PGAGrid: *A Parallel Genetic Algorithm of Fine-Grained implemented on GPU to find solutions near the optimum to the Quadratic Assignment Problem (QAP)*

ROBERTO MANUEL POVEDA CHAVES
MATHEMATICAL, PhD(C), MSc COMPUTER SYSTEMS ENGINNERING

Universidad Nacional de Colombia
Facultad de Ingeniería
Departamento de Ingeniería de Sistemas e Industrial
Bogotá, D.C.
2018

PGAGrid: *A Parallel Genetic Algorithm of Fine-Grained implemented on GPU to find solutions near the optimum to the Quadratic Assignment Problem (QAP)*

ROBERTO MANUEL POVEDA CHAVES
MATHEMATICAL, PhD(C), MSc COMPUTER SYSTEMS ENGINNERING

Thesis to qualify for the title of
PhD in Mathematics COMPUTER SYSTEMS ENGINNERING

Advisor
JONATAN GOMEZ PERDOMO
PhD in Mathematics, Computer Science concentration
Research line
Natural Computation
Research group
Research Group on Artificial Life (Alife)

Universidad Nacional de Colombia
Facultad de Ingeniería
Departamento de Ingeniería de Sistemas e Industrial
Bogotá, D.C.
2018

Title in English

PGAGrid: A Parallel Genetic Algorithm of Fine-Grained implemented on GPU to find solutions near the optimum to the Quadratic Assignment Problem (QAP).

Título en español

PGAGrid: Un algoritmo Genético Paralelo de grano Fino implementado sobre GPU para encontrar soluciones cercanas al óptimo al Problema de Asignación Cuadrática.

Abstract: This work consists in implementing a fine-grained parallel genetic algorithm improved with a greedy 2-opt heuristic to find near-optimal solutions to the Quadratic Assignment Problem (QAP). The proposed algorithm was fully implemented on Graphics Processing Units (GPUs). A two-dimensional GPU grid of size 8×8 defines the population of the genetic algorithm (set of permutations of the QAP), and each GPU block consists of n GPU threads, where n is the size of the QAP. Each GPU block was used to represent the chromosome of a single individual, and each GPU thread represents a gene of such chromosome. The proposed algorithm was tested on a subset of the standard QAPLIB data set. Results show that this implementation is able to find good solutions for large QAP instances in few parallel iterations of the evolutionary process.

Resumen: Este trabajo consiste en implementar un algoritmo genético paralelo de grano fino mejorado con una heurística 2-opt voraz para encontrar soluciones cercanas al óptimo al problema de Asignación Cuadrática (QAP). El algoritmo propuesto fue completamente implementado sobre Unidades de Procesamiento Gráfico (GPUs). Una retícula GPU bidimensional de tamaño 8×8 define la población del algoritmo genético (conjunto de permutaciones del QAP) y cada bloque GPU consiste de n hilos GPU donde n es el tamaño del QAP. Cada bloque GPU fue utilizado para representar el cromosoma de un solo individuo y cada hilo GPU representa un gen de tal cromosoma. El algoritmo propuesto fue comprobado sobre un subconjunto de problemas de la librería estándar QAPLIB. Los resultados muestran que esta implementación es capaz de encontrar buenas soluciones para grandes instancias del QAP en pocas iteraciones del proceso evolutivo.

Keywords: Quadratic Assignment Problem (QAP), Genetic Algorithm (GA), Parallel Genetic Algorithm (PGA), Graphics Processing Unit (GPU), Compute Unified Device Architecture (CUDA).

Palabras clave: Problema de Asignación Cuadrática, Algoritmos Genéticos, Algoritmos Genéticos Paralelos, Unidades de Procesamiento Gráfico, Arquitectura Unificada de Dispositivos de Cómputo.

Dedicated to

my wife,

to my mother,

and to the memory of my father for his example of work and honesty.

Acknowledgment

The author expresses his gratitude to all those who in one way or another have collaborated or contributed in the development of this work. Especially, my advisor, Professor PhD Jonatan Gómez Perdomo.

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

The Quadratic Assignment Problem consists in assigning a set of n facilities in a set of n locations, knowing the flow between facilities and the distances between locations. The goal is to assign each facility to each location in such a way that flows and distances are minimized.

The QAP was introduced by Koopmans & Beckmann in 1957 [33] and was mathematically proven to belong to the NP-Complete Problems category proposed by Sahni & Gonzalez in 1976 [59].

The QAP is considered one of the most complex combinatorial optimization problems and is the model for many real life problems such as facility layout, campus planning, backboard wiring, scheduling, computer manufacturing, turbine balancing, and process communications, among other applications [49].

Some exact methods, like Branch and Bound, Cutting Plane and Bender's decomposition [27, 36, 8, 4, 46], have been applied to solve instances of this problem. However, the number of resources required by these methods makes them non-applicable to instances of size $n > 30$ [13, 11]. In order to tackle this issue, some approximation methods, like Greedy Randomized Adaptive Search, Tabu Search, Simulated Annealing, ACO (Ant Colony Optimization), PSO (Particle Swarm Optimization), and Evolutionary Algorithms (specially Genetic Algorithms) [47, 50, 6, 14, 10, 66, 70, 16], have been applied to find near-optimal solutions to instances of the QAP.

Genetic Algorithms is one of the most important approaches in the field of Evolutionary Algorithms and are defined as iterative procedures of general purpose adaptive search with the virtue of abstractly and rigorously describing the collective adaptation of a population of individuals to a particular environment, based on behavior similar to a natural system. Genetic algorithms were invented by John Holland and a group of students at the University of Michigan inspired by the processes that occur in biological evolution. Holland and his students also demonstrated the easy implementation of these adaptive processes in a computer system [28].

Some authors state that parallelizing a genetic algorithm is an efficient way to proceed, first by "saving" time by distributing workloads, and second by the natural behavior of parallelism on spatially distributed populations. The advantages of Parallel Genetic Algorithms are referenced in [68] and [51].

Modern parallel multiprocessing architectures like Graphics Processing Units (GPUs) [30] have significantly evolved in the last ten years, with the aim of increasing the graphic processing capabilities in the video-game industry to make them faster and more realistic. Such multiprocessing has been used in the scientific field for the solution of real world problems (computational biology, computational finances, and cryptography, among others) through Application Programming Interfaces (APIs) like CUDA (Compute Unified Device Architecture), OPEN CL, or Direct Compute, that exploit those GPU advantages. Therefore, GPUs make it possible to execute those parallel algorithms and significantly diminish execution time.

It is usual to combine a GA with a local search technique to solve combinatorial problems of high complexity. The 2-opt local search heuristic or greedy 2-opt are some of the most representative to address the QAP. The advantages of these techniques in the QAP solution are explained in [65] and [5].

In order to determine the performance of any method, the QAPLIB (Quadratic Assignment Problem Library) [55] include a large number of benchmark QAP-instances of different sizes, and provides the best known solution for them up to date.

1.1 Objective

The objective of this research study is to design and implement a new, efficient, and robust algorithm on GPUs which finds optimal solutions (or near-optimal) to large instances of the QAP through a Fine-Grained Parallel Genetic Algorithm and a heuristic optimization technique. The (specific) objectives of this work are:

1. To solve the QAP by developing a Fine-Grained Parallel Genetic Algorithm and a local search optimization technique (2-opt heuristic or greedy 2-opt)—called **PGAGrid**:

   Deterministic mathematical methods, heuristic procedures, and more recently evolutionary computation methods have been used to solve the QAP; the model presented in this work combines precisely an evolutionary strategy (a Fine-Grained Parallel Genetic Algorithm) and a heuristic (2-opt or greedy 2-opt) to solve the QAP.

   The Fine-Grained model (Grid model or Cellular model) is based on individuals distributed in a two-dimensional grid, one individual per cell [68]. The genetic interaction is restricted to nearby neighbors of each individual. The 2-opt (or greedy 2-opt) heuristic consists in performing all pairwise exchanges of all possible facilities on each location in a particular "assignment" of the QAP [65].

By combining the Fine-Grain parallel model with the local search heuristic, greater efficiency is expected in the QAP solution, given that the Fine-Grain model provides a slow diffusion of information among individuals in the population, while the heuristic methods improve the search for the solutions found with the previous parallel model.

2. To implement the **PGAGrid** model on a graphics processing unit (GPU):

   The GPUs are designed for highly data-parallel computations with high arithmetic intensity. In contrast with a CPU, a GPU dedicates more transistors to data processing, so the GPU performs more float-point operations per second. The Evolutionary Algorithms are inherently parallel; their implementation in GPUs is therefore favorable, and they make it possible not only to significantly increase the computation speed but also offer a greater possibility of to convergence in the global optimum (or near-optimal), as explained in [38].

   The main advantage of a GPU is its structure, each one containing up to hundreds of cores grouped in multiprocessors of architecture SIMD (Single Instruction, Multiple Data). The number of threads that can be executed in parallel on those devices is currently in the order of thousands. Processing is based on many threads that are grouped into blocks, organized into a grid at the same time. Memory hierarchy is an important attribute of modern GPUs. Nowadays, GPUs have four levels of memory: registers, shared block memory, local memory, and global memory; each level provides some programming features.

3. To contrast results obtained by the **PGAGrid** model on GPU of a series of benchmark problems in the QAPLIB standard library with the same results referred to in similar studies.

4. To develop statistical tests of the performance of the **PGAGrid** model on GPU to validate the results obtained.

5. To find appropriate values for the genetic operators in the **PGAGrid** model on GPU that improve the results for the instances to be solved.

1.2 Main contributions

Solving the QAP is not an easy task. It is considered one of the most difficult problems in combinatorial optimization as well as one of the most important NP-complete problems. This problem has great application in diverse fields, especially those in facility layout and campus planning. The contributions of each objective are the following:

From, first objective: when the Fine-Grained parallel genetic model and the 2-opt (or greedy 2-opt) technique are combined, greater efficiency is obtained in the QAP solution. The Fine-Grained model provides evolution as a consequence of a slow diffusion of genetic information among individuals in the population. The 2-opt heuristic (or greedy 2-opt heuristic) accelerates the search for the good solutions found with the previous genetic model. This heuristic is applied in order to deeply exploit promising regions already explored by the genetic algorithm.

From, second objective: the model was fully implemented on a parallel multiprocessing architecture such as Graphical Processing Units (GPUs). The implementation of a two-dimensional GPU grid of size 8×8 defines the population of the genetic algorithm (set of permutations of the QAP), and each GPU block consists of n GPU threads where n is the size of QAP. Each GPU block was used to represent the chromosome of a single individual and each GPU thread represents a gene of such chromosome. This way, the intrinsically parallel architecture of the GPU was fully exploited and the results are obtained more quickly.

As the problem was completely implemented on the GPU, the data transfer bottlenecks between the main memory of the PC and the main memory of the GPU were eliminated. In addition, a convenient handling of memory spaces of the GPU was performed. Each individual in the genetic algorithm was defined on the shared memory of each GPU block and the distance and flow matrices were defined on the constant memory space.

The configuration of the GPU grids was also important to calculate the fitness function and to implement the local optimization heuristics. Both procedures use a matrix formulation appropriate to be implemented in GPU as a vector multiprocessing device.

From, third objective: ten different instances of the QAPLIB reference source were examined; each of them belongs to a different kind of problem of QAPLIB. The best results found by the **PGAGrid** of each instance were compared with the results reported by QAPLIB and the literature.

The best results obtained by **PGAGrid** on GPU of each of the problems were compared with a similar sequential implementation in CPU, together with the corresponding times obtained.

Using these comparisons, it is expected to revalidate the efficiency of the **PGAGrid** model and to emphasize it as an interesting procedure in the QAP solution as well as other problems that derive from it.

From, fourth objective: emphasis was placed on the average results obtained and on the standard deviation, as well as, on the median and on the median absolute deviation in relation to the multiple tests performed. Different neighborhood topologies were implemented in the **PGAGrid** model. A non-parametric Wilcoxon signed rank test was performed to determine which of them is the most appropriate in the QAP solution.

From, fifth objective: in order for the **PGAGrid** to reach greater efficiency, different values of genetic operators were tested until obtaining the appropriate ones that optimize the results.

1.3 Dissertation

This work is divided as follows:

- **Chapter 2** describes some preliminary concepts like the Quadratic Assignment Problem (QAP), the most usual techniques to solve the QAP (particularly, genetic algorithms, parallel genetic algorithms, 2-opt heuristic and greedy 2-opt heuristic), and graphical processing units (GPUs).

- **Chapter 3** develops the **PGAGrid** model implemented on GPU that combines the Fine-Grained parallel genetic model and the local optimization techniques discussed in the previous chapter.

- **Chapter 4** present the experimentation part. The **PGAGrid** model was tested using the QAPLIB library benchmark problems. The results obtained by the **PGAGrid** model were compared with the results presented in other bibliographical references, in addition to the results hitherto reported in the same library.

- **Chapter 5** present conclusions and outlines future works.

CHAPTER 2

Preliminaries

This chapter describes the preliminaries necessary to find an optimal (or near optimal) solution for the QAP.

Section 2.1 presents the definition of QAP, introducing it as a problem of location theory. Subsequently, some equivalent formulations of the QAP are presented. One of these formulations is more convenient for the solution of the problem by means of parallel multiprocessing. Finally, the complexity of the QAP is highlighted, categorized as a strongly NP-Complete problem, and as a generality of many other combinatorial problems that are also NP-Complete.

Section 2.2 highlights the most common techniques to reach the QAP solution. First, the exact methods are presented (deterministic methods), and then the approach methods (metaheuristics), based on trajectories and on population, are shown. Finally, the corresponding parallel models of some approach techniques are described.

Section 2.3 gives a brief description of Graphical Processing Units (GPUs) as parallel processing architectures and the CUDA language as a GPU programming interface.

2.1 Quadratic Assignment Problem (QAP)

The QAP consists in finding the optimal assignment in n facilities to n locations, where the cost is a function of the distance between locations and the flow between facilities. The goal is to assign each facility one by one to each location, so that the cost is minimized.

The mathematical model (corresponding to the original formulation proposed by Koopmans & Beckmann) is:

$$\min_{\sigma \in S_n} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} f_{ij} d_{\sigma(i)\sigma(j)} \tag{2.1}$$

where $F = (f_{ij})$ is a flow matrix, $D = (d_{kl})$ is a distance matrix (both F and D have a of size $n \times n$), and $S_n = \{\sigma \mid \sigma : N \to N,\ \sigma \text{ bijective}\}$, where $N = \{0, 1, \cdots, n-1\}$ (it is often said that n is the QAP size). Each individual product $f_{ij} d_{\sigma(i)\sigma(j)}$ of the previous formula is the cost of assigning facility $\sigma(i)$ to location i, and facility $\sigma(j)$ to location j. Figure 2.1 shows an example of a QAP-instance of size $n = 5$.

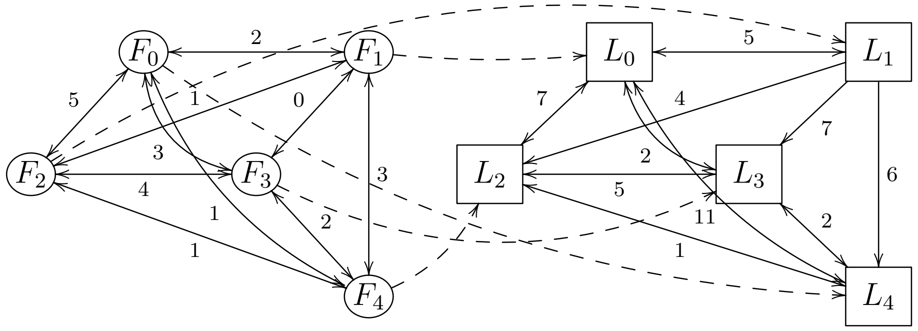


FIGURE 2.1. Example of a QAP-instance of size $n = 5$.

In the example of the previous figure

$$F = \begin{pmatrix} 0 & 2 & 5 & 3 & 1 \\ 2 & 0 & 1 & 0 & 3 \\ 5 & 1 & 0 & 4 & 1 \\ 3 & 0 & 4 & 0 & 2 \\ 1 & 3 & 1 & 2 & 0 \end{pmatrix}, \quad D = \begin{pmatrix} 0 & 5 & 7 & 2 & 11 \\ 5 & 0 & 4 & 6 & 7 \\ 7 & 4 & 0 & 5 & 1 \\ 2 & 6 & 5 & 0 & 2 \\ 11 & 7 & 1 & 2 & 0 \end{pmatrix} \text{ and } \sigma = (1, 2, 4, 3, 0)$$

$$\text{cost}(\sigma) = \text{cost}(1, 2, 4, 3, 0) = \sum_{i=0}^{4} \sum_{j=0}^{4} f_{ij} d_{\sigma(i)\sigma(j)} = 222$$

The total number of permutations for a QAP of size n is $n!$

2.1.1 Alternative formulations of the QAP

The original formulation proposed by Koopmans & Beckmann (formulation 2.1) expresses the combinatorial structure of the QAP. However, other equivalent formulations can provide useful approaches for their solution. Some of these alternative formulations are:

1. **Quadratic Integer Program formulation**

   This formulation corresponds to an integer programming problem with an objective quadratic function (hence the name Quadratic Assignment Problem).

The Quadratic Integer Program formulation uses permutation matrices instead of the simple permutations of the original formulation. Each permutation σ of the set $\{0, 1, \cdots, n-1\}$ can be represented by an $n \times n$ matrix $X = (x_{ij})$ such that

$$x_{ij} = \begin{cases} 1 & \text{if } \sigma(i) = j, \\ 0 & \text{otherwise} \end{cases}$$

(i.e. $x_{ij} = 1$ if facility i is placed in location j, and $x_{ij} = 0$ otherwise).

X is called a permutation matrix. The Quadratic Integer Program formulation is then:

$$\begin{aligned} \text{Min } & \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} a_{ij} b_{kl} x_{ik} x_{jl} \\ \text{subject to } & \sum_{i=0}^{n-1} x_{ij} = 1, \quad 0 \leq j \leq n-1 \\ & \sum_{j=0}^{n-1} x_{ij} = 1, \quad 0 \leq i \leq n-1 \\ & x_{ij} \in \{0, 1\} = 1, \quad 0 \leq i, j \leq n-1 \end{aligned}$$

The restrictions in this formulation imply that each "facility" must be assigned to a single "location," and that each "location" is assigned a single "facility."

2. **Inner Product formulation**

The inner product of two matrices A and B of sizes $n \times n$ is defined as:

$$< A, B >= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_{ij} b_{ij}$$

If X is a permutation matrix, AX_σ^t permute the columns of A, and $X_\sigma A$ permute the rows of A, therefore $X_\sigma A X_\sigma^t = (a_{\sigma(i)\sigma(j)})$.

A more compact alternative formulation for the QAP is:

$$\begin{aligned} \text{Min } & < F, XDX^t > \\ \text{subject to } & \sum_{i=0}^{n-1} x_{ij} = 1, \quad 0 \leq j \leq n-1 \\ & \sum_{j=0}^{n-1} x_{ij} = 1, \quad 0 \leq i \leq n-1 \\ & x_{ij} \in \{0, 1\} = 1, \quad 0 \leq i, j \leq n-1 \end{aligned}$$

3. **Trace formulation**

   It is known that the trace of a matrix A of size $n \times n$ is defined as:

$$\text{trace}(A) = \sum_{i=0}^{n-1} a_{ii},$$

   As trace(AB)=tr(BA) and trace(A) =trace(A^t),

$$< A, B >= \text{trace}(AB^t)$$

   and

$$< F, XDX^t >= \text{trace}(FXD^tX^t)$$

   Therefore, the QAP can be formulated as:

$$\begin{aligned} &\text{Min trace}(FXD^tX^t) \\ \text{subject to } &\sum_{i=0}^{n-1} x_{ij} = 1, \quad 0 \le j \le n-1 \\ &\sum_{j=0}^{n-1} x_{ij} = 1, \quad 0 \le i \le n-1 \\ &x_{ij} \in \{0,1\} = 1, \quad 0 \le i, j \le n-1 \end{aligned} \tag{2.2}$$

   The trace formulation, is the formulation used in this work.

2.1.2 Computational complexity

The QAP is considered one of the most complex combinatorial optimization problems and is the model for many real life problems, such as facility layouts, campus planning, backboard wiring, scheduling, computer manufacturing, turbine balancing and communication processes, among other applications [49].

