A Comparative Study for Efficient Synchronization of Parallel ACO on Multi-core Processors in Solving QAPs

Shigeyoshi Tsutsui
Management Information
Hannan University
Matsubara, Osaka 580-8402, Japan
Email: tsutsui@hannan-u.ac.jp

Noriyuki Fujimoto
Graduate School of Science
Osaka Prefecture University
Sakai, Osaka 599-8531, Japan
Email: fujimoto@mi.s.osakafu-u.ac.jp

***Abstract*—This paper describes three types of parallel synchronization models of ant colony optimization (ACO) on multi-core processors in solving quadratic assignment problems (QAPs). These three models include (1) Synchronous Parallel (SP), (2) Asynchronous Parallel (AP), and (3) Distributed Asynchronous Parallel (DAP). Parallel executions are studied up to 16-core. Among three models, the DAP shows the most promising results over various sizes of QAP instances. It also shows a good scalability up to 16-core.**

I. Introduction

Recently, microprocessor vendors supply processors which have multiple cores of 8, 16, or more, and PCs which use such processors are available at a reasonable cost. They are normally configured with symmetric multi-processing (SMP) architecture.

Since the main memory is shared among processors in SMP, parallel processing can be performed efficiently with less communication overhead among processors. Programming for parallel processing in SMP can be performed using multi-thread programming such as OpenMP and is much easier compared to other options such as GPGPUs [1].

In a previous paper, we proposed a variant of the ACO (ant colony optimization) algorithm called the cunning Ant System (cAS) [2] and evaluated it using TSP. The results showed that the cAS could be one of the most promising ACO algorithms. In this paper, we describe the parallelization of cAS on a multi-core processor for fast solving QAP (Quadratic Assign Problem), a typical NP-hard problem in permutation domains.

Many parallel ACO algorithms have been studied [3]. Brief summaries can be found in [4] and [5]. The most commonly used approach to parallelization is to use an island model where multiple colonies exchange information (i.e., solutions, pheromone matrix values, or parameters) synchronously or asynchronously [6], [4], [5].

In many of the above-mentioned studies, attention is mainly focused on the parallelization using multiple colonies. In contrast to these studies, in this study parallelization is performed at the agent (or individual) level in one colony aiming at speedup of the ACO algorithm on a computing platform with a multi-core processor. With this scheme, we studied parallel ACO to solve TSP with up to 4-core in [7].

In this paper, we study three types of parallel synchronization models with a new improved synchronization parallel model for cAS. They are (1) Synchronous Parallel (SP), (2) Asynchronous Parallel (AP), and Distributed Asynchronous Parallel (DAP) to solve QAP instances in QAPLIB [8]. We studied parallel executions up to 16-core. Among three models, the DAP showed the most promising results over various sizes of QAP instances. It also showed a good scalability up to 16-core.

In the remainder of this paper, Section II gives a brief review of cAS and how we apply it to solving QAPs. Then, in Section III, three types of parallel synchronization models are described. In Section IV, experimental results are presented. Finally, Section V concludes the paper.

II. A Brief Overview of cAS and Its Application to Solving QAPs

A. A Brief Introduction of cAS

cAS introduced two important schemes. One is a scheme to use partial solutions, which we call *cunning*. In constructing a new solution, cAS uses pre-existing partial solutions. With this scheme, we may prevent premature stagnation by reducing strong positive feedback to the pheromone trail density. The other is to use the colony model, dividing a colony into units, which has a stronger exploitation feature, while maintaining a certain degree of diversity among units.

cAS uses an agent called the cunning ant (*c-ant*). It constructs a solution by borrowing a part of an existing solution (we call it a donor ant (*d-ant*)). The remainder of the solution is constructed based on $\tau_{ij}(t)$ probabilistically as usual.

Let l_s represent the number of nodes of a partial solution in permutation representation that are constructed based on $\tau_{ij}(t)$ (i.e., l_c, the number of nodes of partial solutions from its *d-ant*, is $n - l_s$, where n is the problem size). Then cAS introduces the control parameter γ which can define $E(l_s)$ (the average of l_s) by $E(l_s) = n \times \gamma$. Using γ values in [0.2, 0.5] is a good choice in cAS.

978-1-4799-7560-0/15 $31.00 © 2015 IEEE
DOI 10.1109/SSCI.2015.160

The colony model of cAS is shown in Fig. 1. It consists of m units. Each unit consists of only one $ant^*_{k,t}$ ($k = 1, 2, \cdots, m$). At iteration t in unit k, a new *c-ant*$_{k,t+1}$ is generated using the existing ant in the unit (i.e., $ant^*_{k,t}$) as the *d-ant*$_{k,t}$. Then, the newly generated *c-ant*$_{k,t+1}$ and *d-ant*$_{k,t}$ are compared, and the better one becomes the next $ant^*_{k,t+1}$ of the unit. Thus, in this colony model, $ant^*_{k,t}$, the best individual of unit k, is always reserved. Pheromone density $\tau_{ij}(t)$ is then updated with $ant^*_{k,t}$ ($k = 1, 2, \cdots, m$) and $\tau_{ij}(t+1)$ is obtained as:

$$\tau_{ij}(t+1) = \rho \cdot \tau_{ij}(t) + \sum_{k=1}^{m} \Delta^* \tau^k_{ij}(t), \quad (1)$$

$$\Delta^* \tau^k_{ij}(t) = 1/f^*_{k,t} : \text{ if } (i,j) \in ant^*_{k,t}, \ 0 : \text{ otherwise}, \quad (2)$$

where the parameter ρ ($0 \leq \rho < 1$) models the trail evaporation, $\Delta^* \tau^k_{ij}(t)$ is the amount of pheromone by $ant^*_{k,t}$, and $f^*_{k,t}$ is the fitness of $ant^*_{k,t}$.

