Journal of Engineering Science and Technology
Vol. 15, No. 5 (2020) 3082 - 3095

HYBRID OF CELLULAR PARALLEL GENETIC ALGORITHM AND GREEDY 2-OPT LOCAL SEARCH TO SOLVE QUADRATIC ASSIGNMENT PROBLEM USING CUDA

ROBERTO POVEDA[1,*], EDUARDO CARDENAS[2], ORLANDO GARCIA[1]

[1]Facultad de Ingeniería, Universidad Distrital "Francisco José de Caldas" Bogotá, Colombia
2Departamento de Matemáticas, Universidad Nacional de Colombia, Bogotá, Colombia
*Corresponding Author: rpoveda@udistrital.edu.co

Abstract

This article presents an implementation of a cellular parallel genetic algorithm and a local optimization heuristic to solve the Quadratic Assignment Problem (QAP). First, we implemented a cellular model that combines the most representative characteristics of the population in the Genetic Algorithm, later, we resort to a greedy 2- opt heuristic optimization which performs a meticulous genetic exploitation of the spaces previously explored by the Genetic Algorithm. Our algorithm was completely implemented with CUDA on a Graphical Processing Unit (GPU), where a GPU grid represents the population of the Genetic Algorithm (GA), a GPU block represents each particular individual (chromosome) of the population, and each GPU thread represents a gene of such a chromosome. The problems examined correspond to prominent instances of the QAPLIB library. Our algorithm solves these problems efficiently.

Keywords: Greedy 2-opt local search heuristics, Graphical processing unit (GPU), Parallel genetic algorithm, Quadratic assignment problem (QAP).

1. Introduction

The QAP consists in assigning, one by one, a set of n facilities in a set of n locations so as to minimize the flow between the facilities and the distances between the locations. The QAP is considered a strongly NP-Hard problem [1] and it is a general model of other NP-Hard combinatorial problems [2].

Several methods have been used to resolve QAP- instances; Branch & Bound is perhaps the most outstanding exact method [2, 3], but it is inappropriate in large-size QAP-instances. An alternative in the solution of the QAP are methods like metaheuristics based on population - Ant Colony Optimization (ACO) [4], Particle Swarm Optimization (PSO) [5], Genetic Algorithms (GA) [6], among others - generally combined with meta heuristics based on trajectories - local search techniques (LS) [7, 8], Tabu Search (TS) [9, 10], Simulated Annealing (SA) [11], among others. Parallel models of the previous metaheuristics have been more efficient in the QAP-instances solution [12, 13]. These models have been implemented in recent years on multicore architectures or GPUs, significantly reducing the execution time [14-16]. The CUDA computing platform is the most popular development tool nowadays to program on GPUs. CUDA is a C language with SIMD extensions for programming without taking into account graphic programming concepts, which were necessary with interfaces such as DirectX or OpenGL [17].

The performance of our algorithm is measured by executing some reference instances of different sizes present in the standard QAPLIB library. This library includes information on such instances (distance and flow matrices) as well as the best known solution to date [18].

This article is organized as follows: Chapter 2 presents a related works. Chapter 3 presents a background on QAP, Parallel Genetic Algorithms, Local Search Heuristics (greedy 2-opt), and GPUs. Chapter 4 presents our algorithm. Chapter 5 explains the experiments and the results obtained. Finally, Chapter 6 highlights conclusions and future works.

2. Related Work

Mohassesian and Karasfi [16] present an algorithm with a balanced dispersion of the solutions in the search space of the problem. They also perform 10 executions on each problem, each with two 220 iterations. The algorithm is executed on CPU.

Tsutsui [19] implemented ACO on a PC with a 4 GPUs array. An islands model, where each colony resides in a GPU and individuals are interchanged through the CPU following 4 different procedures, is followed. All the algorithm data is localized in the global memory of the GPU, but the flow and distance matrices are located in the texture memory. For the first procedure, in the islands model, two independent executions are made, when an ACO in a GPU finds an acceptable solution, then the algorithm ends, there is no interchange of information. The second procedure is elitist, the best solution found from the 4 GPUs is distributed to the remaining that did not obtain that solution and replaces the worst. The third uses an annealing model, the best solution in the GPU g replaces the worst solution in the GPU (g+1)mod4. The last procedure is elitism with massive annealing connection, the best solution of each GPU/island, with some other solutions are distributed to the other 3 GPUs. The best results are obtained with the last procedure. They also develop a Master-Slave

implementation with only one colony of m individuals (m varies between 1 and 150) runs on the CPU and distributes $m/4$ individuals in each one of the GPUS. A Tabu Search is carried out on each GPU.

