone trails by ants.

In ACO, artificial ants build a solution for a combinatorial optimization problem by traversing a fully connected graph. ACO algorithm is building the solution 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 GC (*V*,*E*).

In each solution component c*ij*, a value of pheromone trail denoted as *τij* is associated with it. The pheromone values are used and updated by the ACO algorithm during the search.

By the help of pheromone values, ants move along the edges of the graph. Incrementally, the solution building process is updated. Ants deposit a certain amount of pheromone on the traversed edges. The following ants would be guided by this pheromone information and would be able to discover more areas of the search space.

ACO algorithm consists of three main procedures which are (i) Construct Ants Solutions by all ants, (ii) Update Pheromones, and (iii) the optional Local Search which improve the construction solution. These three phases are explained as follows:

Construct Ants Solutions (edge selection) phase: The moving of ants through adjacent neighbour nodes of the graph is made by a stochastic local decision according to two main, pheromone trails and heuristic information. The solution

Construction phase starts with a partial solution $s^p=\phi$. From the adjacent neighbours a feasible solution component $N(s^p) \subseteq C$ is added to the partial solution. The solution construction phase would be described as a path on the construction graph GC(*V*,*E*). The partial built solution made by an ant is evaluated to be used later in the Update Pheromones procedure and according to this a pheromone amount is decided to be released. Dorigo et al. [16] formed an equation for the probability of selecting solution component:

$$p\left(C / S^p\right) = \frac{\tau_{ij}^{\alpha}.\eta_{ij}^{\beta}}{\sum_{C_{il} \in N(S^p)} \tau_{ij}^{\alpha}.\eta_{ij}^{\beta}}, \forall C_{ij} \in N(S^p) \quad (1)$$

Where *τij* is the deposited pheromone value in the transition from state *i* to state *j*, and *ηij* is the heuristic value between *i, j*. Both *τij, ηij* associated with the component *cij*. Where α and β are two parameters which controls the parameters of *τij and*

ηij respectively, where $\alpha \geq 0, \beta \geq 1$

Local Search (Daemon Actions) phase: is an optional phase in

some specific problems. This step is started after solution construction phase and before pheromone update. The result of this step is locally optimized solutions. This is required - as a centralized action - to improve the solution construction phase.

Update Pheromones 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:

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

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

$$\Delta\tau_{ij}^k = \begin{cases} \frac{Q}{L_k} if\ ant\ is\ transverseedge(i,j) \\ 0 else \end{cases} \qquad (3)$$

Where Q is a constant, L_k is tour length by the ant k.

Continues increase pheromone levels in each iteration would produce an attractive path to the following iterations. This leads to the trap of local optima when direct-ing all the ants to this solution and discarding the exploration of other connections. Therefore, pheromone evaporation activated participating in lowering pheromone levels in each tour. Because of this, new areas would be explored in the search space.

The ACO algorithm for TSP

In the algorithm of TSP, while the salesman heuristically searching for the shortest path from his home city to a set of cities in a graph; he only once visits each city. The weighted graph G = (N, A) where N is the number of nodes representing cities, A is the connections between cities. The connection between cities (i,j) ϵ A and dij is the distance between (i,j). The τij representing the desirability of visiting city j directly after visiting city i according to pheromone trails, ηij depicts the heuristic information where ηij =1/ dij and there will be a matrix of τij which includes pheromone trails.

The value of pheromone at initial state for TSP is:

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

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 determine the probability P of visiting the next city according t

$$P_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in N_i^K} [\tau_{ij}]^\alpha\ [\eta_{ij}]^\beta}, j \in N_i^K \qquad (5)$$

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 (τij) against heuristic information (ηij=1/dij), tabuk 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, then pheromone levels are increased only for the visited connections by ants, pheromone evaporation determined by:

$$\tau\, ij \leftarrow (1-\rho)_{\tau\ ij} \qquad (6)$$

Consider ρ as pheromone evaporation rate, where 0 < ρ ≤ 1. After number of iterations, the ants release pheromone in all visited connections during their tour due to:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^m \Delta\tau_{ij}^k \qquad (7)$$

$\Delta\tau_{ij}^k(t)$ The amount of pheromone ant k deposits on the trip finished between *i, j*.

$$\Delta\tau_{ij}^k(t) = \begin{cases} \frac{Q}{L_k} if\ ant\ k\ traverseedge(i,j) \\ 0 else \end{cases} \qquad (8)$$

Where Q is a constant, Lk is tour length by the ant k.