Values of $\tau_{ij}(t+1)$ are set to be within $[\tau_{min}, \tau_{max}]$ as in MAX-MIN ant system (MMAS) [9]. Here, τ_{max} and τ_{min} for cAS is defined as

$$\tau_{\max}(t) = \frac{m}{1-\rho} \cdot \frac{1}{f^{best-so-far}}, \quad (3)$$

$$\tau_{\min}(t) = \frac{\tau_{\max}}{2n}, \quad (4)$$

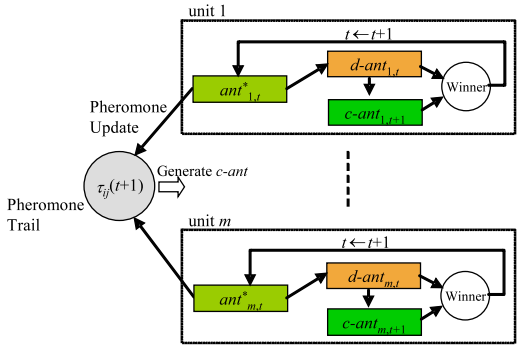

Fig. 1. Colony model of cAS

B. cAS on QAP

1) Quadratic Assignment Problem (QAP): The QAP is a problem which assigns a set of facilities to a set of locations and can be stated as a task to find permutations ϕ which minimize

$$f(\phi) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} b_{\phi(i)\phi(j)}, \quad (5)$$

where $A = (a_{ij})$ and $B = (b_{ij})$ are two $n \times n$ matrices and ϕ is a permutation of {1, 2, …, n}. Matrix A is a flow matrix between facilities i and j, and B is the distance between locations i and j. Thus, the goal of the QAP is to place the facilities in locations in such a way that the sum of the products between flows and distances are minimized.

2) c-ant and d-ant in QAP: The *c-ant* in QAP acts in a slightly different manner from *c-ant* in TSP. In TSP, $\tau_{ij}(t)$ are defined on each edge between city i and j. In QAP, the pheromone trails $\tau_{ij}(t)$ correspond to the desirability of assigning a location i to a facility j. Fig. 2 shows how the *c-ant* acts in QAP. In this example, the *c-ant* uses part of the node values at positions 0, 2, and 4 of the *d-ant*, where these positions are determined randomly. The *c-ant* constructs the remainder of the node values for positions 1 and 3 according to the following probability:

$$p_{ij} = \frac{\tau_{ij}(t)}{\sum_{k \in F(i)} \tau_{ik}}, \quad (6)$$

where $F(i)$ is the set of facilities that are yet to be assigned to locations.

In cAS for TSP, the number of nodes to be sampled, l_s, is determined probabilistically, so that $E(l_s) = n \times \gamma$ [2]. In cAS for QAP in this study, we simply determine l_s as $l_s = n \times \gamma$. Then we copy the number of nodes, $l_c = n - l_s$, from *d-ant* at its random positions and sample l_s number of remaining nodes according to Eq. (6) with random sequence.

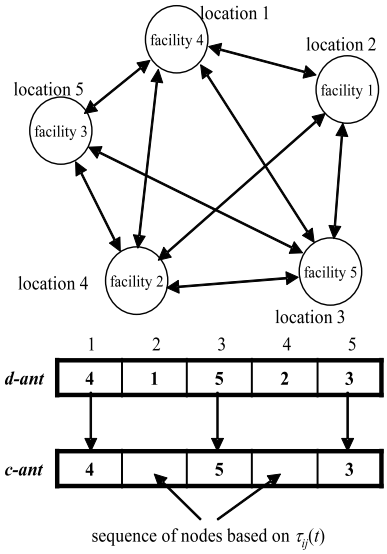

Fig. 2. *c-ant* and *d-ant* in QAP

In this study, we apply 2-OPT local search (LS) to newly generated solutions by cAS. 2-OPT local search explores its neighborhood $N(\phi)$ and accepts a solution according to a given pivoting rule. Here we use the best improvement pivoting rule. The process is repeated until an IT_{max} number of iterations is reached or no improvement solutions are found as shown in Fig. 3.

Sequential algorithm cAS with 2-OPT can be summarized as shown in Fig. 4.

III. Synchronization Models for Parallel ACO on a Multi-core Processor

In this study, parallelization is performed at the agent level, i.e., operations for each agent are performed in parallel in one

```
2-opt(int *ϕ, int n){
    it = 0
    do{
        Δmax= 0;
        for(r=1; r<n; i++){
            for(s=0; s<r; s++){
                if(Δ(ϕ, r, s) < Δmax){
                    R = r; S = s; Δmax = Δ(ϕ, r, s)
                }
            }
        }
        if(Δmax < 0)
            apply exchange (R, S) to ϕ;
        it++;
    }while(it < ITmax && Δmax != 0);
}
```

Fig. 3. 2-OPT local search (LS).