Sahi & Gonzalez show in [59] that the QAP is not only a NP-Complete problem, but also that it is impossible to find an approximate solution (or ϵ-approximate solution) in polynomial time within some constant factor of the optimal solution, unless P = NP. The foregoing can be formalized in the following definition and in the following theorem:

Definition 2.1. *Given a real number $\epsilon > 0$, an algorithm γ for the QAP is said to be an ϵ-approximation algorithm if*

$$\left| \frac{Z(F, D, \sigma_\gamma) - Z(F, D, \sigma_{opt})}{Z(F, D, \sigma_{opt})} \right| \le \epsilon,$$

holds for every instance QAP, where $Z(F, D, \sigma)$ is the objetive function value of a solution σ for a QAP with flow matrix F and distance matrix D; σ_γ is the solution of QAP computed by algorithm γ and σ_{opt} is an optimal solution of QAP.

Theorem 2.1. *The quadratic assignment problem is strongly NP-Complete.*

For an arbitrary $\epsilon > 0$, the existence of a polynomial time ϵ-approximation algorithm for the QAP implies $P = NP$.

Since it is assumed that P $\neq$ NP, it seems very unlikely to find an ϵ-approximation algorithm in polynomial time for the QAP, for some $\epsilon > 0$. Therefore, solving the QAP to optimality, or even finding an ϵ-approximate solution to it are considered to be hard problems.

Queyranne confirms in [56] the difficulty of solving the QAP in comparison with other difficult combinatorial problems. He shows (unless P = NP) that the QAP is not approximable in polynomial time within some finite ratio, even if D is a distance matrix of a set of points on the Euclidean line, and F is a block-diagonal symmetric matrix. For example, Christofides [18] demonstrates that the Traveling Salesman Problem (TSP) is approximable in polynomial time (with $\epsilon = 3/2$) if the distance matrix is symmetric and satisfies the triangular inequality.

2.1.3 Other NP-Complete problems formulated as QAPs

Another reason that highlights the structure and complexity of QAP is that many others significant NP-Complete combinatorial optimization problems result in particular cases of QAP. Some of them are:

1. **The Traveling Salesman Problem (TSP) [37]**

   This problem can be stated as follows: How should an agent visit a set of n cities returning to the origin city in such a way that each city is just visited once and the cost of the tour is the minimized?

   If, in the QAP formulation, $D = (d_{ij})$ is defined as the matrix of distances between cities of the TSP, and $F = (f_{ij})$ is defined as a hamiltonian cycle adjacency matrix with a number n of vertices, for example:

$$F = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ 1 & 0 & 0 & \dots & 0 \end{pmatrix}$$

$$\begin{aligned}
\text{TSP} : \min_{\sigma \in S_n} & \sum_{i=0}^{n-2} d_{\sigma(i)\sigma(i+1)} + d_{\sigma(n-1)\sigma(0)} \\
\equiv \min_{\sigma \in S_n} & \sum_{i=0}^{n-2} f_{i(i+1)} d_{\sigma(i)\sigma(i+1)} + f_{(n-1)0} d_{\sigma(n-1)\sigma(0)} \\
\equiv \min_{\sigma \in S_n} & \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} f_{ij} d_{\sigma(i)\sigma(j)} : \text{QAP}
\end{aligned}$$

2. **The Linear Arrangement Problem (LAP)** [26]

   This problem consists in finding an arrangement of the nodes of a weighted graph on n nodes, so as to minimized the sum of the weighted edge lengths.

   If, in QAP formulation, $D = (d_{ij})$ given by $d_{ij} = |i - j|$, for all i, j and flow matrix $F = (f_{ij})$ is the (weighted) adjacency matrix of a given graph $G = (V, E)$.

$$\begin{aligned}
\text{LAP} : \min_{\sigma \in S_n} & \sum_{(i,j) \in E} w_{ij} |\sigma(i) - \sigma(j)| \\
\equiv \min_{\sigma \in S_n} & \sum_{i=0}^{n-2} f_{i(i+1)} d_{\sigma(i)\sigma(i+1)} + f_{(n-1)0} d_{\sigma(n-1)\sigma(0)} \\
\equiv \min_{\sigma \in S_n} & \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} f_{ij} d_{\sigma(i)\sigma(j)} : \text{QAP}
\end{aligned}$$

3. **Maximum Clique Problem (MCP)** [48]

   Given a graph $G = (V, E)$ with $|V| = n$, this problem consists in finding the maximum number $k \leq n$ such that there is a subset $V_1 \subseteq V$ with k vertices that induces a clique in G.

   If, in the QAP formulation, $D = (d_{ij})$ equal to the adjacency matrix of the given graph G, and the flow matrix $F = (f_{ij})$ given as the adjacency matrix of a graph consisting of a clique of size k and $n - k$ isolated vertices, multiplied by -1.

   G has a clique of size k if and only if the optimal value of the QAP problem is $-k^2 + k$. In fact:

$$
\begin{aligned}
\text{QAP} : \min_{\sigma \in S_n} & \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} f_{ij} d_{\sigma(i)\sigma(j)} \\
&= 2\Big(f_{01}d_{\sigma(0)\sigma(1)} + f_{02}d_{\sigma(0)\sigma(2)} + \dots f_{0(k-1)}d_{\sigma(0)\sigma(k-1)} \\
&\qquad + f_{12}d_{\sigma(1)\sigma(2)} + \dots + f_{1(k-1)}d_{\sigma(1)\sigma(k-1)} \\
&\qquad\qquad + f_{(k-2)(k-1)}d_{\sigma(k-2)\sigma(k-1)}\Big) \\
&= -2\Big((k-1) + (k-2) + \dots + 1\Big) = -2\Big(\frac{(k-1)k}{2}\Big) = -k^2 + k
\end{aligned}
$$

4. **Graph Partitioning Problem (GPP) [35]**

   Given a graph $G = (V, E)$ with $|V| = n$ and a number k which divides n, the problem consists in partitioning the set V into k subsets $V_1, V_2, \dots, V_k$ of equal cardinality such that the set $E_{cut} = \{(u, v)|(u \in V_i) \wedge (v \in V_j), i \neq j\}$ is minimized. This problem is a special case of QAP if, in QAP formulation, $D = (d_{ij})$ equal to the adjacency matrix of the given graph G, and the flow matrix $F = (f_{ij})$ given as a block diagonal matrix with k blocks, where each block is the negative adjacency matrix of a complete subgraph with n/k vertices, thus:

$$
F = \begin{pmatrix} \mathbf{B_1} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{B_2} & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{B_k} \end{pmatrix}_{n \times n} \quad \text{with } \mathbf{B_j} = \begin{pmatrix} 0 & -1 & \dots & -1 \\ -1 & 0 & \dots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \dots & 0 \end{pmatrix}_{\frac{n}{k} \times \frac{n}{k}}
$$

   $\forall j,\ 1 \leq j \leq k$. and $\mathbf{0}$ is a matrix of zeros.

5. **Packing Problem in Graphs [9]**

   Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with $|V_1| = |V_2| = n$, a permutation σ of $\{0, 1, \dots, n-1\}$ is called a packing of G_2 into G_1, if $e_{ij} \in E_1$ implies $e_{\sigma(i)\sigma(j)} \notin E_2\ \forall i, j,\ 0 \leq i, j \leq n-1$ (or vice versa). The graph packing problem consist in finding or not a packing of G_2 into G_1.

   If, in the QAP formulation, $D = (d_{ij})$ equal to the adjacency matrix of the given graph G_2, and $F = (f_{ij})$ equal to the adjacency matrix of given graph G_1, a packing of G_2 into G_1 exists if only if the optimal value of this QAP is equal to 0.

2.2 Techniques to solve the QAP

The most usual techniques applied to solve the QAP are exact methods and approach methods.

2.2.1 Exact methods

These methods correspond to mathematical programming techniques. Some of them are based on the divide and conquer strategy, i.e. they work by partitioning the search space of the problem into sub-problems and by optimizing each one of them separately. The most common are:

Branch and bound (B&B). It is based on an implicit enumeration of all of the solutions to the problem. The search is carried out on the whole problem domain; the set of solutions is thought of as forming a rooted tree, where the root represents the problem itself, and the leaves represent additional restrictions that bound the solution to the problem. This solution improves as the iterations advance. The method uses the *branching* operator that determines the order in which the branches are explored (for example, a deep search or a wide search), the *pruning* operator that eliminates these solutions that do not lead to the best, considering lower bounds for every partial solution. The method ends when there are no more branches or nodes to explore, or when all the nodes have been eliminated. The quality of the bounds and the branching strategy determine the quality of the method. Generally, the QAP is solved by B&B from its representation as a linear assignment problem [7] and using bounding techniques, such as the Gilmore-Lawler lower bounds method [27].

Dynamic programming. It is based on Richard Bellman's optimality principle that "All sub-policies of an optimal policy must be optimal as well." It solves a general problem in a recursive manner, positing less complex problems that have the same structure of the original problems. This process is applied until problems with an immediate solution are found, and the process starts again until a solution to the general problem is obtained.

Relaxations. A problem is solved with less requirements than the original problem. The most common relaxations are the linear programming relaxations and Lagrangian relaxations. The former solve a problem with real variables when the original requires only integer solutions; with this, lower bounds are found, and they can be used with the B&B method for the *branching* part. In the Lagrangian relaxation, some restrictions of the problem (usually the most difficult ones) are removed and incorporated to the objective function through a penalizing function [67]. Some relaxations for QAP appear in [36, 25, 3]

Benders decomposition. When the QAP formulated as a linear programming (mixed integer) problem, it can be solved by fixing the integer variables and by solving the corresponding dual problem. The time for this convergence to happen is long, and it is usually applied only to small instances, but when applying cutting plans, good sub-optimal solutions are produced [8].

Cutting planes. Cutting planes were proposed by Gomory [29]. They are based on adding specific restrictions to relaxed Linear Programming problems. The method approximates the polyhedron represented by the convex shell of all feasible solutions of the original problem by the polytope of the relaxed problem.

2.2.2 Approach methods

They are divided into metaheuristics based on trajectory, and metaheuristics based on population.

2.2.2.1 Metaheuristics based on trajectories

These methods consist in making determined searches within a space of solutions. They start with only one initial solution, and the solution in each iteration is then replaced by another (frequently a better one). The spirit of the methods is aimed at exploting promising regions of the search space (in order to intensify the search). The most common in the solution of the QAP are:

Construction methods. They are considered the most simple heuristics for the QAP. The quality of the solutions is not the best, but, they are very simple to implement computationally and, given their properties, they can be used as part of more intelligent methods for the QAP. Basically, they start with an empty permutation, and recursively assign places to facilities according to a certain criteria until all facilities have been assigned. These methods were formulated for the first time by Gilmore [27] around 1960. Another method of construction with better results is the one proposed by Muller in [42].

Local Search methods (LS). They are algorithms that produce optimal local solutions in the following way: a neighborhood $N(\sigma_0)$ of a permutation σ_0 of the QAP consists of all permutations that in some sense are close to σ_0. Therefore, a local optimal QAP solution is a permutation $\overline{\sigma}$ such that:

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} f_{ij} d_{\overline{\sigma}(i)\overline{\sigma}(j)} = \min_{\sigma \in N(\sigma_0)} \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} f_{ij} d_{\sigma(i)\sigma(j)}$$

The procedure starts with an initial solution, and is improved through a movement to a solution within its neighborhood; this procedure is repeated until no solution gets better. In order to obtain better results, the local search algorithms are carried out many times starting at different initial solutions (it is therefore recommended that a parallel implementation be used). A more complete study appears in [1].

2-opt heuristic. The 2-opt local optimization heuristic is applied in order to improve candidate QAP solutions [65]. This method consists in performing all pairwise exchanges of all possible facilities on each of the locations in a particular permutation. The current permutation is updated with the permutation with the lowest negative value Δ_{ij} (difference between the cost of the permutation found and the cost of the current permutation) in the following formula:

$$\begin{aligned}\Delta_{ij} = {} & (f_{ji} - f_{ij})(d_{\sigma(i)\sigma(j)} - d_{\sigma(j)\sigma(i)} \\ & + \sum_{\substack{k=0 \\ k\neq i,j}}^{n-1} ((f_{jk} - f_{ik})((d_{\sigma(i)\sigma(k)} - d_{\sigma(j)\sigma(k)})) \\ & + (f_{kj} - f_{ki})(d_{\sigma(k)\sigma(i)} - d_{\sigma(k)\sigma(j)})\end{aligned}$$

with asymmetric distance and flow matrices [14], (i, j are facilities that are exchanged).

This formula is evaluated in linear time for all possible $n(n-1)/2$ swaps. The Koopmans-Beckmann's original formulation (formula 2.1) is evaluated in quadratic time.

A matrix formulation for Δ_{ij} is:

$$\begin{aligned}\Delta_{ij} = {} & (f_{ij} - f_{ji})(d_{\sigma(j)\sigma(i)} - d_{\sigma(i)\sigma(j)}) \\ & + (F_{i\cdot} - F_{j\cdot})\bullet((DX^t)_{\sigma(j)\cdot} - (DX^t)_{\sigma(i)\cdot}) \\ & + (F_{\cdot i} - F_{\cdot j})\bullet((XD)_{\cdot\sigma(j)} - (XD)_{\cdot\sigma(i)})\end{aligned} \tag{2.3}$$

where $\bullet$ interprets an internal product, $f_{ij} = f_{ji} = 0$ in addends 2 and 3. $F_{k\cdot}$ is the row k of the matrix F and $F_{\cdot k}$ is the column k of the matrix F.

Greedy Randomized Adaptive Search Procedure (GRASP). It is a very usual heuristics for combinatorial optimization problems. GRASP is a combination of greedy elements and randomized search elements, and it is composed of a construction phase, where two facilities are assigned to two locations among all those with minimal costs, and a improvement phase that includes randomized elements to avoid falling into local optima. A detail implementation for QAP is described in [50].

Greedy 2-opt heuristic. This method is a variant of the 2-opt heuristic. The Greedy 2-opt heuristic performs one by one all pairwise exchanges of all possible facilities on each of the locations in a particular permutation, but, immediately find an permutation with better cost, this updates the previous permutation (the cost is also updated), i.e. the current permutation is updated with the first permutation found such that $\Delta_{ij} < 0$ (formula 2.3). The greedy 2-opt heuristic continues on the permutation updated in the next swap. This heuristic and other greedy heuristics for the QAP are explained in [5].

Tabu Search (TS). This technique is used to "remember" which solutions have already been visited, and to abandon neighborhoods that have local optimal locations. In QAP, the movements used are usually swaps, but are controlled by a tabu list, which does not allow certain movements on the current solution; as movements change, the list is updated during the search. The solution starts with an initial feasible solution; only if the selected solution is not in the tabu list, the initial solution is updated by the selected solution (this new solution is not necessarily better than the initial), and then the search in the neighborhood is repeated. Different studies refer to the convenient size of the tabu list, with respect to the QAP—in [6] there is a deep study on the subject. The tabu search algorithm has a parallel nature in its implementation dividing its search load between various processors. Parallel implementations are proposed in [64, 58].

Simulated Annealing (SA). It is an approach that exploits the analogy between combinatorial optimization problems and mechanical statistics problems (a physical system composed of many particles). The feasible solutions to a combinatorial optimization problem correspond to the states of the physical system, and the values of the objective function correspond to the energy of the physical system state. A material is heated and then slowly cooled to change its physical properties. The heat causes the atoms to augment their energy and move from the initial positions that would correspond to local minimums in an optimization problem. The slow cooling produces low energy states (thermic balance) that would correspond to global minimums in an optimization problem. Wilhelm et al. [75] obtained good results for QAP with a sophisticated simulated annealing approach.

2.2.2.2 Metaheuristics based on populations

They are iterative techniques that apply stochastic operators on a set of individuals (population), where each individual one corresponds to a coded version of a possible problem solution. The performance (quality) of each individual is evaluated through an aptitude function; variation operators on some of the individuals guide the whole population to high quality solutions, which confer these metaheuristics a good exploration power. The most common metaheuristics applied to the QAP are the following:

Genetic Algorithms (GAs). Genetic algorithms (GAs) are some one of the most outstanding approaches in the field of evolutionary algorithms, and are defined as general-purpose iterative adaptive search procedures. This is the metaheuristic considered in this work. GAs have the advantage of describing in an abstract and rigorous way the collective adaptation of a population of individuals to a given environment, based on a behavior similar to that of a natural system.

A simple Genetic Algorithm usually begins with a population of individuals randomly generated, or sometimes pre-established by previous experiences, or generated by any heuristic procedure. The simple GA keeps population size constant, and works iteratively as follows:

During each step in the iteration (called *generation*), individuals are evaluated and assigned a fitness value. To form a new population (from the previous one), the *selection operator* is applied, which consists in choosing individuals with a probability proportional to their relative aptitude; this ensures that the expected number of times an individual is chosen is proportional to their relative performance in the population. It is expected that individuals above the average have a higher copies in the new population (higher probability of reproduction), while individuals below the average have more risk of disappearing. This operator acts as a generator of "intermediate parents," who will be responsible for giving birth to a new population, better than the previous one. To incorporate new individuals into the population, some genetic operators are required, such as the *crossover operator* (which simulates sexual reproduction), and/or the *mutation operator* (which simulates asexual reproduction). The crossover, which is the most important operator of recombination, consists in taking two individuals called parents (chosen by the selection operator) and generating two new individuals called offsprings, thus exchanging parts of the parents; sub-strings of parent chains are exchanged from a certain crossing point chosen randomly. With the crossover, the search is guided towards good regions in the problem domain. The mutation operator essentially makes it possible to avoid a premature convergence to a local optimum, changing each component in some chains with a reduced probability. In more sophisticated genetic algorithms, crossover and especially mutation do not necessarily have to remain constant throughout the simple AG process. In general, the crossover aims to combine the most characteristic features of parent chains, and therefore to increase the fitness of new individuals. The mutation only affects one individual at a time, and its intention is to avoid a premature convergence to a local optimum. Without the mutation operator, potentially useful genetic material could be lost. Since these algorithms are stochastic procedures, their performance varies from one execution to another (unless the same random number generator is used). Because of this, the average performance of several executions is more reliable—and therefore more used—than the results generated by a single execution of the algorithm.

Genetic algorithms, like most stochastic iterative algorithms, do not guarantee convergence. They end after a maximum number of iterations, or when a satisfactory solution is reached. It may also happen that the quality of the results is not improved in the iterative process. Therefore, the algorithm must be stopped before reaching the maximum number of iterations. Algorithm 1 shows a simple genetic algorithm.

Algorithm 1 Simple Genetic Algorithm.

```
Produce an initial population of individuals
while termination condition not met do
   Evaluate the fitness of all individuals
   Select individuals suitable for reproduction
   Produce new individuals
   Generate a new population by inserting some good new individuals and
      discarding some bad old individuals
   Mutate some individuals
end while
```

A great number of research projects have been proposed for the QAP, but generally this method works better considering a hybrid approach with a Local Search algorithm or a Tabu Search [76, 72].

Ant Colony Optimization (ACO). This heuristic method imitates the behavior of ants when searching for food. The analogy between the performance of ants and the solution of a combinatorial optimization problem resides in the following factors: the search space of ants corresponds to the set of feasible solutions to the optimization problem, and each source of food corresponds to the value of the objective function. The adaptive memory component of each ant is the track of pheromones that accumulate in the less traveled roads. With respect to the QAP, the pheromone track of ACO is the measure of attraction to locate a facility i in a location j. This procedure is described in detail in [66], and other studies are gathered in [19]. Tseng and Liang, in [70], apply first a GA to find an initial population. A parallel ACO procedure for the QAP appears in [71].

Particle Swarm Optimization (PSO). In this method, a number of particles move through the search space with the objective of finding an optimal position (a good solution). The particles communicate with each other, and the one with the best position (measured according to an aptitude function) exerts an influence on the rest. The particles adjust their movements systematically (position and speed) according to their own experience, and according to the experience of the rest of the swarm. This method is inspired in the social behavior of organisms, such as flocks of birds or schools of fish. Although in principle it is a method for continuous search spaces, it has also been applied to discrete optimization problems like the QAP (see [2, 45]) and to the TSP.

2.2.2.3 Parallel metaheuristic methods based on trajectories

Parallel movement model (or Iteration level model). The evaluation of neighborhoods is made in parallel. At the beginning of each iteration, a master process distributes the current solution on slave processes, and each one, separately, handles a particular neighborhood and obtains a new solution. These solutions then return to the master process (see figure 2.2).

