A Parallel Iterated Local Search Algorithm on GPUs for Quadratic Assignment Problem

Erdener Ozcetin*, Gurkan Ozturk**

* Industrial Engineering Department, Hitit University Engineering Faculty, Corum, Turkey

** Industrial Engineering Department, Eskisehir Technical University Engineering Faculty, Eskisehir, Turkey

(erdenerozcetin@hitit.edu.tr, gurkan.o@anadolu.edu.tr)

‡ Corresponding Author; Erdener Ozcetin, Industrial Engineering Department, Hitit University Engineering Faculty, Corum, Turkey , Tel: +90 364 227 4533,

Fax: +90 364 227 4535, erdenerozcetin@hitit.edu.tr

Received: 25.10.2018 Accepted: 21.06.2018

Abstract- In this study, quadratic assignment problem, which is a hard combinatorial optimization problem, is examined to solve by a new approach. To reach the optimal results by using mathematical programming approaches cannot be possible even for some sorts of small and middle scaled problems in a reasonable time interval. Huge amounts of data are being progressed simultaneously by graphics processing units located on computers' graphics card. Therefore, a parallel iterated local search algorithm has been proposed to solve the quadratic assignment problem by using graphics processing units' simultaneously progressing property. This parallel algorithm and the sequential one on central processing units are tested and compared for test problems in literature. Indeed, it is observed that the parallel algorithm works averagely 6.31 times faster for Skorin problems and 11.93 times faster for Taillard problems faster than sequentially one.

Keywords Quadratic assignment problem (QAP), local search algorithms, graphics processing units (GPU), CUDA.

1. Introduction

Combinatorial optimization is a topic that consists of finding optimal solution from a finite set of alternative solutions (n!) that generally studied in applied mathematics and theoretical computer sciences. Quadratic assignment problem (QAP) is a type of combinatorial optimization problem that was introduced by Koopmans and Beckmen in 1957 [1].

As a real life problem QAP can be defined facility planning, circuit designing and assigning the air-crafts to the gates. If we investigate QAP as a facility planning problem, the objective is assigning n facilities to n candidate locations by minimizing a cost function. The mathematical formulation of QAP can be as:

$$\min z = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}\sum_{l=1}^{n} f_{ij} d_{kl} x_{ik} x_{jl} \qquad (1.1)$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad j = \overline{1,n} \qquad (1.2)$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad i = \overline{1,n} \qquad (1.3)$$

$$x_{ij} \in \{0,1\} \; 1 \le i,j \le n \qquad (1.4)$$

In this formulation, f_{ij} is the flow between two facilities and d_{kl} is the distance between two candidates. Complexity study of this mathematical model is handled by Sahni and Gonzalez [2] in 1976. They showed that QAP is *NP-Hard* and even finding and ϵ approximate solution is extremely hard.

With the reason of QAP is *NP-Hard*, working with exact methods such as mathematical programming is so difficult. For this reason, researchers are tended to study on meta-heuristics. Although meta-heuristic algorithms do not guarantee to find optimal solution, they can converge to optimal solution much faster than exact methods. There are many studies in literature that subjected solving QAP with meta-heuristics. Simulated annealing is an iteration based meta-heuristic algorithm that developed by Kirkpatrick et al. [3]. Burkard and Rendl [4], Wilhelm and Ward [5] and Abreu et al. [6] used this algorithm for QAP in their studies. Ant colony optimization (ACO) is a population based meta-heuristic algorithm. Stützle and Dorigo [7] and Dorigo et al. [8] implemented this algorithm for QAP. Another population based algorithm is genetic algorithm (GA) that published in QAP literature such as Kochhar et al. [9] and Drezner [10]. Glover [11], developed iteration based tabu search (TS) algorithm to solve problems that oriented to integer programming, in 1989. Main concentration of this algorithm is to search neighbors with a method to escape local optima. Skorin [12], Taillard [13], Misevicius and Ostereika [14] proposed a TS algorithm for QAP.