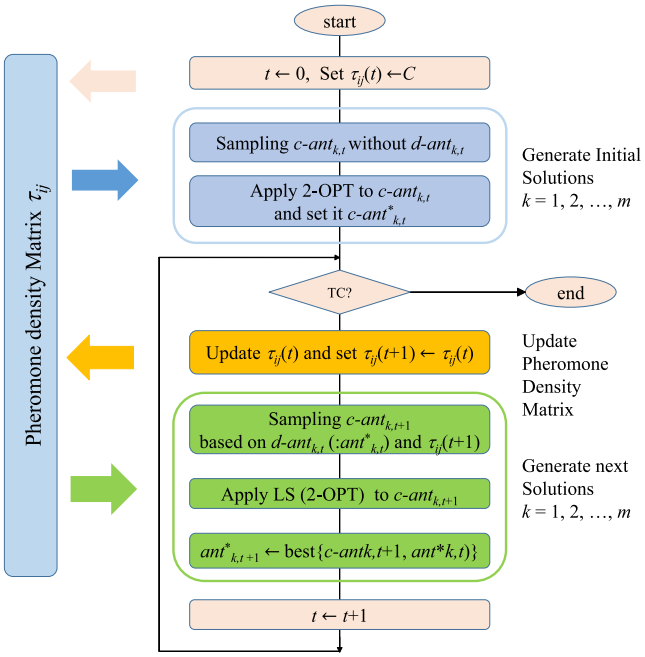

Fig. 4. Algorithm description of sequential cAS with 2-OPT local search (LS)

colony. A set of operations for an agent is assigned to a thread in OpenMP. Usually, since the number of agents (m) may be larger than or equal to the number of cores of the platform, we generate n_{core} threads, where n_{core} is the number of cores to be used. These threads execute operations for m agents of the cAS.

Fig. 5 shows the parallel cAS on a multi-core processor. The Agent Pool maintains agents of cAS. The Agent Assignor assigns agents to n_{core} number of threads, CasThread$_1$, CasThread$_2$, $\cdots$, and CasThread$_{n_{core}}$. In the parallel execution, an important factor is how to synchronize among n_{core} threads in their parallel runs for efficient execution of cAS. We implement three types of synchronization parallel models and run them up to $n_{core} = 16$, much bigger than previous study [2].

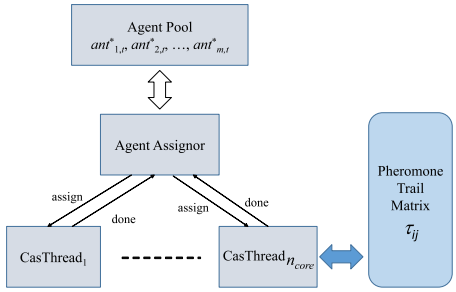

Fig. 5. Algorithm description of sequential cAS with 2-OPT local search (LS)

A. Synchronous Parallel cAS (SP-cAS)

In the synchronous parallel cAS (SP-cAS), all agents are processed in each thread, independently as shown in Fig. 6. However, in the SP-cAS, pheromone density updating is performed after all $ant^*_{k,t}$ $(k = 1, 2, \cdots, m)$ are generated in each iteration, the same way as is done in the sequential cAS in Fig. 4. Thus, we call this parallel model synchronous.

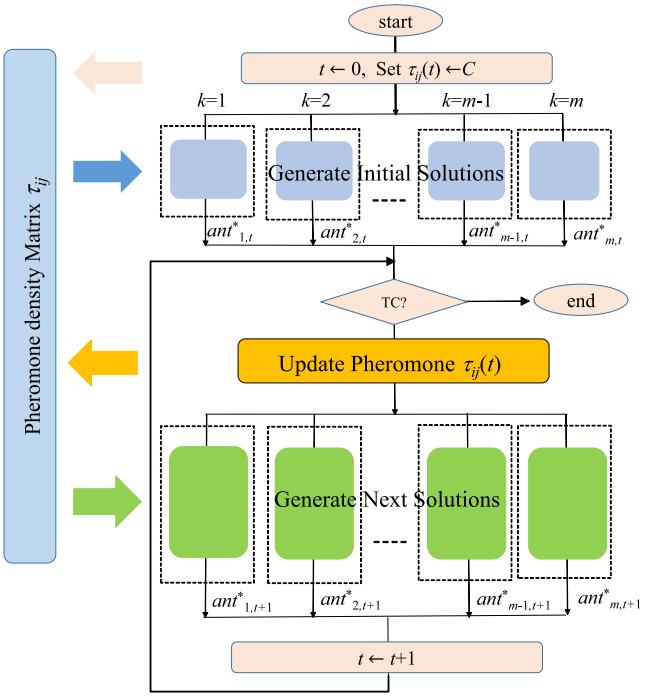

Fig. 6. Synchronous parallel cAS (SP-cAS)

B. Asynchronous Parallel cAS (AP-cAS)

As described in previous section, In SP-cAS the pheromone density updating is performed after all $ant^*_{k,t}$ $(k = 1, 2, \cdots, m)$ are generated in each iteration. This may cause some idling time in the iteration in Fig 6 when some threads

have no tasks to perform while the others have. AP-cAS is intended to remove this idling time. To attain this feature, all steps of cAS are performed asynchronously as shown in Fig. 7. As seen in this figure, each agent has its own iteration counter.

