Fine-grained Parallel Ant Colony System for Shared-Memory Architectures

Ali Hadian
Iran University of Science and Technology
Tehran, Iran

Saeed Shahrivari
Tarbiat Modares University
Tehran, Iran

Behrouz Minaei-Bidgoli
Iran University of Science and Technology
Tehran, Iran

ABSTRACT
Although Ant Colony Systems (ACS) have gained much attention in last two decades but slow execution and convergence speed are still two challenges for these meta-heuristic algorithms. Many parallel implementations have been proposed for faster execution. However, most of available implementations use coarse-grained synchronization mechanisms that are not efficient and scalable. In this work, we have taken a fine-grained (ant-level) approach that is more efficient and scalable. We have used traveling salesman problem as a test case and have presented a parallel fine-grained implementation for shared-memory multi-core systems. Our experimental results show that our proposed parallel implementation can achieve considerably higher speedup values on modern multicore processors.

General Terms
Optimization, Parallel Processing.

Keywords
Ant Colony System, Parallel Computing, Traveling Salesman Problem, Shared Memory System.

1. INTRODUCTION
Swarm intelligence methods have recently become common tools for combinatorial optimization problems (COPs). These methods try to find the optimal solution using collective behavior of agents via an iterative process. The interactions in such systems may be local direct message passing between the individuals or indirectly through the environment i.e. stigmergy.

Ant Colony System (ACS) [1] is a well-known swarm-based optimization method, in which the optimization is performed by the foraging behavior of ants trying to find the shortest path to a food source in the environment [2]. As in its ancestor, the Ant System (AS) [3], the density of deposited pheromone is used to share prior experience between the colony members.

Due to the iterative nature of ACS, the process is highly time-consuming. As in other population based meta-heuristic methods, such as genetic algorithm, it may take a long time, (in some cases, several days) to run the ACS for a reasonable number of iterations.

Many research works have been done on parallelizing ant colony optimization. Most of them have focused on clustered computing techniques. In such systems, the communication between the computers is the bottleneck[4]. Therefore, most of the previous works have used coarse-grained implementations to minimize this overhead, and yield better parallel performance.

In this paper, we introduce a new approach for shared memory multi-processor systems, e.g. multi-core CPUs, in which the communication overhead is very light and effective. A multi-threaded implementation with maximum asynchronism is proposed by analyzing the performance bottlenecks and relaxing avoidable barrier points. Our results show promising speedup and efficiency using 12 processors.

The remainder of the paper is organized as follows. In the next section we discuss state-of-the-art parallel ACS implementations in the literature. Section III gives a brief introduction to the Ant Colony Optimization. Then in section IV, the proposed method is presented. Section V presents our experiment results. Finally, section VI offers some concluding remarks and avenues for future research.

2. RELATED WORK
The time-consuming nature of ant colony systems has encouraged many researchers to overwhelm this issue. The proposed solutions can be categorized into two major approaches. The first approach is to change the nature of ant colony system, in order to avoid unnecessary computations and yield faster convergence. Few research have been done in this area and they yield relatively minor speed-ups where the speedup vary among different problems [5].

The second approach is to take advantage of parallel computing techniques, to run some of the calculations simultaneously. The taxonomic discrimination among these algorithms is based on the granularity level and the exchange strategy. The granularity level is the level to which the problem is decomposed for parallel processing [4]. Each single process may be run a single ant, a subset of ants, i.e. cellular model [6], or the whole ant colony system. These different approaches result to different granularities. *Fine-grained* methods focus on completing a single ACS process as fast as possible, by taking the advantage of parallel computing solutions [7]. This is usually done by running a subset of ants in each processing unit, and exchange some sort of information between them [8]. On the other hand, the *coarse-grained* methods, also known as *Multi-instance* or *Colony-based* methods, try to run multiple instances of ant colony systems concurrently [9–11]. The simplest case in a cluster computing environment is to run the ACS independently in each machine, and then select the best solution among the results collected from the systems [12–14]. More complex models may arrange some kind of periodic information exchange [15–18].

