# Parallel Improved Genetic Algorithm for the Quadratic Assignment Problem

Huda Alfaifi[1]
College of Computer & Information Sciences
Al-Imam Mohammad Ibn Saud Islamic University
Dept. of Computer Science, Riyadh, Saudi Arabia

Yassine Daadaa[2]
College of Computer & Information Sciences
Al-Imam Mohammad Ibn Saud Islamic University (IMSIU)
Riyadh, Saudi Arabia

***Abstract*—Quadratic Assignment Problem is one of the most common combinatorial optimization problems that represents many real-life problems. Many techniques are applied to solve Quadratic Assignment Problem, these include exact, heuristic, and metaheuristic methods. A Genetic Algorithm is a powerful heuristic approach used to find optimal solutions or near-to-optimal for Quadratic Assignment problems. In this paper, we developed a Genetic Algorithm with a new crossover operator with new technology closer to that found in nature without a crossover point and a new suggested intelligent mutation operator, then we developed a Parallel Genetic Algorithm using the same crossover and mutation. The sequential Genetic Algorithm will be implemented in the Central Processing Unit (CPU), and the Parallel Genetic Algorithm will be implemented in the Graphical Processing Unit (GPU). This paper presents two comparisons, first calculates elapsed time for crossover, mutation, and selection in both CPU and GPU, then compares the results. This comparison clearly shows the enhancement degree of computation time in the parallel environment, which is around half the time executed in the sequential environment. The second comparison, iterates these operators into several generations, using twenty benchmark instances reported in Quadratic Assignment Problem Library with sizes from (12-70), population size equal to 600, the number of generations equal to 2000, and the maximum number of parallel threads will be 600. Proposed crossover and mutation give the optimal solutions with ten benchmarks with problem sizes from 12 to 32 in both Sequential Genetic Algorithm and Parallel Genetic Algorithm, the next ten benchmarks give solutions closed to the optimal solution with a small error rate.**

***Keywords*—Component; Quadratic Assignment Problem (QAP); Genetic Algorithm (GA); Parallel Genetic Algorithm (PGA); Sequential Genetic Algorithm (SGA); Central Processing Unit (CPU); Compute Unified Device Architecture (CUDA); Quadratic Assignment Problem Library (QAPLIB); Best Known Solution (BKS); Average Percent Deviation (APD)***

## I. INTRODUCTION

The Quadratic Assignment Problem (QAP) is one of the most common combinatorial optimization problems that represents many real-life problems. The QAP involves the assignment of n facilities that have flows (weights) among them to n possible locations that also have distances among them to achieve the minimum sum of the distances multiplied by flows, this minimum sum will be reached by assigning high facilities to nearby locations and small facilities to far locations. The problem was first introduced as a mathematical model for economic activities in 1957[1], then it was becoming a fundamental and important problem to represent several applications in different areas, such as computer backboard wiring, locating clinics with a hospital, locating machine and electronic components, assignment of buildings in a university campus, etc.

The quadratic assignment problem (QAP) consists of n facilities and n possible locations, exactly one facility for each location. For each pair of facilities, a flow matrix, F = $[f_{ij}]$ is defined, which consists of flow values that must be required to move from facility i to facility j. Also, for each pair of locations, a distance matrix, D = $[d_{kl}]$ is defend and it consists of distance values between location k to location l. The assignment of facility i is not independent of other assignments, so when assigning facility i to location k we must consider the assignment for all other facilities that have nonzero relationships with facility i. Let $a = \{a(1), a(2), ....., a(n)\}$ be an assignment, where a(i) represents the location of the facility i. The problem is to assign to each location exactly one facility to minimize the cost of the objective function as shown in Equation. 1.

$$Z_a = \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{a(i)a(j)} \quad (1)$$

Since the solution is derived from n! possible assignments, it makes the problem impossible to solve in polynomial time with moderate problem size, even with modern computers.