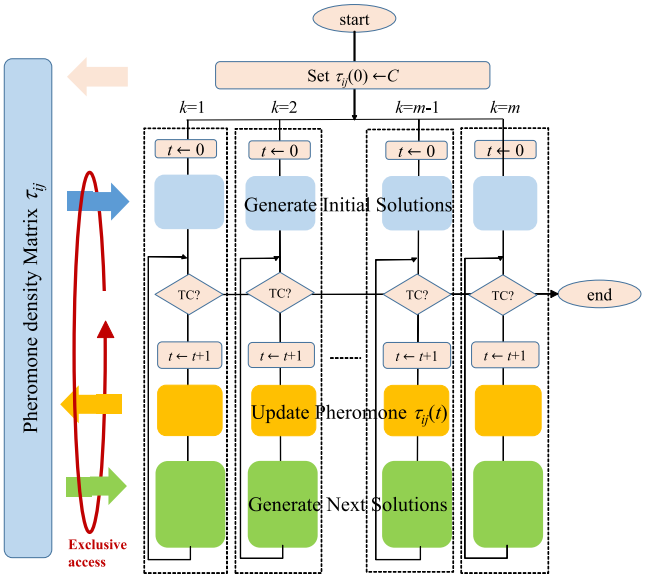

Fig. 7. Asynchronous parallel cAS (AP-cAS)

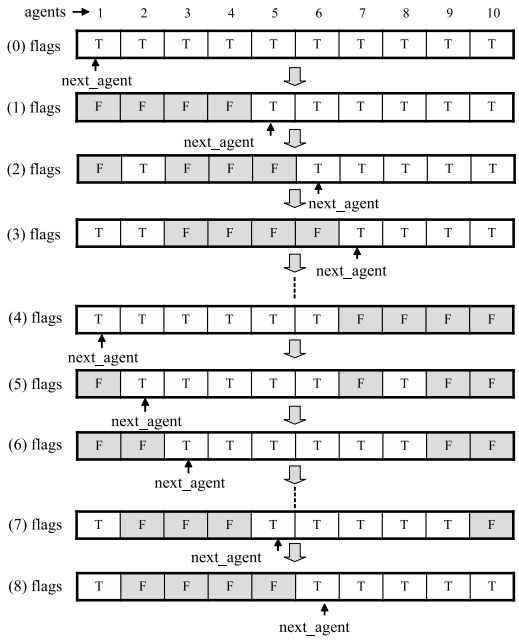

Fig. 8. An example of agents assignment to threads in AP-cAS

Fig. 8 shows an example how each agent is assigned to each of n_{core}. The Agent Assignor has an array of flags, here $m = 10$ and $n_{core} = 4$ are assumed. At each start of run, all of the entries of flags are set to T (true) indicating that all agents are yet to be processed, as shown in Fig. 8 (0). $next\_agent$ indicates the next agent to be processed, and its initial value is set to 1.

Fig. 8 (1) shows a situation in which the four CasThreads are processing one iteration for each agent 1, 2, 3, and 4. *next_agent* is set to 5. Entries of flags at 1, 2, 3, and 4 have value F (false) indicating these agents are being processed. Fig. 8 (2) shows the situation where processing of one iteration for agent 2 has been completed and agent 5, previous indicated by *next_agent* is being processed and *next_agent* has been newly set to 6. Fig. 8 (3) shows the situation where processing of one iteration for agent 1 has been completed and one iteration for agent 6, which was previously indicated by *next_agent*, is now being processed and *next_agent* is set to 7.

Fig. 8 (4) shows the situation where the four CasThreads are processing one iteration for each agent 7, 8, 9, and 10. The *next_agent* is 1. Fig. 8 (5) shows the situation where processing of one iteration for agent 8 is done and agent 1, which was previously indicated by *next_agent* is being processed and *next_agent* is newly set to 2. Note that at this moment agent 1 is being processed for the next iteration. Fig. 8 (6) shows the situation where processing of one iteration for agent 7 has been completed and one iteration for agent 2, which was indicated by *next_agent*, is being processed and *next_agent* is newly set to 3. Fig. 8 (7) shows the situation where one iteration for each agent 10, 2, 3, and 4 is being processed. Fig. 8 (8) shows the situation where one iteration of processing for agent 10 has been completed and one iteration for agent 5, which was previously indicated by *next_agent*, is being processed and *next_agent* is now set to 6.

In this way, in the AP-cAS, Agent Assignor assigns agents to each CasThread repeatedly without any synchronization among agents. As described here, when each agent is assigned to a CasThread by Agent Assignor, only one iteration is performed. Here, one problem arises in the updating pheromone density, defined by Eq. (1). Strictly following Eq. (1), update procedure requires all agent members $ant^*_{k,t}$ for $k = 1, 2, \cdots, m$ at the same time. But this situation is impossible in asynchronous parallel execution. To perform the pheromone update asynchronously, we modified the pheromone density updating for asynchronous parallel execution as follows:

$$\tau_{ij}(t+1) = \sqrt[m]{\rho} \cdot \tau_{ij}(t) + \Delta^* \tau^k_{ij}(t), \tag{7}$$

where $\Delta^* \tau^k_{ij}(t)$ is defined by Eq. (2). Although the pheromone density updating by Eq. (7) is not strictly equivalent to Eq. (1), we can say that it emulates the pheromone updating process of Eq. (1) in an asynchronous processing mode. Note here that the pheromone density updating and accessing to generate new solutions must be treated as a critical section as indicated by *Exclusive access* in Fig. 7.