Luong et al. [14] based their work on three approaches: the first is to distribute, in an efficient manner, a Local Search process between GPU and CPU. The second, consist in making an adequate relation between a neighbour of one current solution and each GPU thread. The third, to efficiently optimize the data transfer between GPU memory hierarchies, handling their capacity limitations. The CPU-GPU communication is performed through the VRAM of the GPU. Here the input data (flow and distance matrices) as well as the coding of the solutions of the problem are stored. Given that the global memory is slow, a new implementation is made on the texture memory as an optimization alternative for the storing and access to the problem's input data. They argue that an important way to drastically reduce the CPU-GPU data transfer (which represents the greatest obstacle in the GPU performance) can be achieved by moving the complete Local Search process to the GPU.

In another work, Tsutsui and Fujimoto [15] implement a coarse grain parallel genetic algorithm, where each sub-population is hosted in the shared memory of each block of threads and evolves in parallel on each one of these blocks (30 blocks of 128 threads were used). In a block, each individual is implemented as an independent thread. The QAP flow and distance matrices data are stored in the constant memory of the GPU. After a certain number of generations, individuals in the subpopulations are mixed in the host and then put in the GPU global memory (VRAM). Each multiprocessor selects an unprocessed block and copies the GPU individuals from the global memory to the shared memory, 500 generations are made and the evolved individuals are copied again to the VRAM, this process continues until all of the blocks are processed, then, all individuals are copied to the host memory and mixed. The process is repeated until a certain stop criterion. In this implementation there is no individuals interchange among blocks. The algorithm ends when one of those subpopulations finding an "acceptable" solution. The size of the QAP instances varies between 25 and 50.

Tsutsui and Fujimoto [20] proposed a parallel genetic algorithm with independent executions. The base of this evolutionary model for the QAP on GPU is exactly the same of the immediately previous one, except that here it is considered a restart strategy, that is, it "mutates" individuals with the best solution to avoid premature stagnation but leaving also a certain quantity of the best found up to the moment.

Chaparala et al. [21] implemented a 2-opt local search heuristic over the GPU, configuring different thread numbers per block.

Semlali et al. [22] presented a hybrid algorithm that combines chicken swarm optimization and Greedy randomized adaptive search procedures, in order to find a better initial population. The implementation was in GPU, and each instance was tested 20 times, each with 100 iterations.

Mohammadi et al. [23] proposed a parallel genetic algorithm on GPU. The authors combine the previous and current populations (the latter obtained by crossover and mutation) and obtain a new one, half of it through a deterministic fitness and the other half randomly. Each problem was executed 20 times. The authors do not report the number of iterations used in each execution.

Szwed et al. [24] implemented on OpenCL a massively parallel multi-swarm algorithm, which can be executed in independent swarms or with migrations. The authors highlight the fact that they can process large populations thanks to parallelism. They do not show algorithm execution times for all problems. Instances sc64a and tai60b were solved in 71 and 2220 iterations respectively - too many compared with our algorithm.

3. Background

3.1. Quadratic assignment problem (QAP)

The QAP is an outstanding problem of optimization that has served as a model to describe some problems of distribution and communication of scientific scope [2]. The QAP consists in assigning, one by one, a set of n facilities in a set of n locations so as to minimize the flow between the facilities and the distances between the locations.

The original formulation of the QAP consists in finding a permutation σ such that:

$$\min_{\sigma \in S_n} \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} f_{ij}\, d_{\sigma(i)\sigma(j)} \tag{1}$$

where $F = f_{ij}$ is a flow matrix, $D = d_{kl}$ is a distance matrix (both F and D have a n x n size), and $S_n = \{\sigma \mid \sigma : N \rightarrow N\}$, where $N = \{0,1, \ldots, n\text{-}1\}$ (it is often said that n is the QAP size). Each individual product $f_{ij}\, d_{\sigma(i)\sigma(j)}$ of the previous Eq. (1) is the cost of assigning facility $\sigma(i)$ to location i, and facility $\sigma(j)$ to location j.