The QAP solving methods can be categorized into three main classifications: exact methods, heuristic methods, and meta-heuristic methods. The exact methods give the exact optimal solution, but the drawback of such methods is the long computational time that makes the solution impossible. Therefore, the problem was restored to be solved using heuristics and meta-heuristic methods which overcome the problem of long computational time, but they also have their drawback. Heuristics and meta-heuristic methods do not guarantee to provide the exact optimal solution, but they instead provide a good solution, near to optimal solution, in reasonable computational time. Genetic algorithms, simulated annealing, tabu search, artificial neural network, etc., are some well-known heuristic methods, and genetic algorithm is considered as one of the best heuristic methods.

A Genetic Algorithm (GA) provides individual candidate solutions that do not hold any dependencies between them, so, it will be easy to implement such an algorithm in parallel to get a more considerable speedup.

This paper uses a parallelism concept which in turn becomes an effective way to simplify the difficult problems and reduce its computational time. Additionally, GA is a popular effective heuristic approach in both computation time and solution quality. So, they have motivated us to take the advantage of both GA and parallelism to solve that difficult problem.

This work exploits the recent improvement in the graphical processing unit (GPU) which is expanded to include parallel computation rather than just graphical purpose. So, we will propose a solution for QAP using a proposed genetic algorithm with enhancement in crossover and mutation. These enhancements are suggested new crossover operator with new technology which closer to that found in nature without crossover point and new intelligent mutation operator which in turn improve solution quality.

## II. Background and Related Work

Genetic algorithms were first invented on QAP by John Holland at the University of Michigan in 1975[2]. The first applied for GA in QAP was in 1994 by Fleurent and Ferland[3]. GA is considered as a type of stochastic and local search technique, which are based on three natural operators: selection, crossover, and mutation. Also, there are many recent efficient algorithms, we will present a brief study about them to explore the new techniques and take advantage of them.

Radomil Matousek et al [4] presented Metaheuristic Optimization Using HC12 Algorithm. It is categorized as a parallel algorithm implemented on GPU. It used HC12 which is a Genetic Algorithm using binary encoding which depends on the next population is a population from the current solution neighborhood. This algorithm gives the optimal solutions for 8 problems with sizes (12 -32) in a short run time of an average of 1.89 seconds.

Takeshi Okano et al [5] proposed variant k-opt local search (vKLS) which is categorized as a sequential algorithm in a CPU environment, vKLS used a variable depth approach that depends on exchanging multiple nodes at a time rather than just two nodes. They combine two strategies best-improvement move and the first-improvement move. vKLS tested on 48 QAPLIB instances with a range of 20 - 150 in a fixed period equal to 60 seconds.

Ensieh et al improved the performance of the (NIFLS) Fast Local Search algorithm in the sequential environment by adding Temperature characteristics from simulated annealing to conduct the search to explore the search space wider[6]. The algorithm gets 0.26 APD in average execution time 1207 seconds.

Erdener et al developed ILS (Iterated Local Search) algorithm using GPU parallelism[7]. They implement the multi-start technique, use the delta function instead of calculating object function for each neighbor and design a mutation operator to escape the local optimum. The algorithm works 6.31 to 11.93 times faster than sequentially one.

Omar Abdelkaf et al. [8] suggested Parallel iterative Tabu Search (PITS) by parallelizing an existing TS algorithm called Ro-Ts using a grid of 5000 CPUs. PITS works with 350 iterations inside the process, 100 global iterations, and 40 processes. PITS gives an average standard deviation equal to 12.19 in average time equal to 13.01 minutes with problems with size 343.

Also Emrullah et al presented an algorithm called the Parallel Simulated Annealing method with multi-start technique (PMSA) using GPU parallelism[8]. PMSA starts the next SA algorithm with the best previous generated value rather than a random permutation, this technique is called the multi-start approach. It provides the optimal solution for 196 instances except for 14 instances in time less than 60 seconds.

Lopez et al presented GA-CPLS algorithm which is a type of CPU level parallelism[9]. CPLS operation depends on a group of nodes called explorers. GA-CPLS performed the Genetic algorithm as the main explorer to generate the population as a head node, other explorer nodes execute the Extremal Optimization Algorithm and robust Tabu search. GA-CPLS gives 0.054 APD on an average time of 82.7 minutes.