C. Distributed Asynchronous Parallel cAS (DAP-cAS)

Usually an asynchronous run can be expected to be efficient. However, there may be worry that AP-cAS has critical sections in accessing to the pheromone density $\tau_{ij}(t)$. This may consume the efficiency of the asynchronous parallel run of AP-cAS. The distributed asynchronous parallel cAS (DAP-cAS) in this section is devised to solve this problem.

In DAP-cAS, we have pheromone density matix τ_{ij} in each of n_{core} threads as $\tau_{ij}^{(1)}(t), \tau_{ij}^{(2)}(t), \cdots, \tau_{ij}^{(n_{core})}(t)$ as shown in Fig. 9. Each $\tau_{ij}^{h}(t)$ $(h = 1, 2, \cdots, n_{core})$ is size of $n \times n$ as usual.

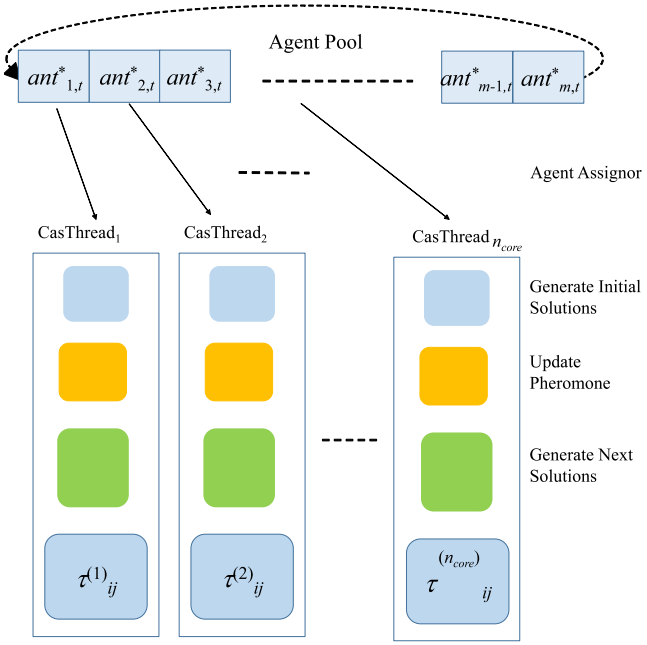

Fig. 9. Distributed asynchronous parallel cAS (DAP-cAS)

Agent assignment is performed in the same manner as AS-cAS. When an agent $ant_{k,t}^{*}$ $(k = 1, 2, \cdots, m)$ is assigned to a CasThread$_h$ $(h = 1, 2, \cdots, n_{core})$, only one iteration is performed as in AS-cAS. Pheromone update is applied to pheromone density $\tau_{ij}^{(h)}(t)$ by agent $ant_{k,t}^{*}$ according to the following equation.

$$\tau_{ij}^{(h)}(t+1) = \rho \cdot \tau_{ij}^{(h)}(t) + \Delta^{*}\tau_{ij}^{k}(t), \tag{8}$$

where $\Delta^{*}\tau_{ij}^{k}(t)$ is defined by Eq. (2).

Note here the pair (k, h) is not fixed in the run, i.e., the pair (k, h) may change every iteration according to agent assignment by the Agent Assignor. It means that a new solution will be generated based on the pheromone density modified by other agents. In this sense, DAS-cAS can be expected to have a similar behavior as an island model of genetic algorithms with high immigration rate [10].

IV. Experiments

A. Experimental Conditions

In this study, we used a PC which has 2.6GHz Xeon E5-2650 v2 (8-core) × 2 with Windows 8.1 Pro. The total number of cores is 16. The code is written in C++. For C++ program compilation, Microsoft Visual Studio Professional 2013 with optimization option /O2 was used.

The instances on which we tested our algorithm were taken from the QAPLIB benchmark library at [8]. We used the following 6 instances which were classified as "real-life like instances" [9] with the problem size ranging from 35 to 100, i.e., tai35b, tai40b, tai50b, tai60b, tai80b, and tai100b. Note here that the number in each instance name shows the problem size n of the instance. 25 runs were performed for each run. We measured the performance by the average time to find known optimal solutions T_{avg} over 25 runs. In each experiment, the algorithms found optimal solutions 25 times. Table I shows the control parameter values used in the experiment.

TABLE I. Control Parameter Values

Parameters		Values
cAS	m	n
	ρ	0.9
	γ	0.5
2-OPT	IT_{max}	n

B. Results and Their Analysis

Fig. 10 shows speedup values ($Speedup$ = (T_{avg} of sequential cAS) / (T_{avg} of parallel models of cAS)) of three syncronization models, SP-cAS, AP-cAS, and DAP-cAS described in Section III. The number of cores (n_{core}) was changed from 2 to 16 with step 2. From this figure, DAP-cAS outperforms the other two models on all test instances. It also shows a good scalability on all size of problems. On tai80b and tai100b, it shows superliner scalability (exact values are shown in Table IV-B). SP-cAS shows a stable scalability constantly for all instances.

AS-cAS shows a poor scalability with increases in the number of cores on instances of small problem size (tai35b - tai50b). However, we can observe that AS-cAS has the scalability on larger size instances, i.e., on tai60b, tai80b, and tai100b. On tai100b, it shows better performance than SP-cAS, though the difference is not significant from statistical tests (see Table IV-B).