Figure 1 shows an example of a QAP-instance of size $n = 5$. In the example of the Fig.1, $\sigma = (1,2,3,0)$ with

$$\text{cost}(\sigma) = \sum_{j=0}^{4} \sum_{i=0}^{4} f_{ij}\, d_{\sigma(i)\sigma(j)} = 222 \tag{2}$$

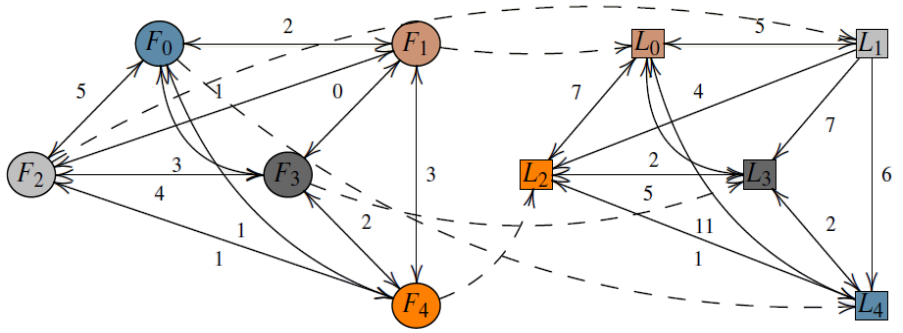

Fig. 1. Example of a QAP-instance of size *n* = 5.

Another equivalent formulation is the trace formulation; this consists in finding a permutation matrix X (associated with the permutation σ above) such that:

$$\min \text{trace}(FXD^tX^t) \tag{3}$$
$$\text{Subject to } \sum_{i=0}^{n-1} x_{ij} = 1,\ 0 \le j \le n-1$$

$$\sum_{j=0}^{n-1} x_{ij} = 1,\ 0 \leq i \leq n-1$$

$$x_{ij} \in \{0,1\},\quad 0 \leq i,j \leq n-1$$

where Eq. (3) is appropriate to be implemented in a vector device such as a GPU [25].

3.2. Parallel genetic algorithm

Parallel Genetic Algorithms in addition to significantly reducing the execution time in comparison with a simple genetic algorithm, consider each individual as a computationally independent unit and explore the search space for the problem more appropriately. There are several models of Parallel Genetic Algorithms, and they only differ in how individuals (or sub-populations) interact within a population, with the Master-Slave, Islands, and Cellular models being the best known [26, 27]. The latter is the model implemented in this work.

Cellular model. The individuals of the population are distributed one by one in each cell of a two-dimensional mesh. The fitness of each individual is simultaneously evaluated and the crossover genetic operator for each individual occurs locally within a previously defined neighbourhood. With this model, a diffusion of information between individuals from different regions of the search space of the problem can be performed.

3.3. Local search heuristics

These methods consist in finding an optimal solution in a neighbourhood of an initial solution. The solution found updates the initial solution and the process continues until no better solution is found.

Greedy 2-opt Local Search Heuristics. For the QAP, these heuristic orderly interchanges all pairs of components of an initial permutation (all possible facilities on each of the locations). As the permutation improves (permutation with better cost), this updates the previous permutation.

3.4. Graphic processing unit (GPU)

Graphic processing unit or GPU is a dedicated coprocessor. Due to its independent RAM, GPU has high parallel processing capacity. This unit can use the NVidia's unified computing device architecture (CUDA); it becomes a programming language that is an extension of C / C ++, whose procedures run in parallel following the SIMD programming paradigm in the Flynn's taxonomy [28]. The programming language CUDA has largely improved; it makes it possible to develop generic applications (GPGPU) [29] - **mobile** phones, tablets and laptops; Conquering Disease, Improving Diagnoses, and Scientific Innovation - thanks to numerous cores that a GPU offers. Genetic algorithms are inherently parallel in nature, so they are favourable to be implemented in GPU but considering the challenge of how to adequately handle the access to the device memory.

4. Proposed Approach

The approach presented in this paper is a cellular parallel genetic algorithm (exposed in its initial phase in [30]) improved from the use of the Mutation and

Transposition genetic operators and a Greedy 2-opt local optimization heuristic. Our algorithm was implemented completely on GPU.

In our implementation a permutation $\sigma = (\sigma(0), \sigma(1),\ldots, \sigma(n\text{-}1))$ of the QAP is individual (chromosome) of the Genetic Algorithm and corresponds to a GPU block; each component $\sigma(i)$, $0 \leq i \leq n\text{-}1$ it interpret that facility $\sigma(i)$ assigned to location i) is a gene on that chromosome and corresponds to a GPU thread. The population of the Genetic Algorithm is a GPU grid.