Seyda et al improved sequential Hybrid GA called IHGA[10]. Its idea takes from combining genetic algorithm, simulated annealing algorithm, and the greedy algorithm. It enhances the solution by 13.33, 7.94, 2.50, and 0.29 percent better than the greedy algorithm, DA, classical GA, and SA respectively.

Soukaina et al developed a Hybrid Chicken Swarm Optimization (HCSO)[11]. HCSO applies GPU level parallelism and integrates Chicken Swarm Optimization CSO with Greedy Randomized Adaptive Search Procedure GRASP. GRASP run with a 2-opt Local Search for constructing the initial population. HCSO finds the optimal solution for 85% of 30 QAP instances.

Mohamed et al enhanced Whales Optimization Algorithm by integrating it with Tabu Search (WAITS)[12]. WAITS was applied in a sequential environment, and it enhances the speed of convergence and local search inside the Whales Algorithm (WA). WAITS provides the optimal solutions for 86 instances out of 122 instances.

Previous studies explored many recent heuristics and metaheuristics algorithms in solving QAP either in parallel or in a sequential environment. Parallelism can be designed at the CPU level or GPU level. As we see from reviewed algorithms, parallel algorithms designed by GPU produced better results in computational time and algorithms like GA will provide a high-quality solution in a reasonable time. This will motivate us to design a new GA with a new crossover operator with new technology closer to that found in nature, it depends on arranging genes in a specific way without the need for a crossover point, and also suggested an intelligent mutation operator in the GPU environment.

The proposed method will be implemented and tested in a sequential environment and then in parallel to compare results and to show the degree of parallel improvement using benchmark instances available in QAPLIB[13].

This paper was organized into sections, each section treats a part of our works. The second section shows the methodology of our works, the next section illustrates the overall structure of the proposed algorithm, the fourth section analyzes and explores the results, and finally the conclusion.

## III. METHODOLOGY

### A. Population Initialization Method

Population sets will be initialized randomly concerning the problem size. Additionally, make sure this population does not have incomplete or invalid individuals and all nodes are existing and forming a complete solution. Also, be sure the individual does not have redundant nodes or invalid nodes.

### B. Selection Method

The proposed GA applied the selection to two places in the algorithm. First, parents' selection is called the stochastic remainder selection method. It works by assigning a probability to every individual to be chosen as a parent. This method takes each individual's fitness then divides it by average fitness, the integer part of the division represents the number of appearances of the individual as a parent, and the remaining fractional part is used to stochastically fill the remaining parents to stochastic places.

The second application of selection was after crossover operation when deciding about if a current parent will stay for the next generation or be replaced by its best offspring. This type of survivor selection is called the steady-state approach.

### C. Crossover Operator

In this paper, we propose a new crossover method that produces an individual who inherits from parent's characteristics as much as possible. This method will preserve the order of the inherited nodes from both parents without making a crossover point.

The following example will illustrate the proposed crossover method by using the facility matrix and distance matrix that is used in the "Hud12" benchmark. If we have two parents parent1 with cost = 1956 and parent2 with cost = 1936 each with size 12, as shown in Fig. 1 and Fig. 2, and offspring will be as shown in Fig. 3.

Parent1:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 4 | 12 | 6 | 10 | 9 | 7 | 1 | 8 | 3 | 11 | 2 |

Fig. 1. Crossover _ parent1

Parent2:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 6 | 9 | 2 | 4 | 11 | 10 | 1 | 5 | 8 | 7 | 3 |

Fig. 2. Crossover _ parent2.

Offspring:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 4 | 12 | 6 | 10 | 9 | 2 | 11 | 1 | 8 | 7 | 3 |

Fig. 3. Crossover_offspring.

There are two indexes (index1= 0) which point to the first index in parent1, (index2=size-1=11) which point to the last index in parent2. Start filling offspring by these two indexes, at the same time, as shown in Fig. 4.

Step1: index1=0, index 2=11, offspring will be:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | | | | | | | | | | | 3 |

Fig. 4. Crossover First Step.

