



ACO with Tabu Search on a GPU for Solving QAPs using Move-Cost Adjusted Thread Assignment

Shigeyoshi Tsutsui and Noriyuki Fujimoto

MEDAL Report No. 2011005

April 2011

Abstract

This paper proposes an ant colony optimization (ACO) for solving quadratic assignment problems (QAPs) on a graphics processing unit (GPU) by combining tabu (TS) with ACO in CUDA (compute unified device architecture). In TS for QAP, all neighbor moves are tested. These moves form two groups based on computing of move cost. In one group, the computing of cost is $\mathcal{O}(1)$ and in the other group, the computing of move cost is $\mathcal{O}(n)$. We compute these two groups of moves in parallel by assigning the computations to threads of CUDA. In this assignment, we propose an efficient method which we call *Move-Cost Adjusted Thread Assignment (MATA)*. The results with GPU computation with MATA show a promising speedup compared to computation with the CPU. It is also shown that MATA is effective in applying 2-opt local search.

Keywords

QAP, ACO, Local search, Tabu Search, 2-opt, GPU computation.

Missouri Estimation of Distribution Algorithms Laboratory (MEDAL)
Department of Mathematics and Computer Science
University of Missouri–St. Louis
One University Blvd., St. Louis, MO 63121
E-mail: medal@cs.ums1.edu
WWW: <http://medal.cs.ums1.edu/>

ACO with Tabu Search on a GPU for Solving QAPs using Move-Cost Adjusted Thread Assignment

Shigeyoshi Tsutsui

Hannan University
5-4-33 Amamihigashi, Matsubara
Osaka 580-8502, Japan
tsutsui@hannan-u.ac.jp

Noriyuki Fujimoto

Osaka Prefecture University
1-1 Gakuenmachi, Nakaku,
Sakai, Osaka 599-8531, Japan
fujimoto@mi.s.osakafu-u.ac.jp

April 15, 2011

Abstract

This paper proposes an ant colony optimization (ACO) for solving quadratic assignment problems (QAPs) on a graphics processing unit (GPU) by combining tabu (TS) with ACO in CUDA (compute unified device architecture). In TS for QAP, all neighbor moves are tested. These moves form two groups based on computing of move cost. In one group, the computing of cost is $\mathcal{O}(1)$ and in the other group, the computing of move cost is $\mathcal{O}(n)$. We compute these two groups of moves in parallel by assigning the computations to threads of CUDA. In this assignment, we propose an efficient method which we call *Move-Cost Adjusted Thread Assignment (MATA)*. The results with GPU computation with MATA show a promising speedup compared to computation with the CPU. It is also shown that MATA is effective in applying 2-opt local search.

Keywords: QAP, ACO, Local search, Tabu Search, 2-opt, GPU computation.

1 Introduction

Recently, parallel computations using graphics processing units (GPUs) have become popular with great success, especially in scientific fields such as fluid dynamics, image processing, and visualization using particle methods [20]. These parallel computations are reported to see a speedup of tens to hundreds of times compared to CPU computations.

Studies on parallel evolutionary computation with GPU computation are found in genetic programming (GP) [12, 13, 30, 3], genetic algorithms (GAs) [31, 4, 32, 16, 9], evolutionary programming (EP) [8], evolutionary strategies (ESs) [14], ant colony optimization (ACO) [2], and others.

Studies solving the quadratic assignment problem (QAP) on GPUs using an evolutionary model are found in [27, 28, 21, 15]. In [27, 28], distributed GA models were used and no local searches were applied. In [21], a cellular GA model was used and no local searches were used. In [15], parallel hybrid evolutionary algorithms on CPU and GPU were proposed and applied to QAPs.

In our previous studies [27, 28], we applied GPU computation to solve quadratic assignment problems (QAPs) using a distributed parallel GA model on GPUs. However, in those studies no local searches were applied. In this paper, we propose a parallel ACO for QAPs on a GPU by combining tabu search (TS) with ACO in CUDA (Compute Unified Device Architecture [17]).

In a QAP, a solution ϕ is presented by a permutation of $\{1, 2, \dots, n\}$ where n is the problem size. Here we consider neighbors $N(\phi)$ which are obtained by swapping two elements (i, j) of ϕ . In $N(\phi)$, there are $n(n-1)/2$ neighbors. In a TS which uses $N(\phi)$ as neighbors, we need to compute move costs for all neighbors in $N(\phi)$. Depending on the pair value (i, j) , these moves can be divided into two groups based on computing cost.

In one group, the computing of move cost is $\mathcal{O}(1)$ and in the other group, the computing of move cost is $\mathcal{O}(n)$ [23]. We compute these groups' moves in parallel by assigning the computations to threads in a thread block of CUDA. In this assignment, we devised an efficient method that reduces disabling time, as far as possible, in each thread of CUDA. As for the ACO algorithm, we use the Cunning Ant System (*cAS*) which is one of the most promising ACO algorithms [26].

In the remainder of this paper, Section 2 gives a brief review of GPU computation. Then, *cAS* and how we combine it with a local search are described in Section 3. Section 4 describes a TS for combining with ACO to solve QAPs. Section 5 describes implementation of ACO with a TS on a GPU in detail. In Section 6, results and discussions are given. Finally, Section 7 concludes the paper.

2 A Brief Review of GPU Computation

2.1 GPU Computation with CUDA

In terms of hardware, CUDA GPUs are regarded as two-level shared-memory machines [17]. Processors in a CUDA GPU are grouped into multiprocessors (MPs). Each MP consists of thread processors (TPs). TPs in an MP exchange data via fast shared memory (SM). On the other hand, data exchange among MPs is performed via VRAM. VRAM is also like main memory for processors. So, code and data in a CUDA program are basically stored in VRAM.

The CUDA is a multi-threaded programming model. In a CUDA program, threads form two hierarchies: the *grid* and *thread blocks*. A block is a set of threads. A block has a dimensionality of 1, 2, or 3. A grid is a set of blocks with the same size and dimensionality. A grid has dimensionality of 1 or 2. Each thread executes the same code specified by the *kernel function*. A kernel-function call generates threads as a grid with given dimensionality and size. As for GPU in this study, we use a single GTX 480 [18].

2.2 Single Instruction, Multiple Threads

To obtain high performance with CUDA, here we need to know how each thread runs in parallel. The approach is called *single instruction, multiple threads (SIMT)* [19]. In SIMT each MP executes threads in groups of 32 parallel threads called *warps*.