The initial population is size 64 and each individual is randomly generated simultaneously in each shared memory space of each GPU block in a 8×8 two-dimensional GPU grid; the size of the population is maintained throughout the algorithm. Figure 2 shows this configuration on the GPU.

The flow and distance matrices are stored in the constant memory space of the GPU in order to accelerate the calculations.

The evaluation of the individual fitness of the population was made with the trace formulation for the QAP. This formulation uses the multiprocessing features of the GPU better. Matrix products are calculated by using a two-dimensional GPU grid of size 8 x n^2 where each GPU block consists of n one-dimensional GPU threads. Each row of the GPU grid represents the linearized permutation matrix of each individual of the population. That is, each GPU block is a row of such matrices. These rows are housed in the corresponding shared memory spaces of each GPU block to accelerate operations.

For the implementation of the selection operator in GPU, we obtain a permuted population from the current population P, therefore, $\overline{P} = X * P$, where X is a permutation matrix associated with a random permutation of the set {0,1,…,63}.

The binary tournament is carried out simultaneously at the block level between the corresponding individuals of the populations P and $\overline{P}$. Individuals with better fitness make up an intermediate population.

The crossover is a Modified Order Crossover (MOX) [19]. For the implementation in GPU a permuted population is obtained in the same way as before, but in this case the permutation matrix X is derived from a permutation $\theta = (\theta(0), \theta(1),\ldots, \theta(63))$ the set {0,1,…,63} from a specific topology of neighbourhoods; $\theta(i)$ represents the couple with the best fitness in the neighbourhood of the individual i, $0 \leq i \leq 64$ It is here where the cellular parallel genetic model is applied. We consider two different neighbourhood topologies. These are:

- Topology 4*n* (or Von Neumann neighbourhood; see Fig. 3).
- Topology 8*n* (or Moore neighbourhood; see Fig. 4).

The meshes that represent these topologies are toroidal. The offspring is obtained between the corresponding individuals of each block of the current and permuted population. This procedure is per- formed for all individuals simultaneously. The offspring will only replace the individual of the current population in the corresponding block if their fit- ness is better or equal to the current individual's fitness (replacement of neutral mutants).

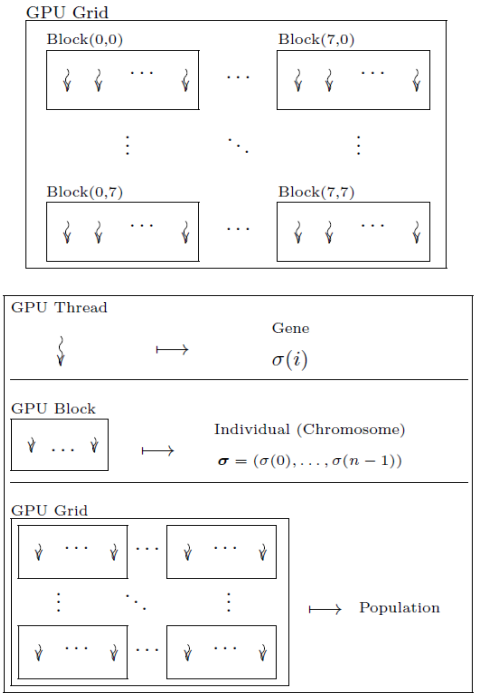

Fig. 2. Population in GPU.

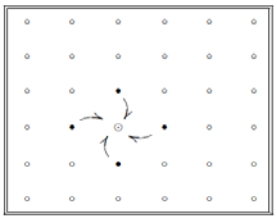

Fig. 3. Topology 4*n*.

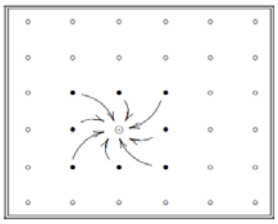

Fig. 4. Topology 8*n*.

The probability of crossover for each of the individuals is 0.6 throughout the algorithm. At this moment we identify the best individual of the current population, and then reincorporate it after applying the genetic mutation and transposition operators.

The mutation is an Exchange Mutation (EM) [19] and applied to each individual of the current population simultaneously (i.e., in each GPU block) with a probability of 0.01.

Transposition consists in inverting the genes of a substring (random) of the chromosome that represents each individual. This operator is also applied simultaneously at the block level and it has a probability of 0.4 for all individuals. The probabilities of mutation and transposition do not change throughout the algorithm.