The exchanged information may be small-sized information, such as the best-so-far solutions, or the whole pheromone matrix. More information exchange generally improves the results quality, but imposes more overhead, that yields longer

execution time [19]. Fine-grained methods usually use more information exchange, and may need some extra system-level facilities, e.g. synchronization points [4], [7].

Most of the related works are based on cluster computing platforms. Due to large communication overhead in such systems that is caused by the delay in networking, the community have believed that the fine-grained implementations will suffer from this overhead and hence are less efficient comparing to coarse-grained methods [15], [20]. However, this issue does not hold in shared-memory systems because the processes communicate by accessing common memory blocks that is a processor-level lightweight action. Recent parallel ACS proposals for shared memory systems have chosen the coarse-grained model. Delisle et al [8] have compared these methods with respect to varying parameters, such as the number of ants, the problem size, etc.

Following a different approach, the implementation of ACS for some specific hardware has been proposed. These implementations have utilized inherent parallel capabilities of the hardware, such as hierarchical memory model in General Purpose Graphics Processing Units (GPGPU)[13], [21], redundant processor connections in Optical Pipelined Reconfigurable Mesh (PR-Mesh) systems [22], runtime reconfigurable processor arrays [23], and FPGA [24]. A number of papers have also proposed ACO implementations for the MapReduce framework, a programming model and software framework for distributed data processing[25], [26]

3. BACKGROUND

As in the original ACS paper, we will use Travelling Salesman Problem (TSP) as the test case [1]. Assume $C=\{c_1,c_2,\ldots,c_n\}$ the set of n cities with cost matrix δ, where $\delta(r,k)$ is the cost, i.e. distance, from c_r to c_k. The aim is to find the shortest Hamiltonian path in the weighted graph. It is proved that the problem is NP-hard [27], and has been the most famous combinatorial optimization problem in the meta-heuristics literature [28].

The family of ant optimization techniques use pheromone trails in the environment as a natural memory model for the optimization system. This is modeled by τ, the matrix of desirability measure, in which $\tau(r,k)$ is the density of pheromone trails in the path between c_r and c_k. Ants create their paths randomly, but this random selection is biased toward the paths with higher pheromone densities. Ants move from one city to another using the state transition rule. Assume ant a_k to be in c_r and $J_k(r)$ the set of cities which are not observed by a_k. The ant decides whether to choose the best known path i.e. exploitation or select a path in a probabilistic manner i.e. exploration. The next city c_s is selected from the following rule: [4].

$$s=\begin{cases}\arg\max_{u\in J_k(r)}[\tau(r,u)].[\eta(r,u)]^{\beta} & \text{if } q\le q_0 \text{ (exploitation)}\\ \text{Probabilistic selection (Eq. 2)} & \text{if } q>q_0 \text{ (exploration)}\end{cases} \quad (1)$$

$$p_{ij}(r,s)=\begin{cases}\dfrac{[\tau(r,s)].[\eta(r,s)]^{\beta}}{\sum_{u\in J_k(r)}\tau(r,s)].[\eta(r,s)]^{\beta}} & \text{if } s\in J_k(r)\\ 0 & \text{if } s\notin J_k(r)\end{cases} \quad (2)$$

In order to select a city, q is sampled from a uniform distribution from [0,1]. If $q\le q_0$, the best path is selected.

The parameter $0\le q_0\le 1$ determines the preference of exploitation over exploration. $\eta(r,s)$ is the heuristic function. For instance, in TSP the heuristic function is defined as the inverse of the distance (cost) function $\delta(r,s)$. β is a positive value which controls the importance of heuristic function versus the amount of pheromone trails.