5 is the first node in parent1, 3 is the last node in parent2, increment index 1, decrement index2, index1=1, index2=10.

Step2: index1=1, index2=10, before inserting must check if the new node exists in new offspring if not just insert it, if exist go to the next node in the corresponding parent, offspring will be , as shown in Fig. 5.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 4 | | | | | | | | | 7 | 3 |

Fig. 5. Crossover Second Step.

4 is the second node in parent1, 7 is the second node from the last in parent2, increment index 1, decrement index2, index1=2, index2=9.

Step3: index1=2, index 2= 9, before inserting must check if the new node exists in new offspring if not just insert it, if exist go to the next node in the corresponding parent, offspring will be as shown in Fig. 6:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 4 | 12 | | | | | | | 8 | 7 | 3 |

Fig. 6. Crossover Third Step.

12 is the third node in parent1, 8 is the third node from the last in parent2, increment index1, decrement index2, index1=3, index2=8.

Step4: index1=3, index 2= 8, before inserting must check if a new node exists in new offspring if not just insert it, if exist go to the next node in the corresponding parent, offspring will be , as shown in Fig. 7.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 4 | 12 | 6 | | | | | 1 | 8 | 7 | 3 |

Fig. 7. Crossover Fourth Step.

6 is the fourth node in parent1, 5 is the fourth node from the last in parent2 but 5 exists in offspring, so go to the fifth node from the last in parent2 which is 1 then check if doesn't exist in offspring insert 1, increment index1, decrement index2, index1=4, index2=7.

Step5: index1=4, index 2= 7, before inserting must check if the new node exists in new offspring if not just insert it, if exists go to the next node in the corresponding parent, offspring will be as shown in Fig. 8.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 4 | 12 | 6 | 10 | | | 11 | 1 | 8 | 7 | 3 |

Fig. 8. Crossover Fifth Step.

Ten (10) is the fifth node in parent1, 10 is the sixth node from the last in parent2 but 10 exists in offspring, so go to the

seventh node from the last in parent2 which is 11 then check if does not exist in offspring insert 11, increment index 1, decrement index2, index1=5, index2=6.

Step 6: index1=5, index 2= 6, before inserting must check if the new node exists in new offspring if not just insert it, if exist go to the next node in the corresponding parent, offspring will appear as shown in Fig. 9.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 4 | 12 | 6 | 10 | 9 | 2 | 11 | 1 | 8 | 7 | 3 |

Fig. 9. Crossover Sixth Step.

Nine (9) is the sixth node in parent1, 4 is the eighth node from the last in parent2 but 14 exists in offspring, so go to the ninth node from the last in parent2 which is 2 then check if doesn't exist in offspring insert 2. The cost for the generated offspring = 1868 which is better than the cost of parents.

Crossover must be simple as possible to achieve maximum utilization of GPU benefits. The generated offspring was produced by simple crossover but inherit many features from parents selected by a strong selection method.

### *D. Mutation Operators*

The proposed GA uses a new mutation operator that works as scanning the individual to find the maximum product (flow * distance) located between facility(i) to the facility (i+1). Then swap facility (i+1) with random node from the individual.

This proposed mutation can be illustrated as shown in the following example: The following individual belongs to the "Had12" benchmark with cost = 1902, as shown in Fig. 10.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 6 | 3 | 1 | 8 | 2 | 9 | 10 | 7 | 12 | 5 | 11 |

Fig. 10. Individual before Mutation.

After applying the mutation operator, the cost will be = 1834, and the individual will be as shown in Fig. 11.

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 6 | 3 | 1 | 7 | 2 | 9 | 10 | 8 | 12 | 5 | 11 |

Fig. 11. Individual after Mutation.

## IV. Structure of the Proposed Parallel Genetic Algorithm

The following algorithm shows the general structure of the proposed PGA, followed by a system diagram to represent the PGA structure, as shown in Fig. 12.