Arnautović et al. [7,8] describes a sequential ant colony optimization algorithm pseudo code, as shown in Figure 1:

Proposed ACO Parallelization Using Open MP

ACO is a potential candidate for parallelization for different reasons, including:

i. The individual independent behaviour of ants.

ii. The large number of iterations required in searching and updating pheromone trails on visited edges.

iii. The huge computation power needed for the single ant to construct a solution in the graph.

Parallel ACO could be implemented with two different strategies [5]:

- 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 [6]. Fine-grain parallelization strategy has been adopted in this paper to study the behaviour of multithreading with relation to the multicores available in CPU with a single colony.

```
1  optimalPath ← initializeOptimalPathToZero()
2  optimalPrice ← optimalPath→calculatePrice()
3  for every edge e in the graph
4 e→initializePheromoneTrail()
5  end for
6  repeat
7 for each ant in the colony do
8   path ← ant→buildSolution()
9   price ← path→calculatePrice()
10  if price is better than optimalPrice
11    optimalPath ← path
12    end if
13  end for
14  for every edge e in the graph
15    e→evaporatePheromones()
16  end for
17  for every visited edge A in the graph
18    A→updatePheromones()
19  end for
20  until maximum number of iterations is reached
```

Figure 1: The pseudo code of sequential ACO [8].

The Open MP API library provides an effective way to manage the shared-memory model and the communication overhead between CPUs.

An improvement in ACO algorithm could be achieved mainly by using multi-thread programming with multi-core processors. The independent behaviour of each ant from other ants makes ACO a suitable candidate for parallel/independent threads. This section introduces an implementation for parallel ACO using Open MP 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 "Construct Ant Solutions" and "Update Pheromones" processes.

This method will decrease communication overhead where the same memory blocks are shared between different threads. Using Open MP parallel directive for loops can be executed using #pragma omp parallel for with the available number of threads, and number of threads could be specified with #pragma omp parallel for num_threads(n) where n is the number of threads to be used. The goal of using Open MP is to reduce the execution time and not altering the ACO algorithm with major change. For better performance, the #pragma omp parallel num threads (n) should be the main parallel region. While the #pragma omp for directly before the *for* loops.

The main effort here is to analyse and select the places which consume most execution time in the sequential ACO and to overcome the problem of communication overhead by using the Open MP directives. Using parallel regions indiscriminately would limit the expected speedup. Fragmented parallel regions increase the overhead of creating and terminating threads. Larger and in-place Open MP parallel directives are better than wrapping every loop with Open MP parallel pragmas.

Tuning optimal number of threads

One of the major questions here when implementing parallel regions is answering the question: what is the optimal number of threads to execute through *for* loops? To find the answer of this question, a hypothesis has 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 three factors.

- Amdahl's law, which means that adding more threads, would be neglected with no significant speedup because of sequential part.
- Ability to assign threads to cores (CPUs). This is related to Gustafson's law where the optimal number of threads would depend on how much cores available on sys-tem.
- 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. Hyper threading availability in modern CPUs provides management for many threads per core.

With the nature of the ACO, no threads will be blocked waiting for a resource , thus we can conclude that the number of the threads will be typically the number of cores, creation of extra threads will not be of any benefits.

Tuning parallel regions

The pseudo code of ACO is shown in Figure 2, which simplifies the three main components of the algorithm. The "Construct Ant Solutions" is the function of asynchronous concurrent ants while visiting neighbour nodes of the graph. Ants progressively build their path towards the optimal solution with the help of "Update Pheromones" function. In the function of "Update Pheromones" 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.

As shown previously by pseudo code of ACO in Figure 2, the three main steps of the algorithm are: *Construct Ant Solutions*, *Apply Local Search*, *Update Pheromones*. Through the experimental performance analysis, parallel regions of Open MP will be implemented over the most time consuming parts in the ACO algorithm. For this reason, those three main parts were selected as an initial direction to parallelize the ACO algorithm. And by getting the performance and time consumed in each region results would reveal which part would be the most time consuming and therefor requiring to be parallelized.