At the beginning of process, the pheromone on the path of each two cities have some initial value τ_0. It is suggested to let $\tau_0=(nL_{nn})^{-1}$, where n is the size of the problem (i.e. number of cities) and L_{nn} is the cost of a rule produced by a heuristic, such as Nearest Neighbor. When an ant moves from c_i to. c_j., the pheromone is updated by the local updating rule:

$$\tau_{ij}\leftarrow(1-\alpha)\tau_{ij}+\alpha\tau_0 \quad (3)$$

$\alpha\in[0,1]$ is the pheromone decay factor, which determines the rate of pheromone evaporation. The local pheromone updating rule prevents the ACS from falling into local optima. When all ants finish their tours, the best path, which is the path with least cost (P^*) is used to apply the global updating rule. Let C^* be the total cost of P^*, then the pheromone is updated according to:

$$\tau_{ij}\leftarrow(1-\alpha)\tau_{ij}+\frac{\alpha}{C^*} \quad \textbf{if}\,(c_i\rightarrow c_j)\in P^* \quad (4)$$

The pseudo code for the Ant Colony System is given in Fig. 1.

4. THE PROPOSED METHOD

Our approach is to optimize the performance of state of the art fine-grained ACS implementations. We realized that OS-level performance analysis, an area that has received less attention in the parallel ACO community, yields more performance improvement than new heuristic modifications and results in a simple but effective implementation.

Fig.2 shows a general fine-grained parallel ACS implementation. For better comparison, the line numbering in both algorithms are kept similar. The logic of the algorithms is the same, but the requirements for multi-threaded processing are added to the algorithm. In such implementation, each ant is assigned to a thread, and the pheromone matrix is shared between the threads (line 4). The ant threads can simultaneously read the pheromone values from the pheromone and distance matrixes. Any concurrent modification of the matrix element, i.e. updating the pheromone values, should be done in a thread-safe manner. Otherwise, the concurrent update may cause an unpredictable effect that is known as race condition. The thread-safe settings are usually implemented by putting a lock, e.g. semaphore, on the updating element of the pheromone matrix. This lock mechanism ensures that no other thread may modify the value of element that is updated by the current thread.

Algorithm 1: Ant Colony System

1: **for** each ***Iteration***
3: **for** each ***ant***
4: Initialize the ant
5: **while** a *tour* is not finished
6: Choose the next city (Eq. 1,2)
7: Local pheromone update (Eq. 3)
8: **end while** //end of ant's travel
9: Global pheromone update (Eq. 4)
10: **end for**
11: **end for** //end of iteration

Fig. 1. Ant Colony System.

The settings described above are common in all the previous fine-grained ACS systems. We have analyzed the components of that system for finding the bottlenecks. A brief review of our observations, which builds the strategies of the proposed method, is as follows.

1) The global pheromone update (lines 9.1 – 9.3) is a very lightweight process. Conversely to most of the previous parallel configurations [8], we found that performing this task in parallel will cause a huge synchronization overhead. Thus, running the task in serial mode may conjure much better performance.

2) Launching each ant as a new thread (line 4) requires initializing local information, such as the list of visited cities and hence, it will cause some performance overhead. Additionally, the thread initialization overhead is heavy itself. To our knowledge, all the previous parallel ACS proposals create threads in each iteration. In such setting, as an ant thread finishes, its allocated resources will be removed. To run a new thread, all the required resources, such as descriptors, heap and call stack will be initialized, as well as the thread's private data, namely the list of previously visited cities. We have taken the advantage of thread pooling technique, specially the Java *ExecuterService* mechanism to avoid such overhead and reuse the resources from the previous iterations.

3) In order to avoid unsafe memory access conflicts in the shared graph, in which the ant threads need to read and update pheromone values by the local updating rule (lines 7.1 – 7.3), a synchronization mechanism, such as barrier points or memory lock have been proposed in almost all of the previous works[2], [20]. This thread-safe implementation not only causes a huge overhead, but also decreases the speedup when running on the machines with more processing units. We discuss that this mechanism is not useful, and hence can be avoided.

Assume two ant threads T_i and T_j are trying to update the pheromone trails for the same path between two cities c_r and c_s. Obviously, this is a race condition for the threads. In order to analyze how much does it degrade the algorithm's performance we discuss on what may occur to the ACS in the case of race condition, and how much it is likely to happen.