PGA exploited graphical processing unit (GPU) for non-graphical parallel computation, the proposed algorithm uses a large single population of individuals which is distributed among several threads in GPU. Each thread performs three GA operators Crossover, Mutation Survivor, and Selection because they are suitable to implement in the parallel environment as shown in Fig. 12. This means, does not need to force threads to communicate between each other or lock other threads, or wait for other threads until unlocking, this parallelism technique maintains data integrity and consistency also threads' waiting time is almost non-existent. The proposed algorithm was shown in Table I.

TABLE I. Algorithm1

**Algorithm1: proposed Parallel Genetic Algorithm**

*Population Size= N*

*Problem Size = n*

*Number of threads =N*

*Termination condition=2000 generation*

**Create** *random Population with size N*

**while** termination condition **is not reached do**

**calculate** fitness values

**Reorder** individuals according to stochastic probability

**For each** thread **i , in parallel do**

**Select** parent 1 =individual i

**Select** parent 2 = individual i+1,

**For each** facility j in **offspring**

offspring = *Assign facility j from left of individual i*

offspring =*Assign facility n - j+1 right of individual i+1*

**Find** maximum product (flow * distance) between facility(k) to facility (k+1) **in** individual i

offspring = **swap** *facility (k+1) with random picked facility.*

**If** offspring cost < parent 1 cost

**Replace** (individual i = offspring)

**end while**

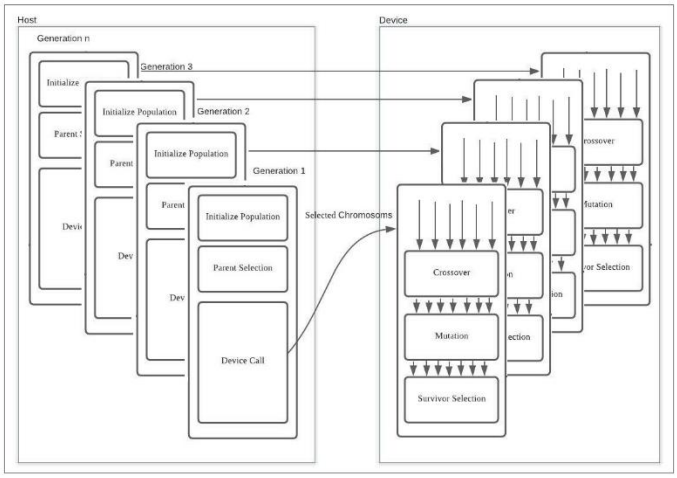


Fig. 12. Overall Proposed PGA Structure.

## V. Result and Discussion

This section will show an analysis, discussion, and illustration of the output and results of the proposed method in this paper, results are going to be analyzed in two ways. The

first analysis will present six tables that show elapsed time for GPU and CPU during the execution of proposed crossover, mutation, and selection. The second analysis presents a test of the proposed method in GPU and CPU after embedding it inside several iterations (generations).

The proposed method was tested in CPU of type intel® core™ i7-8565U CPU @ 1.80GHz (8 CPUs) and GPU of kind NVIDIA GeForce MX250 using both CUDA (Compute Unified Device Architecture) and C++ programming languages.

The following four figures show a comparison between CPU and GPU using common QAP benchmarks, while N means the size of population, CPU and GPU time is measured in milliseconds.

### A. First Test Illustration

This test shows elapsed time for GPU and CPU during the execution of proposed crossover, mutation, and selection.