The main experimental objective here is to apply a *pragma omp* ***parallel*** region to the main parts of ACO, first on *Construct Ant Solutions* only and then add *update Pheromone* to the parallel region, where a parallel "for" applied with "n" number of threads. At the end of each parallel region will be an implicit automatic barrier, its mission is to synchronize with the main thread before starting new parallel region. Figure 3 Show places where parallel regions in blue colour, automatic barriers exist.

Tuning Open MP scheduling clause

Three types of Open MP schedule clause could be used to control the granularity of thread execution: static (which is the default), dynamic, and guided:

- *Static* schedule is implicitly applied even if schedule clause doesn't appear in the code. During compile time, chunks are scheduled to threads. Static schedule characterized by each thread in the team is nearly exposed to the same number of iterations as the other threads do. In addition to that, distribution of work requires no synchronization, and nearly all threads converge at the same finishing time. Iteration assignment to threads is determined as a function of iteration/thread number.
- *Dynamic* schedule controls and organizes the delivery and receipt of the chunk iterations over the thread at runtime. *Dynamic* schedule is well used when threads are assigned to different work load or time. Synchronization is required to dynamically assign available thread to iterations. Hence, the speedup is the result, signifying the fact that the available threads have no

```
1.  procedure ACOMetaheuristic
2.  Begin
3.      Set parameters, initialize pheromone trails
4.      while (termination condition not met) do
5.          ConstructAntSolutions
6.          ApplyLocalSearch  % optional
7.          UpdatePheromones
8.      end while
9.  end
```

Figure 2: The pseudocode of ACO [16].

```
1. optimalPath ← initializeOptimalPathToZero()
2. optimalPrice ← optimalPath→calculatePrice()
3. start parallel region1
4. for every edge e in the graph
5. e→initializePheromoneTrail()
6. end for
7. end parallel region
8. repeat
9. start parallel region2
10. for each ant in the colony do
11. path ← ant→AntSolutionConstruction()
12. price ← path→calculatePrice()
13. if price is better than optimalPrice
14. optimalPath ← path
15. end if
16. end for
17. end parallel region
18. Start parallel region3
19. for every edge e in the graph
20. e→evaporatePheromones()
21. end for
22. for every visited edge A in the graph
23. A→updatePheromones()
24. end for
25. end parallel region
26. until maximum number of iterations is reached
```

Figure 3: The pseudocode of parallel ACO.

idle time waiting for busy or slow threads. When not specified, default chunk size is 1.

- *Guided* schedule, where chunk size is first determined by implementation, then decreased to the minimum size specified by the developer.

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 over iteration distribution over 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. This means that the chunk size should be considered to obtain the most performance from load balance, synchronization and computation costs.

Analysis of race condition hazards

The race condition would occur when many threads update the same memory location at the same time. Obviously, ACO algorithm would suffer from this problem, especially when two or more ants are trying to update the pheromone matrix at the same time. One of the ants would read the value of pheromone while the other ant is trying to update the same value. Race condition effect would occur while increasing or de-creasing pheromone levels by ants. 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 up-dating the pheromone level of each edge. Thus, the value of pheromone level is the sole responsibility of a single thread. Accordingly, race hazards can be eliminated.

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 Open MP was applied as a parallelization API. After that, results were gathered from the experiment. Finally, the performance of ACO algorithm with Open MP was analysed.

ACO parallelization environment

In the conducted experiment of this paper, Open MP 4.0 and Visual Studio Ultimate 2013, ACO algorithm has been implemented in C++. Two different computers are used. The first one is Intel® Core™ i5-460M 2.53GHz, CPU– L3 cache 3MB, 4GB RAM. The second is Intel® Pentium4-2.8GHz, CPU – L2 cache 1MB, 512MB RAM.