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. However, if threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path.

In our implementation to be described in Section 5, we designed the kernel function so that the threads that belong to the same warp will have as few branches as possible.

3 Sequential ACO with a Local Search

As a bio-inspired computational paradigm, ACO has been applied with great success to a large number of hard problems such as traveling salesman problem (TSP), QAP, scheduling problems, and vehicle routing problems [6]. The first example of an ACO was Ant System (AS) [5]. Since then, many variant ACO algorithms have been proposed as extensions of AS. Typical of these are Ant Colony System (ACS) and MAX-MIN Ant System (MMAS) [7]. In our previous study we proposed a new ACO algorithm called the Cunning Ant System (*cAS*). In this research we use *cAS*.

Although the ACO is a powerful metaheuristic, in many applications of ACO in solving difficult problems, it is very common to combine it with a local search or metaheuristics [7]. In this study, we combine *cAS* with a tabu search (TS) [11] which is also a powerful metaheuristic.

cAS introduced two important schemes [26]. 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 trail density. The other is to use the colony model, dividing colonies 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 partial solution that are constructed based on $\tau_{ij}(t)$ (note that 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* [26].

The colony model of *cAS* is shown in Figure 1. It consists of m units. Each unit consists of only one $ant_{k,t}^*$ ($k = 1, 2, \dots, m$). 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 $ant_{k,t}^*$ ($k = 1, 2, \dots, m$) and $\tau_{ij}(t+1)$ is obtained as usual as:

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

$$\Delta^* \tau_{ij}^k(t) = 1/C_{k,t}^* : \text{if } (i, j) \in ant_{k,t}^*, \quad 0 : \text{otherwise}, \quad (2)$$

where the parameter ρ ($0 \leq \rho < 1$) models the trail evaporation, $\Delta^* \tau_{ij}^k(t)$ is the amount of pheromone by $ant_{k,t}^*$, and $C_{k,t}^*$ is the fitness of $ant_{k,t}^*$. Values of $\tau_{ij}(t+1)$ are set to be within $[\tau_{min}, \tau_{max}]$ as in MMAS [22]. Sequential *cAS* with a local search can be summarized as shown in Figure 2 (please see [26] for detail).

4 Tabu Search for Combining with ACO to Solve QAPs

4.1 Quadratic Assignment Problem (QAP)

The QAP is the problem which assigns a set of facilities to a set of locations and can be stated as a problem to find a permutation ϕ which minimizes

$$cost(\phi) = \sum_{i=1}^n \sum_{j=1}^n a_{ij} b_{\phi(i)\phi(j)}, \quad (3)$$

where $A = (a_{ij})$ and $B = (b_{ij})$ are two $n \times n$ matrices and ϕ is a permutation of $\{1, 2, \dots, n\}$. Matrix A is a flow matrix between facilities i and j , and B is the distance between locations i and

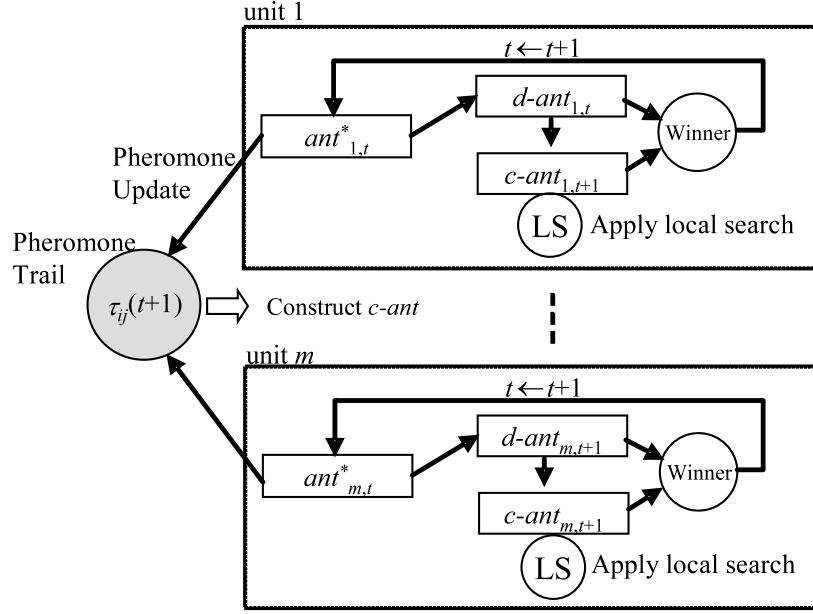


Figure 1: Colony model of cAS with a local search

1. $t \leftarrow 0$
2. Set the initial pheromone density ($\tau_{ij}(t) = \tau_0$)
3. Sample $c-ant_{k,t}$ randomly without using donor ($k=1, 2, \dots, m$)
4. Apply local search (LS) to $c-ant_{k,t}$ and set it $ant^*_{k,t}$ ($k=1, 2, \dots, m$)
5. Update $\tau_{ij}(t+1)$ using Eqs. (1) and (2)
6. Construct $c-ant_{k,t+1}$ based on $d-ant_{k,t}$ ($ant^*_{k,t}$) and $\tau_{ij}(t+1)$ ($k=1, 2, \dots, m$)
7. Apply local search (LS) to $c-ant_{k,t+1}$ ($k=1, 2, \dots, m$)
8. Compare $c-ant_{k,t+1}$ and $d-ant_{k,t}$ and set the best one as $ant^*_{k,t+1}$ ($k=1, 2, \dots, m$)
9. $t \leftarrow t+1$
10. If the termination criteria are met, terminate the algorithm. Otherwise, go to 5.

Figure 2: Algorithm description of sequential cAS with a local search

j . Thus, the goal of the QAP is to place the facilities on locations in such a way that the sum of the products between flows and distances are minimized. The functional value $cost(\phi)$ in Eq. (3) relates to both distances between locations and flows between facilities. As a result, the problem structure of QAP is more complex and harder to solve than TSP [22].

4.2 Tabu Search for QAP of This Study

Tabu search (TS) [11] has been successfully applied to solving large combinatorial optimization problems [10]. Although TS is a powerful metaheuristic, it is often used in conjunction with other solution approaches such as evolutionary computation. In [22], the Robust TS (Ro-TS) by Taillard [23] was combined with MMAS and was used as a local search in solving QAPs.