We conducted a two-sided t-test of T_{avg} for two pairs of (SP-cAS, AP-cAS) and (SP-cAS, DAP-cAS) to show the statistical significance of the obtained $Speedup$ for n_{core} of 2, 4, 8, and 16 and showed the results by the p-values in Table IV-B. We showed p-values which are smaller than 0.05 by boldface. Except tai40b with $n_{core} = 2$ and 4, this test shows a clear advantage of DAP-cAS over SP-cAS as shown in Fig. 10.

Now, let us discuss two questions which arise from the results in Fig. 10 and Table IV-B. One question is why DAP-cAS works much better than SP-cAS and another question is why the values of $Speedup$ of AP-cAS are so poor compared to SP-cAS.

Let I_{ave} be the average of total iterations by m agents to find optimal solutions over 25 runs on an instance. Fig. 11 shows I_{ave} for $n_{core} = 2, 4, 8$, and 16. Here, values are normalized by setting I_{ave} of SP-cAS to 1.0.

Observing this figure, DAP-cAS has the smallest values among three parallel models for all cases except for two; tai40b with $n_{core} = 4$ and tai50b with $n_{core} = 16$ (for these two cases, AP-cAS has the smallest values). Especially, on large

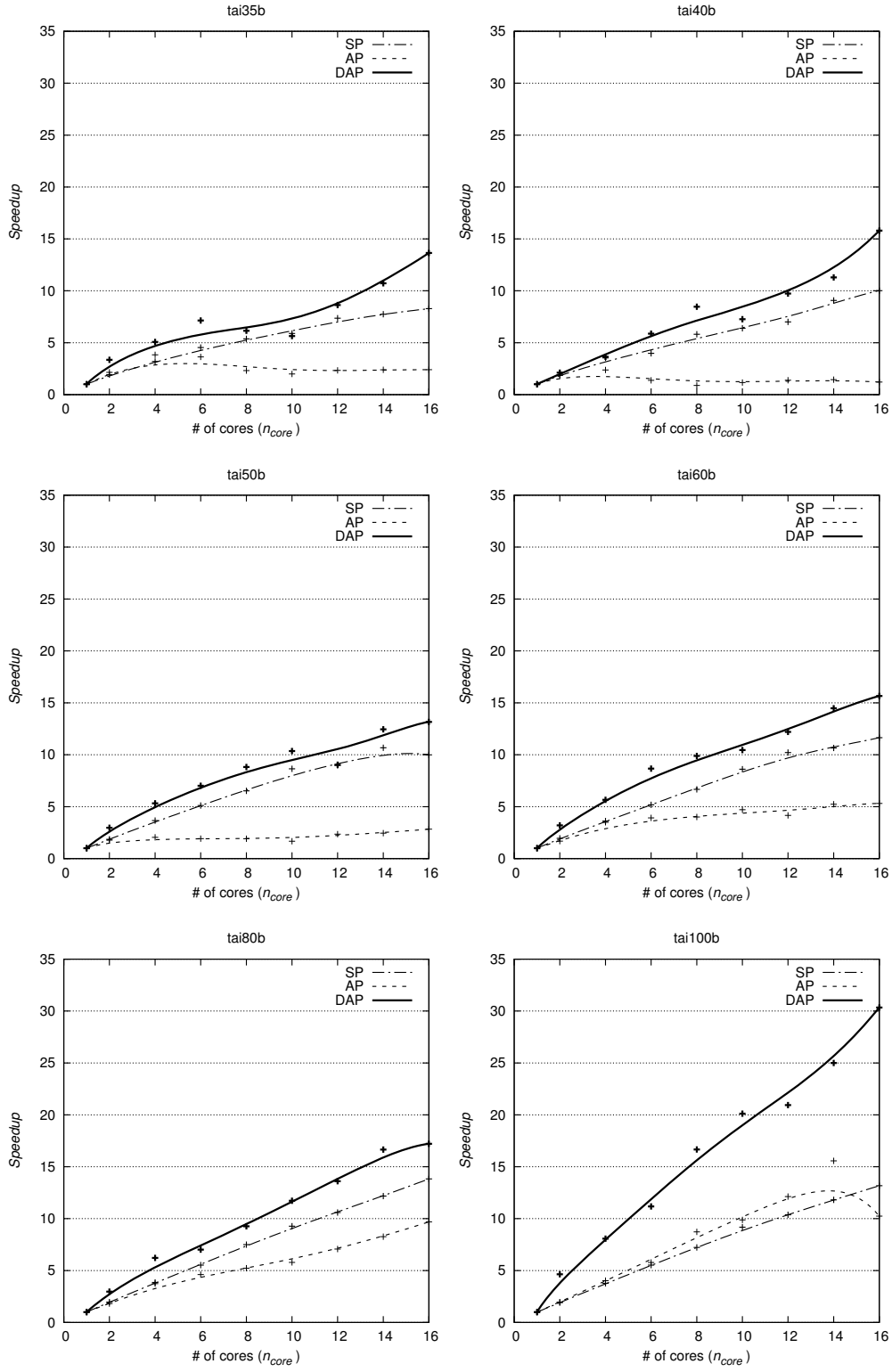

Fig. 10. *Speedup* of three synchronization models for different number of cores.