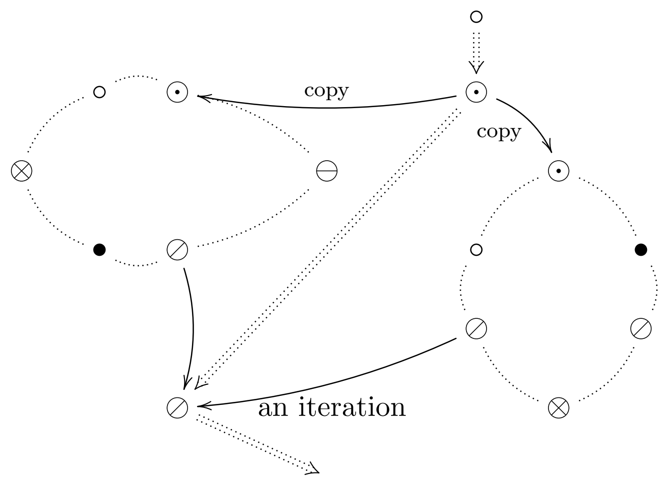


Figure 2.2. Parallel movement model, parallel exploration of the neighborhood, taken from [38].

Parallel multi-start model (or Algorithm level model). Several metaheuristic methods based on trajectories are executed simultaneously, from the same or from a different solution (see figure 2.3).

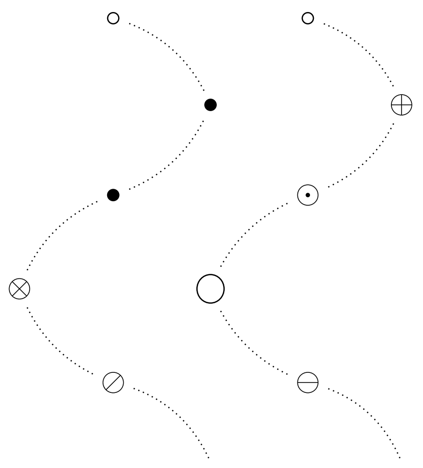


Figure 2.3. Parallel multi-start model, taken from [38].

Acceleration movement model (Solution level model). A unique solution is evaluated in parallel. In this case, the aptitude function is a succession of sub-functions that can be executed in parallel (see figure 2.4).

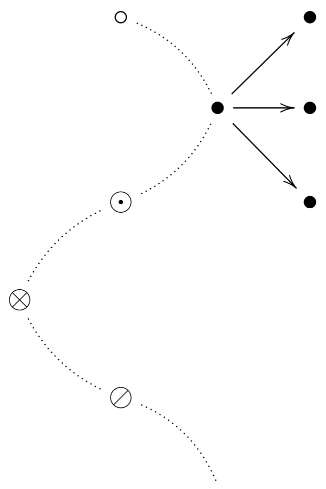


FIGURE 2.4. Acceleration movement model, taken from [38].

2.2.2.4 Parallel models for metaheuristic methods based on population

Sharing workload in terms of computing resources (memory, processors, interconnected equipment) can consider the possibility of trying to solve increasingly complex and interesting problems. The parallelism is inherent in this type of metaheuristics, since each individual in the population is practically an independent unit [62]. There are different types of parallel models for evolutionary algorithms classified according to the way the population individuals interact and to how its size is defined. The most important are:

Embarrassingly parallel algorithm. The same evolutionary algorithm is run under different initial conditions in a parallel way. When all the different configurations have been executed, the configuration showing the best behavior is chosen. The embarrassingly parallel genetic algorithm does not have a behavior similar to that of a natural environment, but the time savings in obtaining final results is important.

The following models correspond to genuine parallel evolutionary algorithms.

Master-Slave model (Global parallelization model). This model parallelizes an evolutionary algorithm in the level of its fitness function. A master process distributes multiple individuals to various slave processes, which then return right away the fitness of those individuals. This procedure turns out to be efficient, given that the evaluation of the fitness function is the task within an evolutionary algorithm that consumes a greater processing time (see figure 2.5).

Coarse-grain model (Island model or distributed model). A master process is in charge of the distribution of N tasks to slave processes, that is, the total

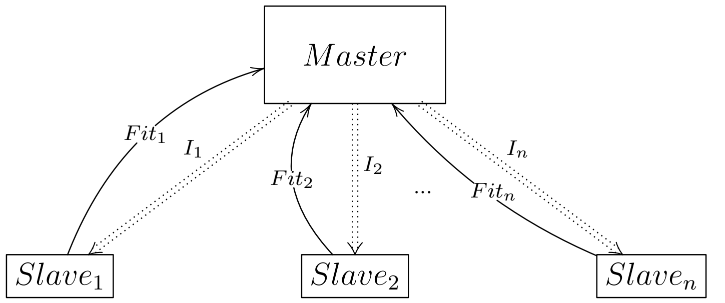


FIGURE 2.5. Master-Slave model. I_i: i-th individual, Fit_i: i-th individual's fitness.

population is distributed in N sub-populations (islands) and on each one of them a population-based model is executed [51]. This model is inspired by the spatially distributed structure of natural populations. After working each slave (processor) simultaneously and independently, an information interchange is executed among them in certain stages of the execution of the algorithm, replacing the k individuals with lower aptitude of each "island" by the k individuals with greater aptitude of the previous "island" (see figure 2.6).

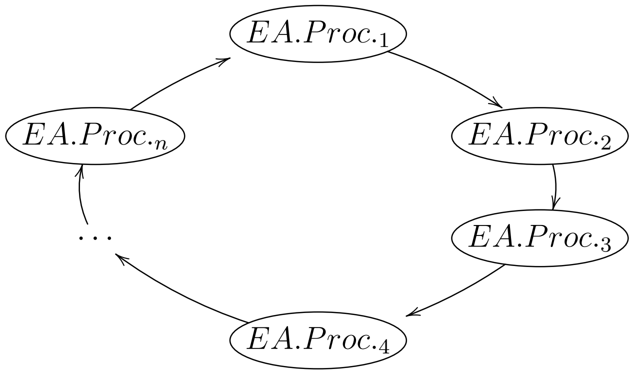


FIGURE 2.6. Coarse-grain model. $EA.Proc._i$: Simple evolutionary algorithm executed on the i-th processor.

The intention of this approach is to periodically re-inject diversity to sub-populations that, in principle, made an exploration in particular search space and that, therefore, tend to converge to a local or premature optimality.

Fine-grain model (Cellular model or Grid model). This model is so called, because it has a strong resemblance to cellular automata, but with stochastic transition rules [51]. In this model, individuals are placed in a two-dimensional mesh, one individual per cell. Figure 2.7 shows a neighborhood (in black) for a particular individual (in gray), this particular neighborhood is known as Moore's neighborhood.

The fitness evaluation for each individual is done simultaneously, and the crossover operator takes place locally within each neighborhood. The way an individual is selected in the neighborhood to mate with the "central" individual (previous figure) is to select some individuals from the neighborhood with a uniform probability and to make a selection per tournament between them. The selection per

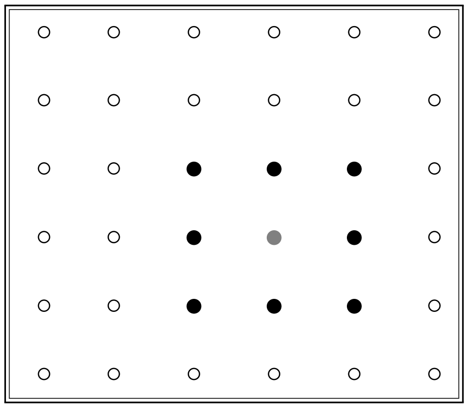


FIGURE 2.7. Neighborhood in the Fine-grain model.

tournament is done deterministically or probabilistically: if it is done in a deterministic way, the most suitable individual is simply selected among the participants, while if it is done in a probabilistic way, the most apt individual will be more likely to be chosen. Subsequently, the crossover between the selected individual and the central individual is made. The central individual will be replaced by the generated offspring if and only if their fitness is less or equal than the generated offspring (maximization problem). More sophisticated algorithms do not restrict the selection of the central individual's mate only to the neighbors, but rather carry it out through more distant random paths, which makes the algorithm much more dynamic. Algorithm 2 shows a Fine-grain model.

Algorithm 2 Fine-grained Model.

```
for each cell $i$ in the mesh, in parallel do
    Assign a random individual
end for
while termination condition not met do
    for each individual $i$, in parallel do
        Evaluate
        Select an individual $k$ in the neighboring of $i$
        Produce offspring from $i$ and $k$
        if the offspring is better or equal than $i$ then
            Assign the offspring to $i$
        end if
    end for
end while
```

2.3 Graphics Processing Unit (GPUs)

A graphic processing unit (GPU) is a coprocessor whose purpose is graphics processing. However, GPUs have been used recently to accelerate general computations, now, they are also known as General-Purpose Graphics Processing Unit (GPGPU). The multiprocessing architecture in parallel of a GPU is ideal for developing algebraic operations of vectors and matrices.

A GPU consists of a set of Streaming Multiprocessor (SMXs), each one equipped with enough cores (Streaming Processors (SPs)) designed for parallel performance. Each multiprocessors shares the same control unit, i.e. SIMD (Single instruction, multiple data) execution (see figure 2.8).

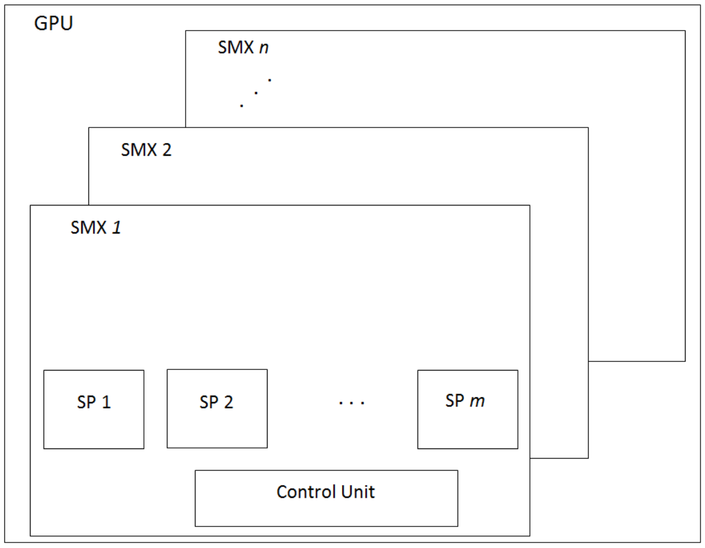


FIGURE 2.8. Hardware model of a GPU.

The GPU has its own off-chip DRAM memory (GDDR5), called *device memory* (*global memory or VRAM*), which is approximately three times faster than the main memory (DDR3) of the CPU. Each multiprocessor has a bank of memory registers, a shared memory, and a space of constants and textures—the last two are read-only (see figure 2.9). The lifetime of the shared memory is the same as the lifetime of a kernel (a procedure that runs on a GPU), while the global, constant and texture memory spaces remain throughout the application. Shared memory (on-chip memory) is approximately five hundred times faster than VRAM and constant memory is approximately one hundred times faster than VRAM, [73]. Shared memory is actually a SRAM (Static Random Access Memory) memory.

CUDA (Compute Unified Device Architecture) is a computer language designed to take advantage of the multiprocessing power of a GPU in general-purpose computing. CUDA is basically C programming language with SIMD extensions. With CUDA, programmers do not require any knowledge about vertices, pixels, or textures (as was the case with graphical programming interfaces like OpenGL or DirectX), nor do they require any knowledge of graphic programming [60].

The CUDA programming model is based on massive data parallelism and fine grain parallelism. In addition, the model is scalable, that is, the code is executed on any number of cores without the need to recompile [73]. The hierarchical structure

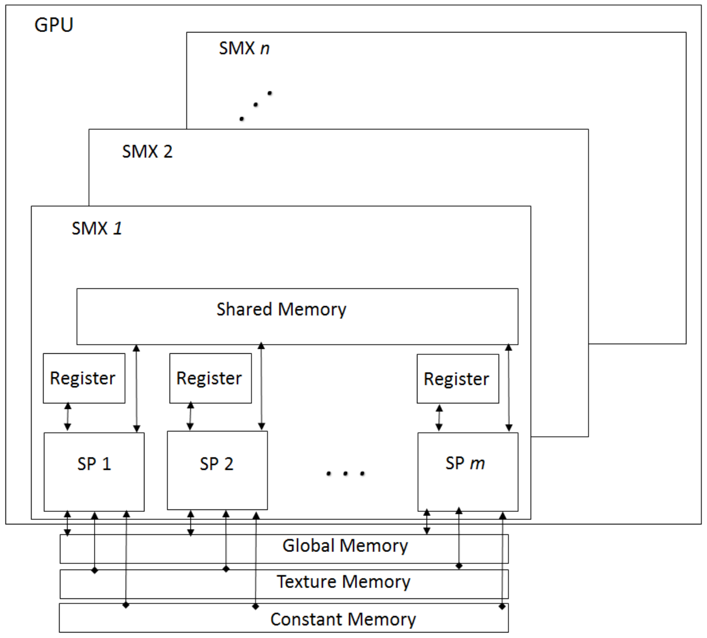


FIGURE 2.9. Memory hierarchy on a GPU.

in a GPU device with CUDA is composed of execution threads, thread blocks, and block grids. The threads are organized in a one-dimensional, two-dimensional or three-dimensional way within a block, just like the blocks inside a grid (see figure 2.10).

The GPU is a coprocessor that is highly branched in threads. The threads run in parallel on the cores of a multiprocessor. Each multiprocessor processes batches of blocks, one after the other. A correspondence between the hierarchical structure in a GPU device with CUDA and the memory hierarchy appears in figure 2.11.

The hardware is free to plan the execution of a block in any multiprocessor. The blocks can be executed sequentially or concurrently according to the availability of resources [73] (see figure 2.12).

The skilled programmer takes advantage of plasticity of the CUDA development, and adapts it to the technology of the logical structure of the grid in the GPU using the data structure implied by the problem to be solved.

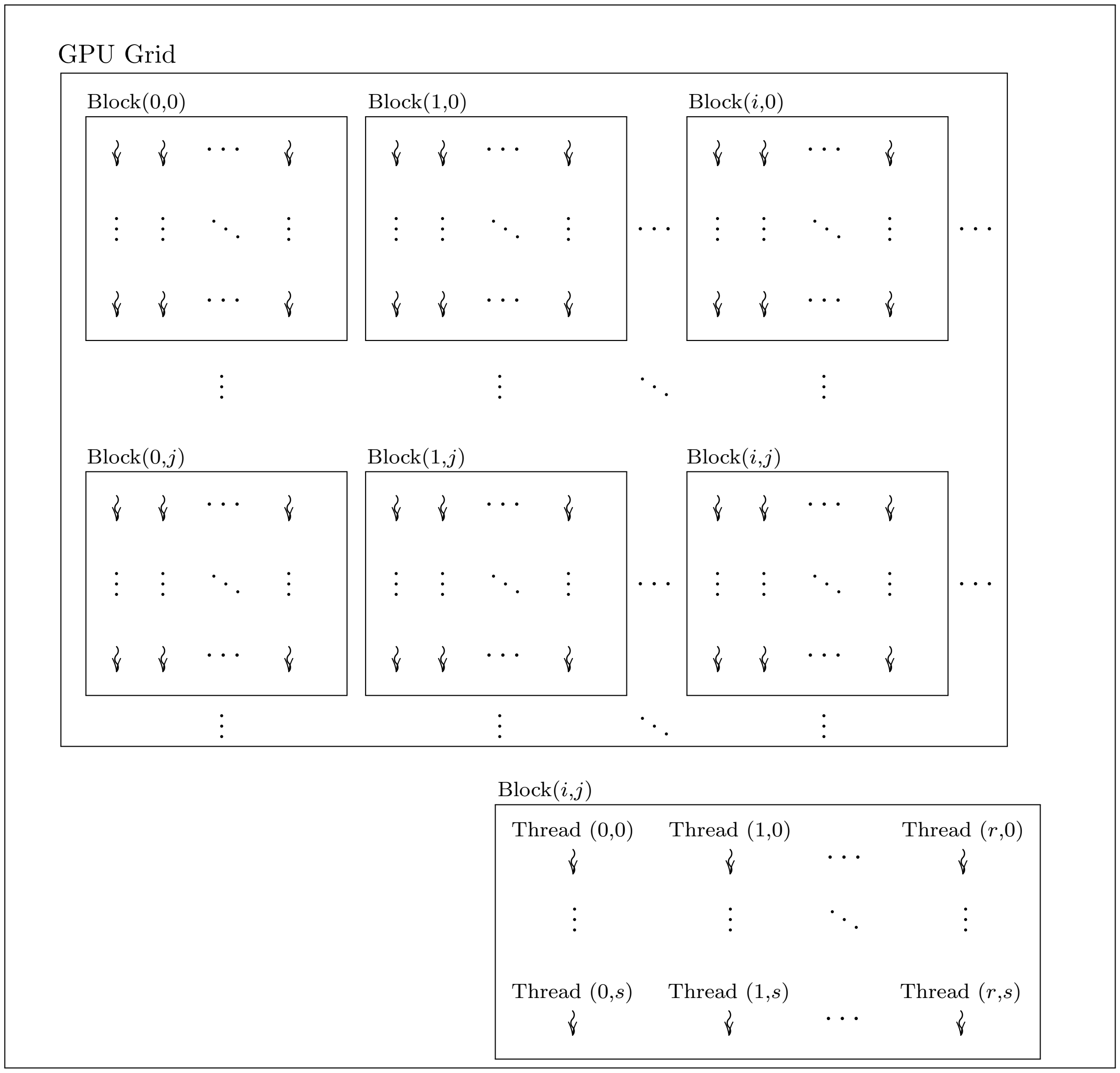


FIGURE 2.10. Logical organization of GPU with CUDA.

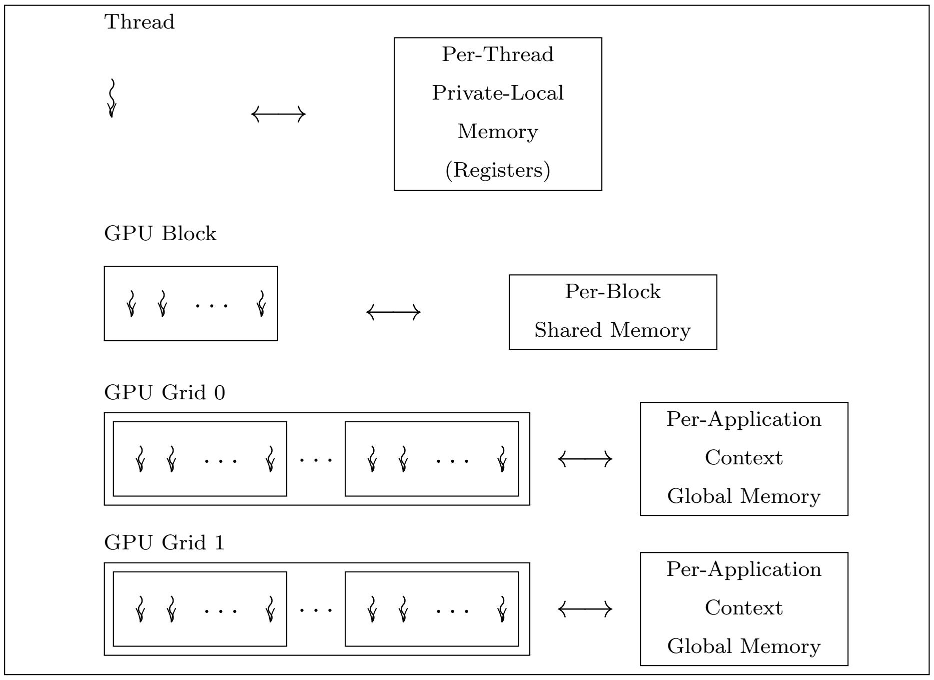

Thread
Per-Thread Private-Local Memory (Registers)
GPU Block
Per-Block Shared Memory
GPU Grid 0
Per-Application Context Global Memory
GPU Grid 1
Per-Application Context Global Memory

FIGURE 2.11. Correspondence between the hierarchical structure in a GPU device with CUDA and the memory hierarchy.

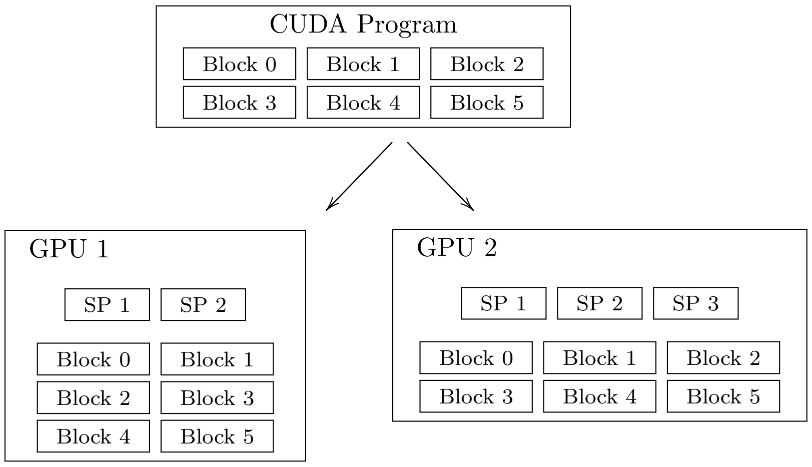


FIGURE 2.12. Execution of a CUDA program.