The main idea of TS is as follows [23]. TS defines a neighborhood, or a set of moves that may be applied to a given solution to produce a new one. Among all the neighboring solutions, TS seeks one with the best evaluation. If there are no improving moves, TS chooses one that least degrades the objective function. In order to avoid returning to the local optimum just visited, a *taboo list* is used.

There are several implementations of TS for QAP. In this study, we implement a TS to combine cAS on a GPU and use it as a local search with short runs. In this implementation, we used Ro-TS with some modification. In the following, we describe the TS for QAP in this study.

Following [24], the taboo list is a 2-dimensional array ($n \times n$) of integers where the element (i, j) identifies the value of the future iteration at which these two elements may again be exchanged with each other. For every facility and location, the latest iteration which the facility occupied that location is recorded. A move is taboo if it assigns both interchanged facilities to locations they had occupied within the taboo list size ($T_{list-size}$) most recent iterations.

The choice of $T_{list-size}$ is critical; if its value is too small, cycling may occur in the search process while if its value is too large, appealing moves may be forbidden and lead to the exploration of lower quality solutions, resulting in a larger number of iterations [24]. To overcome the problem related to the search of the optimal taboo list size, Ro-TS introduced randomness in the value of $T_{list-size}$. Following [25], we set the taboo list size as $T_{list-size} \times r^3$, where r is a uniform random number in $[0, 1)$.

We use the classical aspiration criterion that allows a taboo move to be selected if it leads to a solution better than the best found so far. In [24], an aspiration function that is useful for a longer term diversification process is proposed. In this research, we use TS as a local search in ACO, we do not use this kind of aspiration function.

4.3 Move and Computation of Move Cost in QAP

As described in Section 4.2, TS seeks one with the best evaluation among all the neighboring solutions. If there are no improving moves, TS chooses one that least degrades the objective function. Thus, we need to calculate costs of all neighboring solutions efficiently. Let $N(\phi)$ be the set of neighbors of the current solution ϕ . Then a neighbor, $\phi' \in N(\phi)$, is obtained by exchanging a pair of elements (i, j) of ϕ . Then, we need to compute move costs $\Delta(\phi, i, j) = cost(\phi') - cost(\phi)$ for all the neighbouring solutions. The neighborhood size of $N(\phi)$ ($|N(\phi)|$) is $n(n-1)/2$ where n is the problem size.

When we exchange r and s elements of ϕ (i.e., $\phi(r), \phi(s)$), the change of $cost(\phi)$, $\Delta(\phi, r, s)$, can be computed in computing cost $\mathcal{O}(n)$ as follows:

$$\begin{aligned}
\Delta(\phi, r, s) = & a_{rr}(b_{\phi(s)\phi(s)} - b_{\phi(r)\phi(r)}) + \\
& a_{rs}(b_{\phi(s)\phi(r)} - b_{\phi(r)\phi(s)}) + \\
& a_{sr}(b_{\phi(r)\phi(s)} - b_{\phi(s)\phi(r)}) + \\
& a_{ss}(b_{\phi(r)\phi(r)} - b_{\phi(s)\phi(s)}) + \\
& \sum_{k=0, k \neq r, s}^{n-1} \begin{pmatrix} a_{kr}(b_{\phi(k)\phi(s)} - b_{\phi(k)\phi(r)}) + \\ a_{ks}(b_{\phi(k)\phi(r)} - b_{\phi(k)\phi(s)}) + \\ a_{rk}(b_{\phi(s)\phi(k)} - b_{\phi(r)\phi(k)}) + \\ a_{sk}(b_{\phi(r)\phi(k)} - b_{\phi(s)\phi(k)}) \end{pmatrix}
\end{aligned} \tag{4}$$

Let ϕ' be obtained from ϕ by exchanging r and s elements of ϕ , then fast computation of $\Delta(\phi', u, v)$ is obtained in computing cost $\mathcal{O}(1)$ if u and v satisfy the condition $\{u, v\} \cap \{r, s\} = \emptyset$, as follows [24]:

$$\begin{aligned}
\Delta(\phi', u, v) = & \Delta(\phi, u, v) + \\
& (a_{ru} - a_{rv} + a_{sv} - a_{su}) \times \\
& (b_{\phi'(s)\phi'(u)} - b_{\phi'(s)\phi'(v)} + b_{\phi'(r)\phi'(v)} - b_{\phi'(r)\phi'(u)}) + \\
& (a_{ur} - a_{vr} + a_{vs} - a_{us}) \times \\
& (b_{\phi'(u)\phi'(s)} - b_{\phi'(u)\phi'(v)} + b_{\phi'(v)\phi'(r)} - b_{\phi'(u)\phi'(r)})
\end{aligned} \tag{5}$$

To use this fast update, additional memorization of the $\Delta(\phi, i, j)$ values for all pairs (i, j) in a table are required.

5 Implementation Details of ACO with TS on a GPU with CUDA

5.1 Overall Configuration

We coded the process of each step in Figure 2 as a *kernel function* of CUDA. The overall configuration of ACO with TS for solving QAPs on a GPU with CUDA is shown in Figure 3.

All of the data of the algorithm are located in VRAM of GPU. They include ACO data (the population pools ($ant_{k,t}^*$, $c\text{-}ant_{k,t}$), the pheromone density matrix τ_{ij}), TS data (the temporal memory for move costs, the tabu list), and QAP data (the flow matrix A and distance matrix B). Since matrices A and B are constant data, reading them is performed through texture fetching. On SM, we located only working data which are shared among threads tightly in a block.

As for the local search in Figure 2, we implement TS described in Section 4.2. Construction of a new candidate solution ($c\text{-}ant$) is performed by the kernel function “Construct_solutions(...)” in a single block. Then each m solutions are stored in VRAM. In the kernel function “Apply_tabu_search(...)”, m solutions are distributed in m thread blocks.

The Apply_tabu_search(...) function, in each block, performs the computation of move cost in parallel using a large number of threads. Kernel function “Pheromone_update(...)” consists of 4 separate kernel functions for implementation easiness. Table 1 summarizes these kernel functions.

Kernel functions are called from the CPU for each ACO iteration. In these iterations of the algorithm, only the *best-so-far* solution is transferred to CPU from GPU. It is used for checking whether termination conditions are satisfied. Thus, in this implementation, overhead time used for data transfer between CPU and GPU can be ignored. In the end of a run, whole solutions are transferred from GPU to CPU.