TABLE II. T_{avg} VALUES OF THREE SYNCHRONIZATION MODELS FOR n_{core} OF 2, 4, 8, AND 16. RESULTS OF THE TWO-SIDED t-TEST OF T_{avg} FOR TWO PAIRS OF (SP-cAS, AP-CAS) AND (SP-cAS, DAP-cAS) WERE SHOWN BY THE p-VALUES.

	n_{core}		2			4			8			16		
	models		SP-c AS	AP-c AS	DAP-c AS	SP-c AS	AP-c AS	DAP-c AS	SP-c AS	AP-c AS	DAP-c AS	SP-c AS	AP-c AS	DAP-c AS
Instances	tai35b	T_{avg} *1	0.64	0.57	0.36	0.38	0.32	0.24	0.23	0.53	0.20	0.15	0.51	0.09
		p-value *2	-	0.42	**0.00**	-	0.13	**0.00**	-	**0.00**	0.34	-	**0.00**	**0.00**
		Speedup *3	1.9	2.1	3.4	3.2	3.8	5.1	5.4	2.3	6.1	8.3	2.4	13.6
	tai40b	T_{avg} *1	0.51	0.52	0.46	0.27	0.41	0.26	0.17	1.09	0.11	0.10	0.79	0.06
		p-value *2	-	0.93	0.27	-	0.01	0.75	-	**0.00**	**0.00**	-	**0.00**	**0.00**
		Speedup *3	1.9	1.9	2.1	3.5	2.4	3.6	5.8	0.9	8.5	10.0	1.2	15.8
	tai50b	T_{avg} *1	3.85	4.00	2.40	1.95	3.49	1.34	1.09	3.72	0.81	0.71	2.52	0.54
		p-value *2	-	0.79	**0.00**	-	**0.00**	**0.00**	-	**0.00**	**0.00**	-	**0.00**	**0.00**
		Speedup *3	1.9	1.8	3.0	3.7	2.0	5.3	6.5	1.9	8.8	10.0	2.8	13.2
	tai60b	T_{avg} *1	9.63	11.18	5.88	5.21	5.38	3.32	2.82	4.69	1.90	1.62	3.54	1.20
		p-value *2	-	0.08	**0.00**	-	0.79	**0.01**	-	**0.00**	**0.00**	-	**0.00**	**0.02**
		Speedup *3	2.0	1.7	3.2	3.6	3.5	5.7	6.7	4.0	9.9	11.6	5.3	15.7
	tai80b	T_{avg} *1	122.00	131.95	80.55	62.14	64.11	38.52	31.97	45.77	25.87	17.32	24.72	13.91
		p-value *2	-	0.31	**0.00**	-	0.69	**0.00**	-	**0.00**	**0.04**	-	**0.00**	**0.04**
		Speedup *3	2.0	1.8	3.0	3.9	3.7	6.2	7.5	5.2	9.3	13.8	9.7	17.2
	tai100b	T_{avg} *1	341.23	332.94	140.47	173.26	161.76	80.78	90.26	74.76	39.13	49.49	63.64	21.49
		p-value *2	-	0.80	**0.00**	-	0.45	**0.00**	-	0.32	**0.00**	-	0.33	**0.00**
		Speedup *3	1.9	2.0	4.6	3.8	4.0	8.1	7.2	8.1	16.7	13.2	14.5	30.3

*1 The average time to find known optimal solutions over 25 runs in second.

*2 p-values by two-sided t-test of T_{avg} for two pairs of (SP-c AS, AP-c AS) and (SP-c AS, DAP-c AS).

*3 (T_{avg} of single cAS) / (T_{avg} of parallel c AS).

size problems (tai80b, tai100b), I_{ave} values of DAP-cAS are much smaller than the other two models.

As for SP-cAS and AP-cAS, we can see their I_{ave} values are almost the same and thus the synchronous or asynchronous model does not effect I_{ave} values much.

Now consider why I_{ave} values of DAP-cAS are smaller than other models. As described in Section III-C, an agent k of DAP-cAS ($k = 1, 2, \cdots, m$) accesses $\tau_{ij}^{(h)}(t)$ in threads CasThread$_h$ ($h = 1, 2, \cdots, n_{core}$). The pair (k, h) changes iteration by iteration. This may cause a degree of diversity among $\tau_{ij}^{(h)}(t)$ ($h = 1, 2, \cdots, n_{core}$). As a result, this model could a good convergence property resulting in small I_{ave} values.

Next observe the computational *efficiency* of execution of each model. To achieve this, we can see the run time to execute one iteration by calculating T_{ave}/I_{age} for each parallel model. Fig. 12 shows T_{ave}/I_{age} for $n_{core} = 2, 4, 8$, and 16. Here again we normalized the values so that T_{ave}/I_{age} of SP-cAS are 1.0.

Observing this figure, values of AP-cAS are large values compared to the other two models for all instances except for tai100b. We can also observe that the values of AS-cAS increase as n_{core} becomes large. As shown in Fig. 7 in Section

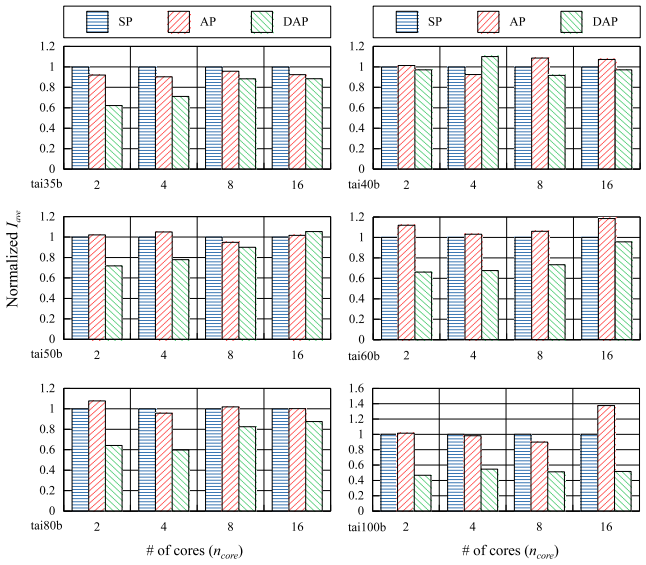

Fig. 11. The average total iterations by m agents to find known optimal solutions over 25 runs. The values are normalized by setting values for SP-cAS is 1.0.