Introduction of CUDA by nVIDIA in 2006 can be considered as a turning point of using for computational sciences of Graphics Processing Units (GPUs) [15]. Thousands of applications have been proposed from different disciplines of science since that day. As a reason of complexity of QAP and compatibility of meta-heuristics are lead researchers in this area to work on GPUs. The motivation of starting these studies is to accelerate the algorithms and to decrease the time of solutions, reasonably. On the other hand, there are only several studies that related to combinatorial optimization with meta-heuristics on GPUs. Tsutsui and Fujimoto [16] published a parallel GA on GPUs for QAP and they gained between 3-12 times speed-ups according to Central Processing Unit (CPU) implementation. In another study, same authors [17], applied a hybrid ACO algorithm. They focused on local search to parallelize that the most time consuming part of algorithm and they observed up to 24.6 times speed-ups. Czapinski [18] proposed multi-start TS algorithm on GPUs to solve QAP. They gained up to 50 times speed-ups for symmetric instances. Again for symmetric instances, Ozcetin and Ozturk [19] handled averagely 17 times up to 51 times speed-ups with a hybrid evolutionary algorithm on GPUs.

2. Iterated Local Search Algorithm

Iterated local search algorithms (ILS) can be defined as a hybrid local search (LS) algorithms that improved with global search procedures. Basically, LS stop when a local optimum solution found. Two procedures are often used to continue searching from the solution found with LS. First one is perturbing the current local minimum. This is the simplest way to start the search from another starting point. Second one is modifying solution according to a mutation based procedure.

In this study, a multi start ILS algorithm is developed for solving symmetric QAP instances efficiently. In the first phase of algorithm the instances is read to two matrices. *D* is the distance matrix of locations and *F* is the flow matrix of facilities. After that, objective function evaluation is calculated for each solution only once. Then, the algorithm starts to search. There are $\frac{n(n-1)}{2}$ alternative changes for each solution, so calculation of objective function for each iteration is a computationally expensive operation. For this reason, during the search objective function evaluation is not considered again. Instead of objective function calculation a delta function is used like in equation 2.1. A solution is defined as a permutation. If we consider about changing a^{th} and b^{th} neighbors in permutation, *π(a)* and *π(b)* will be the values in permutation of a and b, respectively This candidate changing is done, if and only if δ value is lower than zero.

$$\delta = \sum_{t=1\ a\neq t\ b\neq t}^{n} (d_{at} - d_{bt})(f_{\pi(b)\pi(t)} - f_{\pi(a)\pi(t)}) \qquad (2.1)$$

Basically, the algorithm has two main operators. One of them is LS and the other one is mutation operator. Mutation operator is designed to escape the local optimum in a systematic procedure with the cross exchange of two elements in a solution.

3. Parallelization of Algorithm

Some parts of proposed ILS algorithm is available for parallelization. For p starting elements objective function evaluation is handled in a parallel fashion like in Figure 1.

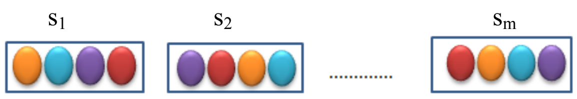

Fig 1. Parallelization of elements

For CPU implementation, loops of an objective function evaluation are also a demanding issue for parallelization. It takes n^2 steps for sequential version. This means that there are n^2 multiplication of distance and flow and summation. In parallel version, a kernel is organized to carry out all these multiplications simultaneously.

Another kernel is organized for parallel LS procedure. For *p* elements each neighborhood search is investigated in a parallel manner. Following figure is an example of 2nd and 4th neighbors changing.

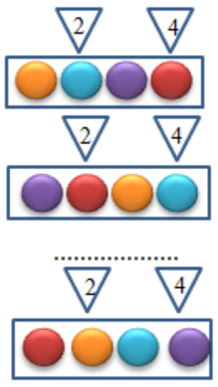

Fig 2. An illustrative example for parallel local search

Let we think about four facilities will be assigned four candidate locations. In this problem T_1, T_2, T_3, T_4 are the elements of flow matrix between facilities *F* and Y_1, Y_2, Y_3, Y_4 are the elements of distance matrix between locations *D*.