Table 2 shows the distribution of computation time of cAS in solving QAP with sequential runs on a CPU. Here, we used QAP instances tai40a, tai50a, tai60a, tai80a, tai100a, tai50b, tai60b, tai80b, tai100b, and tai150b which will be used in experiments in Section 6. The conditions of the

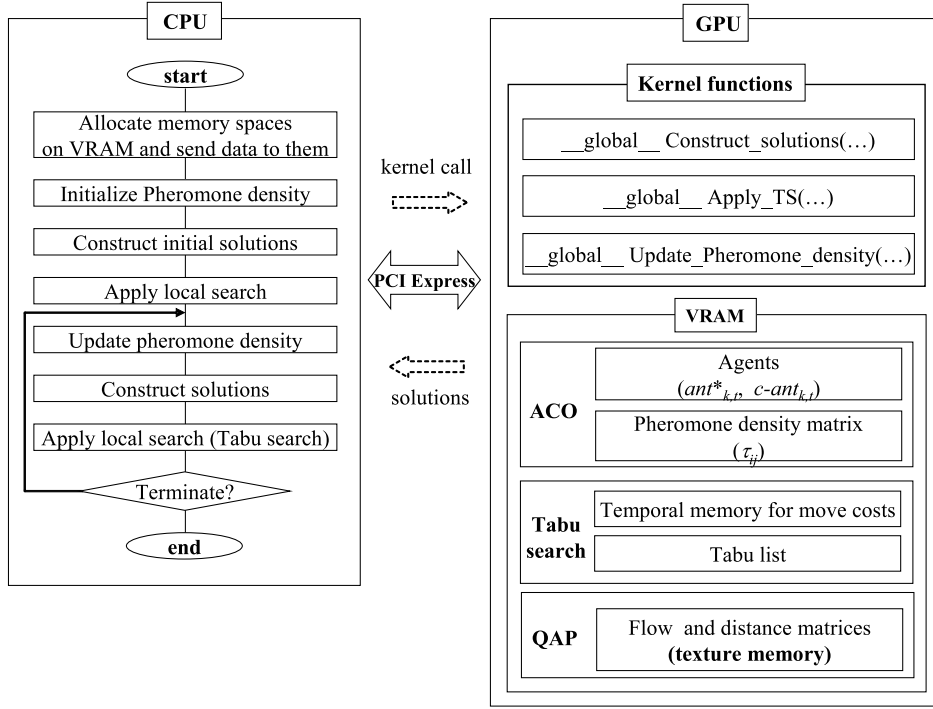


Figure 3: Configuration of ACO with TS for solving QAPs on a GPU with CUDA

Table 1: Summary of kernel functions. n is the problem size, m is the number of agents, and T_{TOTAL} is the total number of threads in a block (see Section 5.2.2)

Kernel functions		dim3		Function
		grid	block	
Update pheromone density	Initialize_pheromone_density()	n	n	$\tau_{ij} = t_0$
	Evaporate_pheromone()	n	n	$\tau_{ij} *= \rho$
	Lay_pheromone()	m	1	$\tau_{ij} += 1/cost$
	Max_min_pheromone()	n	n	adjust τ_{ij} in $[\tau_{min}, \tau_{max}]$
Construct solutions	Construct_solutions(...)	m	1	Construct $c-ant$
Local search	Apply_tabu_search(...)	m	T_{TOTAL}	improve solutions by TS

runs are the same as will be described in Section 6. From this table, we can see that TS uses over 99.9% of the computation time. Thus, we can see that the efficient implementation of TS is the most important factor in increasing speedup of this algorithm.

Table 2: Distribution of computation time of *cAS* in solving QAP with sequential runs on a CPU

Instances	Construction of solutions	TS	Updating Trail
tai40a	0.007%	99.992%	0.001%
tai50a	0.005%	99.994%	0.000%
tai60a	0.004%	99.996%	0.000%
tai80a	0.002%	99.997%	0.000%
tai100a	0.002%	99.998%	0.000%
tai50b	0.022%	99.976%	0.002%
tai60b	0.017%	99.982%	0.001%
tai80b	0.011%	99.988%	0.001%
tai100b	0.008%	99.991%	0.000%
tai150b	0.005%	99.995%	0.000%

5.2 Efficient Implementation of TS with CUDA

5.2.1 An Inefficient Assignment of Move Cost Computations to Threads in a Block

In TS in this study, we use a table which contains move costs so that we can compute the move cost in $\mathcal{O}(1)$ using Eq. (5). For each move, we assign an index number as shown in Figure 4. In this example, we assume a problem size of $n = 8$. Thus, the neighborhood size $|N(\phi)|$ is $8 \times 7/2 = 28$. As described in Section 5.1, each set of move cost calculations of an agent is being done in one block. The simplest approach to computing the move costs in parallel in a block is to assign each move indexed i to the corresponding sequential thread indexed i in a block.

		v							
		0	1	2	3	4	5	6	7
0									
1	0								
2	1	2							
3	3	4	5						
4	6	7	8	9					
5	10	11	12	13	14				
6	15	16	17	18	19	20			
7	21	22	23	24	25	26	27		

Figure 4: Indexing of moves ($n = 8$)

Here, consider a case in which a solution ϕ' is obtained by exchanging positions 2 and 4 of a

current solution ϕ in a previous TS iteration. Then the computation of $\Delta(\phi', u, v)$, shown numbers in gray squares, must be performed in $\mathcal{O}(n)$ using Eq. (4). The computation of the remaining moves are performed in $\mathcal{O}(1)$ quickly using Eq. (5).

Thus, if we simply assign each move to the block thread, threads of a warp diverge via the conditional branch ($\{u, v\} \cap \{2, 4\} = \emptyset$) into two calculations, threads in one group run in $\mathcal{O}(n)$ of Eq. (4) and threads in the other group run in $\mathcal{O}(1)$ of Eq. (5). In threads of CUDA, all instructions are executed in SIMT (please see Section 2.2 for detail). As a result, the computation time of each thread in a warp becomes longer, and we cannot receive the benefit of the fast calculation of Eq. (5) in GPU computation. Figure 5 shows this situation for the case shown in Figure 4. Thus, if threads which run in $\mathcal{O}(1)$ and threads which run in $\mathcal{O}(n)$ co-exist in the same warp, then their parallel computation time in the warp becomes longer than their own respective computation time.