The best individual previously identified is reincorporated into the population. This is done in order to avoid losing important genetic material obtained before

applying the mutation and transposition operators, i.e., our implementation develops a strategy based on elitism.

Now, to further exploit the regions explored by the Genetic Algorithm, we use a local optimization heuristic. Applied heuristics is a greedy 2-opt search.

For this heuristic, the current individual evaluates the $n(n-1)/2$ swaps (all pairwise exchanges of all possible facilities on each of the locations) Fig. 5 shows the case for a QAP-instance of size $n = 5$.

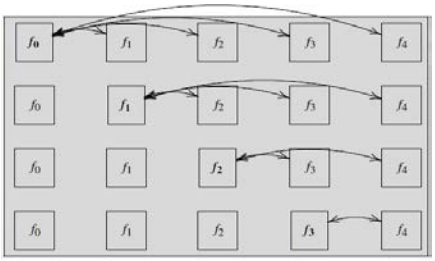

Fig. 5. 10 possible 2-opt swaps for a QAP of size $n = 5$. f_i is the *i-th* facility.

This heuristic leads the search more quickly to areas that have not been exploited, since immediately a better individual is found, this replaces the current individual (of course, fitness is also updated); and continues to be applied on the individual updated in the next swap; by contrast, with the basic 2-opt heuristic that replaces the current individual (and its fitness) with the best solution found only after making all possible exchanges.

The Greedy 2-opt local search takes advantage of the elitist parallel GA which prevents return to optimal local solutions already visited; that is, our cellular parallel GA is an efficient perturbation technique for Greedy 2-opt local search. Bashiri and Karimi in [31] compare several local search methods (2-opt, 2-opt Greedy, 3-opt and 3-opt Greedy) to solve the QAP. For them, the Greedy 2-opt local heuristic is the least efficient as the implementation lacks a perturbation technique.

This heuristic is implemented with a matrix type formulation (Eq. (4)) to take advantage of the benefits of the GPU.

$$\begin{aligned}\Delta_{ij} = {} & (f_{ij} - f_{ji})\left(d_{\sigma(i)\sigma(j)} - d_{\sigma(j)\sigma(i)}\right) \\ & + (F_{i\cdot} - F_{j\cdot}) \cdot \left((DX^t)_{\sigma(j)\cdot} - (DX^t)_{\sigma(i)\cdot}\right) \\ & + (F_{\cdot i} - F_{\cdot j}) \cdot \left((XD)_{\cdot\sigma(j)\cdot} - (XD)_{\cdot\sigma(i)}\right) \end{aligned} \quad (4)$$

where, Δ_{ij} compute the change in the fitness value after a pair-wise exchange (i, j facilities that are exchanged).

The current individual is updated with the first individual found such that $\Delta_{ij} < 0$. The greedy 2-opt heuristic continues on the individual updated in the next swap.

The operator $\cdot$ interprets an internal product, $f_{ij} = f_{ji} = 0$ in the second and third additions of the Eq. (3). $F_{k\cdot}$ is the row k of the matrix F, and $F_{\cdot k}$ is the column k of the matrix F.

This Eq. (3) is evaluated in $O(n)$ operations for all possible $O(n^2)$ swaps. The cost with Koopmans-Beckmann's original formulation (Eq. (1)) requires $O(n^2)$ operations. Algorithm 1 presents our model implemented completely on GPU.

Algorithm 1. Hybrid Parallel Algorithm implemented on GPU.

```
for each GPU block i in the GPU grid, in parallel do
   Assign a random individual
end for
generation number ← 1
while termination condition not met do
   for each individual i, in parallel do
      Fitness
      Select a different individual k
      if k is better than i then
         Assign k to i
      end if
      Select an individual k in the neighbouring of i
      Produce an offspring from i and k
      if the offspring is better than i then
         Assign the offspring to i
      end if
   end for
   To identify the best individual so far
   for each individual i, in parallel do
      Mutate
      Transpose
   end for
   To reincorporate the best individual
   for each individual i, in parallel do
      Apply greedy 2-opt local optimization heuristic
   end for
   generation number ← generation number + 1
end while
```

5. Experiments and Results Obtained

For testing purposes, a custom CUDA program was written. The algorithms were run on an Intel®Core™i7 - 4700HQ CPU @ 2.40GHz, RAM 8 GB and GPU NVidia GeForce GTX 760M. This device has 64 kB of constant memory, therefore the maximum size of QAP instances to consider is 90, because 2 x 90^2 x (4 bytes) ≤ 64 kB (two matrices of integers - flow and distance).