CHAPTER 3

Proposed Approach

As seen in the previous chapters, it is not an easy task to find an optimal solution (not even near the optimum) to the QAP by exact methods. This chapter shows the implementation of a Fine-Grained Parallel Genetic Algorithm improved with a 2-opt (greedy 2-opt) local search heuristic, called **PGAGrid**. **PGAGrid** is fully implemented on graphics processing unit (GPUs), and aims to find near-optimal solutions to significant instances of the QAP.

Section 3.1 explains the coding used to represent a chromosome (permutation of the QAP) and how to define the initial population on a GPU. This section also shows the correspondence between the logical components of a GPU and the data structures of a Genetic Algorithm in the **PGAGrid** model.

Section 3.2 explains how the fitness of each of the individuals in the population is calculated, and how their matrix representation is favorable to be implemented in a GPU. It also explains how different memory spaces of the GPU are used to speed up calculations. Finally, this section highlights the computational complexity of each of the procedures involved in the calculation of the fitness function in the **PGAGrid** model.

Section 3.3 presents the implementation of the selection operator in the **PGA-Grid** model. The technique used is an elitist binary tournament. The tournament takes place simultaneously between each individual residing in each of the GPU blocks of the GPU grid defined in section 3.1 and another individual from a randomly assigned population. Each random individual is uniquely assigned to the corresponding GPU block.

The crossover operator for **PGAGrid** is implemented in section 3.4. The crossover used was a modified order crossover. This crossover is performed simultaneously between each of the individuals of the population that reside within each GPU block of the GPU grid and another individual assigned to the same GPU block, using a neighborhood topology according to the fine-grained parallel genetic model described in paragraph of the subsection 2.2.2.4. The modified order crossover is implemented completely in parallel. Each procedure in the section describes the corresponding computational complexities.

The mutation and transposition operators used in the **PGAGrid** model are described in sections 3.5 and 3.6, respectively. Both operators are implemented at block level within the GPU. Exchange mutation was used as the mutation operator, where two genes (GPU threads) of each individual (GPU block) are exchanged. The transposition operator simply inverts the genes (GPU threads) of a substring of the chromosome that represents each individual (GPU block).

Section 3.7 explains two heuristics of local optimization (2-opt and greedy 2-opt), as well as the way they were implemented in the GPU. Each of these heuristics was applied simultaneously to each of the individuals (GPU blocks) of the population (GPU grid). These techniques were fully implemented in the GPU according to the matrix formulation 2.3. Of course, this formulation takes advantage of the architecture of the GPU as a vector device.

A summary of the **PGAGrid** model is described in section 3.8.

Finally, section 3.9 lists two links to articles published on the solution of some combinatorial problems from a QAP by the PGAGrid model. The first article deals with the solution of some NP-Complete combinatorial problems as particular cases of QAP (see subsection 2.1.3). The second article focuses on the solution of classical chess problems posed as a QAP. The NP-Complete problems require only properly defining the flow and distance matrices in the **PGAGrid** model. Chess problems also require defining those matrices, as well as redefining the coding of the individual in the **PGAGrid** model.

3.1 Coding and initial population

An integer coding was used to represent the chromosome of an individual in the GA. This chromosome is a permutation $\boldsymbol{\sigma} = (\sigma(0), \sigma(1), \ldots, \sigma(n-1))$ of the QAP. The position $i,\ 0 \leq i < n$ in vector $\boldsymbol{\sigma}$ corresponds to a location of the QAP and the value $\sigma(i)$ in $\boldsymbol{\sigma}$ corresponds to a facility of the QAP associated with location i, (see an example in figure 3.1).

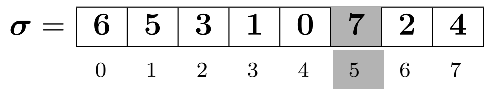


FIGURE 3.1. Representation scheme of the GA. The shaded gene means that facility 7 is placed at location 5 in a QAP of size $n = 8$.

The initial population was defined in a two-dimensional GPU grid of size 8×8 (set of permutations of the QAP). Each GPU block consists of 128 GPU threads, but only n are active[1] (n is the size of the QAP).

Each GPU block and each GPU thread are completely referenced by the variables blockIdx.x, blockIdx.y and threadIdx.x:

- blockIdx.x and blockIdx.y are the indices that correspond to each GPU block in dimension x and dimension y, respectively.

  $0 \leq$ blockIdx.x, blockIdx.y < 8.

- threadIdx.x is the index thet correspond to each GPU thread inside each GPU block.

  $0 \leq$ threadIdx.x < 128, but only the first n are active threads.

Each GPU block represents the chromosome of a particular individual, and each GPU thread represents a gene on that chromosome (see Figure 3.2). Ujaldón states in [73] that the best fine-grain parallelism is achieved when a single value is assigned to each thread. The GPU used in this work was a GPU Nvidia GeForce GTX 760M with 4 SMX and CUDA Compute Capability (CCC) 3.0. For this CCC, each SMX processes 16 blocks concurrently, and all the individuals of the population were therefore generated simultaneously.

Each individual was randomly generated simultaneously in the corresponding shared memory space of each GPU block (see Figure 3.3).

Algorithm 3 specifies how each individual was generated in each GPU block and copied to the population in the global memory.

3.2 Fitness

The fitness of each individual of the population was evaluated according to the *trace formulation* (Formulation 2.2). This formula takes advantage of the GPU, since its multiprocessing architecture in parallel is ideal for developing algebraic operations of vectors and matrices [73]. The distance and flow matrices of the QAP reside in the constant memory space of the GPU in order to speed up the calculations; constant memory is approximately 100 times faster than global memory.

To deal with large instances of the QAP, a configuration of a two-dimensional GPU grid of size $n \times 8^2$ was used[2], where each GPU block consists of n one-dimensional active GPU threads[3]; in this GPU grid:

[1]Each GPU block consists of 128 GPU threads for efficiency in processing. It is recommended that the number of GPU threads in a GPU block be a multiple of 32, which is the size of a warp, i.e. the set of GPU threads processed simultaneously by a streaming processors [73]. The statement *threadIdx.x* $< n$ in a IF statement in a CUDA code specifies that only the first n threads are active.

[2]n columns and 64 rows, contrary to the orders of matrices of linear algebra.

[3]Again, 128 GPU threads were defined in each GPU block, but only the first n are active.

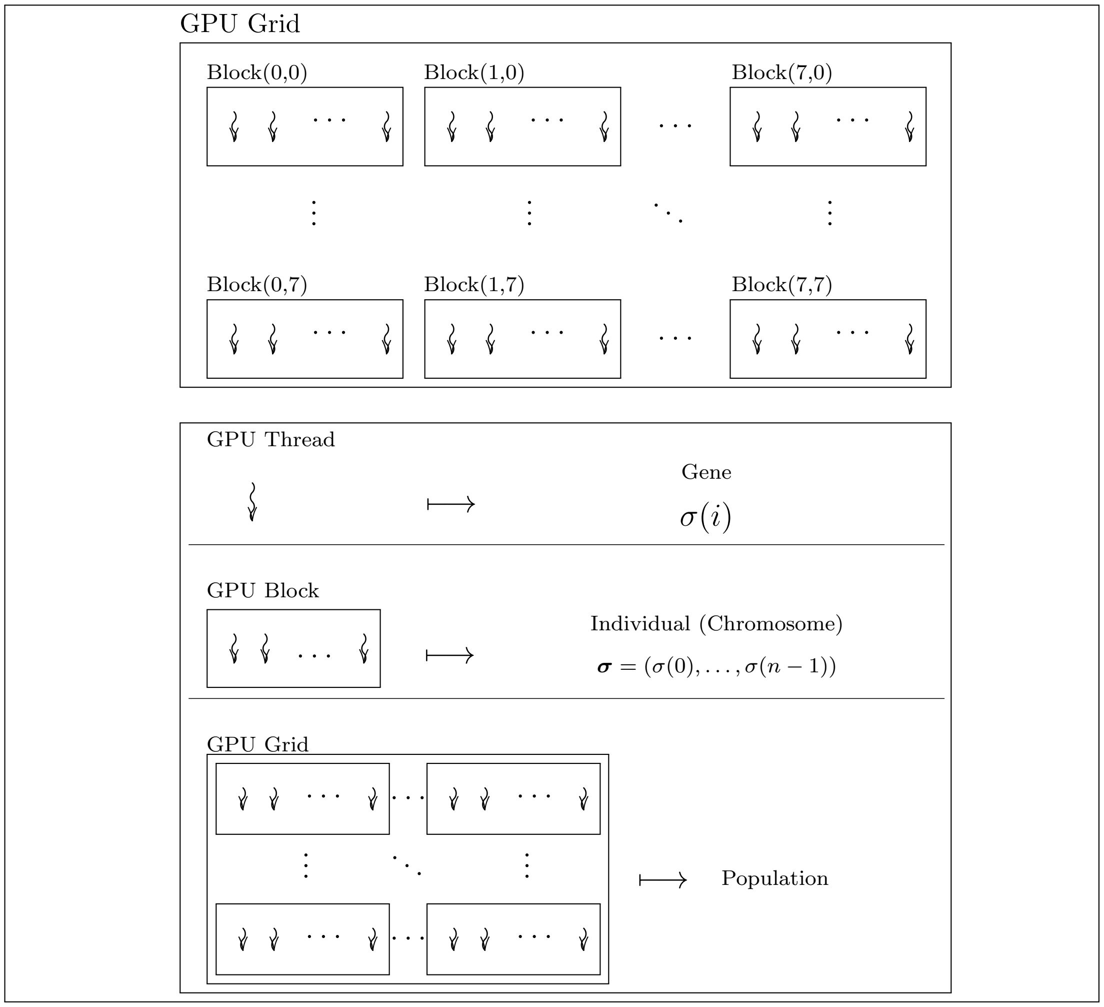


FIGURE 3.2. Population in GPU.

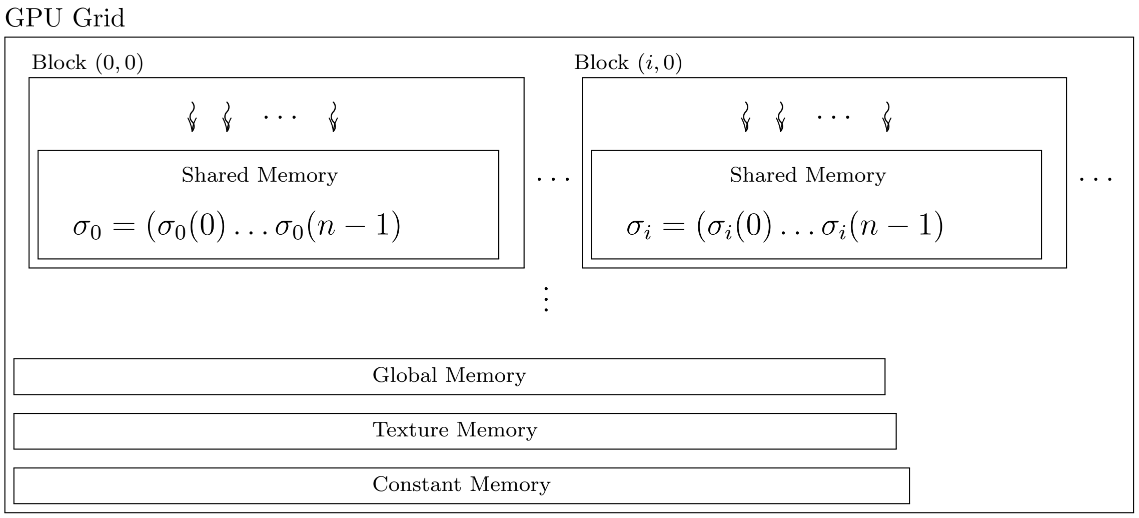


FIGURE 3.3. The ith individual of the population resides in the shared memory space of the ith GPU block.

Algorithm 3 Procedure to generate to generate the initial population in **PGAGrid**

```
$IndexBlock \leftarrow blockIdx.x + blockIdx.y * n$ // is the global index (linearized) of each GPU block in the GPU grid
$IndexThread \leftarrow threadIdx.x + IndexBlock * n$ // is the global index (linearized) of each GPU thread in the GPU grid
$\sigma[threadIdx.x] \leftarrow threadIdx.x$ // $\sigma$ is in shared memory
for $k$ from 0 to $(n-1)$ do
    generate random $i, 0 \leq i < n$
    generate random $j, 0 \leq j < n$
    if $threadIdx.x = 0$ then
        exchange $\sigma(i)$ and $\sigma(j)$
    end if
end for
Synchronize threads // This order ensures a synchronization in the writing
if $threadIdx.x < n$ then
    $Population[IndexThread] \leftarrow \sigma[threadIdx.x]$ // Population in global memory
end if
// The statement $threadIdx.x < n$ in sentence IF specifies that only the first $n$ threads are active
```

$0 \leq$ blockIdx.x $< n$, $0 \leq$ blockIdx.y < 64 and $0 \leq$ threadIdx.x < 128, but only the first n are active GPU threads.

Each row of the GPU grid represents the linearized X permutation matrix associated with each individual of the population, so that the GPU block(i,j), $0 \leq i < n$, $0 \leq j < 64$ represents the ith row of the permutation matrix of the jth individual, denoted $\boldsymbol{x^j_{i\cdot}} = x^j_{i0}, x^j_{i1}, \ldots, x^j_{i(n-1)}$ (see Figure 3.4).

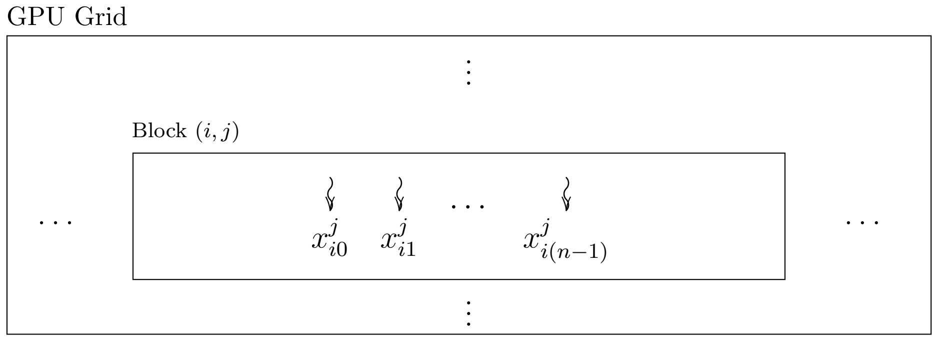


FIGURE 3.4. GPU block(i,j), $0 \leq i < n$, $0 \leq j < 64$ represents the ith row of the permutation matrix of the jth individual, denoted $\boldsymbol{x^j_{i\cdot}} = x^j_{i0}, x^j_{i1}, \ldots, x^j_{i(n-1)}$

To speed up the calculations, each of these rows was processed simultaneously in the corresponding shared memory space of each block. Algorithm 4 shows how the permutation matrices of each of the individuals in the population were obtained.

The same two-dimensional GPU grid of size $n \times 8^2$ was configured to calculate the transposed matrix of each previous permutation matrix, as well as the products involved in the trace formulation. The transposed matrix is calculated from the global memory of the GPU (Algorithm 5).

Algorithm 4 Procedure to generate the permutation matrices of each of the individuals of the population in **PGAGrid**

```
$IndexBlock \leftarrow blockIdx.x + blockIdx.y * n$ // is the global index (linearized) of each GPU block in the GPU grid
$IndexThread \leftarrow threadIdx.x + IndexBlock * n$ // is the global index (linearized) of each GPU thread in the GPU grid
$X_{row}[threadIdx.x] \leftarrow 0$ // $X_{row}$ is a row of the $X$ permutation matrix, $X_{row}$ is in shared memory
$X_{row}[Population[IndexBlock]] \leftarrow 1$ // Population is housed in global memory
if $threadIdx.x < n$ then
    $X[IndexThread] \leftarrow X_{row}[threadIdx.x]$ // $X$ permutation matrix in global memory
end if
```

Algorithm 5 Procedure to generate the transposed matrix

```
$IndexBlock \leftarrow blockIdx.x + blockIdx.y * n$ // is the global index (linearized) of each GPU block in the GPU grid
$IndexThread \leftarrow threadIdx.x + IndexBlock * n$ // is the global index (linearized) of each GPU thread in the GPU grid
if $threadIdx.x < n$ then
    $X^t[IndexThread] \leftarrow X[threadIdx.x * n + blockIdx.y * n^2]$ // $X$ and $X^t$ matrices in global memory
    Synchronize threads // This order ensures a synchronization in the writing
end if
```

For the FX product, each column $\boldsymbol{x^j_{\cdot i}} = x^j_{0i}, x^j_{1i}, \ldots, x^j_{(n-1)i}$ of the permutation matrix of the individual j was loaded into the corresponding shared memory space of the GPU block(i,j), $0 \le i < n$, $0 \le j < 64$. The τ-th GPU thread runs the internal product

$$F_{\tau\cdot} \bullet \boldsymbol{x^j_{\cdot i}} \quad \forall i, j$$

for each of the individuals, simultaneously. ($F_{k\cdot}$ indicates the row k of the matrix F). The result was stored in the element $r^j_{\tau i}$, in a vector $\boldsymbol{r^j_{\cdot i}}$ that was also defined in the corresponding shared memory space. Therefore, the FX product has complexity $O(n)$. Algorithm 6 is a pseudo-code of this procedure.

Algorithm 6 Procedure to calculate the FX product matrix in **PGAGrid**

```
if $threadIdx.x < n$ then
    $sum \leftarrow 0$
    for $i$ from 0 to $(n-1)$ do
        $sum \leftarrow F[threadIdx.x * n + i] * X_{col}[i]$ // $X_{col}$ is each column of each $X$ permutation matrix, $X_{col}$ is in shared memory
        Synchronize threads // This order ensures a synchronization in the writing
    end for
    $r[threadIdx.x] \leftarrow sum$ // $r$ is each column of each $FX$ matrix in shared memory
end if
```

For the D^tX^t product, each row $\boldsymbol{x^j_{i\cdot}}$ was loaded into the corresponding shared memory space of the GPU block(i,j), $0 \le i < n$, $0 \le j < 64$. The τ-th GPU thread runs the internal product

$$D^t_{\tau\cdot} \bullet \boldsymbol{x}^{\boldsymbol{j}}_{\boldsymbol{i\cdot}} \quad \forall i, j$$

for each of the individuals, simultaneously. The result was stored as in the previous product. The complexity of D^tX^t is also $O(n)$.

Finally, for the FXD^tX^t product, the i-th column of the matrix $H = D^tX^t$ corresponding to the j-th individual, denoted $\boldsymbol{h}^{\boldsymbol{j}}_{\boldsymbol{\cdot i}}$, was loaded into the corresponding shared memory space of the GPU block(i,j), $0 \leq i < n,\ 0 \leq j < 64$. The FX linearized matrix of size $n^2{\times}64$ resides in the global memory. The τ-th GPU thread runs the internal product

$$(FX)_{\tau\cdot} \bullet \boldsymbol{h}^{\boldsymbol{j}}_{\boldsymbol{\cdot i}} \quad \forall i, j$$

for each of the 64 individuals. The complexity of FXD^tX^t is also $O(n)$.

The i-th row of the previous matrix, corresponding to the j-th individual, resides in the corresponding shared memory space of the GPU block(i,j), $0 \leq i < n,\ 0 \leq j < 64$. Therefore, for the j-th individual, a j-th vector $(t^j_0, t^j_1, \ldots, t^j_{n-1})$ was formed with the i-th GPU thread of each GPU block(i,j), $0 \leq i < n,\ 0 \leq j < 64$, whose sum is precisely its corresponding fitness (trace of the matrix FXD^tX^t of the j-th individual). This sum was calculated simultaneously for each individual in the corresponding shared memory space of each GPU block of a one-dimensional GPU grid size 64. Each GPU block has 128 threads (n of which are active) (see Figure 3.5).