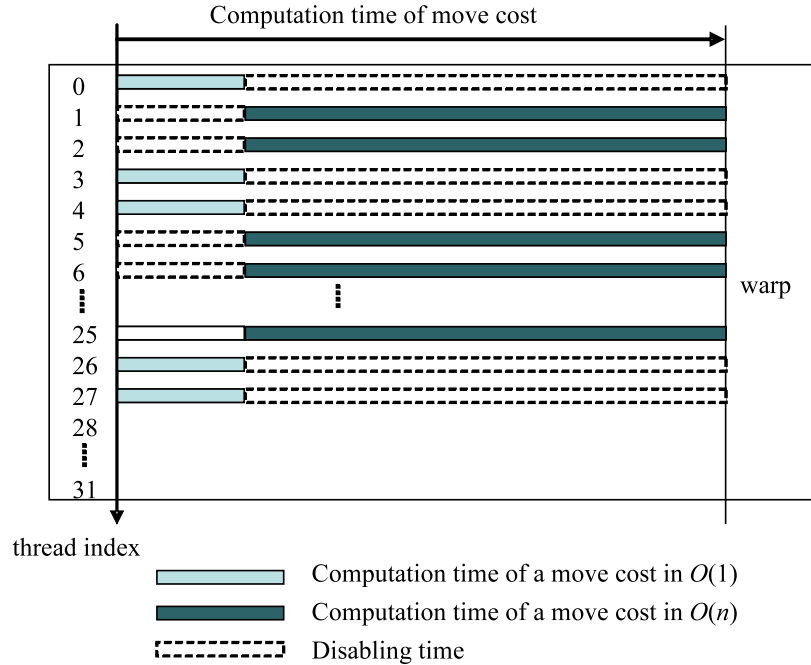


Figure 5: Simple assignment of calculations of move costs to threads in a block. Due to disabling time in SIMT, the parallel computation by thread in a block is inefficient.

5.2.2 Move-Cost Adjusted Thread Assignment (MATA)

In general, for problem size n , the number of moves having move cost in $\mathcal{O}(1)$ is $(n-2)(n-3)/2$ and the number of moves having move cost in $\mathcal{O}(n)$ is $2n-3$. Table 3 shows these values for various problem sizes n . For larger size problems, ratios of $|N(\phi)|$ in $\mathcal{O}(n)$ to $|N(\phi)|$ have smaller values than those in smaller sized problems.

In this research, we assign move cost computations of a solution ϕ which are in $\mathcal{O}(1)$ and in $\mathcal{O}(n)$ to threads which belong to different warps in a block as described below.

Since the computation of a move cost which is $\mathcal{O}(1)$ is smaller than the computation which is $\mathcal{O}(n)$, we assign multiple number N_S of computations which are $\mathcal{O}(1)$ to a single thread in the block. Also, it is necessary to assign multiple calculations of the move costs to a thread, because the maximum number of threads in a block is limited (1024 for GTX 480 [18]).

Table 3: Neighborhood sizes for various problem sizes

Problem size n	Neighborhood size $ N(n) $ $n(n-1)/2$	$ N(n) $ in $O(1)$ $(n-2)(n-3)/2$	$ N(n) $ in $O(n)$ $2n-3$	$ N(n) $ in $O(n)$ $ N(n) $
40	780	703	77	0.099
80	3160	3003	157	0.050
120	7140	6903	237	0.033
160	12720	12403	317	0.025
200	19900	19503	397	0.020

Let C be $|N(\phi)|$ ($C = n(n-1)/2$). Here, each neighbor is numbered by $\{0, 1, 2, \dots, C-1\}$ (see Figure 4). Then, thread indexed as $t = \lfloor k/N_S \rfloor$ computes moves for $k \in \{tN_S, tN_S + 1, \dots, tN_S + N_S - 1\}$. In this computation, if k is a move in $O(n)$, then the thread indexed as t skips the computation. The total number of threads assigned for computations in $O(1)$ is $T_S = \lceil C/N_S \rceil$.

For each thread indexed as t , we need to know the move pair values (i, j) corresponding to each move assigned to it. In a thread indexed as t , if the pair (i, j) for its initial move tN_S is given, move pairs for $tN_S + 1, \dots, tN_S + N_S - 1$ can be easily calculated. So, we prepared a lookup table to provide the pair values only for initial move in each t (move indexed as tN_S).

For the computation in $O(n)$, we assign only one computation of move cost to one thread in the block. Although the total number of moves in $O(n)$ is $2n-3$, we used $2n$ threads for these parallel computations for implementation convenience. Since the threads for these computations must not share the same warp with threads used for computations in $O(1)$, the starting thread index should be a multiple of warp size (32), which follows the index of the last thread used for computation in $O(1)$. Thus, the index of the first thread that computes move in $O(n)$ is $T_{L-START} = \lceil T_S/32 \rceil \times 32$.

This assignment is performed according to move pairs as follows. Let r and s be a pair of previous move. Then we assign pairs (r, x) , $x \in \{0, 1, 2, \dots, n-1\}$ to threads indexed from $T_{L-START}$ to $T_{L-START} + n - 1$ and assign pairs (y, s) , $y \in \{0, 1, \dots, n-1\}$ to threads indexed from $T_{L-START} + n$ to $T_{L-START} + 2n - 1$. Among these $2n$ threads, 3 threads assigned pairs (r, r) , (s, s) , and (r, s) do nothing. Note that thread assigned pair (s, r) do the move cost computation.

Thus, the total number of threads $T_{TOTAL} = T_{L-START} + 2n$ and this kernel function is called from CPU by kernel call “Apply_TS<<< m, T_{TOTAL} >>> (...argument...)”.

Figure 6 shows the thread structure in a block in computing move costs for TS in this study. Hereafter, we refer to this thread structure as *Move-Cost Adjusted Thread Assignment*, or *MATA* for short.