Algorithm 2: General fine-grained implementation for parallel ant colony system

1: **for** each ***Iteration***
3: **for** each ***ant: IN PARALLEL***
4: Start a new thread, Initialize the ant, and share distance and pheromone graph for the thread
5: **while** a *tour* is not finished
6: Choose the next city (Eq. 1,2)
7.1: LOCK PHEROMONE PATH
7.2: Local pheromone update (Eq. 3)
7.3: UNLOCK PHEORMONE PATH
8: **end while** //end of ant's travel
9.1: *LOCK THE PATH*
9.2: Global pheromone update (Eq. 4)
9.3: *UNLOCK THE PATH*
10.1: Terminate the thread, release resources.
10.2: **end for**
13: **end for** //end of iteration

Fig.2. General fine-grained parallel ACS.

Let t^r and t^w to be the time of reading and updating the pheromone trail, respectively. Without loss of generality, suppose T_i to be first thread which reads the pheromone trail. From the thread management literature, three situations are possible.

Situation 1: $t_i^r > t_i^w > t_j^r > t_j^w$: in this case, threads T_i and T_j have sequentially read and written the values, no inconsistency occurs.

Situations 2: $t_i^r > t_j^r > t_j^w > t_i^w$: in this case, the T_i reads the pheromone value, then T_j updates (reads and writes) the value, and T_i *overwrites* the value. It is equivalent to the case that T_j has ignored executing the local update in the edge between c_r and c_s.

Situation 3: $t_i^r > t_j^r > t_i^w > t_j^w$: it is similar to situation 2, but the T_i's local updating has been ignored.

To discuss about the occurrence likelihood of such unsafe event, we analyze the accumulated time for executing each component. Each ant should apply equations 1 and 2 for all paths in the graph. Then, it selects one path and applies the local pheromone updating rule on the selected one. On average, the first operation is O{(n/2) × (time to compute eq. 1)}. If the exploration is chosen from eq. 1 that occurs with probability $(1\text{-}q_0)$, another similar time should be considered. Note that both equations 1 & 2 are raised to the power β, which is computed using the time-consuming Taylor series expansion. On the other hand, the local pheromone updating is a light process, consisting of four simple operations. As a result, when the number of nodes (cities) is large enough, it is very improbable for two ant threads to simultaneously be in

the critical region, and hence the unsafe states are unlikely to happen. As a rule of thumb, when the number of processing nodes is more than 20 that is usual in the benchmark datasets of parallel ACO, the local updating between the threads never overlaps. Therefore, we drop the synchronization and lock mechanisms in the proposed parallel ACS.

5. EXPERIMENTAL ANALYSIS

We have used the pr2392 TSP problem, having 2392 cities and their positions, as one of the largest and most frequently used data set in the literature. The ant colony system with 20, 50, 100, and 200 ants is tested by setting the number of active threads to 1, 2, 4, 8, 12, 24 and 36. We used a HP server system with two 6-core Intel® Xeon™ X5660 processors with hyper threading technology. The number of iterations is fixed to 100. Due to the stochastic behavior of ACS, the experiments in each configuration are repeated for 5 times and the average values are reported.

The proposed modification to the ACS is a fine-grained method because the effect of local pheromone update by each ant is instantly applied to the graph. The path selection by each ant of the colony is concurrent, asynchronous, and independent of the others. Thus, the parallelized ant colony system at the agent level i.e. the ant level, is roughly identical to the natural serial implementation, which makes us to intend expect similar results in multiprocessing, but in less time. Therefore we just report performance results that are caused by parallel execution.

Several metrics can be used to evaluate the parallel performance, from which the *speedup* and *computational efficiency* are the most commons. Assume T_m to be the execution time of the parallel program with m processors and T_1 be the execution time of sequential algorithm which is in our settings equal to time of executing the parallel version with one thread. The speedup is defined as the ratio of T_1/T_m. For stochastic algorithms where the execution time varies between multiple runs, the average value of T_1 and T_m are used:

$$S_m = \frac{E[T_1]}{E[T_m]} \quad (5)$$

Similarly, the computational efficiency $e_m = S_m / m$, is the normalized value of speedup. We should mention that when the number of active threads is more than the number of available processors, m is set to the number of processors [2].