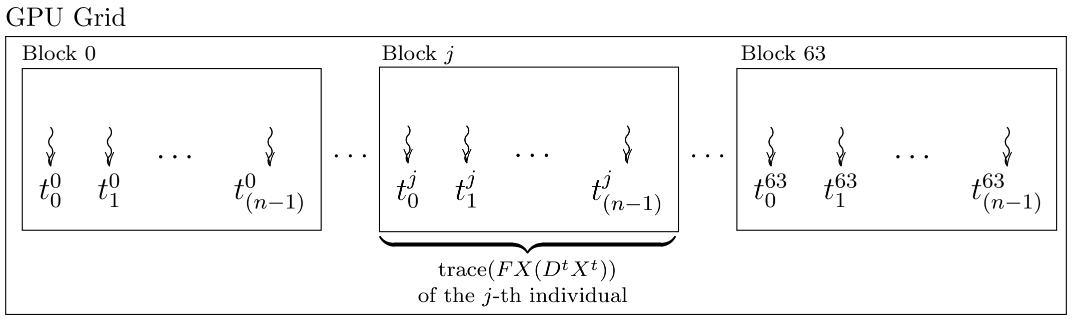

GPU Grid
Block 0
Block j
Block 63
t^0_0 t^0_1 … $t^0_{(n-1)}$
t^j_0 t^j_1 … $t^j_{(n-1)}$
t^{63}_0 t^{63}_1 … $t^{63}_{(n-1)}$
trace($FX(D^tX^t)$)
of the j-th individual

Figure 3.5. Configuration of a grid for the calculation of the trace.

A reduction algorithm was used to obtain the previous sum. The general idea is that the original vector is reduced by half, by adding pairs of components and by storing the result again in the vector the process continues until the final result is stored in the first position of the vector. This algorithm requires the size of the original vector to be a power of 2, something that can always be achieved by completing it with zeros. This procedure has logarithmic complexity. A pseudo-code and an iteration of the reduction algorithm are shown in Algorithm 7 and Figure 3.6, respectively.

After evaluating the fitness of each one of the individuals in parallel, the algorithm continues to select the fittest individuals.

Algorithm 7 Reduction method to add the components of a vector of size m, m power of 2

```
$i \leftarrow m/2$
while $i \neq 0$ do
  if $threadIdx.x < i$ then
    $vector[threadIdx.x] \leftarrow vector[threadIdx.x] + vector[threadIdx.x + i]$
    Synchronize threads  // This order ensures a synchronization in the writing
    $i \leftarrow i/2$
  end if
end while
```

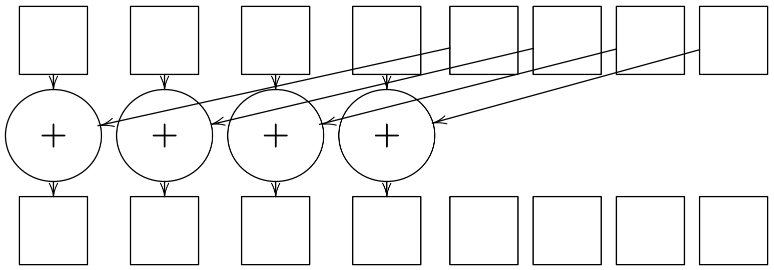


FIGURE 3.6. One step of a summation reduction.

3.3 Selection

The implemented selection consists in an elitist binary tournament. For this, a GPU grid similar to the GPU grid of Figure 3.2 was considered. The corresponding individual of the current population and another individual of the population assigned randomly reside in each GPU block. To ensure that all individuals in the population were randomly assigned in a unique way, the current population was multiplied to the left with a permutation matrix X associated with a $\boldsymbol{\phi}$ permutation of the set $\{0, 1, \ldots, 63\}$, thus:

$$\text{Population}_{\text{permuted}} = X \cdot \text{Population}_{\text{current}} \tag{3.1}$$

Each individual of the permuted population inherits his corresponding fitness, doing:

$$\text{Fitness}_{\text{permuted}} = X \cdot \text{Fitness}_{\text{current}} \tag{3.2}$$

The $\boldsymbol{\phi}$ permutation was generated in the CPU and copied to the global memory of the GPU.

Each row of the permutation matrix X was generated orderly, in each shared memory space of each GPU block of a one-dimensional GPU grid of size 64, (see Figure 3.7); in this GPU grid: $0 \leq$ blockIdx.x < 64 and $0 \leq$ threadIdx.x < 128, but only the first 64 are active threads.

An pseudo-code to calculate these rows is shown in Algorithm 8.

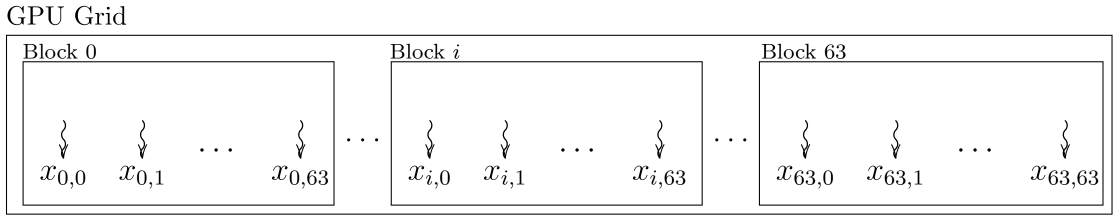

GPU Grid
Block 0
Block i
Block 63
$x_{0,0}$ $x_{0,1}$ … $x_{0,63}$
$x_{i,0}$ $x_{i,1}$ … $x_{i,63}$
$x_{63,0}$ $x_{63,1}$ … $x_{63,63}$

FIGURE 3.7. Permutation matrix generated in GPU. $(x_{i,0}, x_{i,1}, \ldots, x_{i,63})$ is the i-th row of the X permutation matrix.

Algorithm 8 Procedure to generate a permutation matrix for the selection operator in **PGAGrid**

```
$IndexBlock \leftarrow blockIdx.x$ // is the global index (linearized) of each GPU block in the GPU grid
$X_{row}[threadIdx.x] \leftarrow 0$ // $X_{row}$ is a row of the $X$ permutation matrix, $X_{row}$ is in shared memory
$X_{row}[\phi[IndexBlock]] \leftarrow 1$ // Population is housed in global memory
```

Products 3.1 and 3.2 were implemented in parallel in GPU, in the following way: the τ-th GPU thread simultaneously runs the internal product:

$$X_{i\cdot}\bullet(\text{Population})_{\cdot\tau} \quad \forall i,\ 0 \le i < 64 \tag{3.3}$$

and the product:

$$x_\tau \cdot (\text{Fitness})_\tau \tag{3.4}$$

x_j is the j-th element of a row of X (it does not matter which one, since all rows were executed simultaneously), and $(\text{Fitness})_j$ is the fitness of the j-th individual, $0 \le j < 64$.

A pseudo-code for the previous products appears in Algorithms 9 and 10, respectively. Product 3.3 has complexity $O(n)$, while product 3.4 has constant complexity.

Algorithm 9 Procedure to calculate $\text{Population}_{\text{permuted}}$ in **PGAGrid**

```
if $threadIdx.x < n$ then
    $sum \leftarrow 0$
    for $i$ from 0 to $(n-1)$ do
        $sum \leftarrow X[i] * Population[i * n + threadIdx.x]$
        Synchronize threads // This order ensures a synchronization in the writing
    end for
    $S[threadIdx.x] \leftarrow sum$ // $S$ is each row of $\text{Population}_{\text{permuted}}$ matrix in shared memory
end if
```

Finally, a binary tournament was made between the corresponding individuals of each GPU block (GPU grid of Figure 3.8) to build an intermediate population. The current individual is selected if its fitness is better or equal to the fitness of the permuted individual.

Algorithm 10 Procedure to calculate $\text{Fitness}_{\text{permuted}}$ in **PGAGrid**.

```
if $threadIdx.x < 64$ then
    $S[threadIdx.x] \leftarrow X[threadIdx.x] * Fitness[threadIdx.x]$   // $S$ is a vector in shared memory
end if
Apply reduction algorithm to vector $S$ (add its components)   // $S[0]$ is the corresponding fitness of each permuted individual
```

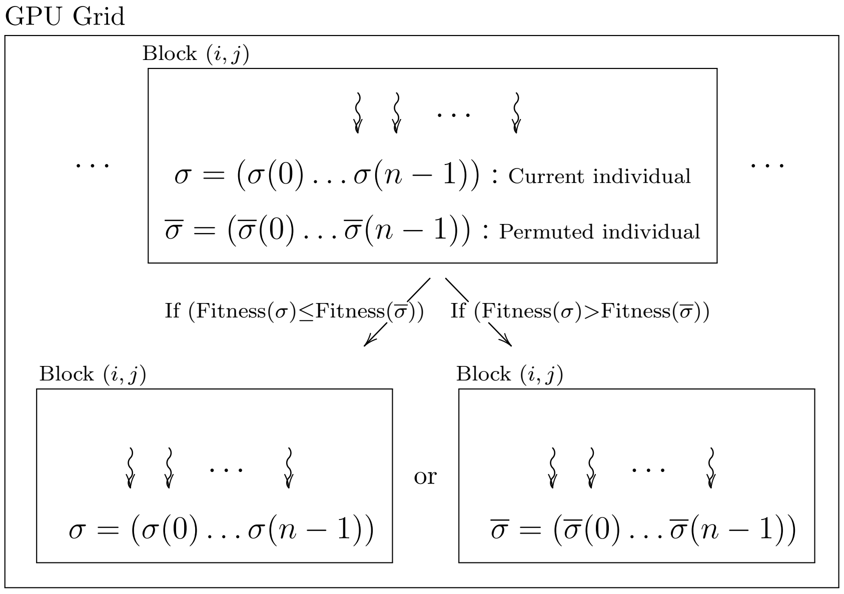

GPU Grid
Block (i, j)
$\sigma = (\sigma(0) \dots \sigma(n-1))$: Current individual
$\overline{\sigma} = (\overline{\sigma}(0) \dots \overline{\sigma}(n-1))$: Permuted individual
If (Fitness(σ)≤Fitness($\overline{\sigma}$))
If (Fitness(σ)>Fitness($\overline{\sigma}$))
Block (i, j)
$\sigma = (\sigma(0) \dots \sigma(n-1))$
or
Block (i, j)
$\overline{\sigma} = (\overline{\sigma}(0) \dots \overline{\sigma}(n-1))$

FIGURE 3.8. Selection by binary tournament on GPU.

3.4 Crossover

The crossover operator implemented was a Modified Order Crossover (MOX). This operator acts as follows: a crossover point that is common to the two parents is randomly selected; the genes to the left of the crossover point of parent 1 are copied to the offspring, and then the remaining genes of the other parent are copied in the order in which they are placed (this avoids repeated genes) (see Figure 3.9).

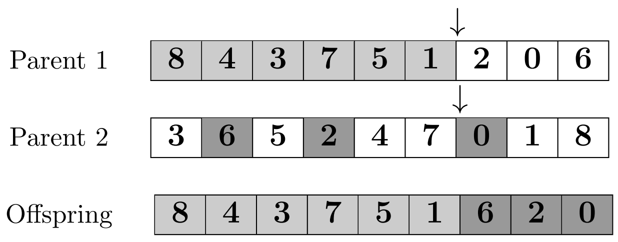
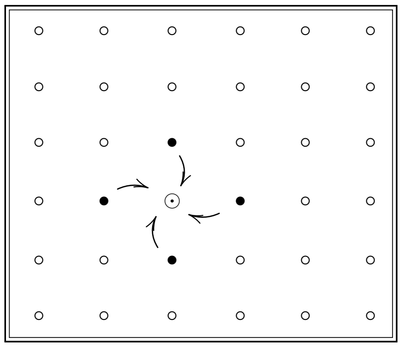


FIGURE 3.9. Example of MOX.

MOX was also implemented at the GPU block level as the selection operator (see Figure 3.2). The current population was permuted as in the selection operator (see Formula 3.1) in order to assign a “couple” to each individual of the current population. This permuted population was generated according to a neighborhood topology.

Four neighborhood topologies were considered in this work, each one represented by a toroidal reticule. These were:

1. Topology 4n or Von Neumann´s topology: horizontal and vertical neighbors (see Figure 3.10 (a)).
2. Topology 8n or Moore´s topology: all neighbors at a distance of one (see Figure 3.10 (b)).
3. Topology 16n: all neighbors at a distance of two (see Figure 3.11 (a)).
4. Topology 20n: equal to topology 4n is joined to topology 16n (see Figure 3.11 (b)).


(a)

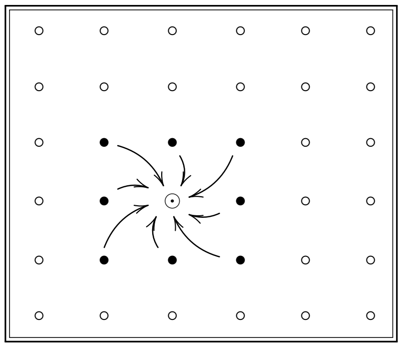
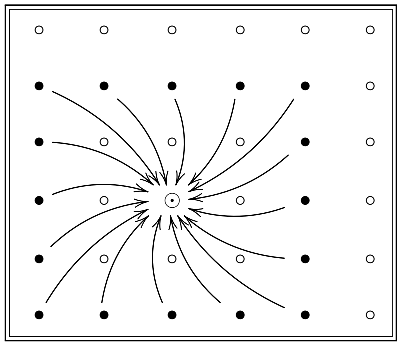

(b)

FIGURE 3.10. (a) Topology 4n, (b) Topology 8n.


(a)

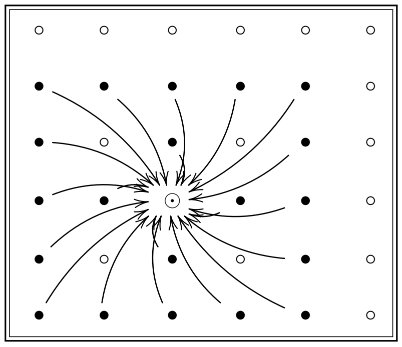

(b)

FIGURE 3.11. (a) Topology 16n, (b) Topology 20n.

For vector $\boldsymbol{\phi}$ generated in CPU, the position i is the reference of the "central individual" (Parent 1) and the content $\phi(i)$ is the reference of the "assigned couple" (Parent 2). More precisely, $\phi(i)$ is the is the reference of the individual with the best fitness in the neighborhood topology (see Figure 3.12).

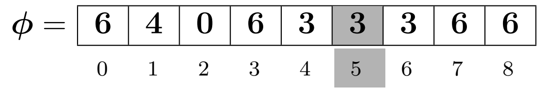


FIGURE 3.12. Couples assigned to the crossover operator. The shaded values mean that the couple of the individual 5 is the individual 3. This individual has the best fitness of some neighborhood topology around individual 5.

Assigning a couple to a central individual according to some topology, with the effect of genetic interaction between individuals from nearby neighborhoods, corresponds to a Fine-Grained Parallel Genetic Algorithm model (Cellular Parallel Genetic Algorithm).

MOX is fully implemented in GPU, as follows:

Each pair of individuals (parent 1 and parent 2) resides in the corresponding shared memory spaces of each GPU block, and they have a probability of crossover $p_c = 0.6$. The crossover point for each pair of individuals is a random value between 0 and $n - 1$.

Initially, the genes (GPU threads) to the left of the crossover point of the first parent are copied simultaneously to the offspring. Then, all the genes (GPU threads) of parent 2 that match the genes of parent 1 before the crossing point are identified and changed to a particular value different from 0 to $n-1$ (for example -1). Figure 3.13 shows an example.

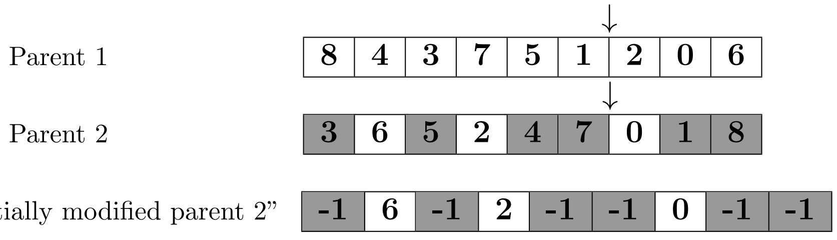


FIGURE 3.13. Construction of the offspring with respect to the genes of the parent 2.

This procedure has a complexity of $O(n)$ at most, since all the threads of parent 2 are simultaneously compared with each of the genes of parent 1 up to the point of crossover, which in the worst case is equal to $n - 1$. Algorithm 11 shows a pseudo-code of this procedure.

Now, all -1 values of the "partially modified parent 2" were moved to the left of the string. This was done by exchanging consecutive contents two to two, one with a value -1 and the other with a value different from -1. Each GPU thread examines them, and if it is the case, it changes such values. Obtaining all values of

Algorithm 11 Procedure to get the "partially modified parent 2" of Figure 3.13.

```
// a different crossing point ($point_c$) for each pair of parents
for $i$ from 0 to $(point_c - 1)$ do
  if $Parent2[threadIdx.x] = Parent1[threadIdx.x]$ then
    $Parent2[threadIdx.x] \leftarrow -1$  // Parent1 and Parent 2 in shared memory
  end if
end for
Synchronize threads  // This order ensures a synchronization in the writing
```

-1 to the left of the string takes at most an number n of iterations, and therefore this procedure has linear complexity. Finally, the genes (GPU threads) to the right of the crossover point of the "Partially modified parent 2" were copied simultaneously to the right of the crossover point of the offspring. A pseudo-code of this last procedure is described in Algorithm 12.

Algorithm 12 Procedure to build offspring from "partially modified parent 2".

```
// a different crossing point ($point_c$) for each pair of parents
$temp[threadIdx.x] \leftarrow 0$ // temp in shared memory
for $i$ from 0 to $n - 1$ do
  if $threadIdx.x < n - 1$ then
    if $(Parent2[threadIdx.x] \neq -1)$ && $(Parent2[threadIdx.x + 1] = -1)$ then
      $temp[threadIdx.x] \leftarrow Parent2[threadIdx.x]$ // Parent1 and Parent2 are in shared memory
      $Parent2[threadIdx.x] \leftarrow Parent2[threadIdx.x + 1]$
      $Parent2[threadIdx.x + 1] \leftarrow temp[threadIdx.x]$
    end if
  end if
end for
if $threadIdx.x \geq point_c$ then
  $Offspring[threadIdx.x] \leftarrow Parent2[threadIdx.x]$ // Offspring in shared memory
end if
Synchronize threads  // This order ensures a synchronization in the writing
```

The offspring replaces parent 1 ("central individual" in the topology of neighborhoods) if and only if the fitness of the offspring is less or equal than the fitness of parent 1 (QAP is a minimization problem). This obeys to a procedure called replacement of neutral mutants. Figure 3.14 shows a sequence of the previous procedures.

It should be emphasized that what was done with a couple of parents occurs simultaneously with all other couples, but under parameters that are not necessarily the same (the croosver may or may not happen in some couples and/or not all couples may have the same crossover point). This happens because each couple of parents resides in a different shared memory space of each GPU block, and a GPU is a SIMD architecture device.