6 Experiments and Discussions

6.1 Experimental platform

In this study, we used a PC which has one Intel Core i7 965 (3.2 GHz) processor and a single NVIDIA GeForce GTX480 GPU. The OS was Windows XP Professional with NVIDIA graphics driver Version 258.96. For CUDA program compilation, Microsoft Visual Studio 2008 Professional

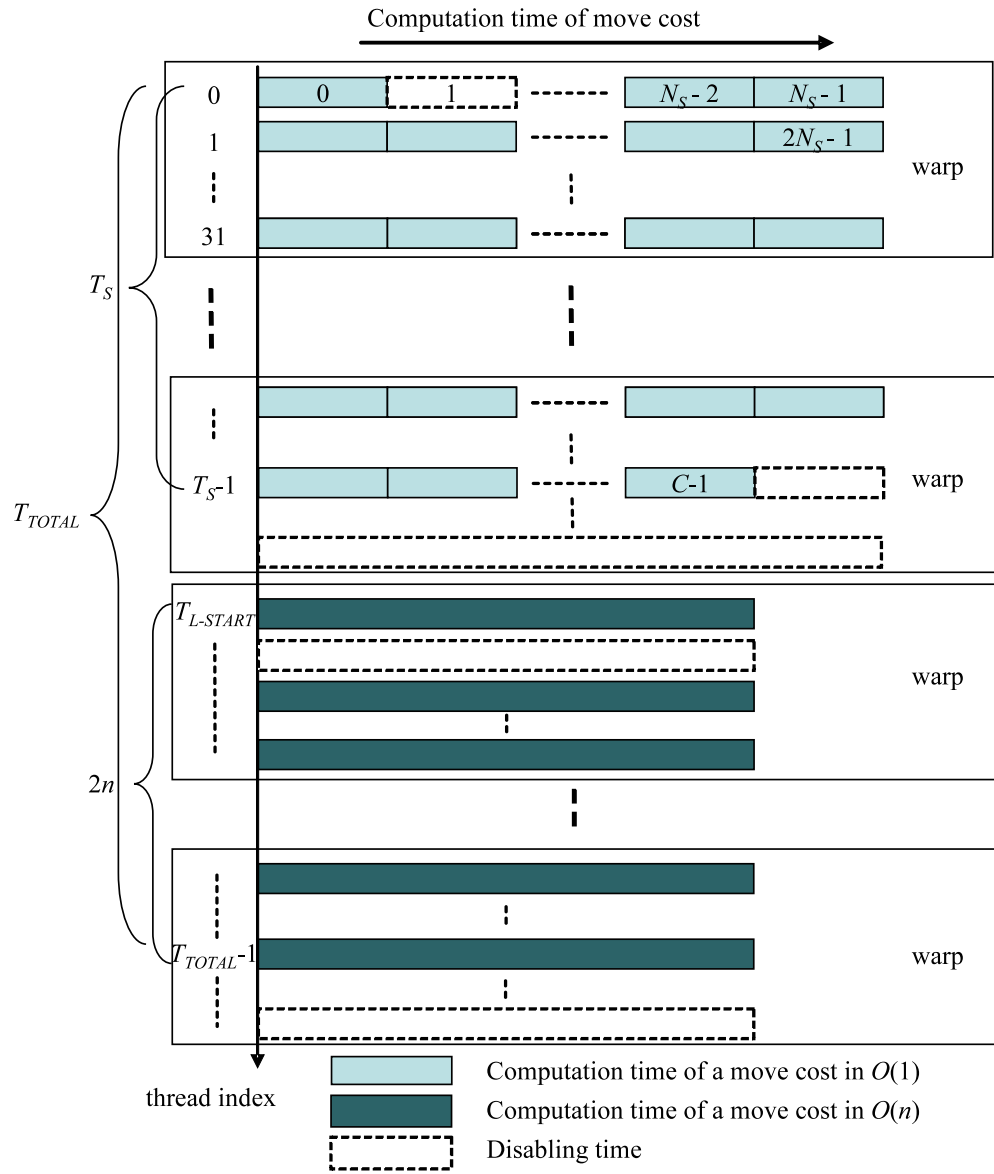


Figure 6: The thread structure in a block in computing move costs for TS (MATA)

Edition with optimization option /O2 and CUDA 3.1 SDK were used.

The instances on which we tested our algorithm were taken from the QAPLIB benchmark library [1]. QAP instances in the QAPLIB can be classified into 4 classes; (i) randomly generated instances, (ii) grid-based distance matrix, (iii) real-life instances, and (iv) real-life like instances [24]. In this experiment, we used the following 10 instances which were classified as either (i) or (iv); tai40a, tai50a, tai60a, tai80a, tai100a, tai50b, tai60b, tai80b, tai100b, and tai150b (the numbers indicate the problem size n).

Here, note that instances classified into class (i) are much harder to solve than those in class (iv). 10 runs were performed for each instance. We did the two types of experiments in the following subsections.

6.2 Experiment 1: Running the Algorithms for Fixed Iterations

Let IT_{TS} be the length of TS applied to one solution which is constructed by cAS , and IT_{ACO} be the iterations of cAS , respectively. Then, $IT_{TOTAL} = m \times IT_{ACO} \times IT_{TS}$ represents a total length of TS in the algorithm. We fix the value of $IT_{TOTAL} = n \times 50000$. In this experiment, if IT_{TOTAL} reaches $n \times 50000$ or a known optimal solution is found, the algorithm terminates.

In runs, we set the following three types of models: (1) runs on a GPU with the $MATA$ in Section 5.2.2, (2) runs on a GPU without using $MATA$, and (3) sequential runs on a CPU using a single thread. In runs without using $MATA$, we assign to one thread N_S number of move calculations, similar to how it is done in $MATA$. Thus, in these runs we used a total number of $T_S = \lceil C/N_S \rceil$ threads in a block. Hereafter we will refer to this as non- $MATA$. In runs on a CPU, we use Intel Core i7 965 (3.2 GHz) processor using a single thread. The parameter values are summarized in Table 4.

Table 4: Control parameter values

Parameters		Values	
		On class (i) QAP	On class (iv) QAP
cAS	m	n	$4n$
	ρ	0.5	0.5
	γ	0.4	0.5
TS	IT_{TS}	$16n$	$4n$
	$T_{list-size}$	n	n
	N_S	$n/4$	$n/4$

The results are summarized in Table 5. First, see the effect of the approach including $MATA$. As seen in the table, the run time results with $MATA$ in T_{avg} were faster than those of the non- $MATA$ run time in T_{avg} , although these speedup values are different among instances. For example, on tai60a, T_{avg} with $MATA$ is 11.9 and T_{avg} with non- $MATA$ is 74.6, respectively. The speedup ratio by $MATA$ is x6.3. On tai60b, the speedup ratio by $MATA$ is x6.0. These speedup values range from x3.5 to x6.3 and the average value of the speedup values over 10 instances is x5.0. Thus, we can confirm that the $MATA$ in Section 5 is a useful approach for fast execution in solving QAPs by ACO with TS on a GPU.

Now see the results of GPU computation of the proposed approach ($MATA$) compared to the results with CPU computations. Please note that in runs on a CPU, there is no parameter N_S in Table 4. Other parameter values are the same as runs on a GPU.

