Computação paralela e bio-inspirada: algoritmos genéticos multipopulação e autômatos celulares híbridos

Bruno Well Dantas Morais

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2020

Bruno Well Dantas Morais

Computação paralela e bio-inspirada: algoritmos genéticos multipopulação e autômatos celulares híbridos

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Gina Maira Barbosa de Oliveira

Uberlândia
2020

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

M827 Morais, Bruno Well Dantas, 1993-
2020 Computação paralela e bio-inspirada: algoritmos genéticos multipopulação e autômatos celulares híbridos [recurso eletrônico] / Bruno Well Dantas Morais. - 2020.

Orientadora: Gina Maira Barbosa de Oliveira.
Dissertação (Mestrado) - Universidade Federal de Uberlândia, Pós-graduação em Ciência da Computação.
Modo de acesso: Internet.
Disponível em: http://doi.org/10.14393/ufu.di.2021.29
Inclui bibliografia.
Inclui ilustrações.

1. Computação. I. Oliveira, Gina Maira Barbosa de, 1967-, (Orient.). II. Universidade Federal de Uberlândia. Pós-graduação em Ciência da Computação. III. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:

Gizele Cristine Nunes do Couto - CRB6/2091

Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

The road to wisdom?—Well, it's plain
and simple to express:
Err
and err
and err again,
but less
and less
and less.

Resumo

Este trabalho consiste em uma investigação sobre a aplicação de técnicas de computação paralela a modelos bio-inspirados baseados em autômato celular e algoritmo genético no contexto de aplicações para criptografia e para escalonemanto de tarefas, respectivamente. O modelo *Hybrid Cellular Automata* (HCA) possui dois algoritmos para evolução, em que os estados de uma grade de células são atualizados iterativamente de acordo com regras de transição. Este modelo é aplicado a criptografia, que visa comunicação segura por meio da codificação de mensagens. O *multipopulation genetic algorithm* (MPGA) é uma variação do algoritmo genético projetado para a aplicação de computação paralela. Este modelo consiste na evolução de múltiplos conjuntos de soluções por meio de operadores estocásticos para aplicações de busca e otimização. Neste trabalho, este algoritmo é aplicado para escalonamento de tarefas, um problema computacionalmente intratável que consiste na minimização do tempo de execução de tarefas interdependentes atribuídas a um conjunto de processadores. Implementações sequenciais e paralelas destes modelos foram desenvolvidas com a linguagem Python, sendo destinadas para processadores *multicore* (CPU) e processadores gráficos (GPU) no caso do HCA, e abordagens baseadas em memória distribuída e memória compartilhada para CPU no caso do MPGA. Com estes programas, experimentos foram conduzidos para quantificar os ganhos de desempenho de cada abordagem paralela em comparação com as implementações sequenciais. O desempenho dos algoritmos de evolução do HCA foi beneficiado pela execução paralela em GPU, enquanto as implementações paralelas em CPU resultaram em perda de desempenho devido a *overhead*. Os experimentos que envolvem a parametrização do MPGA demonstraram um conflito entre a qualidade das soluções e o tempo de execução. Neste caso, uma análise multiobjetivo foi empregada, que demonstrou configurações altamente eficientes considerando ambas as métricas de desempenho.

Palavras-chave: autômato celular, algoritmo genético, criptografia, escalonamento de tarefas, computação paralela.

Parallel and bio-inspired computing: multipopulation genetic algorithms and hybrid cellular automata

Bruno Well Dantas Morais

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2020

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Programa de Pós-Graduação em Ciência da Computação
Av. João Naves de Ávila, nº 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br

ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Mestrado Acadêmico, 35/2020, PPGCO				
Data:	14 de dezembro de 2020	Hora de início:	14:00	Hora de encerramento:	15:45
Matrícula do Discente:	11812CCP010				
Nome do Discente	Bruno Well Dantas Morais				
Título do Trabalho:	Parallel Computing approaches applied to the bio-inspired models Cellular Automaton and Genetic Algorithm				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Inteligência Artificial				
Projeto de Pesquisa de vinculação:	-				

Reuniu-se, por videoconferência, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: Rodrigo Sanches Miani - FACOM/UFU; Alexandre Cláudio Botazzo Delbem - ICMC/USP e Gina Maira Barbosa de Oliveira - FACOM/UFU, orientadora do candidato.

Os examinadores participaram desde as seguintes localidades: Alexandre Cláudio Botazzo Delbem - São Carlos/SP; Rodrigo Sanches Miani e Gina Maira Barbosa de Oliveira - Uberlândia/MG. O discente participou da cidade de Uberlândia/MG.

Iniciando os trabalhos a presidente da mesa, Prof.ª Dr.ª Gina Maira Barbosa de Oliveira, apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação do Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir a senhora presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir o candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

Aprovado.