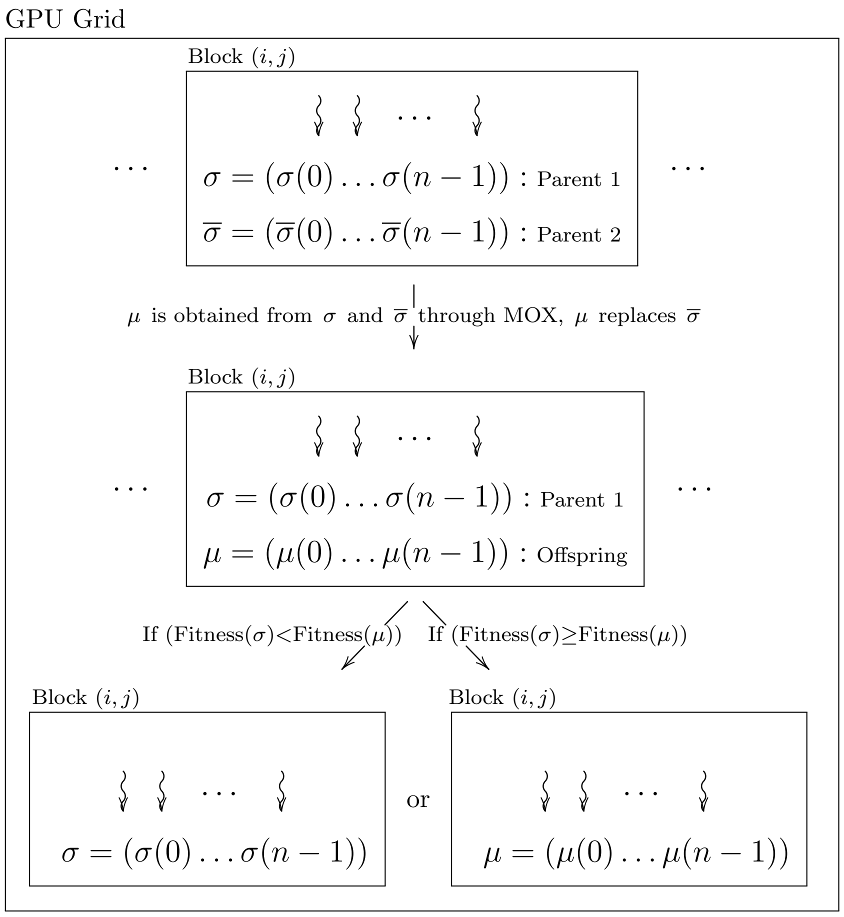

GPU Grid
Block (i,j)
$\sigma = (\sigma(0) \dots \sigma(n-1))$: Parent 1
$\overline{\sigma} = (\overline{\sigma}(0) \dots \overline{\sigma}(n-1))$: Parent 2
μ is obtained from σ and $\overline{\sigma}$ through MOX, μ replaces $\overline{\sigma}$
Block (i,j)
$\sigma = (\sigma(0) \dots \sigma(n-1))$: Parent 1
$\mu = (\mu(0) \dots \mu(n-1))$: Offspring
If (Fitness(σ)<Fitness(μ))
If (Fitness(σ)≥Fitness(μ))
Block (i,j)
$\sigma = (\sigma(0) \dots \sigma(n-1))$
or
Block (i,j)
$\mu = (\mu(0) \dots \mu(n-1))$

FIGURE 3.14. Crossover on GPU.

3.5 Mutation

Before applying the genetic mutation operator, the best individual of the population so far was identified, to be reincorporated later.

To get this best individual, a reduction algorithm was applied for this purpose on a GPU grid of a single GPU block; such a GPU block consists of 128 GPU threads (64 are active threads): blockIdx.x=0 and $0 \leq$ threadIdx.x < 128, but only 64 are active threads (see Figure 3.15).

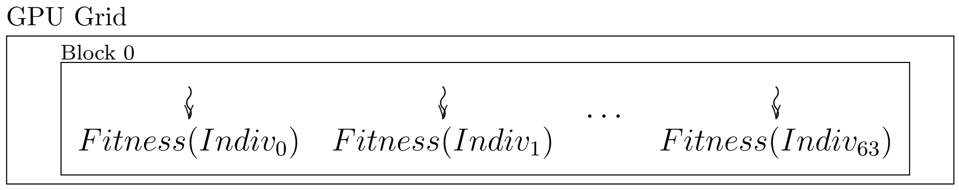


FIGURE 3.15. GPU grid to get the best individual from the population. The i-th thread represents the fitness of the i-th individual of the population.

Like the reduction algorithm used to add the components of a vector, the fitness vector was reduced by half, pairs of components were compared and the best fitness was stored back in the vector. The process continues until the best fitness is stored in the first component of the vector. This reduction algorithm has logarithmic complexity. A pseudo-code of a reduction algorithm to find the minimum value of a vector is shown in Algorithm 13.

Algorithm 13 Reduction method to find the minimum value of Fitness vector size $N = 64$.

```
$i \leftarrow N/2$
**while** $i \neq 0$ **do**
    **if** $threadIdx.x < i$ **then**
        **if** $Fitness[threadIdx.x] > Fitness[threadIdx.x + i]$ **then**
            $Fitness[threadIdx.x] \leftarrow Fitness[threadIdx.x + i]$
        **end if**
        Synchronize threads // This order ensures a synchronization in the writing
        $i \leftarrow i/2$
    **end if**
**end while**
```

After identifying the best individual (and its corresponding fitness), the mutation operator was implemented. The implemented mutation was an Exchange Mutation (EM). For each individual, two genes were selected randomly and exchanged (see Figure 3.16).

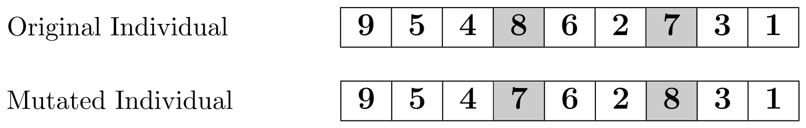


FIGURE 3.16. Example of EM.

The mutation operator was used to generate diversity in the population, with the aim of avoiding premature convergences and stagnation in the local optimum.

The probability of mutation used for each of the individuals was the usual probability in a Genetic Algorithm ($p_m = 0.01$).

EM was implemented in the GPU at the block level, as the previous operators (see Figure 3.2), that is, simultaneously each individual (GPU block) undergoes mutation with the previously described probability, exchanging genes (GPU threads) chosen randomly. Of course, these random genes are not necessarily the same for all individuals. Algorithm 14 presents a pseudo-code of EM.

3.6 Transposition

The last genetic operator implemented in **PGAGrid** was the transposition operator. The transposition operator simply reversed the genes (GPU threads) between

Algorithm 14 EM in **PGAGrid**.

```
generate random $r, r \in (0,1)$ // $r$ is different in each block
if $(threadIdx.x < n)$ &&$(r < p_m)$ then
    generate random $\text{gene1}_m, 0 \leq \text{gene1}_m < n$
    generate random $\text{gene2}_m, 0 \leq \text{gene2}_m < n$
    if $threadIdx.x = 0$ then
        $\sigma(\text{gene1}_m) \longleftrightarrow \sigma(\text{gene2}_m)$
    end if
end if
Synchronize threads // This order ensures a synchronization in the writing
```

two points randomly generated on the chromosome (see Figure 3.17). The probability of transposition used for all individuals in the population was $p_t = 0.4$. This operator was also implemented at the block level (see Figure 3.2), and therefore the transposition was done simultaneously in each individual (GPU block) of the population.

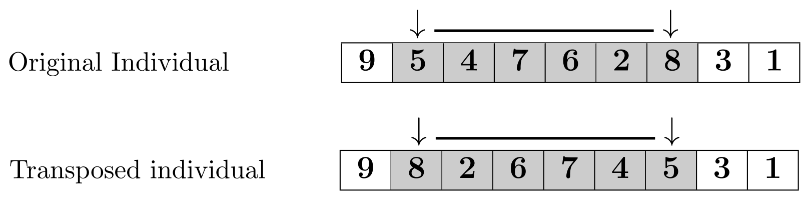


FIGURE 3.17. Example of Transposition.

Algorithm 15 presents a pseudo-code of transposition.

Algorithm 15 Transposition in **PGAGrid**.

```
generate random $r, r \in (0,1)$ // $r$ is not necessarily the same in each GPU block
$\mu[threadIdx.x] \leftarrow \sigma[threadIdx.x]$ // a copy is made to avoid unsynchronization
if $(threadIdx.x < n)$ &&$(r < p_t)$ then
    generate random $\text{point1}_t, 0 \leq \text{point1}_t < n$
    generate random $\text{point2}_t, 0 \leq \text{point2}_t < n$
    if $\text{point1}_t > \text{point2}_t$ then
        $\text{point1}_t \longleftrightarrow \text{point2}_t$
    end if
    if $threadIdx.x \leq (\text{point2}_t - \text{point1}_t)$ then
        $\sigma[threadIdx.x + \text{point1}_t] \leftarrow \mu[\text{point2}_t - threadIdx.x]$
    end if
end if
Synchronize threads // This order ensures a synchronization in the writing
```

Finally, a strategy based on elitism was applied, that is, the best individual identified before applying the mutation and transposition operators, was reincorporated back into the population. Using this strategy, the performance of the Genetic Algorithm was increased, by preventing it from losing the best chromosome found.

3.7 2-opt and greedy 2-opt local optimization heuristics

At this stage, a 2-opt (or greedy 2-opt) local optimization heuristic was applied. This local search heuristic exploits the promising regions already explored by the Genetic Algorithm.

- For the 2-opt local search heuristics, the current individual evaluates the $n(n-1)/2$ swaps (all pairwise exchanges of all possible facilities on each of the locations). Figure 3.18 shows the case for a QAP-instance of size $n = 5$.

  After evaluating all the swaps (which correspond to one iteration within the **PGAGrid** model), the best individual found updates the current individual. Similarly the fitness of the best individual found, updated the fitness of the current individual, Figure 3.19 shows an example for QAP of the Figure 2.1.

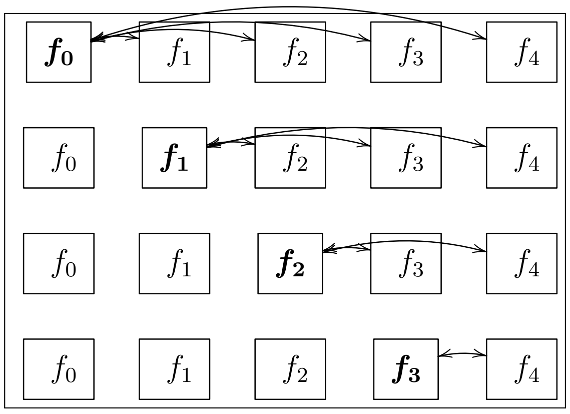

$\boldsymbol{f_0}$ f_1 f_2 f_3 f_4
f_0 $\boldsymbol{f_1}$ f_2 f_3 f_4
f_0 f_1 $\boldsymbol{f_2}$ f_3 f_4
f_0 f_1 f_2 $\boldsymbol{f_3}$ f_4

FIGURE 3.18. 10 possible 2-opt swaps for a QAP of size $n = 5$. f_i is the i-th facility.

- For the greedy 2-opt heuristic, the current individual evaluates the $n(n-1)/2$ swaps, but immediately a better individual is found, this replaces the current individual (of course, fitness is also updated); the greedy 2-opt heuristic continues to be applied on the individual updated in the next swap. An example of this heuristic for QAP of the Figure 2.1 is presented in Figure 3.20.

These heuristics were implemented in a one-dimensional GPU grid of size 64, where each GPU block consists of 128 GPU threads (n are active):

$0 \leq$ blockIdx.x < 64, $0 \leq$ threadIdx.x < 128 (n are active) (see Figure 3.21). Each GPU block corresponds to an individual of the current population and on each one of them the local search heuristic is carried out simultaneously.

Initial Permutation: $\boxed{\boldsymbol{\sigma}_0 = (3\ 2\ 1\ 4\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0) = 256}$

$$\text{Possible swaps (total } n(n-1)/2\text{) :} \left\{ \begin{array}{l} \boldsymbol{\sigma}_0^0 = (2\ 3\ 1\ 4\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0^0) = 214 \\ \boldsymbol{\sigma}_0^1 = (1\ 2\ 3\ 4\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0^1) = 244 \\ \boldsymbol{\sigma}_0^2 = (4\ 2\ 1\ 3\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0^2) = 224 \\ \boldsymbol{\sigma}_0^3 = (0\ 2\ 1\ 4\ 3),\ \text{Fitness}(\boldsymbol{\sigma}_0^3) = 262 \\ \boldsymbol{\sigma}_0^4 = (3\ 1\ 2\ 4\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0^4) = 194 \Leftarrow \\ \boldsymbol{\sigma}_0^5 = (3\ 4\ 1\ 2\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0^5) = 252 \\ \boldsymbol{\sigma}_0^6 = (3\ 0\ 1\ 4\ 2),\ \text{Fitness}(\boldsymbol{\sigma}_0^6) = 210 \\ \boldsymbol{\sigma}_0^7 = (3\ 2\ 4\ 1\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0^7) = 222 \\ \boldsymbol{\sigma}_0^8 = (3\ 2\ 0\ 4\ 1),\ \text{Fitness}(\boldsymbol{\sigma}_0^8) = 228 \\ \boldsymbol{\sigma}_0^9 = (3\ 2\ 1\ 0\ 4),\ \text{Fitness}(\boldsymbol{\sigma}_0^9) = 208 \end{array} \right.$$

Best solution and best fitness updated: $\boxed{\boldsymbol{\sigma}_0 = \boldsymbol{\sigma}_0^4 = (3\ 1\ 2\ 4\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0^4) = 194}$

FIGURE 3.19. Example of a 2-opt implementation for QAP of the Figure 2.1.

The matrix formula used in both heuristics was Formula 2.3, which takes advantage of the features of the GPU, as a multiprocessing vector device. The products of matrices of this formula were calculated as in Algorithm 6.

Algorithms 16 and 17 show the pseudo-codes respectively of these implementations in GPU.

3.8 Summary of the PGAGrid algorithm

PGAGrid is a Fine-Grained Parallel Genetic Algorithm improved with a 2-opt (or greedy 2-opt) local optimization heuristic, fully implemented on GPU, whose purpose is finding optimal (or near-optimal) solutions to large instances of the QAP.

PGAGrid performs an iteration with each of the concepts covered so far in this chapter. The number of iterations was established according to the complexity of each problem analyzed (problems of the standard QAPLIB library, or other problems raised as particular cases of the QAP).

Initial Permutation: $\boxed{\boldsymbol{\sigma}_0 = (3\ 2\ 1\ 4\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0) = 256}$

$$\text{Possible swaps (total } n(n-1)/2) : \left\{ \begin{array}{l}
\boldsymbol{\sigma}_0^0 = (2\ 3\ 1\ 4\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0^0) = 214 \\
\boxed{\boldsymbol{\sigma}_0 = \boldsymbol{\sigma}_0^0 = (2\ 3\ 1\ 4\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0^0) = 214} \\
\boldsymbol{\sigma}_0^1 = (1\ 3\ 2\ 4\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0^1) = 204 \\
\boxed{\boldsymbol{\sigma}_0 = \boldsymbol{\sigma}_0^1 = (1\ 3\ 2\ 4\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0^1) = 204} \\
\boldsymbol{\sigma}_0^2 = (4\ 3\ 2\ 1\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0^2) = 170 \\
\boxed{\boldsymbol{\sigma}_0 = \boldsymbol{\sigma}_0^2 = (4\ 3\ 2\ 1\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0^2) = 170} \\
\boldsymbol{\sigma}_0^3 = (0\ 3\ 2\ 1\ 4),\ \text{Fitness}(\boldsymbol{\sigma}_0^3) = 214 \\
\boldsymbol{\sigma}_0^4 = (4\ 2\ 3\ 1\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0^4) = 212 \\
\boldsymbol{\sigma}_0^5 = (4\ 1\ 2\ 3\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0^5) = 172 \\
\boldsymbol{\sigma}_0^6 = (4\ 0\ 2\ 1\ 3),\ \text{Fitness}(\boldsymbol{\sigma}_0^6) = 192 \\
\boldsymbol{\sigma}_0^7 = (4\ 3\ 1\ 2\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0^7) = 200 \\
\boldsymbol{\sigma}_0^8 = (4\ 3\ 0\ 1\ 2),\ \text{Fitness}(\boldsymbol{\sigma}_0^8) = 266 \\
\boldsymbol{\sigma}_0^9 = (4\ 3\ 2\ 0\ 1),\ \text{Fitness}(\boldsymbol{\sigma}_0^9) = 228
\end{array} \right.$$

Best solution and best fitness updated: $\boxed{\boldsymbol{\sigma}_0 = \boldsymbol{\sigma}_0^2 = (4\ 3\ 2\ 1\ 0),\ \text{Fitness}(\boldsymbol{\sigma}_0^2) = 170}$

FIGURE 3.20. Example of a greedy 2-opt implementation for QAP of the Figure 2.1.

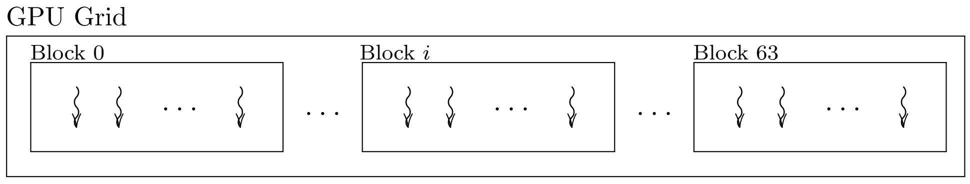


FIGURE 3.21. Configuration of a GPU Grid to implement the 2-opt heuristic. Block i corresponds to the i-th individual of the population, $0 \leq i \leq 63$.

PGAGrid was presented at the 10th International Conference, Computational Collective Intelligence ICCCI 2018, held in the city of Bristol UK, from September 5 to 7, 2018. The article was published by Springer in the series LNCS / LNAI, [54].

The author of this work et. al. published other two papers on the solution of QAP in journals [15] and [20] respectively. Both papers show a particular implementation of the data structure of the Genetic Algorithm with respect to the logical structure of the GPU. Paper [20] presents a more refined parallel implementation than paper [15]. These documents were a basis for finally designing a more robust parallel genetic model, such as **PGAGrid**, with the capacity to solve larger QAP instances.

Algorithm 16 2-opt heuristics in **PGAGrid**.

```
$Index \leftarrow threadIdx.x + n * blockIdx.x$ // is the global index (linearized) of each GPU thread in the GPU grid
$Population_{\text{aux}} \leftarrow Population$ // generates a copy of the population in global memory
$Fitness_{\text{aux}} \leftarrow 0$ // initializes an auxiliary fitness vector to zeros in global memory
for $i$ from 0 to $n-2$ do
    for $j$ from $i+1$ to $n-1$ do
        $Delta[threadIdx.x] \leftarrow 0$ // Delta is in shared memory
        $flag \leftarrow Fitness_{aux}[blockIdx.x]$ // flag in shared memory
        if $threadIdx.x < n$ then
            $\sigma[threadIdx.x] \leftarrow Population_{aux}[Index]$ // $\sigma$ is an individual in shared memory
            // The following vectors: Delta$_F$, Delta$_{F^t}$, Delta$_{DX^t}$, and Delta$_{XD}$ are in shared memory
            $Delta_F[threadIdx.x] \leftarrow F[threadIdx.x + i * n] - F[threadIdx.x + j * n]$
            if $(threadIdx.x = i) \mid\mid (threadIdx.x = j)$ then
                $Delta_F[threadIdx.x] \leftarrow 0$
            end if
            $Delta_{F^t}[threadIdx.x] \leftarrow F[i + threadIdx.x * n] - F[j + threadIdx.x * n]$
            if $(threadIdx.x = i) \mid\mid (threadIdx.x = j)$ then
                $Delta_{F^t}[threadIdx.x] \leftarrow 0$
            end if
            $Delta_{DX^t}[threadIdx.x] \leftarrow DX^t[threadIdx.x + \sigma(j) * n + blockIdx.x * n^2]$ -
                                      $DX^t[threadIdx.x + \sigma(i) * n + blockIdx.x * n^2]$
            $Delta_{XD}[threadIdx.x] \leftarrow XD[\sigma(j) + threadIdx.x * n + blockIdx.x * n^2]$ -
                                      $XD[\sigma(i) + threadIdx.x * n + blockIdx.x * n^2]$
            $Delta[threadIdx.x] \leftarrow (F[j + i * n] - F[i + j * n])*$
                                  $(D[\sigma(i) + \sigma(j) * n] - D[\sigma(j) + \sigma(i) * n])+$
                                  $Delta_F[threadIdx.x] * Delta_{DX^t}[threadIdx.x]+$
                                  $Delta_{F^t}[threadIdx.x] * Delta_{XD}[threadIdx.x]$
            Apply reduction algorithm to vector $Delta$ (add its components)
            if $(Delta(0) < flag)$ && $(threadIdx.x = 0)$ then
                $\sigma(i) \longleftrightarrow \sigma(j)$
                Synchronize threads // This order ensures a synchronization in the writing
                $Fitness_{aux}[blockIdx.x] \leftarrow Delta(0)$ // updates auxiliary fitness in global memory
                $Population_{\text{aux}}[Index] \leftarrow \sigma[threadIdx.x]$ // updates the auxiliary population
                                                                  in global memory
            end if
            Synchronize threads // This order ensures a synchronization in the writing
            if $(i = (n-2))$ && $(j = (n-1))$ then
                $Fitness[blockIdx.x] \leftarrow Fitness[blockIdx.x] + Fitness_{aux}[blockIdx.x]$
                                                                  // updates fitness in global memory
            end if
        end if
    end for
end for
$Population \leftarrow Population_{aux}$ // copy in global memory
```

The improved **PGAGrid** model with the 2-opt (greedy 2-opt) local optimization heuristic is summarized in Algorithm 18.