On tai100a, for example, GPU computation with MATA obtained T_{avg} of 89.9 and CPU computation obtained T_{avg} of 1246.6 showing a speedup of x13.9. Note here if we compare T_{avg} of GPU with non-MATA and T_{avg} of CPU, the speedup ratio on this instance is only x3.3. The speedup ratios of GPU with MATA to CPU are in the range from x13.8 to x30.3, showing their average is x19.2.

Table 5: Results of Experiment 1

QAP instances		GPU Computation (GTX 480)				CPU Computation (i7 965 3.2GHz)		Speedup in T_{avg}	
		T_{avg} (sec)		ratio of T_{avg}	Average $Error(\%)$	T_{avg} (sec)	$Error(\%)$	CPU MATA	CPU non-MATA
		MATA	non-MATA	$\frac{\text{non-MATA}}{\text{MATA}}$					
class (i)	tai40a	4.4	27.8	6.2	0.29	74.4	0.31	16.7	2.7
	tai50a	6.8	42.6	6.2	0.49	144.1	0.45	21.0	3.4
	tai60a	11.9	74.6	6.3	0.54	253.7	0.52	21.3	3.4
	tai80a	42.9	143.9	3.4	0.51	612.6	0.50	14.3	4.3
	tai100a	89.9	372.5	4.1	0.55	1246.6	0.51	13.9	3.3
class (iv)	tai50b	0.3	1.5	5.6	0.0	6.318	0.0	22.8	4.1
	tai60b	0.5	3.1	6.0	0.0	15.656	0.0	30.3	5.0
	tai80b	8.8	30.7	3.5	0.0	138.684	0.0	15.7	4.5
	tai100b	16.7	72.6	4.3	0.0	373.03	0.0	22.3	5.1
	tai150b	367.7	1715.9	4.7	0.12	5088.874	0.21	13.8	2.97
average		-	-	5.0	-	-	-	19.2	3.88

6.3 Experiment 2: Comparison of the TS and 2-opt Local Search

As a local search, 2-opt iterative local search is a popular one and is often used in research solving QAPs. In this experiment, we compare the results of ACO with TS and the results of ACO with 2-opt when we solve the QAP on a GPU.

In a 2-opt iterative local search, it explores its neighborhood 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_{2OPT} number of iterations is reached or no improvement solutions are found. Here we can note that 2-opt using best improvement pivoting rule needs to compute all neighbor move costs. This computation is the same with the TS when we solve QAPs. Thus, we can apply MATA to 2-opt local search implementation [29].

When we perform a fair comparison of different algorithms, sometimes it is difficult to determine their termination criteria. In experiment 2, we run the algorithms until predetermined acceptable solutions are obtained. For acceptable solutions for class (i), we set them to be within 1.0% of the known optimal solutions. For class (iv) instances, except tai150b, we set them be known optimal solutions. For tai150b, we set them to be within 0.2% of the known optimal solution. 10 runs were performed for each instance. We measured the performance by the average time to find acceptable solutions T_{avg} over 10 runs. For 2-opt, we set the IT_{2OPT} and m of class (iv) QAPs to $4 \times n$ and $4 \times n$, respectively. Other parameter values are the same as described in Table 4.

The results are summarized in Table 6. First, see the ratios of T_{avg} of TS with MATA and 2-opt using MATA. These values on instances in class (i) are much larger than those in class (iv). For example, on tai40a the value is 5.3, showing TS is 5.3 times faster than 2-opt. These values become progressively larger with progressively larger problems. For example, on tai100a the value is 127.9

times faster. However, instances in class (iv), these values are smaller than 1.0 except for on tai60b, showing 2-opt local search is faster than TS local search. TS typically gives higher quality solutions at the cost of higher run times. It is especially useful on class (i) instances (randomly generated instances), but on class (iv) instances (real-life like instances), 2-opt local search work well. This results coincide with the earlier study of MMAS in [22].

Table 6: Results of Experiment 2

QAP instances		Acceptable error (%)	Local search	GPU Computation (GTX 480)				CPU Com. (i7 965)	Speedup in T_{avg}
				T_{avg} (sec)		ratios of T_{avg}		T_{avg} (sec)	$\frac{\text{CPU}}{\text{GPU}}$ with MATA
				MATA	non-MATA	$\frac{2\text{-opt}}{\text{TS}}$ with MATA	$\frac{2\text{-opt}}{\text{TS}}$		
class (i)	tai40a	1.0	TS	0.1	0.6	5.3	7.0	2.0	22.8
			2-opt	0.5	2.0		4.3	7.7	16.7
	tai50a	1.0	TS	0.4	2.5	18.5	6.6	9.9	25.4
			2-opt	7.2	31.3		4.4	398.5	55.5
	tai60a	1.0	TS	1.0	5.5	25.1	5.3	23.3	22.7
			2-opt	25.7	150.7		5.9	697.4	27.1
	tai80a	1.0	TS	3.2	12.1	81.4	3.8	56.7	17.7
			2-opt	261.1	719.4		2.8	3872.7	14.8
	tai100a	1.0	TS	7.1	29.5	127.9	4.1	108.6	15.3
			2-opt	909.9	4337.0		4.8	20062.5	22.0
class (iv)	tai50b	0.0	TS	0.3	1.5	0.8	5.7	6.3	23.2
			2-opt	0.2	1.6		7.1	5.0	22.5
	tai60b	0.0	TS	0.5	3.1	1.1	6.2	15.7	31.2
			2-opt	0.5	4.4		8.2	11.4	21.4
	tai80b	0.0	TS	8.1	30.8	0.8	3.8	138.7	17.1
			2-opt	6.8	39.0		5.7	121.6	17.9
	tai100b	0.0	TS	16.2	66.9	0.7	4.1	373.0	23.1
			2-opt	11.8	69.2		5.8	280.1	23.6
	tai150b	0.2	TS	138.6	697.4	0.7	5.0	1815.1	13.1
			2-opt	98.8	612.8		6.2	1596.8	16.2
average			TS	-	-	-	5.2	-	21.2
			2-opt	-	-	-	5.5	-	23.8

Now see these results of GPU computation of 2-opt with MATA. Comparing results by CPU computation, we can confirm that MATA is useful with 2-opt local search as is seen with TS.