Think that we have 3 individual solutions with the permutations Π_1, Π_2, Π_3, 1-2-3-4, 4-1-3-2 and 1-3-4-2, respectively. For third solution, assignment will be like this: T1→ Y1, T3→Y2, T4→Y3 and T2→Y4. According to this information, multiplications of objective function calculation will be like in Figure 3.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
S	d_{12}	d_{13}	d_{14}	d_{23}	d_{24}	d_{34}	d_{12}	d_{13}	d_{14}	d_{23}	d_{24}	d_{34}	d_{12}	d_{13}	d_{14}	d_{23}	d_{24}	d_{34}
R	f_{12}	f_{13}	f_{14}	f_{23}	f_{24}	f_{34}	f_{41}	f_{43}	f_{42}	f_{13}	f_{12}	f_{32}	f_{13}	f_{14}	f_{12}	f_{34}	f_{32}	f_{42}
A	$s_1.r_1$	$s_2.r_2$	$s_3.r_3$	$s_4.r_4$	$s_5.r_5$	$s_6.r_6$	$s_7.r_7$	$s_8.r_8$	$s_9.r_9$	$s_{10}.r_{10}$	$s_{11}.r_{11}$	$s_{12}.r_{12}$	$s_{13}.r_{13}$	$s_{14}.r_{14}$	$s_{15}.r_{15}$	$s_{16}.r_{16}$	$s_{17}.r_{17}$	$s_{18}.r_{18}$
Obj.	$2(A_1+A_2+A_3+A_4+A_5+A_6)$						$2(A_7+A_8+A_9+A_{10}+A_{11}+A_{12})$						$2(A_{13}+A_{14}+A_{15}+A_{16}+A_{17}+A_{18})$					

Fig 3. An illustrative example for parallel evaluations

There are $\frac{4(4-1)}{2} = 6$ multipcations for each element's objective function calculation. Each multiplication corresponds to unique thread for GPU and is collected in the array of *A*. Objective function values is calculated by summarizing the A with parallel reduction via *thrust library*.

For parallel local search same neighborhood search is handled like in Figure 4. Calculation of delta function is computed via a device kernel. First step of neighborhood search for this example, for first individual changing of 1-2 orders in permutation, for second 4-1 and for third 1-3 handled simultaneously. Only second individual's changing is done as a reason of delta value (-240). This procedure continues as the same.

F, *D* matrices and permutations are allocated on GPU side with *thrust* vectors. For each iteration only solutions are copied. Shared memory and constant memory cannot be used as a reason of capacity. For this reason, only device memory has been used for the implementation on GPUs

Individual	Orders	Facilities	Delta
Π_1	$\pi_1\ \pi_2$	1-2	120
	$\pi 1\ \pi 3$	1-3	
	$\pi_1\ \pi_4$	1-4	
	$\pi_2\ \pi_3$	2-3	
	$\pi_2\ \pi_4$	2-4	
	$\pi_3\ \pi_4$	3-4	
Π_2	$\pi_1\ \pi_2$	4-1	-240
	$\pi_1\ \pi_3$	4-3	
	$\pi_1\ \pi_4$	4-2	
	$\pi_2\ \pi_3$	1-3	
	$\pi_2\ \pi_4$	1-2	
	$\pi_3\ \pi_4$	3-2	
Π_3	$\pi_1\ \pi_2$	1-3	180
	$\pi_1\ \pi_3$	1-4	
	$\pi_1\ \pi_4$	1-2	
	$\pi_2\ \pi_3$	3-4	
	$\pi_2\ \pi_4$	3-2	
	$\pi_3\ \pi_4$	4-2	

Fig 4. Parallel neighborhood search

4. Results

A work station which have 6 cores CPU with 32 GB RAM and a GTX580 chipset GPU is used for comparisons. There are 1000 individual solutions for multi-start. In addition to this, probability of accepting worse solution in mutation operator is 0.4. Grid parameters are used for objective function evaluation *n* block in grid, *n* threads in a block and for parallel local search 125 blocks in grid, and 8 threads in a block (This numbers defined with preliminary studies). Number of iterations is not a stopping criterion for these implementations. Stopping criterion is done with gap of solution $(\frac{Avg.Obj.Val-BK}{BK})$ (BK=best known solution).

In tables 1, 2 GAP CPU and GAP GPU columns represent the average gap of ten iterations both on CPU and GPU. CPU(s) and GPU(s) columns represent solution times. In addition, HIT column represents number of hits to best known solution for ten iterations. Finally, speed factor is calculated with $(\frac{CPU(s)}{GPU(s)})$.