III-B, each agent in AP-cAS accesses a single pheromone density matrix τ_{ij} simultaneously with asynchronous mode. But these accesses must be performed as the critical sections.

As a result, waiting time occurs during these accesses. This waiting time increases in relation to increases of n_{core}. On instances of small size, this waiting time occupies a large part of the processing time. Thus the performance of AP-cAS is poor especially on small size instances or cases of large n_{core}. On tai100b, waiting time does not occupy as much processing time compared to the processing times of ACO, and AP-cAS runs efficiently in asynchronous mode as we expected in Section III-B.

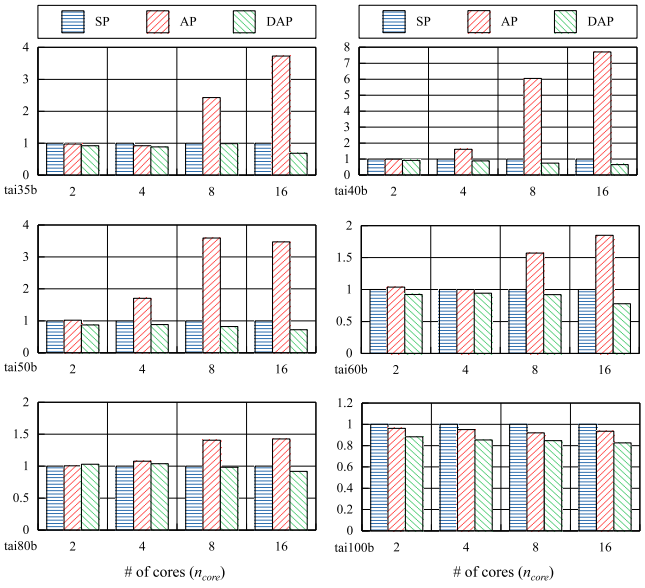

Fig. 12. Run time to execute one iteration by T_{ave}/I_{age}. The values are normalized by setting values for SP-cAS is 1.0.

V. Conclusion

Microprocessor vendors supply processors which have multiple cores of 8, 16, or more, and PCs which use such processors are available at a reasonable cost. In this paper, we studied parallel synchronization models for the ACO algorithms on a multi-core processor to solve QAPs. As for ACO algorithm, we used cAS which we previously proposed. As for parallel synchronization models, we studied three models, i.e., (1) Synchronous Parallel (SP), (2) Asynchronous Parallel (AP), and (3) Distributed Asynchronous Parallel (DAP).

Among them, DAP shows the most promising results over various sizes of QAP instances. It also showed a good scalability up to 16-core. As a natural progression from this study, we need further study using machines with more cores such as Intel® Xeon® Phi™coprocessors.

Acknowledgment

This research is partially supported by the Ministry of Education, Culture, Sports, Science and Technology of Japan under Grant-in-Aid for Scientific Research number 1381211400.

References

[1] S. Tsutsui and N. Fujimoto, "ACO with tabu search on a GPU for solving QAPs using move-cost adjusted thread assignment," in *Genetic and Evolutionary Computation Conference*. ACM, 2011, pp. 1547–1554.

[2] S. Tsutsui, "cAS: Ant colony optimization with cunning ants," *Parallel Problem Solving from Nature*, pp. 162–171, 2006.

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

[4] Q. Lv, X. Xia, and P. Qian, "A parallel aco approach based on one pheromone matrix," *Proc. of the 5th Int. Workshop on Ant Colony Optimization and Swarm Intelligence (ANTS 2006)*, pp. 332–339, 2006.

[5] M. Manfrin, M. Birattari, T. Stützle, and M. Dorigo, "Parallel ant colony optimization for the traveling salesman problems," *Proc. of the 5th Int. Workshop on Ant Colony Optimization and Swarm Intelligence (ANTS 2006)*, pp. 224–234, 2006.

[6] S. Benkner, K. Doerner, R. Hartl, G. Kiechle, and M. Lucka, "Communication strategies for parallel cooperative ant colony optimization on clusters and grids," *Complimentary Proc. of PARA'04 Workshop on State-of-the-art in Scientific Com-puting*, pp. 3–12, 2005.

[7] S. Tsutsui and N. Fujimoto, "Parallel ant colony optimization algorithm on a multi-core processor," in *ANTS Conference*. Springer, 2010, pp. 488–495.

[8] R. E. Burkard, E. Çela, S. E. Karisch, and F. Rendl, "QAPLIB - a quadratic assignment problem library," 2009, www.seas.upenn.edu/qaplib.

[9] T. Stützle and H. Hoos, "Max-Min Ant System," *Future Generation Computer Systems*, vol. 16, no. 9, pp. 889–914, 2000.

[10] E. Cantú-Paz, *Efficient and Accurate Parallel Genetic Algorithms*. Kuwer Academic Publishers, 2000.