7 Conclusions

In this paper, we propose an ACO for solving quadratic assignment problems (QAPs) on a GPU by combining TS local search in CUDA. In TS on QAPs, there are $n(n-1)/2$ neighbors in a candidate solution. These TS moves form two groups based on computing cost. In one group, the computing of move cost is $\mathcal{O}(1)$ and in the other group, the computing of move cost is $\mathcal{O}(n)$.

We compute these groups of moves in parallel by assigning the computations to threads of CUDA. In this assignment, we proposed an efficient method which we call *Move-Cost Adjusted Thread Assignment (MATA)* that can reduce disabling time, as far as possible, in each thread of CUDA. As for the ACO algorithm, we use the Cuning Ant System (*cAS*).

The results showed that GPU computation with MATA showed a promising speedup compared to computation with CPU. We also confirmed MATA is useful 2-opt local search which uses less computation cost than TS local search. Although we used a single GPU, study on colony models

using multiple GPUs is an interesting future research direction.

8 Acknowledgments

This research is partially supported by the Ministry of Education, Culture, Sports, Science and Technology of Japan under Grant-in-Aid for Scientific Research No. 22500215.

References

- [1] QAPLIB - a quadratic assignment problem library, 2009. www.seas.upenn.edu/qaplib.
- [2] H. Bai, D. OuYang, X. Li, L. He, and H. Yu. Max-Min ant system on GPU with CUDA. In *International Conference on Innovative Computing*, pages 801–804, 2009.
- [3] W. Banzhaf, S. Harding, W. Langdon, and G. Wilson. Accelerating genetic programming through graphics processing units. *Genetic Programming Theory and Practice VI*, 12(12):1–19, 2009.
- [4] T. F. Clayton, L. N. Patel, G. Leng, A. F. Murray, and I. A. B. Lindsay. Rapid evaluation and evolution of neural models using graphics card hardware. In *Genetic and Evolutionary Computation Conference*, pages 299–306, 2008.
- [5] M. Dorigo, V. Maniezzo, and A. Coloni. The ant system: Optimization by a colony of cooperating agents. *IEEE Trans. on SMC-Part B*, 26(1):29–41, 1996.
- [6] M. Dorigo and T. Stützle. Ant colony optimization: Overview and recent advances. *Handbook of Metaheuristics*, 2.
- [7] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, Massachusetts, 2004.
- [8] K. Fok, T. Wong, and M. Wong. Evolutionary computing on consumer-level graphics hardware. *IEEE Intelligent Systems*, 22(2):69–78, 2007.
- [9] N. Fujimoto and S. Tsutsui. A highly-parallel TSP solver for a GPU computing platform. *International Conference on Numerical Methods and Applications*.
- [10] M. Gendreau and J. Potvin. Tabu search. *Handbook of Metaheuristics*, 2.
- [11] F. Glover and M. Laguna. *Tabu Search*. Kluwer, Boston, 1997.
- [12] W. Langdon and W. Banzhaf. A SIMD interpreter for genetic programming on GPU graphics cards. In *European Conference on Genetic Programming*, pages 73–85. Springer, 2008.
- [13] W. Langdon and A. Harrison. GP on SPMD parallel graphics hardware for mega bioinformatics data mining. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 12(12):1169–1183, 2008.
- [14] T. V. Luong, N. Melab, and E.-G. Talbi. GPU-based island model for evolutionary algorithms. In *Genetic and Evolutionary Computation Conference*, pages 1089–1096, 2010.
- [15] T. V. Luong, N. Melab, and E.-G. Talbi. Parallel hybrid evolutionary algorithms on gpu. In *IEEE Congress on Evolutionary Computation*, pages 2734–2741, 2010.

- [16] O. Maitre, L. A. Baumes, N. Lachiche, A. Corma, and P. Collet. Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA. In *Genetic and Evolutionary Computation Conference*, pages 1403–1410. ACM, 2009.
- [17] NVIDIA, 2010. www.nvidia.com/object/cuda_home_new.html.
- [18] NVIDIA, 2010. www.nvidia.com/object/fermi_architecture.html.
- [19] NVIDIA, 2010. developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf 1.
- [20] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S.-Z. Ueng, S. S. Baghsorkhi, and W. mei W. Hwu. Program optimization carving for gpu computing. *J. Parallel Distrib. Comput.*, 68(10):1389–1401, 2008.
- [21] N. Soca, J. L. Blengio, M. Pedemonte, and P. Ezzatti. PUGACE, a cellular evolutionary algorithm framework on GPUs. In *IEEE Congress on Evolutionary Computation*, pages 3891–3898, 2010.
- [22] T. Stützle and H. Hoos. Max-Min Ant System. *Future Generation Computer Systems*, 16(9):889–914, 2000.
- [23] É. Taillard. Robust taboo search for quadratic assignment problem. *Parallel Computing*, 17:443–455, 1991.
- [24] É. Taillard. Comparison of iterative searches for the quadratic assignment problem. *Location Science*, 3(2):87–105, 1995.
- [25] É. Taillard. `taboo search tabou_qap code`, 2004. mistic.heig-vd.ch/taillard/codes.dir/tabou_qap.cpp.
- [26] S. Tsutsui. cAS: Ant colony optimization with cunning ants. *Parallel Problem Solving from Nature*, pages 162–171, 2006.
- [27] S. Tsutsui and N. Fujimoto. Solving quadratic assignment problems by genetic algorithms with GPU computation: a case study. In *Genetic and Evolutionary Computation Conference (Companion)*, pages 2523–2530. ACM, 2009.
- [28] S. Tsutsui and N. Fujimoto. An analytical study of GPU computation for solving QAPs by parallel evolutionary computation with independent run. In *IEEE Congress on Evolutionary Computation*, pages 889–896, 2010.
- [29] S. Tsutsui and N. Fujimoto. Fast qap solving by aco with 2-opt local search on a gpu (in press). In *IEEE Congress on Evolutionary Computation*, 2011.
- [30] G. Wilson and W. Banzhaf. Linear genetic programming GPGPU on Microsoft’s Xbox 360. In *IEEE Congress on Evolutionary Computation*, pages 378–385, 2008.
- [31] M. Wong and T. Wong. Parallel hybrid genetic algorithms on consumer-level graphics hardware. In *IEEE Congress on Evolutionary Computation*, pages 2973–2980, 2006.
- [32] M. L. Wong. Parallel multi-objective evolutionary algorithms on graphics processing units. In *Genetic and Evolutionary Computation Conference (Companion)*, pages 2515–2522. ACM, 2009.