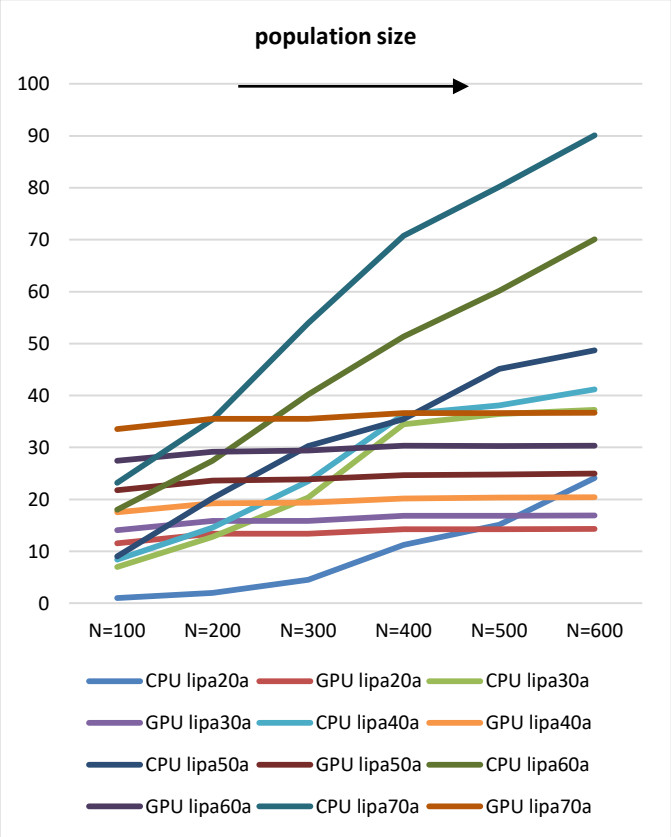


Fig. 13. CPU and GPU Time Illustration with Problem Sizes.

*1) For the "lipa20a" benchmark:* "lipa20a" benchmark with problem size equal to 20. We notice that when population size equal to 100, 200, 300, and 400 CPU show better results than GPU. Here the problem size is small, and we will only see the enhancement in GPU when the size of the problem and population increase. After increasing the population size to 600 we will observe the GPU enhancement and CPU time become approximately twice the time of GPU. Fig. 13 shows a graphical representation of this problem.

*2) For the "lipa30a"* benchmark: The enhancement on GPU begins at N=300, then the CPU time will take an increasing rate when population size increases. Compared to GPU time, GPU time does not take a significantly increasing rate while the population size increases, it just took a small increasing rate $\approx$ of 0.56 milliseconds, as shown in Figure IV.1. CPU continues increasing until it reaches more than 2x time of GPU time at N= 600. Fig. 13 Shows a graphical representation of this problem.

*3) For the "lipa40a"* benchmark: The improvement on GPU starts when N=300, then the CPU time will increase when population size increases until it reaches nearly 2x the time of GPU at N= 600. On the other hands, GPU time takes a small increasing rate $\approx$ of 0.58 while the population size increases. Fig. 13 shows a graphical representation of this problem.

*4) For the "lipa50a"* benchmark: shows the same result as "lipa40". The CPU looks better than GPU when N=300, but after that, it becomes worse when N>300. CPU becomes around 2x time of GPU at N= 600. As we noted earlier GPU time is not affected much by population increase, as shown in Fig. 13.

*5) For "lipa60a" and "lipa70a" benchmarks:* CPU looks worse than GPU when population size > 200, it becomes around 2.3x time of GPU at problem size =60 and N= 600 and it becomes around 2.5x time of GPU at problem size = 70 and population size N= 600, as shown in Fig. 13.

*6) Fig. 14:* shows the worst CPU time state when population size = 600, it becomes around 2x GPU time and it increases at a significant rate when the problem size increases.

### B. Second Test

Table II presents a test set using a group of common QAP benchmarks in size range (12 - 70) and population size N =600. After inserting the proposed tested method into the whole Genetic Algorithm program, we will show their elapsed CPU and GPU time after 2000 generations then show the best solution found in both CPU and GPU. Also, measure Average Percent Deviation to measure how much the solution is closed to best known solution (BKS), as shown in Equation. 2.

$$APD = \frac{\text{solution} - \text{BKS}}{\text{BKS}} \quad (2)$$

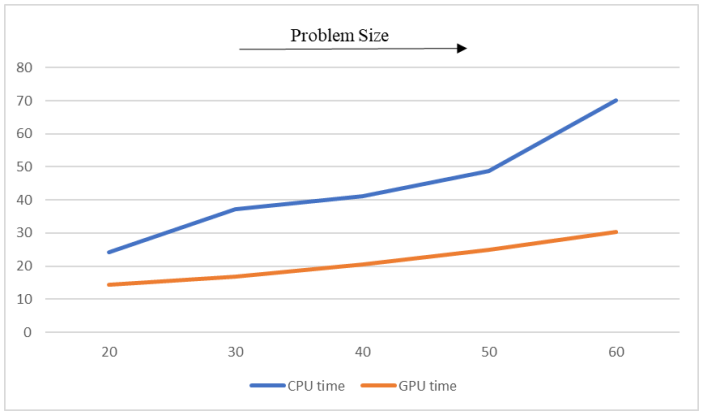


Fig. 14. CPU and GPU Time when Increased Problem Size and N=600.