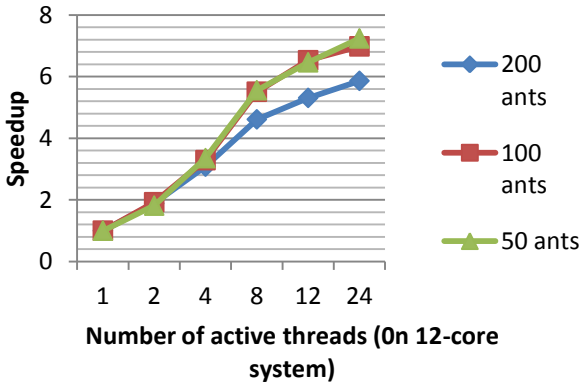

Fig. 3. Speedup of parallel ACS on pr2392.

Fig. 3 shows the speedup of the enhanced parallel ACS on pr2392. As we increase the number of active threads, the speedup increases, as expected. If the number of threads is more than the available processors, around twice or forth, the speedup is slightly increased. This is caused by the hyper threading technology, in which the processor can keep the active running of two tasks i.e. threads, and can switch between them in a limited number of system clocks. It is hence recommended to set the number of thread to twice the number of processors.

Fig. 4 shows the computational efficiency of the proposed method. Detailed results are tabulated[1] in

Table **1**. The only comparable result in the literature, is the work by Delisle et al [8] which is based on shared memory systems. Table 2 shows the speedup and computational efficiency of both methods on the middle-sized datasets. Our method outperforms the synchronized method in all cases, as expected before.

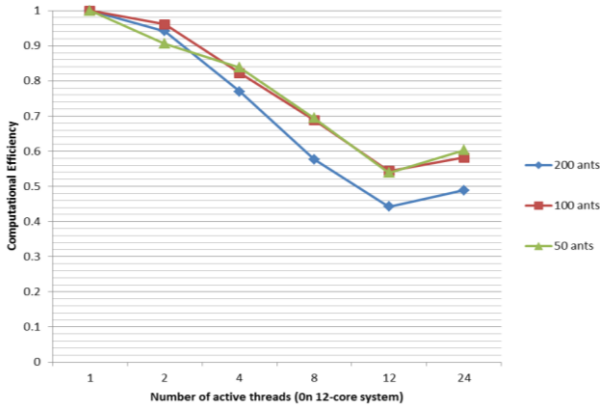

Fig. 4. Computational Efficiency of parallel ACS on pr2392.

6. CONCLUSION

Due to the time-consuming nature of ant colony systems, various methods have been proposed to increase its overall execution speed. The family of fine-grained implementations has gained less interest in prior proposals because the communication overhead is not tolerable in cluster systems. In this work, we have implemented a parallel ant colony system that is specially crafted for systems with shared-memory architecture.

We have optimized the algorithm's overall speed by reviewing the thread-management overheads and skipping neutral thread-safety locks. By relaxing the synchronization mechanisms that is used for consistency, we have reached near-linear speedup that means if we double the assigned CPU-cores for ACS execution; we shall expect approximately double speedup value. Near-linear scalability of our proposed implementation makes it ideal for multicore CPUs that their number of cores is steadily increasing. To validate the performance of our proposed parallel Ant Colony System, we have performed some experiments on a state of the art Intel multicore CPU. Our results show promising speedup value around 6X to 8X on a machine with two 6-core hyper threaded i7 CPUs that is very promising.

[1] For better organization, we have placed the comparison tables to end of the paper.

7. REFERENCES

[1] M. Dorigo and L. M. Gambardella, "Ant colony system: A cooperative learning approach to the traveling salesman problem," Evolutionary Computation, IEEE Transactions on, vol. 1, no. 1, pp. 53–66, 1997.

[2] M. Pedemonte, S. Nesmachnow, and H. Cancela, "A survey on parallel ant colony optimization," Applied Soft Computing, vol. 11, no. 8, pp. 5181–5197, 2011.