Table 1. Skorin Test Problems

Problem	BK	CPU(s)	GPU(s)	GAP CPU	GAP GPU	HIT	Speed Up
Sko42	15812	40.347	2.843	0.0001	0	9/10	14.18x
Sko49	23386	65.814	10.179	0.0005	0.0006	2/10	6.46x
Sko56	34458	66.211	11.161	0.0002	0.0004	1/10	5.93x
Sko64	48498	80.243	18.647	0.0001	0.0003	4/10	4.30x
Sko72	66256	104.15	21.579	0.0013	0.0015	0/10	4.82x
Sko81	90998	118.223	24.813	0.0013	0.0014	0/10	4.76x
Sko90	115534	152.651	31.289	0.0017	0.0019	0/10	4.87x
Sko100a	152002	190.581	36.695	0.0020	0.0022	0/10	5.19x
Avg.		**102.27**	**19.65**	**0.0009**	**0.0009**		**6.31x**

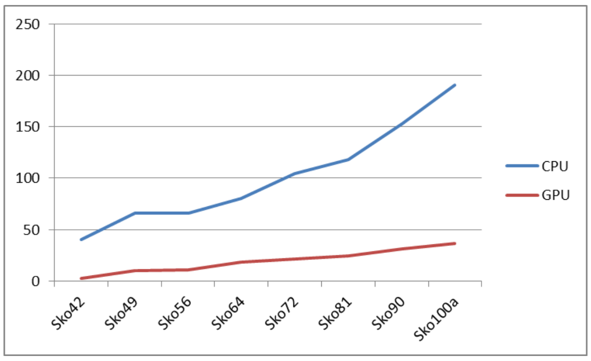

Fig 5. Skorin Speed-Ups

For comparisons all test instances are taken from QAPLIB (http://anjos.mgi.polymtl.ca/qaplib/) with best known solutions up to date. Skorin and Taillard test problems are the well-known hardest test instances for QAP.

Table 1 and Fig. 5 are the results for Skorin test problems. Average gap of these problems are 0.0009 both on CPU and GPU implementation. In addition, 6.31x speed-up gained for these problems. For Sko42 problem, execution time of CPU version is 40.347 seconds and GPU version is 2.843 seconds. The speed-up factor for this problem is 14.18x with the 9/10 best known solution hit. Computational results of the proposed ILS for Taillard problems are shown in Table 2 and Figure 6 According to this, average gap for these problems set are 0.0063 on CPU and 0.0068 on GPU, respectively. If we have an eye on execution times, GPU implementation runs 11.93 times faster than CPU implementation. Best speed-up observed with Tai30a problem with a 16.69x (CPU(s)/GPU(s)) factor.

Table 2. Taillard Test Problems

Prob.	BK	CPU(s)	GPU(s)	GAP CPU	GAP GPU	HIT	Speed Up
Tai20a	703482	1.943	0.493	0	0	10/10	3.93x
Tai25a	1167256	14.95	1.165	0	0	10/10	12.83x
Tai30a	1818146	39.849	2.387	0.0003	0.0004	8/10	16.69x
Tai35a	2422002	116.682	10.472	0.0008	0.0008	7/10	11.14x
Tai40a	3139370	179.576	11.398	0.0004	0.0004	0/10	15.75x
Tai50a	4938796	277.076	18.202	0.0093	0.0091	0/10	15.38x
Tai60a	7208572	360.42	32.112	0.0117	0.0135	0/10	11.25x
Tai80a	13515450	453.78	44.188	0.0122	0.0121	0/10	10.29x
Tai100a	21054656	602.98	59.776	0.022	0.025	0/10	10.19x
Avg.		**227.47**	**20.02**	**0.0063**	**0.0068**		**11.93x**