Algorithm 17 Greedy 2-opt heuristics in **PGAGrid**.

```
$Index \leftarrow threadIdx.x + n * blockIdx.x$ // is the global index (linearized) of each GPU thread in the GPU grid
for $i$ from 0 to $n-2$ do
  for $j$ from $i+1$ to $n-1$ do
    $Delta[threadIdx.x] \leftarrow 0$ // Delta is in shared memory
    if $threadIdx.x < n$ then
      $\sigma[threadIdx.x] \leftarrow Population_{aux}[Index]$ // $\sigma$ is an individual in shared memory
      // The following vectors: Delta$_F$, Delta$_{F^t}$, Delta$_{DX^t}$, and Delta$_{XD}$ are in shared memory
      $Delta_F[threadIdx.x] \leftarrow F[threadIdx.x + i * n] - F[threadIdx.x + j * n]$
      if $(threadIdx.x = i) \mid\mid (threadIdx.x = j)$ then
        $Delta_F[threadIdx.x] \leftarrow 0$
      end if
      $Delta_{F^t}[threadIdx.x] \leftarrow F[i + threadIdx.x * n] - F[j + threadIdx.x * n]$
      if $(threadIdx.x = i) \mid\mid (threadIdx.x = j)$ then
        $Delta_{F^t}[threadIdx.x] \leftarrow 0$
      end if
      $Delta_{DX^t}[threadIdx.x] \leftarrow DX^t[threadIdx.x + \sigma(j) * n + blockIdx.x * n^2] -$
                                    $DX^t[threadIdx.x + \sigma(i) * n + blockIdx.x * n^2]$
      $Delta_{XD}[threadIdx.x] \leftarrow XD[\sigma(j) + threadIdx.x * n + blockIdx.x * n^2] -$
                                    $XD[\sigma(i) + threadIdx.x * n + blockIdx.x * n^2]$
      $Delta[threadIdx.x] \leftarrow (F[j + i * n] - F[i + j * n]) *$
                                  $(D[\sigma(i) + \sigma(j) * n] - D[\sigma(j) + \sigma(i) * n]) +$
                                  $Delta_F[threadIdx.x] * Delta_{DX^t}[threadIdx.x] +$
                                  $Delta_{F^t}[threadIdx.x] * Delta_{XD}[threadIdx.x]$
      Apply reduction algorithm to vector Delta (add its components)
      if $(Delta(0) < 0)$ && $(threadIdx.x = 0)$ then
        $\sigma(i) \longleftrightarrow \sigma(j)$
        Synchronize threads // This order ensures a synchronization in the writing
        $Fitness[blockIdx.x] \leftarrow Fitness[blockIdx.x] + Delta(0)$ // updates fitness in global memory
      end if
      $Population[Index] = \sigma[threadIdx.x]$ // updates the population in global memory
      Synchronize threads // This order ensures a synchronization in the writing
    end if
  end for
end for
```

3.9 Combinatorial problems solved from a QAP by PGAGrid

3.9.1 Solution of some NP-Complete problems formulated as QAPs

As discussed in subsection 2.1.3, many NP-complete combinatorial problems result in particular cases of QAP, defining the flow and distance matrices properly. The

Algorithm 18 The Model **PGAGrid** implemented on GPU.

```
for each GPU block $i$ in the GPU grid, in parallel do
    Assign a random individual
end for
generation number $\longleftarrow$ 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 a individual $k$ in the neighboring 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 2-opt (or greedy 2-opt) local optimization heuristic
    end for
    generation number $\longleftarrow$ generation number+1
end while
```

PGAGrid model was used to solve different instances of each of the three NP-complete problems (*TSP*, *LAP* and *MCP*). The chosen instances correspond to benchmark problems found in the literature.

The solution of these problems by **PGAGrid** can be consulted in [52].

3.9.2 Classic $k \times k$ chessboard problems solved as a QAP by PGAGrid

Some significant classic problems of the $k \times k$ chessboard, can be treated as QAPs and therefore the **PGAGrid** model becomes an option for your solution. The four problems posed as a QAP and solved by **PGAGrid** are: *the Knight's Tour Problem, the Bishop Problem, the k Rooks Problem*, and *the k Queens Problem.*

The solution of these problems by **PGAGrid** can be consulted in [53].

CHAPTER 4

Experimentation and Result Obtained

For testing **PGAGrid** a GPU Nvidia GeForce GTX 760M with 768 CUDA cores was used on an IntelCore^{TM} i7 - 4700HQ CPU @ 2.40GHz, RAM 8GB. **PGAGrid** was written and compiled using CUDA toolkit (v8.0) for C and executed under Windows 10.

Section 4.1 lists and explains each of the ten instances taken from the QAPLIB library that were considered in the **PGAGrid** model.

Section 4.2 describes the parameters used in the **PGAGrid** model to solve each of the instances listed in the previous section. The parameters were: population size, crossover rates, mutation rates and transposition rates, number of tests for each instance, and maximum number of iterations in each of the tests for each instance.

Section 4.3 shows the results obtained by the **PGAGrid** by combining the two different local search heuristics implemented (2-opt and greedy 2-opt) on the four different neighborhood topologies described in section 3.4. This section also shows the results obtained by a similar implementation in CPU. These results were compared with those obtained by the **PGAGrid** model. Finally, some results reported in the literature are shown.

Section 4.4 performs a Wilcoxon signed-rank test to establish the topology, and the local search heuristic with which the best results were obtained.

4.1 Test sets

Ten widely used instances from the standard QAPLIB library [55] were examined, these are:

- Els19: [21]. “The data describe the distances of nineteen different facilities of a hospital and the flow of patients between those locations. The optimal solution was first found by [39]”. It is the only instance of this type of problems

- Esc64: [22]. “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: [32]. “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 $[1, n]$. The proof of optimality of the solution is due to [12]”. It is the largest instance of this type of problems.

- Kra32: [34]. “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: [44]. “The distance matrix contains Manhattan distances of rectangular grids. The solution was found by applying a branch and bound algorithm [11]”. It is the largest instance of this type of problems.

- Scr20: [61]. “The distances of these problems are rectangular. The optimal solution was found by [39]”. It is the largest instance of this type.

- Tai35b: [65]. “The problem were introduced in [65]. This problem is asymmetric and randomly generated”. It is the sixth largest instance of this type of problems.

- Tai40b: [65]. “The problem were introduced in [65]. This problem is asymmetric and randomly generated”. It is the fifth largest instance of this type of problem.

- Tai60b: [65]. “The problem were introduced in [65]. This problem is asymmetric and randomly generated”. It is the third largest instance of this type of problem.

- Tho40: [74]. “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.

4.2 Parameters used by PGAGrid

The population consists of 64 individuals (see figure 3.2). The genetic operators were tuned. For each couple of individuals (in the same GPU block), the same crossover rate was fixed: $p_c = 0.6$. Each individual has the same mutation rate

($p_m = 0.01$) and the same transposition rate ($p_t = 0.4$). Each rate for each genetic operator remains constant throughout the execution of the **PGAGrid**.

2-opt and greedy 2-opt local search techniques always make the $n(n-1)/2$ swaps in each iteration of **PGAGrid**.

PGAGrid conducted ten tests for each instance in relation to each of the four topologies listed in the previous chapter. The maximum number of iterations for each instance was one hundred.

4.3 Results

Tables 4.1 and 4.2 show the performance of **PGAGrid** 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. **PGAGrid** *always found the optimal solution* for these three problems independent of these configurations.

Table 4.1. **Performance of PGAGrid to find a optimal solution, using a 2-opt heuristic.**

QAP	Topology 4n	Topology 8n	Topology 16n	Topology 20n
Els19	12.0±2.0	12.5±1.5	11.5±1.5	23.0±11.0
Esc64a	9.0±1.0	9.5±0.5	9.0±0.5	10.0±0.0
Had20	22.0±6.5	17.5±3.5	31.0±16.5	15.0±4.5

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

Table 4.2. **Performance of PGAGrid to find a optimal solution, using a greedy 2-opt heuristic.**

QAP	Topology 4n	Topology 8n	Topology 16n	Topology 20n
Els19	5.5±1.5	6.0±2.0	4.5±1.5	5.5±1.5
Esc64a	1.0±0.0	1.0±0.0	1.0±0.0	1.0±0.0
Had20	4.0±1.0	3.0±1.0	4.0±1.0	3.5±0.5

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

Tables 4.3 and 4.4 show the performance of **PGAGrid** (arithmetic average and standard 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, combining heuristic implementations (2-opt and greedy 2-opt).

Tables 4.5 and 4.6 show the performance of **PGAGrid** (median and median absolute deviation) for these same seven instances (Kra32, Nug30, Scr20, Tai35b, Tai40b, Tai60b, and Tho40) in relation to the *solutions found* in the executions for each topology, combining heuristic implementations (2-opt and greedy 2-opt).

Table 4.3. **Performance of PGAGrid, using a 2-opt heuristic.**

QAP	QAPLIB	Topology 4n	Topology 8n	Topology 16n	Topology 20n
Kra32	**88700**	91246±538	91013±527	91350±973	91454±674
Nug30	**6124**	6236±37	6203±30	6217±27	6253±25
Scr20	**110030**	111241±890	111158±1000	111742±1421	111478±917
Tai35b	**283315445**	286109725±1473358	285418088±1405123	286117941±1851034	285214600±1558527
Tai40b	**637250948**	654130538±12522178	655960058±10171089	663307321±16281696	661450966±12210870
Tai60b	**608215054**	627301589±8096733	629065013±7732720	622263727±8885398	625016351±5935946
Tho40	**240516**	247264±1613	246402±1380	246003±2705	246070±1774

Value $x \pm y$ indicates an arithmetic average of x (solution found) with a standard deviation of y.

Table 4.4. **Performance of PGAGrid, using a greedy 2-opt heuristic.**

QAP	QAPLIB	Topology 4n	Topology 8n	Topology 16n	Topology 20n
Kra32	**88700**	88922±470	89006±493	89227±566	89220±557
Nug30	**6124**	6136±10	6130±6	6132±12	6134±12
Scr20	**110030**	110065±101	110030±0	110245±347	110097±203
Tai35b	**283315445**	283455144±230811	283455144±230811	283527059±273193	283508047±254633
Tai40b	**637250948**	637250948±0	637262177±23672	637273405±28992	638543295±4047374
Tai60b	**608215054**	608562201±702305	608254081±87568	608301697±89972	608313952±136515
Tho40	**240516**	241504±512	241212±537	241522±914	241034±294

Value $x \pm y$ indicates an arithmetic average of x (solution found) with a standard deviation of y.

The corresponding experiments on a CPU implementation were also carried out. This sequential implementation has the same characteristics as the parallel implementation of **PGAGrid**, but it does not consider any of the previous topologies, only, couples generated with a random order. Ten tests were also conducted for each QAP-instance with a maximum of one hundred iterations.

Table 4.5. **Performance of PGAGrid, using a 2-opt heuristic.**

QAP	QAPLIB	Topology 4n	Topology 8n	Topology 16n	Topology 20n
Kra32	**88700**	91155±455	91105±350	91430±600	91730±410
Nug30	**6124**	6245±24	6195±22	6214±17	6260±10
Scr20	**110030**	111174±923	110878±848	112163±1310	111298±681
Tai35b	**283315445**	285624601±981047	285165264±620114	285446919±1246283	285117037±351396
Tai40b	**637250948**	658136188±9267476	659679797±6701287	661970688±13176897	666262567±7899045
Tai60b	**608215054**	627874603±5834722	629065013±7732720	621929415±8167650	622898346±3299634
Tho40	**240516**	247299±879	246359±938	246116±2515	245540±632

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

Table 4.6. **Performance of PGAGrid, using a greedy 2-opt heuristic.**

QAP	QAPLIB	Topology 4n	Topology 8n	Topology 16n	Topology 20n
Kra32	**88700**	88700±0	88700±0	89100±400	89100±400
Nug30	**6124**	6132±6	6128±0	6128±0	6128±4
Scr20	**110030**	110030±0	110030±0	110030±0	110030±0
Tai35b	**283315445**	283315445±0	283315445±0	283315445±0	283315445±0
Tai40b	**637250948**	637250948±0	637250948±0	637250948±0	637250948±0
Tai60b	**608215054**	608228619±13565	608228578±9633	608290526±74005	608231469±16415
Tho40	**240516**	241524±370	241130±381	241144±439	241019±201

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

Table 4.7 presents the performance of the CPU implementation for problems Els19, Esc64a, and Had20, in relation to the *number of iterations* where the optimal solution was found in the executions, combining heuristic implementations (2-opt and greedy 2-opt). In addition, this CPU implementation always found the optimal solution for these three problems independently of the configurations.

TABLE 4.7. **Performance of a CPU implementation to find the optimal solution, combining heuristic.**

QAP	2-opt	greedy 2-opt
Els19	19.5 ± 6.5	7.0 ± 3.0
Esc64a	10.5 ± 0.5	1.0 ± 0.0
Had20	58.0 ± 29.0	6.5 ± 2.5

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

Table 4.8 presents the performance of the CPU implementation (arithmetic average and standard deviation) for the remaining seven problems (Kra32, Nug30, Scr20, Tai35b, Tai40b, Tai60b, and Tho40) in relation to the *solutions found* in the executions, combining heuristic implementations (2-opt and greedy 2-opt).

TABLE 4.8. **Performance of a CPU implementation to find the optimal solution, combining heuristic.**

QAP	QAPLIB	2-opt	greedy 2-opt
Kra32	**88700**	94385±1847	88780±253
Nug30	**6124**	6260±56	6128±4
Scr20	**110030**	112106±1312	110033±9
Tai35b	**283315445**	287208985±2458063	283748455±260064
Tai40b	**637250948**	659513550±15585339	637329374±58739
Tai60b	**608215054**	646733519±14432654	608826757±338208
Tho40	**240516**	249761±2199	241871±670

Value $x \pm y$ indicates an arithmetic average of x (solution found) with a standard deviation of y.

Table 4.9 presents the performance of the CPU implementation (median and median absolute deviation) for these same seven instances (Kra32, Nug30, Scr20, Tai35b, Tai40b, Tai60b, and Tho40) in relation to the *solutions found* in the executions, combining heuristic implementations (2-opt and greedy 2-opt).

TABLE 4.9. **Performance of a CPU implementation to find the optimal solution, combining heuristic.**

QAP	QAPLIB	2-opt	greedy 2-opt
Kra32	**88700**	95025±1030	88700±0
Nug30	**6124**	6250±39	6128±2
Scr20	**110030**	112113±1029	110030±0
Tai35b	**283315445**	286884376±2049354	283725417±167022
Tai40b	**637250948**	660191766±113462272	637307091±11767
Tai60b	**608215054**	640337053±4217663	608823261±124250
Tho40	**240516**	249642±1318	242038±103

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

Table 4.10 presents the best known solution for each instance reported by QAPLIB, the best solution obtained by **PGAGrid** in all the tests carried out (eighty in total for each instance), and the best solution obtained by a CPU implementation in all the tests carried out (twenty in total for each instance).

Table 4.10. **The best solutions found by PGAGrid against the best solutions found by sequential implementation**

QAP	QAPLIB	**PGAGrid**	CPU
Els19	17212548	17212548 (2)	17212548 (2)
Esc64a	116	116 (1)	116 (1)
Had20	6922	6922 (2)	6922 (2)
Kra32	88700	88700 (5)	88700 (13)
Nug30	6124	6124 (14)	6124 (27)
Scr20	110030	110030 (2)	110030 (4)
Tai35b	283315445	283315445 (12)	283315445 (42)
Tai40b	637250948	637250948 (6)	637250948 (15)
Tai60b	608215054	608215054 (42)	608352432 (81)
Tho40	240516	240542 (76)	240632 (85)

The value in parentheses correspond to the minimum number of iterations in which these solutions were reached.

The best results obtained by **PGAGrid** coincide with the results referenced in QAPLIB, except in problem Tho40. The number of iterations in which **PGAGrid** reached these results is relatively small compared to the sequential implementation.

Tables 4.11, 4.12, and 4.13 show some results referenced by other researchers on the QAP-instances considered. They use either exact methods or approach techniques, or a hybrid between them: in some cases, they also perform an implementation on GPU.

Table 4.11. **Results reported in the literature.**

QAP	Ramkumar [57]	Mohassesian [41]	Chaparala [43]	Chmiel [17]
Els19	17212548	17212548	-	17212548
Esc64a	-		-	132
Had20	6922	6922	-	6922
Kra32	88760	88700	-	88700
Nug30	6146	6130	-	6124
Scr20	110030	110030	-	110030
Tai35b	297719554	286605043	283349722	-
Tai40b	716675355	637250948	637349459	-
Tai60b	631700959	608215054	609612341	-
Tho40	242888	241912	-	240516

Ramkumar et. al. [57] propose a modified iterated fast local search algorithm with a crossover recombination as a perturbation strategy and a self-improvement in the

Table 4.12. **Results reported in the literature.**

QAP	Ziqiang [78]	Bashiri [5]	Semlali [23]	Gunawan [31]
Els19	-	-	-	-
Esc64a	-		-	
Had20	-	-	-	2927.2
Kra32	-	-	-	88700
Nug30	-	6180	-	2124.4
Scr20	-	-	-	110030
Tai35b	-	-	286661392	-
Tai40b	637428558.1	-	649512197	-
Tai60b	-	-	-	-
Tho40	-	-	-	-

Table 4.13. **Results reported in the literature.**

QAP	Mohammadi [40]	Szwed [63]	Fomeni [24]	Zhang [77]
Els19	-	-	-	-
Esc64a	-	116	116	-
Had20	-	-	6922	-
Kra32	-	-	91760 -	93000
Nug30	-	-	6180	6264
Scr20	-	-	110676 -	110030
Tai35b	284703248	-	-	-
Tai40b	647201580	-	-	-
Tai60b	624137807	612078720	-	-
Tho40	-	-	-	-

mutation operator followed by a local search. They perform 10 executions on each problem, each with 220 iterations. The algorithm was executed in CPU.

Mohassesian et al. [41] improve the algorithm proposed by Ramkumar et. al. 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.

Chaparala et. al. [43] implemented a 2-opt local search heuristic over the GPU, configuring different thread numbers per block.

Chmiel et. al. [17] execute an improved ant algorithm with a 2-opt neighborhood technique. Ten executions were made for each instance, oscillating between 100 and 1000 iterations. The algorithm was executed in CPU.

Ziqiang et. al. [78] design a hybrid algorithm between schools of fish and differential evolution called HAFSOA. The algorithm is sequential, and 20 executions were performed for the instances considered, each with 500 iterations.

Bashiri et. al. [5] implement some metaheuristics and heuristics (TS, SA, PSO, 2-opt, greedy 2-opt, 3-opt and greedy 3-opt) and compare their solutions. Each of these techniques was implemented in CPU. The heuristic that provided the best solution was 2-opt. The only problem that matches the one implemented in the **PGAGrid** model, required 1000 iterations to obtain the reported solution. The worst solution found by the authors for most of the problems, in contrast to **PGAGrid**, was provided by the greedy 2-opt heuristic; therefore, it can be argued that the greedy 2-opt technique works well if a metaheuristic (such as a Genetic Algorithm) is used as a perturbation strategy on local solutions.

Semlali et al. [23] present 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.

Gunawan et. al. [31] apply a hybrid metaheuristic, using GRASP to build an initial population, and SA and TS to improve the solution. Each problem was executed 20 times, and the number of iterations was $300n$, where n was the size of the problem. The implementation is executed in CPU, and the authors state that they do not report computational time due to space limitations.

Mohammadi et. al. [40] propose 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. [63] implement 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 (which match **PGAGrid** instances) were solved in 71 and 2220 iterations respectively—too many compared to **PGAGrid** solutions.

Fomeni [24] puts forward a thesis with a new proposal for the QAP solution. The author implements in CPU a deterministic method based on a standard quadratic integer programming formulation called “auxiliary function-based dynamic convexized method”.

Zhang et. al. [77] resort to a general purpose mixed integer linear programming solver to find a deterministic solution to QAP instances of size between twelve and thirty-two. They considered three linearizations:—Gilmore-Lawler Bound, Kaufman-Broeckx, and Xia-Yuan—. The latter had the best results, although with a longer processing time. The results presented in table 4.10 refer to the Xia-Yuan linearization. These implementations were executed in CPU.

4.4 Analysis of results

The results obtained were analyzed with a non-parametric Wilcoxon signed rank test. This test compares the median of two related samples and determines if there is a difference between them [69].

The null hypothesis is $H_0 : \widetilde{\mu_1} = \widetilde{\mu_2}$, that is, the median of the differences of the samples of the paired data is equal to zero. The alternative hypothesis is $H_1 : \widetilde{\mu_1} \neq \widetilde{\mu_2}$, that is, the median of the population of differences is not equal to zero.

The median of two samples was compared to determine the best neighborhood topology and the best local search heuristic for each of the 10 instances examined. The significance level used was 0.05.