Esta defesa faz parte dos requisitos necessários à obtenção do título de Mestre.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Rodrigo Sanches Miani**, **Professor(a) do Magistério Superior**, em 16/12/2020, às 17:06, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **ALEXANDRE CLAUDIO BOTAZZO DELBEM**, **Usuário Externo**, em 21/12/2020, às 13:44, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Gina Maira Barbosa de Oliveira**, **Professor(a) do Magistério Superior**, em 11/03/2021, às 14:33, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **2453653** e o código CRC **B721661B**.

Referência: Processo nº 23117.074752/2020-57 SEI nº 2453653

Abstract

This work consists of an investigation about the application of parallel computing techniques to bio-inspired models based on cellular automata (CA) and genetic algorithms (GA) in the context of their application to cryptography and the task scheduling problem, respectively. The Hybrid Cellular Automata (HCA) model features two algorithms that perform forward and backward evolution, where the states of a grid of cells are iteratively updated according to transition rules and nearby cells. This model is applied to cryptography, which aims for secure communication by encoding messages to prevent unintended access to the information. The Multipopulation Genetic Algorithm (MPGA) is a variation of GA intended for the application of parallel computing. This model consists of the evolution of multiple sets of solutions by means of stochastic operators for search and optimization applications. This algorithm is applied to the task scheduling problem, a computationally intractable problem that consists of minimizing the execution time of interdependent tasks assigned to a set of processors. Sequential and parallel implementations of these models were developed with the Python language, with implementations aimed to multicore processors (CPU) and graphics processing units (GPU) in the case of the HCA, and distributed memory and shared memory approaches for multicore processors in the case of the MPGA. With these implementations, experiments were conducted to quantify the performance gains of each parallel approach in comparison to the sequential implementations. The performance of the HCA algorithms was benefited by the parallel execution on GPU, while the parallel CPU implementations resulted in the loss of performance due to overhead. The experiments involving the parameterization of MPGA demonstrated a trade-off between the quality of solutions and execution time. In this case, a multiobjective analysis was employed, elucidating highly efficient configurations considering both of these performance metrics.

Keywords: cellular automata, genetic algorithms, cryptography, task scheduling, parallel computing.

List of Figures

List of Tables

Acronyms list

CPU Central Processing Unit

GPU Graphics Processing Unit

CA Cellular Automaton

HCA Hybrid Cellular Automata-based cipher system

FCA Forward evolution of Cellular Automata

BCA Backward evolution of Cellular Automata

GA Genetic Algorithm

MPGA Multipopulation Genetic Algorithm

Contents

I hereby certify that I have obtained all legal permissions from the owner(s) of each third-party copyrighted matter included in my thesis, and that their permissions allow availability such as being deposited in public digital libraries.

Bruno Well Dantas Morais

Chapter 1

Introduction

The increasing application of multiprocessor systems has led to increasing demand for parallel implementations of algorithms that are capable of exploiting this increased computing power (MITCHUM, 2013). For instance, multicore processors have been components to most personal computers, as well as graphics processing units (GPU) that can be applied to general purpose programming. The availability of these parallel computing architectures makes the modern consumer-oriented computers effective for the implementation of parallel programs that employ multiple processes or threads in a variety of degrees of interdependence.

Nevertheless, to be able to harness the performance benefits of parallel execution on these architectures, programs must be composed of a number of subtasks that display a considerable degree of independence between them, such that their simultaneous execution is semantically appropriate as well as efficient (Bernstein, 1966). The bio-inspired models cellular automaton (CA) (SARKAR, 2000) and genetic algorithm (GA) (GOLDBERG, 1989) are examples of programs that might have significant potential for performance gains with the use of parallel computing, as some of the basic concepts from their definitions are intrinsically parallel (MITCHELL, 2009).

Cellular automaton is a discrete dynamical system that represents a model of decentralized computation (MITCHELL et al., 1996). It is composed by a regular grid of cells, each featuring a state, and a transition rule that updates a cell according to the states present on its locality. The evolution of CA consists of updating each of its cells with the transition rule for a number of iterations. During a step of the evolution of CA, each cell can be updated independently, which indicates a potential for benefiting from the application of parallel computing.

Genetic algorithm (GA) is a metaheuristic technique inspired by the evolution of the species for the application to search and optimization problems (GOLDBERG, 1989). This is performed by generating and evolving a population of solutions for a number of iterations. On each iteration, a number of stochastic operators are applied to the population, leading to the generation of new solutions via selection and combinations. Each

solution is evaluated according to a given quality metric, denoting its fitness. The independence found in the processes of generation and evaluation of each individual solution also indicates a potential for benefiting from the application of parallel computing.

1.1 Motivation

The hybrid cellular automata (HCA) model (OLIVEIRA; MACEDO, 2019) is a CA model developed to enable the application of a deterministic inverse operation of the conventional CA evolution. This is achieved with the utilization of pairs of transition rules with certain properties. The HCA model is applied to cryptography, which aims to achieve secure communication in the context of possible interference by third-party subjects. This is done through the encryption process, where the messages are transformed into a form that is not readily intelligible to be decoded only by the intended destination. Thus, it is important to employ a cryptographic system that is effective in protecting information and efficient to make communication possible (STALLINGS, 2013).