Ten different instances contained in the standard QAPLIB library [18] were examined, these are:

- Els19: “The data describe the distances of nineteen different facilities of a hospital and the flow of patients between those locations”. It is the only instance of this type of problems
- Esc64: “This example stem from an application in computer science, from the testing of self-testable sequential circuits. The amount of additional hardware

for the testing should be minimized". It is the second largest instance of this type of problems.

- Had20: "The distance matrix represents Manhattan distances of a connected cellular complex in the plane while the entries in the flow matrix are drawn uniformly from the interval $[1, n]$". It is the largest instance of this type of problems.
- Kra32: "The instances contain real world data and were used to plan the Klinikum Regensburg in Germany". It is the largest instance of this type of problems.
- Nug30: "The distance matrix contains Manhattan distances of rectangular grids. The solution was found by applying a branch and bound algorithm". It is the largest instance of this type of problems.
- Scr20: "The distances of these problems are rectangular". It is the largest instance of this type.
- Tai35b: "This problem is asymmetric and randomly generated". It is the sixth largest in- stance of this type of problems.
- Tai40b: "This problem is asymmetric and randomly generated". It is the fifth largest in- stance of this type of problem.
- Tai60b: "This problem is asymmetric and randomly generated". It is the third largest in- stance of this type of problem.
- Tho40: "The distances of this instance are rectangular". It is the second largest instance of this type of problems.

The number in the name of each instance indicates the size of the problem. Ten tests were done for each QAP-instance with respect to each of the two topologies cited in the previous section; each test consisted of one hundred iterations. The probability rates of the genetic operators were tuned to achieve the maximum performance of the algorithm.

To compare the results of our implementation in GPU, a sequential genetic algorithm was implemented in CPU with the same characteristics of our parallel algorithm. Of course, the implementation in CPU does not consider any topology of neighbourhoods; the crossover takes place from a random permutation. Ten tests were also done for each QAP-instance, each with one hundred iterations.

Table 1 shows the performance of our Hybrid Parallel Algorithm for problems Els19, Esc64a, and Had20, in relation to the number of iterations in which the optimal solution was found in the executions for each topology. Our Hybrid Parallel Algorithm always found the optimal solution for these three problems independent of these configurations.

Table 2 shows the performance of our Hybrid Parallel Algorithm (median and median absolute deviation) for the remaining seven instances - Kra32, Nug30, Scr20, Tai35b, Tai40b, Tai60b, and Tho40 - in relation to the solutions found in the executions for each topology.

Table 3 shows some results referenced by other researchers mentioned above on the QAP-instances considered.

Table 1. Performance of our Hybrid Parallel Algorithm in GPU versus performance of a CPU implementation.

QAP	*Topology 4n*	*Topology 8n*	*CPU*
Els19	5.5±1.5	6±2	7±3
Esc64a	1±0	1±0	1±0
Had20	4±1	3±1	6.5±2.5

Value $x \pm y$ indicates a median of x (iterations) with a median absolute deviation of y.

Table 2. Performance of our Hybrid Parallel Algorithm in GPU versus performance of a CPU implementation.

QAP	*Topology 4n*	*Topology 8n*	*CPU*
Kra32	88700±0	88700±0	88700±0
Nug30	6132±6	6128±0	6128±2
Scr20	110030±0	110030±0	110030±0
Tai35b	283315445±0	283315445±0	283725417±0
Tai40b	637250948±0	637250948±0	637307091±11766.5
Tai60b	608228619±13565	608228578±9632.5	608823261±124250
Tho40	241524±370	241130±381	242038±103

Value $x \pm y$ indicates a median of x (solution found) with a median absolute deviation of y.

Table 3. Results reported in the literature versus performance of our Hybrid Parallel Algorithm in GPU.

QAP	*Mohammadi et al. [23]*	*Szwed et al. [24]*	*Chaparala et al. [21]*	*Our Algorithm (Topology 8n)*
Els19	-	-	-	17212548
Esc64a	-	116	-	116
Had20	-	-	-	6922
Kra32	-	-	-	88700
Nug30	-	-	-	6128
Scr20	-	-	-	10030
Tai35b	284703248	-	283349722	283315445
Tai40b	647201580	-	637349459	637250948
Tai60b	624137807	612078720	609612341	608228578
Tho40	-	-	-	24130

6. Conclusions and Future Works

In this paper we proposed a hybrid of a cellular parallel genetic algorithm and a greedy 2-opt local search heuristic to solve large instances of the quadratic assignment problem. The algorithm was completely implemented on GPU, eliminating data transfers between GPU and CPU.