TABLE II. SOLUTIONS AND ELAPSED TIME FOR CPU AND GPU

| Problem Instances | Avg. Whole CPU program time | Avg. Whole GPU program time | CPU solution | GPU solution | Optimal solution | CPU APD | GPU APD |
|---|---|---|---|---|---|---|---|
| Esc16a | 13112.34 | 22242.32 | 68 | 68 | 68 | 0 | 0 |
| Esc16b | 7565.65 | 22133.45 | 292 | 292 | 292 | 0 | 0 |
| Esc16c | 8301.25 | 21650.29 | 160 | 160 | 160 | 0 | 0 |
| Esc16d | 8986.33 | 21722.73 | 16 | 16 | 16 | 0 | 0 |
| Esc16e | 8227.89 | 22620.11 | 28 | 28 | 28 | 0 | 0 |
| Esc16g | 17005.75 | 21929.62 | 26 | 26 | 26 | 0 | 0 |
| Esc32g | 29679.09 | 31093.71 | 6 | 6 | 6 | 0 | 0 |
| Had12 | 4577.27 | 19788.06 | 1652 | 1652 | 1652 | 0 | 0 |
| Had14 | 5973.67 | 20853.74 | 2724 | 2724 | 2724 | 0 | 0 |
| Had16 | 14555.79 | 21911.14 | 3720 | 3726 | 3720 | 0 | 0 |
| Lipa20a | 11250.61 | 24875.72 | 3797 | 3809 | 3683 | 0.03 | 0.03 |
| Lipa30a | 51688.26 | 34824.34 | 13555 | 13565 | 13178 | 0.03 | 0.03 |
| Lipa40a | 60049.76 | 48419.51 | 32340 | 32346 | 31538 | 0.03 | 0.03 |
| Lipa50a | 70703.40 | 65243.23 | 63476 | 63557 | 62093 | 0.02 | 0.02 |
| Lipa60a | 142518.61 | 85710.54 | 109388 | 109429 | 107218 | 0.02 | 0.02 |
| Lipa70a | 133348.29 | 110122.11 | 172941 | 172894 | 169755 | 0.02 | 0.02 |
| Sko42 | 95743.75 | 51448.87 | 16900 | 17286 | 15812 | 0.07 | 0.09 |
| Sko49 | 124436.57 | 62790.45 | 26719 | 27317 | 23386 | 0.14 | 0.17 |
| Sko56 | 176624.76 | 77348.33 | 37858 | 38412 | 34458 | 0.1 | 0.11 |
| Sko64 | 215862.65 | 98184.77 | 53072 | 53504 | 45736 | 0.16 | 0.17 |

The proposed crossover and mutation give the optimal solution with the first ten benchmarks with problem sizes from 12 to 32 in both sequential Genetic Algorithm and Parallel Genetic Algorithm, next ten benchmarks give a solution close to the optimal solution with a small error rate, CPU and GPU time are measured in milliseconds.

## VI. CONCLUSION

After this study, we found that the GPU time is not affected much by increasing either population size or problem size compared to CPU time. GPU time was increased by a small rate $\approx$ of 0.61 when increasing population size and take a small rate $\approx$ of 4.5 when the size of the problem increases. Also, the CPU time shows its worst when the population size = 600, it becomes around 2x GPU time, and it increases at a significant rate when the problem size increases.

This paper concentrates on applying proposed GA to QAP, which in turn, gives a successful result in finding optimal solutions or solutions near to optimal.