The parallel regions of Open MP with number of threads n=2, 4, 8, 16, 32, 64 are applied, utilizing 1000 ants. Different problem sizes 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 efficiency. The achieved parallel performance is measured by using speedup which shows the performance to determine the optimal solution in a specific computing time:

$$speedup = ts / tp \qquad (9)$$

In equation (9), ts is the time required to solve the problem with the fastest sequential code on a specific computer, tp is the time to solve the same problem with the parallel code using P processors on the same computer. And the efficiency of the parallel implementation is calculated through the equation:

$$efficiency = speedup / P \qquad (10)$$

The strategy of implementation described before has been put under experiment by starting from an existing sequential implementation in C++. Then, the appropriate Open MP directives were added, and 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, and the average time was calculated after that. In this experiment, Tables 1, 2, 3, 4 show the results of average execution time 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 as shown in equations (9) and (10).

Analysing the results of execution times in table 2 has proved a better performance by using 4 and 8 threads, and 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 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.

After combining the results from three tables, 1, 2 and 3 in Figure

4, a relative speedup for parallelization over sequential implementation was observed especially on increasing the problem size 40, 80 and then 130 cities.

Sequential code (without Open MP) was applied on hardware with a single core. Tables 4, 5, and 6 proved that there would be an overhead if increasing the thread numbers was attempted on a single core machine. Parallel regions here have no use because all threads have to synchronize and work as if it were sequential. Similarly, the cost of thread creation and killing is very high. Moreover, execution time was even slower than pure sequential execution.

Number of threads	Default Schedule Exec. Time (sec)	Dynamic Schedule Execution time (sec)	Speedup (sequential to dynamic)	efficiency
1	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	0.05

Table 1: Ant colony size, 40 cities, 1000 ants.

Number of threads	Dynamic Exec. Time (sec)	Speed up	efficiency
1	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 2: Ant colony size, 80 cities, 1000 ants.

Number of threads	Dynamic Exec. Time (sec)	Speed up	efficiency
1	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.

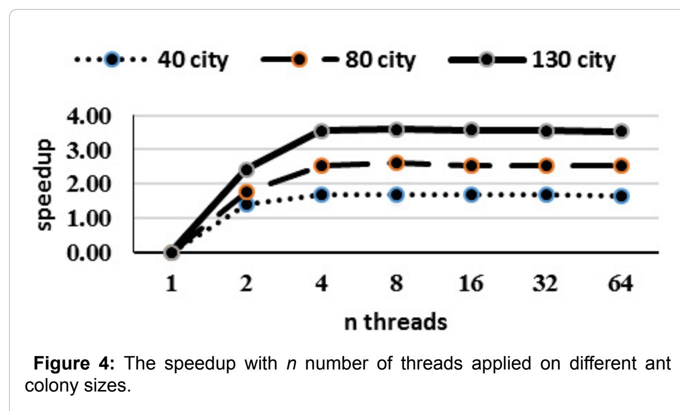

Figure 4: The speedup with *n* number of threads applied on different ant colony sizes.

Number of threads	Dynamic Schedule Execution time (sec)	Speed up (sequential to dynamic)	efficiency
1	1.6580	-	-
2	1.2362	1.34	0.67
4	1.4321	1.16	0.29
8	1.5466	1.07	0.13
16	1.5898	1.04	0.07
32	1.7845	0.93	0.03
64	1.8425	0.90	0.03

Table 4: Ant colony size, 40 cities, 1000 ants, with P4 2.8 GHz, Single core Intel.

Number of threads	Dynamic Schedule Execution time (sec)	Speed up (sequential to dynamic)	efficiency
1	1.6580	-	-
2	1.2362	1.34	0.67
4	1.4321	1.16	0.29
8	1.5466	1.07	0.13
16	1.5898	1.04	0.07
32	1.7845	0.93	0.03
64	1.8425	0.90	0.03

Table 5: Ant colony size, 80 cities, 1000 ants, P4 2.8 GHz GB RAM Intel.

Number of threads	Execution time (sec)	Speed up	efficiency
1	7.5460	-	-
2	6.3175	1.19	0.60
4	7.5460	1.00	0.25
8	7.5140	1.00	0.13
16	8.7851	0.86	0.05
32	10.5492	0.72	0.02
64	11.4560	0.66	0.01

Table 6: 130 cities, 1000 ants, P4 2.8 GHz GB RAM Intel.