The fitness function of the genetic algorithm as well as the incremental function of the greedy 2-opt heuristic were formulated in a matrix fashion to take full advantage of the characteristics of the GPU as a multiprocessing vector device. In addition, an adequate handling of the memory spaces of the GPU was developed, in such a way that the mathematical calculations were made faster.

Not only was our hybrid algorithm in GPU faster than the sequential implementation in CPU, but also the results obtained were also much better. The use of a cellular parallel algorithm was significant in our implementation, highlighting Moore's topology over the Von Neumann's topology.

The greedy 2-opt local search heuristic was important to improve solutions previously found by the genetic algorithm. However, an optimization heuristic with a more rigorous mathematical character and less burden in an exhaustive search is proposed for future research, as was the case of this heuristic implemented.

Resorting to another parallel genetic model such as the distributed model (islands model) and combining it with what has already been implemented will surely improve the results obtained, and per- haps for this it is convenient to configure a cluster of GPUs or combine procedures in multicore architectures.

Nomenclatures

$cost(\sigma)$	Cost of permutation σ
d_{kl}	Component of distance matrix
D^t	Transposed matrix
f_{ij}	Component of flow matrix
$\min_{\sigma \in S_n}$	Least element in S_n
S_n	Permutations set
X	Permutation matrix
x_{ij}	Element in $\{0, 1\}$
Δ_{ij}	Change in the fitness value after a pair-wise exchange (i, j facilities that are exchanged)
$O(\cdot)$	Big O notation, Computational complexity

Greek Symbols

$\sigma(i)$	Facility in location i

Abbreviations

ACO	Ant Colony Optimization
CPU	Central Processing Unit
EM	Exchange Mutation
GA	Genetic Algorithm
GPU	Graphical Processing Unit
GPGPU	General-purpose computing on graphics processing units
LS	Local Search
MOX	Modified Order Crossover
PSO	Particle Swarm Optimization
QAP	Quadratic Assignment Problem
RAM	Random Access Memory
SA	Simulated Annealing
SIMD	Single Instruction - Multiple Data
TS	Tabu Search
VRAM	Video Random Access Memory

References

1. Sahni, S.; and Gonzalez, T. (1976). P-complete approximation problems. *Journal of the Association for Computing Machinery (ACM)*, 23, 555-565.
2. Burkard, R.E.; Cela, E.; Pardalos, P.M.; and Pitsoulis, L.S. (1998). The quadratic assignment problem. *Handbook of Combinatorial Optimization*, Boston, 3, 1713-1809.
3. Gilmore, P. (1962). Optimal and suboptimal algorithms for the quadratic assignment problem. *Journal of the Society for Industrial and Applied Mathematics (SIAM)*, 10 (2), 305-303.
4. Gambardella, L.; Taillard, E.; and Dorigo, M. (1999). Ant colonies for the quadratic assigment problem. *Journal of the Operational* Research *Society*, 50(2), 167-176.
5. Liu, H.; Abraham, A.; and Zhang, J. (2007). A particle swarm approach to quadratic assignment problems. *Soft Computing in Industrial Applications*, Berlin, 39, 213-222.
6. Lim, M.; Yuan, Y.; and Omatu, S. (2002). Extensive testing of a hybrid genetic algorithm for solving qudratic assignment problem. *Computational Optimization and Applications*, 23, 47-64.
7. Taillard, E.D. (1995). Comparision of iterative searches for the quadratic assignment problem. *Location Science*, 3(2), 87-105.
8. Li, Y.; Pardalos, P.M.; and Resende, M.G.C. (1994). A greedy randomized adaptive search procedure for the quadratic assignment problem. *Series in Discrete Mathematics and Theoretical Computer Science (DIMAC)*,16, 237-261.
9. Taillard, E. (1991). Robust tabu search for the quadratic assignment problem, *Parallel Computing*, 17(4-5), 443-455.
10. James, T.; Rengo, C.; and Glover, F. (2009). Multistart tabu serach and diversication strategies for the quadratic assignment problem. *Institute Electrical and Electronics Engineers* (*IEEE) Transaction Systems, Man and Cybernetics - Part A: System and Human*, 39(3), 579-596.
11. Wilhelm, M.R.; and Ward, T.L. (1987). Solving quadratic assignment problems by simulated annealing. *Institute of Industrial and Systems Engineers (IISE) Transaction*, 19(1),107-119.
12. Alba, E.; Luque, G.; and Nesmachnow, S. (2013). Parallel metaheuristics: Recent advances and new trends. *International Transactions in Operational Research*, 20(1), 1-48.
13. Jong, K.A.D.; Spears, W.M.; and Gordon, D.F (1993). Using genetic algorithms for concept learning. *Machine Learning*,13(2), 161-188.
14. Luong, T.V.; Talbi, E.G.; and Melab, N. (2010). Parallel hybrid evolutionary algorithms on GPU. *Institute Electrical and Electronics Engineers* (*IEEE) Congress on Evolutionary Computation*, Barcelona, 1-8 .
15. Tsutsui, S.; and Fujimoto, N. (2009). Solving quadratic assignment problems by genetic algorithms with GPU computation: A case study. *In Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers (GECCO 2009)*, New York, 2523-2530.
16. Mohassesian, E.; and Karasfi, B. (2017). New method for improving the performance of fast local search in solving QAP for optimal exploration of state space. *Artificial Intelligence and Robotics (IRANOPEN)*, Qazvin, 64-72.