[3] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on, vol. 26, no. 1, pp. 29–41, 1996.

[4] I. Ellabib, P. Calamai, and O. Basir, "Exchange strategies for multiple ant colony system," Information Sciences, vol. 177, no. 5, pp. 1248–1264, 2007.

[5] S. P. Tseng, C. W. Tsai, M. C. Chiang, and C. S. Yang, "A fast ant colony optimization for traveling salesman problem," in Evolutionary Computation (CEC), IEEE Congress on, pp. 1–6, 2010.

[6] M. Pedemonte and H. Cancela, "A cellular ant colony optimisation for the generalised Steiner problem," International Journal of Innovative Computing and Applications, vol. 2, no. 3, pp. 188–201, 2010.

[7] M. Randall and A. Lewis, "A parallel implementation of ant colony optimization," Journal of Parallel and Distributed Computing, vol. 62, no. 9, pp. 1421–1432, 2002.

[8] P. Delisle, M. Gravel, M. Krajecki, C. Gagné, and W. Price, "Comparing parallelization of an ACO: message passing vs. shared memory," in Second International Workshop on Hybrid Metaheuristics, 2005.

[9] P. Delisle, M. Krajecki, M. Gravel, and C. Gagné, "Parallel implementation of an ant colony optimization metaheuristic with OpenMP," in Proceedings of the 3rd European Workshop on OpenMP (EWOMP'01), Barcelona, Spain, 2001.

[10] M. S. F. Catalano and F. Malucelli, "Parallel randomized heuristics for the set covering problem," International Journal of Practical Parallel Computing, vol. 10, no. 4, pp. 113–132, 2001.

[11] E. G. Talbi, O. Roux, C. Fonlupt, and D. Robillard, "Parallel ant colonies for the quadratic assignment problem," Future Generation Computer Systems, vol. 17, no. 4, pp. 441–449, 2001.

[12] E. Alba, G. Leguizamon, and G. Ordonez, "Two models of parallel ACO algorithms for the minimum tardy task problem," International Journal of High Performance Systems Architecture, vol. 1, no. 1, pp. 50–59, 2007.

[13] H. Bai, D. OuYang, X. Li, L. He, and H. Yu, "MAX-MIN ant system on GPU with CUDA," in Innovative Computing, Information and Control (ICICIC), 2009 Fourth International Conference on, 2009, pp. 801–804.

[14] M. Rahoual, R. Hadji, and V. Bachelet, "Parallel ant system for the set covering problem," Ant Algorithms, pp. 249–297, 2002.

[15] M. Middendorf, F. Reischle, and H. Schmeck, "Multi colony ant algorithms," Journal of Heuristics, vol. 8, no. 3, pp. 305–320, 2002.

[16] R. Michel and M. Middendorf, "An island model based ant system with lookahead for the shortest supersequence problem," in Parallel Problem Solving from Nature—PPSN V, pp. 692–701, 1998.

[17] R. Michel and M. Middendorf, "An ACO algorithm for the shortest common supersequence problem," in New ideas in optimization, 1999, pp. 51–62.

[18] D. A. L. Piriyakumar and P. Levi, "A new approach to exploiting parallelism in ant colony optimization," in Micromechatronics and Human Science, 2002. MHS 2002. Proceedings of 2002 International Symposium on, pp. 237–243, 2002.

[19] C. Twomey, T. Stützle, M. Dorigo, M. Manfrin, and M. Birattari, "An analysis of communication policies for homogeneous multi-colony ACO algorithms," Information Sciences, vol. 180, no. 12, pp. 2390–2404, 2010.

[20] Q. Lv, X. Xia, and P. Qian, "A parallel aco approach based on one pheromone matrix," Ant Colony Optimization and Swarm Intelligence, pp. 332–339, 2006.

[21] J. Fu, L. Lei, and G. Zhou, "A parallel ant colony optimization algorithm with gpu-acceleration based on all-in-roulette selection," in Advanced Computational Intelligence (IWACI), 2010 Third International Workshop on, pp. 260–264, 2010.