As shown in Table 7, parallel regions of Open MP wraps the most time consuming parts of ACO algorithm. When execution time was measured for each region, Update pheromone was found to be the most time-consuming part. A speedup was achieved after applying Open MP parallel. This is clearly illustrated in Figure 5 which shows a significant time-consuming Update Pheromone function and Ant Solution Construction is the second most time-consuming part. They both gain significant speedup after applying parallel regions of Open MP.

The experiment repeated with different numbers of threads 2, 4, 8, 16, 32, and 64 shown in Figure 6 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 analysed in the results section.

In the current experiment, 1, 4, and 8 threads have been selected

Citation: Abouelfarag AA, Mohamed Aly W, Elbialy AG (2015) Performance Analysis and Tuning for Parallelization of Ant Colony Optimization Using Open MP. Int J Swarm Intel Evol Comput 4: 117. doi: 10.4172/2090-4908.1000117

number of threads	initialize Pheromone Trail	Ant Solution 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

Table 7: Execution time of Parallel regions against different n threads.

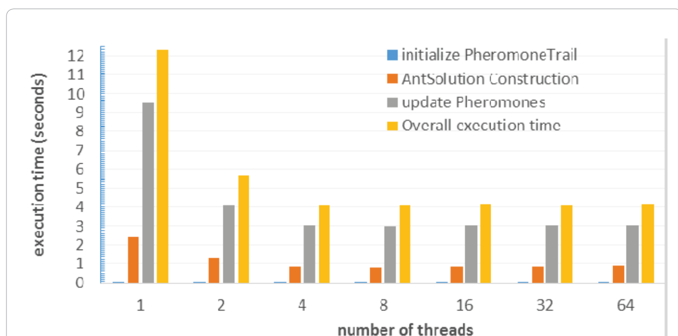

Figure 5: Execution time of Parallel regions against different *n* threads with same problem size.

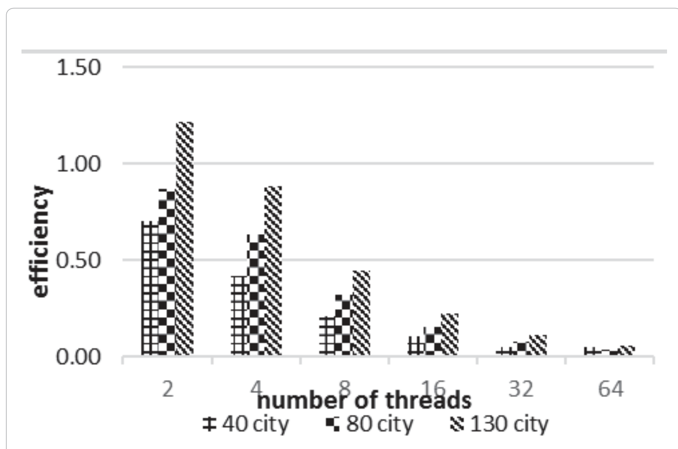

Figure 6: Efficiency values when using 40, 80, and 130 city sizes.

to be analysed 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 behaviour of multithreads and how they execute, block, and synchronize.

As shown in Figure 7, executing the ACO with a bigger number of threads than the number of cores, an overhead of context switching,

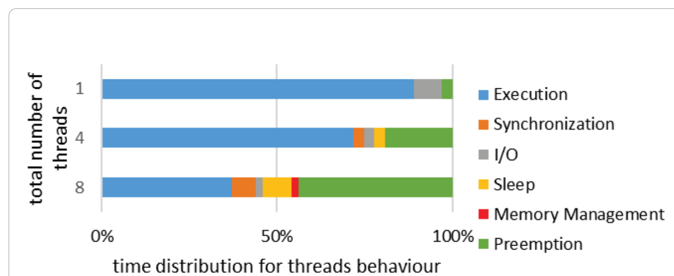

Figure 7: Implementing different numbers of threads on a 2 cores CPU.

Figure 8: The thread states percentage distribution of executing 4 and 8 threads/2 cores CPU with hyper-threading.

synchronization, and pre-emption of the threads is detected. In the meanwhile, Open MP gives a better utilization of the multicore environment. Figure 8, 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 avail-able cores. The advantage of this is less synchronization and pre-emption 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.