Also, this paper applies proposed GA in the parallel environment which shows a good result in execution time enhancement. As mentioned before, the proposed solution uses a large population size; therefore, a lot of synchronous threads will be needed. So, as future works, we can increase the number of threads by increasing the size of the screen card (NVIDIA). Furthermore, the proposed PGA can be generalized to cover other optimization problems such as TSP (traveling salesman problem) or VRP (Vehicle routing problem).

As a future work we try to enhance some drawbacks that make algorithm slower such as sequential parent selection, it needs to convert to be work in parallel environment.

## REFERENCES

[1] T. Koopmans and M. Beckmann, “Assignment problems and the location of economic activities,” Econometrica: Journal of the Econometric Society, vol. 25, no. 1. pp. 53–76, 1957, doi: 10.2307/1907742.

[2] M. Melanie, “An Introduction to Genetic Algorithms,” Cambridge, Massachusetts London, England, Fifth printing, 3, pp. 62–75, 1999.

[3] C. Fleurent and J. Ferland, “Genetic hybrids for the quadratic assignment problem,” Anon. DIMACS Ser. Discret. Math. Theor. Comput. Sci., vol. 16, pp. 173–187, 1994.

[4] R. Matousek, L. Dobrovsky, and J. Kudela, “The quadratic assignment problem: Metaheuristic optimization using HC12 algorithm,” GECCO 2019 Companion - Proc. 2019 Genet. Evol. Comput. Conf. Companion, pp. 153–154, 2019, doi: 10.1145/3319619.3322088.

[5] T. Okano, K. Katayama, K. Kanahara, and N. Nishihara, “A Local Search Based on Variant Variable Depth Search for the Quadratic Assignment Problem,” 2018 IEEE 7th Glob. Conf. Consum. Electron. GCCE 2018, pp. 390–391, 2018, doi: 10.1109/GCCE.2018.8574497.

[6] E. Mohassesian and B. Karasfi, “A new method for improving the performance of fast local search in solving QAP for optimal exploration of state space,” 7th Conf. Artif. Intell. Robot. IRANOPEN 2017, pp. 64–72, 2017, doi: 10.1109/RIOS.2017.7956445.

[7] E. Ozcetin and G. Ozturk, “A Parallel Iterated Local Search Algorithm on GPUs for Quadratic Assignment Problem,” vol. 4, no. 2, pp. 123–128, 2018.

[8] O. Abdelkafi, B. Derbel, and A. Liefooghe, "A Parallel Tabu Search for the Large-scale Quadratic Assignment Problem," 2019 IEEE Congr. Evol. Comput. CEC 2019 - Proc., pp. 3070–3077, 2019, doi: 10.1109/CEC.2019.8790152.

[9] J. Lopez, D. Munera, D. Diaz, and S. Abreu, "On integrating population-based metaheuristics with cooperative parallelism," Proc. - 2018 IEEE 32nd Int. Parallel Distrib. Process. Symp. Work. IPDPSW 2018, pp. 601–608, 2018, doi: 10.1109/IPDPSW.2018.00100.

[10] S. M. Turkkahraman and D. Oz, "An Improved Hybrid Genetic Algorithm for the Quadratic Assignment Problem," pp. 86–91, 2021, doi: 10.1109/ubmk52708.2021.9558978.

[11] S. C. Bourki Semlali, M. Essaid Riffi, and F. Chebihi, "Hybrid chicken swarm optimization with a GRASP constructive procedure using multi-Threads to solve the quadratic assignment problem," Int. Conf. Multimed. Comput. Syst. -Proceedings, vol. 2018-May, 2018, doi: 10.1109/ICMCS.2018.8525992.

[12] M. Abdel-Basset, G. Manogaran, D. El-Shahat, and S. Mirjalili, "Integrating the whale algorithm with Tabu search for quadratic assignment problem: A new approach for locating hospital departments," Appl. Soft Comput. J., vol. 73, pp. 530–546, 2018, doi: 10.1016/j.asoc.2018.08.047.

[13] R. E. Burkard, E. Çela, S. E. Karisch, and F. Rendl, "QAPLIB - A Quadratic Assignment Problem Library," J. Glob. Optim., vol. 10, no. 4, pp. 391–403, 1997.