Table 4.14 specifies the topologies that predominated over the others, first in relation to the 2-opt heuristic and then to the greedy 2-opt heuristic.

TABLE 4.14. **Comparison of neighborhood topologies according to local search heuristics.**

QAP	2-opt	Greedy 2-opt
Els19	Topology 4n $\gg$ Topology 20n	-
	Topology 8n $\gg$ Topology 20n	-
	Topology 16n $\gg$ Topology 20n	-
Esc64a	-	-
Had20	-	-
Kra32	-	-
Nug30	Topology 8n >> Topology 4n	-
	Topology 8n $\gg$ Topology 20n	-
	Topology 16n $\gg$ Topology 20n	-
Scr20	-	-
Tai35b	-	-
Tai40b	-	-
Tai60b	-	Topology 8n $\gg$ Topology 4n
Tho40	-	-

$A \gg B$ means that the results that come from sample A are better than the results that come from sample B.

It can be seen, that in the problems where one topology predominated over the other, the most outstanding was the topology 8n and the less outstanding the topology 20n.

Table 4.15 shows the comparison between the 2-opt and greedy 2-opt heuristics.

The results obtained by **PGAGrid** using the greedy 2-opt heuristic were always better than the results obtained by **PGAGrid** using the 2-opt heuristic for each of the 10 instances considered.

Finally, the median of the results obtained in each of the neighborhood topologies was compared with the median of the results obtained in the sequential implementation, also, with a level of significance of 0.05. Table 4.16 shows this comparison in relation to the 2-opt and greedy 2-opt heuristics respectively.

Neighborhood topologies implemented in GPUs yielded superior results compared to CPU implementation in almost all test instances. Again, the 8n topology was the

Table 4.15. **Comparison between local search heuristics.**

QAP	Heuristics
Els19	Greedy 2-opt $\gg$ 2-opt
Esc64a	Greedy 2-opt $\gg$ 2-opt
Had20	Greedy 2-opt $\gg$ 2-opt
Kra32	Greedy 2-opt $\gg$ 2-opt
Nug30	Greedy 2-opt $\gg$ 2-opt
Scr20	Greedy 2-opt $\gg$ 2-opt
Tai35b	Greedy 2-opt $\gg$ 2-opt
Tai40b	Greedy 2-opt $\gg$ 2-opt
Tai60b	Greedy 2-opt $\gg$ 2-opt
Tho40	Greedy 2-opt $\gg$ 2-opt

$A \gg B$ means that the results that come from sample A are better than the results that come from sample B.

Table 4.16. **Comparison between neighborhood topologies and CPU implementation.**

QAP	2-opt	greedy 2-opt
Els19	Top. $i \gg$ CPU, $i = 4n, \mathbf{8n}, 16n$	-
Esc64a	Top. $i \gg$ CPU, $i = 4n, \mathbf{8n}, 16n, 20n$	Top. $i \gg$ CPU, $i = 4n, \mathbf{8n}, 16n, 20n$
Had20	Top. $i \gg$ CPU, $i = \mathbf{8n}, 20n$	Top. $i \gg$ CPU, $i = 4n, \mathbf{8n}, 16n, 20n$
Kra32	Top. $i \gg$ CPU, $i = 4n, \mathbf{8n}, 16n, 20n$	Top. $i \gg$ CPU, $i = 16n$
Nug30	Top. $i \gg$ CPU, $i = \mathbf{8n}, 16n, 20n$	Top. $i \gg$ CPU, $i = 4n, \mathbf{8n}$
Scr20	Top. $i \gg$ CPU, $i = \mathbf{8n}$	Top. $i \gg$ CPU, $i = 4n, \mathbf{8n}, 16n, 20n$
Tai35b	Top. $i \gg$ CPU, $i = \mathbf{8n}, 20n$	Top. $i \gg$ CPU, $i = 4n, \mathbf{8n}$
Tai40b	-	Top. $i \gg$ CPU, $i = 4n, \mathbf{8n}, 16n$
Tai60b	Top. $i \gg$ CPU, $i = 4n, \mathbf{8n}, 16n, 20n$	Top. $i \gg$ CPU, $i = \mathbf{8n}, 16n, 20n$
Tho40	Top. $i \gg$ CPU, $i = 4n, \mathbf{8n}, 16n, 20n$	Top. $i \gg$ CPU, $i = \mathbf{8n}, 20n$

$A \gg B$ means that the results that come from sample A are better than the results that come from sample B.

most prominent neighborhood topology. This happened in nine of ten test instances, both in the 2-opt heuristic and in the greedy 2-opt heuristics.

Conclusions and Future Works

Solving the QAP is not an easy task. The **PGAGrid** model designed and implemented in this work is an important technique to find optimal or near optimal solutions for significant instances of the QAP.

The **PGAGrid** model is based on an appropriate configuration of a Graphical Processing Unit (GPU). The GPU was configured in such a way, that a two-dimensional GPU grid corresponds to the population of the genetic algorithm, a GPU block corresponds to an individual (chromosome) of the population and a GPU thread to a gene of such a chromosome. Configuring the GPU in this form allowed to manipulate in a simpler way the data structure of the fine-grained parallel genetic algorithm. This configuration allowed, in the same way, to take advantage of the different memory spaces of the GPU to accommodate the structures of the genetic algorithm and speed up the processing calculations.

An alternative formulation of the QAP, as a matrix function (trace formulation) in relation to its original formulation was important in the implementation of the **PGAGrid** model. This formulation exploited the characteristics of the GPU as a parallel multiprocessing vector device.

The type of selection used (selection by binary tournament), was appropriate to be implemented in parallel and therefore, in a GPU. The selection was made simultaneously in the corresponding shared memory spaces of each of the GPU blocks where the individuals that are contrasted reside.

The configuration of the GPU was important at the time of applying the genetic crossing, mutation and transposition operators. These genetic operators were implemented at the block level in the GPU, that is, each operator was applied simultaneously to each of the individuals in the population.

The crossover operator was completely implemented in parallel, unlike all implementations of similar jobs reported in the literature. The crossover operator resorted to one of four neighborhoods topologies, in order to effect a diffusion of highlighting genetic material among individuals of the population. The use of neighborhood topologies was significant, highlighting Moore's topology over the others.

A strategy based on elitism was applied in each iteration of the **PGAGrid** model. This strategy was important, since it prevented the genetic model from losing the best chromosome found before applying the mutation and transposition genetic operators.

Finally, two local search heuristics were implemented with the purpose of carrying out a meticulous genetic exploitation of the spaces previously explored by the fine-grained parallel genetic algorithm. The matrix formulation of the incremental function of the 2-opt (greedy 2-opt) heuristic was important because it took advantage of the characteristics of the GPU as a vector device.

The greedy 2-opt heuristic was the most outstanding to improve the solutions found by the genetic algorithm. **PGAGrid** was completely implemented with CUDA on a Graphical Processing Unit (GPU), eliminating data transfer bottlenecks between the main memory of the PC and the main memory of the GPU.

PGAGrid was examined on ten different benchmark problems of the literature (instances of the QAPLIB reference source) but it could well have been verified its efficiency with any other problem even of bigger size.

The **PGAGrid** model was faster than the sequential algorithm in CPU. In addition the results obtained were also much better. Also, the results obtained by the **PGAGrid** model were in most cases better than the results referenced in other investigations.

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 perhaps for this it is convenient to configure a cluster of GPUs or combine procedures in multicore architectures.

Bibliography

[1] E. Aarts and J. Lenstra, *Local search in combinatorial optimization*, Wiley, Chishester (1997).

[2] H. Liu A. Abraham and J. Zhang, *A particle swarm approach to quadratic assignment problems*, Soft Computing in Industrial Applications. Springer Berlin Heidelberg **39** (2007), 213–222.

[3] W. Adams and H. Sherali, *A tight linearization and an algorithm for zero-one quadratic programming problems*, Management Science **32** (1986), 1274–1290.

[4] E. Balas and J. Mazzola, *Nonlinear programming: I. linearization techniques*, Math. Program **30** (1984), 1–21.

[5] M. Bashiri and H. Karimi, *Effective heuristics and meta-heuristics for the quadratic assignment problem with tuned parameters and analytical comparisons*, Journal of Industrial Engineering International (Springer Open Journal, ed.), vol. 8, 2012, pp. 1–6.

[6] R. Battiti and G. Tecchiolli, *The reactive tabu search*, ORSA Journal on Computing **6** (1994), 126–140.

[7] M. Bazaraa and O. Kirka, *Branch and bound based heuristic for solving quadratic assignment problem*, Naval Res. Logist **30** (1980), 29–41.

[8] M. Bazaraa and H. Sherali, *Benders' partitioning scheme applied to a new formulation of the quadratic assignment problem*, Naval Research Logistics Quarterly **27** (1980), 29–41.

[9] B. Bollobás, *Extremal graph theory*, Academic London (1978).

[10] J. Rodríguez F. MacPhee D. Bonham and V. Bhavsar, *Solving the quadratic assignment and dynamic plant layout problems using a new hybrid meta-heuristic approach*, Int. J. High Perform. Comput. Netw **4** (2006), 286–294.

[11] N. Brixius and K. Anstreicher, *Solving quadratic assignment problem using convex quadratic programming relaxations*, Optimization Methods and Software **16** (2001), 49–68.

[12] A. Bruengger, J. Clausen, A. Marzetta, and M. Perregaard, *Joining forces in solving large-scale quadratic assignment problems in parallel*, Tech. report, University of Copenhagen, 1996.

[13] R. Burkard, *The quadratic assignment problem.*, Pardalos P.M. Handbook of Combinatorial Optimization. (2013).

[14] R. Burkard and F. Rendl, *A thermodynamically motivated simulation procedure for combinatorial optimization problems*, European Journal of Operational Research **17** (1984), no. 2, 169–174.

[15] J. Castellanos, V. Amarillo, and R. Poveda, *Quadratic assignment problem (qap) on gpu through a master-slave pga*, Visión Electrónica. más que un estado sólido **10** (2016), no. 2, 179–183.

[16] P. Szwed W. Chmiel and P. Kad luczka, *Opencl implementation of pso algorithm for the quadratic assignment problem*, Artifcial Intelligence and Soft Computing - 14th International Conference, ICAISC (2015).

[17] W. Chmiel, P. Kadluczka, J. Kwiecien, and B. Filipowicz, *A comparison of nature inspired algorithms for the quadratic assignment proble*, BULLETIN OF THE POLISH ACADEMY OF SCIENCES TECHNICAL SCIENCES **65** (2017), no. 4, 513–522.

[18] N. Christofides, *Worst case analysis of a new heuristic for the traveling salesman problem*, Tech. Report 338, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA, 1976.

[19] A. Colorni and M. Maniezzo, *The ant system applied to the quadratic assignment problem*, IEEE T. Knowl. Dta En **11** (1999), 769–778.

[20] E. Cárdenas, R. Poveda, and O. García, *A solution for the quadratic assignmentproblem (qap) through a parallel geneticalgorithm based grid on gpu*, Applied Mathematical Sciences **11** (2017), no. 57, 2843–2854.

[21] A. Elshafei, *Hospital layout as a quadratic assignment problem*, Operations Research Quarterly **28** (1977), 167–179.

[22] B. Eschermann and H. Wunderlich, *Optimized synthesis of self-testable finite state machines*, In 20th International Symposium on Fault-Tolerant Computing (FFTCS 20), June 1990.

[23] S. Semlali M. Essaid and F. Chebihi, *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), IEEE, 2018.

[24] F. Fomeni, *New solution aprroaches for the quadratic assignment problem*, Master's thesis, University of the Witwatersrand, Johannesburg, Sout Africa, September 2011.

[25] A. Frieze and J. Yadegar, *On the quadratic assignment problem*, Discrete Applied Mathematics **5** (1983), 89–98.

[26] M. Garey and D. Johnson, *Computers and intractability: A gude to the theory of np-completeness*, W.H.Freeman and Company, New York (1979).

[27] P. Gilmore, *Optimal and suboptimal algorithms for the quadratic assignment problem*, SIAMJ Appl Math **10** (1962), 305–313.

[28] D. Goldberg, *Genetic algorithms in search, optimizations and machine learning*, Addison-Wesley, 1989.

[29] R. Gomory, *Outline of an algorithm for integer solutions to linear programs*, Bull AMS **64** (1958), 275–278.

[30] NVIDIA CUDA Programming Guide, *https://developer.nvidia.com/cuda-gpus*, 2016.

[31] A. Gunawan, K. Ming, K. Leng, and H. Chuin, *Hybrid metaheuristics for solving the quadratic assignment problem and the generalized quadratic assignment problem*, IEEE International Conference on Automation Science and Engineering (CASE), August 2014, pp. 119–124.

[32] S. Hadley, F. Rendl, and H. Wolkowicz, *A new lower bound via projection for the quadratic assignment problem*, Mathematics of Operations Research **17** (1992), 727–739.

[33] T. Koopmans and M. Beckman, *Assignment problems and the location of economic activities*, Assigment problemsand the locatin of economic activitics (1957), 53–73.

[34] J. Krarup and P. Pruzan, *Computer-aided layout design*, Mathematical Programming Study **9** (1978), 75–94.

[35] T. Langauer, *Combinatorial algorithms for integrated circuit layout*, Wiley, Chichester (1990).

[36] E. Lawler, *The quadratic assignment problem*, Management Science **9** (1963), 586–599.

[37] E. Lawler and J. Lenstra, *The traveling salesman problem*, Wiley, Chichester (1985).

[38] E. Alba G. Luque and S. Nesmachnow, *Parallel metaheuristics: recent advances and new trends*, International Transactions in Operational Research (2013), 1–48.

[39] T. Mautor, *Contribution à la résolution des problèmes d'implanation: algorithmes séquentiels et parallèles pour l'affectation quadratique*, PhD thesis, Université Pierre et Marie Curie, Paris, France (1992).

[40] J. Mohammadi K. Mirzaie and V. Derhami, *Parallel genetic algorithm based on gpu for solving quadratic assignment problem*, Second Intenational Conference on Knowledge-Based Engineering and Innovation (KBEI), Tehran Iran, IEEE, November 2015, pp. 569–572.

[41] E. Mohassesian and B. Karasfi, *New method for improving the performance of fast local search in solving qap for optimal exploration of state space*, Artificial Intelligenece and Robotics (IEEE) (2017).

[42] H. Muller-Merbach, *Optimale reihenfolgen*, Springer-Verlag, 1970.

[43] A. Chaparala C. Novoa and A. Qasem, *A simd solution for the quadratic assignment problem with gpu acceleration*, ACM, Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment, July 2014.

[44] C. Nugent, T. Vollman, and J. Ruml, *An experimental comparison of techniques for the assignment of facilities to locations*, Operations Research **16** (1968), 150–173.

[45] G. Onwubolu and A. Sharma, *Particle swarm optimization for the assignment of facilities to locations*, New Optimization Techniques in Engineering. Springer (2004), 567–584.

[46] M. Padberg and M. Rijal, *Location, scheduling, design and integer programming*, Kluwer Academic, Boston (1996).

[47] K. Murthy P. Pardalos and Y. Li, *A local search algorithm for the quadratic assignment problem*, Informatica **3** (1992), 524–538.

[48] P. Pardalos and J. Xue, *The maximum clique problem*, Global Optim **4** (1994), 301–328.

[49] R. Burkard E. Cela P. Pardalos and P. Pitsoulis, *The quadratic assignment problem*, **2** (1998), 241–338.

[50] Y. Li M. Pardalos and M. Resende, *A greedy randomized adaptive search procedure for the quadratic assignment problem, in quadratic assignment and related problems*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science **16** (1994), 237–261.

[51] E Cantú Paz, *Survey of parallel genetic algorithms*, Technical Report 97003, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana Champaign (1997).

[52] R. Poveda, E. Cárdenas, and O. García, *Some np-hard problems solved from the quadratic assignment problem through a parallel genetic algorithm based grid on gpu*, Far East Journal of Mathematical Sciences (FJMS) **103** (2018), no. 8, 1289–1302.

[53] R. Poveda, E. Cárdenas, and O. Salcedo, *Classical k x k -chessboard problems solved as a quadratic assignment problem through a fine-grained parallel genetic algorithm on gpu*, Far East Journal of Mathematical Sciences (FJMS) **108** (2018), no. 1, 161–175.

[54] R. Poveda and J. Gómez, *Solving the quadratic assignment problem (qap) through a fine-grained parallel genetic algorithm implemented on gpus*, Springer Nature Switzerland AG (N. T. Nguyen et al. (Eds.): ICCCI 2018., ed.), LNAI 11056, September 2018, pp. 145–154.

[55] QAPLIB, *http://anjos.mgi.polymtl.ca/qaplib/*, 2017.

[56] M. Queyranne, *Performance ratio of polynomial heuristric for triangle inequality quadratic assignment problem*, Operations Research Letters **4** (1986), 231–234.

[57] A. Ramkumar, S. Ponnanbalam, and N. Jawahar, *A new iterated fast local search heuristic for solving qap formulation in facility layout design*, Robotics and Computer-Integrated Manufacturing. Elsevier **25** (2009), no. 3, 620–629.

[58] T. James C. Rengo and F. Glover, *Multistart tabu serach and diversication strategies for the quedratic assignment problem*, IEEE Trans Syst. Man Cybern, Part A Syst. Hum **39** (2009), 579–596.

[59] S. Sahni and T. Gonzalez, *P-complete approximation problems*, Journal of the Association (1976), 555–565.

[60] J. Sanders and E. Kandrot, *Cuda by example*, Addison Wesley, 2010.

[61] M. Scriabin and R. Vergin, *Comparison of computer algorithms and visual based methods for plant layout*, Management Science. **22** (1975), 172–187.

[62] K. Jong W. Spears and D. Gordon, *Using genetic algorithms for concept learning*, Machine Learning **13-2** (1993), no. 2, 161–188.

[63] P. Szwed and W. Chmiel, *Multi-swarm pso algorithm for the quadratic assignment problem: a massive parallel implementation on the opencl platform*, Computing Research Repository (2015).

[64] É. Taillard, *Robust tabu search for the quadratic assignment problem*, Parallel Computing **17** (1991), 443–455.

[65] E. Taillard, *Comparision of iterative searches for the quadratic assignment problem*, Location Science **3** (1995), 87–105.

[66] L. Gambardella E. Taillard and M. Dorigo, *Ant colonies for the qap*, Oper Res. Soc **50** (1999), 167–176.

[67] E. Talbi, *Combining metaheuristics with mathematical programming, constraint programming and machine learning*, Springer-Verlag Berlin Heidelberg (2013).

[68] M. Tomassini, *A survey of genetic algorithms*, Volume III of Annual Reviews of Computational Physics, World Scientific **3** (1995), 87–118.

[69] M. Triola, *Estadística*, 10 ed., Pearson Eduacación. Addison Wesley, 2009.

[70] L. Tseng and S. Liang, *A hibrid metaheuristic for the quadratic assignmnet problem*, Compt. Optim. Appl **34** (2006), 85–113.

[71] S. Tsutsui, *Parallel and colony optimization for the quadratic assignment problem with symmetric multi processing*, in ANTS Springer Berlin **5217** (2008), 363–370.

[72] S. Tsutsui and N. Fujimoto, *An analytical study of parallel ga with independent runs on gpus*, Massively Parallel Evolutionary Computation on GPGPUs, Natural Computing Series Springer-Verlag Berlin Heidelberg (2013), 105–120.

[73] M. Ujaldón, *Programming gpus with cuda*, Tutorial at 18th IEEE CSE'15 and 13th IEEE EUC'15 conferences (Porto (Portugal)), October 2015.

[74] U.Thonemann and A.Bolte, *An improved simulated annealing algorithm for the quadratic assignment problem*, Working paper, School of Business, Department of Production and Operations Research, University of Paderborn, Germany (1994).

[75] M. Wilhelm and T. Ward, *Solving quadratic assignment problems by simulated annealing*, IE Transaction **19** (1987), no. 1, 107–119.

[76] M. Lim Y Yuan and S. Omatu, *Extensive testing of a hybird genetic algorithm for solving qudratic assignment problem*, Comput. Optim. Appl **23** (2002), 47–64.

[77] H. Zhang, C. Beltran, and R. Liang, *Solving the quadratic assignment problem by means of general purpose mixed integer linear programming solvers*, Annals of Operations Research, Springer **207** (2013), no. 1, 261–278.

[78] L. Ziqiang, Y. Yi, and Q. Yang, *A hybrid artificial fish-school optimization algorithm for solving the quadratic assignment problem*, International Conference on Natural Computation IEEE, August 2014, pp. 1099–1104.