Conclusion and Future Work

In this research, parallel implementation of ACO using Open MP API directives effectively solves the common TSP problem. Results were evaluated, and comparison between sequential and parallel multithread were also analysed. Open MP 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 Open MP 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.

Single Instruction Multiple Data (SIMD) has a main role in performance improvements and code acceleration. The new technology of developing processors by Intel SSE, AVX, and AVX-512 provides vectorization to the loops which exist in most metaheuristic algorithms. The parallelization of these loops and using vector units available in new processor architectures are expected to effectively improve the performance and speedup of ACO. To this end, the future work would be oriented to-wards using this kind of important architectures.

References

1. Alba E (2005) “Parallel Met heuristics: A New Class of Algorithms” Wiley ISBN 0-471-67806-6.
2. Dorigo M (1992) “Optimization, Learning and Natural Algorithms,” PhD thesis, Polytechnic di Milano, Italy.
3. Blum C and Roli A (2003) “Met heuristics in combinatorial optimization: Overview and conceptual comparison”, ACM Computing Surveys (CSUR), Volume 35 Issue 3, Pages 268-308.
4. Dumitrescu I and Stützle T (2003) "Combinations of local search and exact algorithms." In *Applications of Evolutionary Computing*: 211-223. Springer Berlin Heidelberg.
5. Xue Xue-dong (2010) “The basic principle and application of ant colony optimization algorithm” Artificial Intelligence and Education (ICAIE) conference, Hangzhou, China.
6. Dorigo M, Gambardella LM (2002). “Ant colony system: A cooperative learning approach to the traveling salesman problem”, Evolutionary Computation, IEEE Transactions.
7. Amdahl G (1967) “Validity of the single processor approach to achieving large Scale Computing capabilities.” In AFIPS Conference Proceedings, Vol.30:483-485, Washington, D.C, Thompson Book.
8. Arnautovic M (May 2013) “Parallelization of the ant colony optimization for the shortest path problem using Open MP and CUDA”, Information and Communication Technology Electronics and Microelectronics (MIPRO), 36th International Convention on, 20-24 May Pages1273 – 1277, Opatija, Croatia.
9. Gendreau M and Potvin J (sep 2010) “Handbook of meta heuristics” Springer.
10. Dorigo M and Socha K (April 2010) “An Introduction to Ant Colony Optimization”, University de Libre de Bruxelles, CP 194/6, Brussels, Belgium. http://iridia.ulb.ac.be.
11. Randall M and Lewis A (2002) “A parallel implementation of ant colony optimization”, J Parallel Distributed Computing.
12. Bullnheimer B, Kotsis G and Strauss C (1997) “Parallelization strategies for the ant system.” In R De Leone, A Murli, P Pardalos, and G Toraldo, editors, High Performance Algo-rithms and Software in Nonlinear Optimization, volume 24 of Applied Optimization, Dordrecht.
13. Stützle T (1998) “Parallelisation strategies for ant colony optimization.” In AE Eiben T, Bäck H, Schwefel P and Schoenauer M editors, Proceedings of the Fifth International Con-ference on Parallel Problem Solving from Nature (PPSN V) Springer-Verlag, New York.
14. Ling C, Hai-Ying S and Shu W (July 2008)“Parallel implementation of ant colony optimization on MPP,” pages 981 - 986 Machine Learning and Cybernetics. International Conference on (Volume: 2) Kunming, China.
15. Xiong J, Liu C and Chen Z (2008) “A New Parallel Ant Colony Optimization Algorithm Based On Message Passing Interface”, Computational Intelligence and Industrial Application, PACIIA '08. Pacific-Asia Workshop, Wuhan, China.
16. Dorigo M and Stutzle T (2004) “Ant colony optimization” A Bradford Book, The MIT Press, Cam-bridge, Massachusetts London, England.

Citation: Abouelfarag AA, Mohamed Aly W, Elbialy AG (2015) Performance Analysis and Tuning for Parallelization of Ant Colony Optimization Using Open MP. Int J Swarm Intel Evol Comput 4: 117. doi: 10.4172/2090-4908.1000117