17. Soyata, T (2018). *GPU Parallel Program Development Using CUDA*, Boca Raton, Florida, Taylor & Francis Group.
18. Burkard, R.; Karisch S.; and Rendl, F. (2017). Quadratic Assigment Problem Library (QAPLIB). Retrieved January 05, 2019, from http://anjos.mgi.polymtl.ca/qaplib.
19. Tsutsui, S. (2012). Aco on multiple GPUs with CUDA for faster solution of QAPs. *12th International Conference on Parallel Problem Solving from Nature - PPSN XII*, Taormina, Italy, 7492,174-184.
20. Tsutsui, S.; and Fujimoto, N. (2013). An analytical study of parallel ga with independent runs on GPUs. *Massively Parallel Evolutionary Computation on GPGPUs, Natural Computing Series*, Berlin, Heidelberg, 105-120.
21. Chaparala, A.; Novoa, C.; and Qasem, A. (2014). A SIMD solution for the quadratic assignment problem with GPU acceleration. *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment (XSEDE'14)*, Atlanta, USA, article No.1, 1-8.
22. Semlali, S.; Essaid, M.;and Chebihi, F. (2018). Hybrid chicken swarm optimization with a grasp constructive procedure using multi-threads to solve the quadratic assignment problem. *6th International Conference on Multimedia Computing and Systems (ICMCS)*, Rabat, Morocoo, 1-6.
23. Mohammadi, J.; Mirzaie, K.; and Derhami, V. (2015). Parallel genetic algorithm based on GPU for solving quadratic assignment problem. *In Second Intenational Conference on Knowledge Based Engineering and Innovation (KBEI)*. Teheran, Iran, 569-572.
24. Szwed, P.; Chmiel, W.; and Luczka, P. (2015). OpenCL implementation of PSO algorithm for the quadratic assignment problem. *14th International Conference Artifcial Intelligence and Soft Computing (ICAISC)*. Zakopane, Poland, 9120, 223-234.
25. Ujaldon, M. (2015). Programming GPUs with CUDA. *Tutorial at 18th IEEE CSE'15 and 13th*. Porto, Portugal.
26. Tomassini, M. (1995). A survey of genetic algorithms. *Annual Reviews of Computational Physics, World Scientific*, 3, 87-118.
27. Cantu, E. (1997). Survey of parallel genetic algorithms, *Technical Report 97003, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana Champaign.*
28. Flynn, M. (1972). Some computer organizations and their effectiveness. *Institute Electrical and Electronics Engineers* (*IEEE) Transactions on Computers*, 21(9), 948-960.
29. NVIDIA Developer (2016). Cuda programming guide. Retrieved February 05, 2018, from https://developer.nvidia.com/cuda-gpus.
30. Poveda, R.; and Gomez, J. (2018). Solving the quadratic assignment problem (QAP) through a fine-grained parallel genetic algorithm implemented on GPUs. *10th International Conference on Computational Collective Intelligence (ICCCI)*, Bristol, England, 145-154.
31. Bashiri, M.; and Karimi, H. (2012). Effective heuristics and meta-heuristics for the quadratic assignment problem with tuned parameters and analytical comparisons. *Journal of Industrial Engineering International*, 8(6), 1-9.