[22] K. D. Nguyen and A. G. Bourgeois, "Ant colony optimal algorithm: fast ants on the optical pipelined r-mesh," in Parallel Processing, 2006. ICPP 2006. International Conference on, pp. 347–354, 2006.

[23] D. Merkle and M. Middendorf, "Fast ant colony optimization on runtime reconfigurable processor arrays," Genetic Programming and Evolvable Machines, vol. 3, no. 4, pp. 345–361, 2002.

[24] B. Scheuermann, K. So, M. Guntsch, M. Middendorf, O. Diessel, H. ElGindy, and H. Schmeck, "FPGA implementation of population-based ant colony optimization," Applied Soft Computing, vol. 4, no. 3, pp. 303–322, 2004.

[25] Q. Tan, Q. He, and Z. Shi, "Parallel Max-Min Ant System Using MapReduce," in Advances in Swarm Intelligence, vol. 7331, Y. Tan, Y. Shi, and Z. Ji, Eds. Springer Berlin / Heidelberg, pp. 182–189, 2012.

[26] Y. Yang, X. Ni, H. Wang, and Y. Zhao, "Parallel Implementation of Ant-Based Clustering Algorithm Based on Hadoop," in Advances in Swarm Intelligence, vol. 7331, Y. Tan, Y. Shi, and Z. Ji, Eds. Springer Berlin / Heidelberg, pp. 190–197, 2012.

[27] E. L. Lawler, "The traveling salesman problem: a guided tour of combinatorial optimization," WILEY-INTERSCIENCE SERIES IN DISCRETE MATHEMATICS, 1985.

[28] C. Blum, J. Puchinger, G. R. Raidl, and A. Roli, "Hybrid metaheuristics in combinatorial optimization: A survey," Applied Soft Computing, 2011.

Table 1. Performance Results of parallel ACS on pr2392.

#threads	50 ants		100 ants		200 ants	
	Com. Eff.	Speedup	Com. Eff.	Speedup	Com. Eff.	Speedup
24	0.60	7.24	0.58	6.98	0.49	5.86
12	0.54	6.46	0.54	6.53	0.44	5.31
8	0.69	5.56	0.69	5.51	0.58	4.61
4	0.84	3.35	0.82	3.29	0.77	3.08
2	0.91	1.81	0.96	1.92	0.94	1.88
1	1	1	1	1	1	1

Table 2. Comparison of performance results on medium-sized datasets with 8 threads.

Problem	Metric	Delisle et al [8]							The proposed method						
		Number of processors							Number of processors						
		2	3	4	5	6	7	8	2	3	4	5	6	7	8
lin318	Speedup	1.65	2.39	3.09	3.64	4.15	4.63	4.77	1.95	2.91	3.76	4.60	5.18	5.79	6.69
	Com. Eff.	0.83	0.8	0.77	0.73	0.69	0.66	0.6	0.97	0.97	0.94	0.92	0.86	0.83	0.84
pcb442	Speedup	1.71	2.47	3.26	4.02	4.57	5.24	5.55	1.94	2.87	3.75	4.57	5.24	5.77	6.81
	Com. Eff.	0.86	0.82	0.81	0.8	0.76	0.75	0.69	0.97	0.96	0.94	0.91	0.87	0.82	0.85
d657	Speedup	1.78	2.54	3.23	3.95	4.62	5.14	5.74	1.96	2.86	3.82	4.72	5.39	6.19	7.10
	Com. Eff.	0.89	0.85	0.81	0.79	0.77	0.73	0.72	0.98	0.95	0.95	0.94	0.90	0.88	0.89
rat575	Speedup	1.74	2.62	3.39	4.12	4.83	5.36	6.14	1.95	2.89	3.89	4.77	5.60	6.20	7.23
	Com. Eff.	0.87	0.87	0.85	0.82	0.8	0.77	0.77	0.98	0.96	0.97	0.95	0.93	0.89	0.90