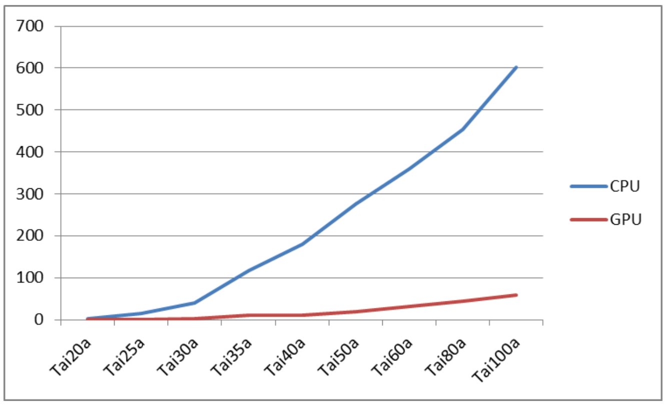

Fig 6. Taillard Speed-Ups

5. Conclusion

Solving combinatorial optimization problems are extremely hard and computationally expensive. For many cases, it can be impossible to solve these problems optimally with exact methods. For these reasons, researchers tend to study with meta-heuristics. Even meta-heuristics converge to approximate solution much more reasonable time, it is still a handicap for large scale instances of combinatorial optimization problems. According to this motivation, we consider about a parallel implementation of ILS algorithm on GPUs. We observed on preliminary studies that our algorithm finds optimal solutions for Escherman and Nuggent problems for all runs in a few seconds. In our results, we prefer to show more difficult problems Skorin and Taillard test instances. Our algorithm converges to best known solution with small gaps. Averagely, 6.31x speed-up gained for Skorin test instances and 11.93x speed-up gained for Taillard test instances. For future studies, GPU implementation of algorithm is available for development. For much more speed-up memory operations can be improved. Finally, this algorithm is also available for other combinatorial optimization problems.

REFERENCES

[1] T. C. Koopmans ve M. J. Beckmann, “Assignment problems and the location of economic activities,” *Econometrica,* vol. 25, pp. 53-76, 1957.

[2] S. Sahni ve T. Gonzalez, “P-complete approximation problems,” *Journal of the Association of Computing Machinery,* vol. 23, pp. 555-565, 1976.

[3] S. Kirkpatrick, C. Gelat ve M. Vecchi, “Optimization by Simulated Annealing,” *Science,* vol. 220, pp. 671-680, 1983.

[4] R. Burkard ve F. Rendl, “A thermodynamically motivated simulation procedure for combinatorial optimization problems,” *European Journal of Operational Research,* vol. 17, pp. 169-174, 1984.

[5] M. Wilhelm ve T. Ward, “Solving quadratic assignment problems by 'simulated annealing',” *IIE Transactions,* vol. 19, pp. 107-119, 1987.

[6] N. Abreu, T. Querido ve P. Boaventura-Netto, “A simulated annealing for the quadratic assignment problem,” *Rairo-Operations Research,* vol. 33, pp. 249-273, 1999.

[7] T. Stützle ve M. Dorigo, “ACO Algorithms for the Quadratic Assignment Problem,” New Ideas in Optimization,McGraw-Hill, 1999.

[8] M. Dorigo, V. Maniezzo ve A. Colorni, “The Ant System: Optimization by a colony of cooperating agents,” *IEEE Transactions on Systems,* vol. 26, pp. 1-13, 1996.

[9] J. Kochhar, B. Foster ve S. Heragu, “A genetic algorithm for the unequal area facility layout problem,” *Computers & Operations Research,* vol. 25, pp. 583-594, 1998.

[10] Z. Drezner, “A New Genetic Algorithm for the Quadratic Assignment Problem,” *Informs Journal on Computing,* vol. 15, pp. 320-330, 2003.

[11] F. Glover, “Tabu Search - Part I,” *ORSA Journal on Computing,* vol. 1, pp. 190-206, 1989.

[12] J. Skorin-Kapov, “Tabu Search Applied to the Quadratic Assignment Problem,” *ORSA Journal on Computing,* vol. 2, pp. 33-45, 1990.

[13] E. D. Taillard, “Robust taboo search for the quadratic assignment problem,” *Parallel Computing,* vol. 17, pp. 443-455, 1991.

[14] A. Misevičius ve A. Ostreika, “Defining Tabu tenure for the Quadratic Assignment Problem,” *Information Technology and Control,* vol. 36, 2007.

[15] “Developer Centers: CUDA Zone,” nVIDIA,. Available: http://developer.nvidia.com/cuda/what-cuda.

[16] S. Tsutsui ve N. Fujimoto, “Solving Quadratic Assignment Problems by Genetic Algorithms with GPU Computation: A Case Study,” *GECCO*, Montreal, 2009.

[17] S. Tsutsui ve N. Fujimoto, “Fast QAP Solving by ACO with 2-opt Local Search on a GPU,” *IEEE Section Congres*, San Francisco, 2011.

[18] M. Czapinski, “An effective Parallel Multistart Tabu Search for Quadratic Assignment Problem CUDA platform,” *J. Parallel Distrib. Comput., vol. 73(11), pp. 1461-1468, 2013 .*

[19] E. Özçetin ve G. Öztürk, “A Hybrid Genetic Algorithm for the Quadratic Assignment Problem on Graphics Processing Units,” *ANADOLU UNIVERSITY JOURNAL OF SCIENCE AND TECHNOLOGY –A Applied Sciences and Engineering,* vol. 17, no. 1, pp. 167-180, 2016.