The multipopulation genetic algorithm (MPGA) is a variation of the standard GA intended for the application of parallel computing (CANTÚ-PAZ, 1998). It extends the genetic algorithm concept by employing multiple populations of solutions instead of one, such that the evolution of each population is mostly independent of the others, with communications between populations occurring with some frequency. The MPGA is applied to the task scheduling problem in this work, a variation of the classic computationally intractable scheduling problem that is also related to the concepts of parallel computing (HOU; ANSARI; REN, 1994). The task scheduling problem is a combinatorial optimization problem that consists in the assignment of a set of interdependent tasks to a set of processors with the aim of reaching the lowest possible total execution time. In this work, parallel implementations of the MPGA are investigated in its application to the task scheduling problem with the aim to improve the execution time of the algorithm and the reliability (proximity between the achieved result and the optimum result) of the task scheduling at the same time.

1.2 Research objectives

The general objective of this work is to contribute to the understanding of the application of parallel computing for the HCA and MPGA bio-inspired models by considering two study cases: the hybrid cellular automata model applied to cryptography and the multipopulation genetic algorithm applied to task scheduling.

In the case of the HCA model, this work aims at the evaluation of new empirical results concerning the performance gain in terms of the execution time of parallel implementations of its evolution algorithms in comparison to the sequential implementations,

thus extending a previous work (MACÊDO, 2007), which lacks parallel computing experiments. To this end, experiments with each HCA evolution algorithm were conducted for the Python implementations of the corresponding sequential programs, as well as parallel implementations for multicore processors and parallel implementations for execution on GPU.

In the MPGA model, this work aims to investigate two parallel implementations, which apply concepts of distributed memory and shared memory development. The parallel implementations of this algorithm require additional parameterization for the number of populations and the frequency of their communication. With the experiments performed, these parameters are shown to influence the performance of the algorithm in terms of the quality of the solutions produced as well as the execution time. From these results, a trade-off between these performance metrics is observed. A multi-objective analysis is then employed to elucidate the effect of the parameters on these metrics.

1.3 Hypothesis

The main hypothesis of this work is that the parallel implementations of the bio-inspired algorithms provide a significant performance gain when compared to the serial implementations.

1.4 Contributions

In this work, the development of the bio-inspired models was done with the use of high-level tools of the Python language (MAROWKA, 2018). Both the sequential and parallel versions of these algorithms were developed, and their performance was evaluated and compared.

The contributions of this work are empirical results concerning the performance of the HCA and MPGA models via their implementation with high-level programming tools and their execution with resources of parallel computing.

Concerning the HCA model, the forward evolution and backward evolution algorithms were investigated with sequential and parallel implementations applied to CPU and parallel implementations applied to GPU.

In the case of the MPGA model, multiple variations of its operators were investigated in the context of the literature of its application to task scheduling, leading to the refinement of the algorithm and the publication of the research paper (MORAIS; OLIVEIRA; CARVALHO, 2019). The investigation of relevant task scheduling problem instances for the purpose of benchmark also contributed to the publication of (CARVALHO; MORAIS; OLIVEIRA, 2018). This work further presents the investigation of two parallel implemen-

tations of the MPGA with the parallel computing approaches of distributed memory and shared memory and their performance in relation to the parameters of the algorithm.

1.5 Organization of the dissertation

The following chapters are organized as follows.

- ❏ The relevant concepts and definitions to this work are described in Chapter 2, regarding cryptography, the task scheduling problem, parallel computing, cellular automata, the HCA model, genetic algorithms, and the multipopulation genetic algorithm.
- ❏ A review of the related literature is presented in Chapter 3, concerning the application of cellular automata to cryptography, the application of genetic algorithms to task scheduling, and parallel approaches for the multipopulation genetic algorithm.
- ❏ The detailed development of this work is described in Chapter 4, concerning preliminary experiments with parallel computing in Python, the development of each version of the HCA evolution algorithms, and the development of the MPGA approaches.
- ❏ In Chapter 5, the details of the experiments conducted and the analysis of their results are described for the HCA evolution algorithms and the parameterization of both MPGA implementations.
- ❏ The conclusions of this work is presented in Chapter 6, along with suggestions of future investigations correlated to the present work.

Chapter 2

Concepts and Definitions

This chapter describes the concepts and definitions concerning the applications of this work to cryptography and task scheduling, the corresponding bio-inspired approaches based on cellular automata and genetic algorithms, and the relevant approaches of parallel computing and multiobjective optimization.

2.1 Application problems

This dissertation is focused on the investigation of the application of parallel computing approaches to the bio-inspired models cellular automaton and genetic algorithm. Before presenting these models in detail, this section presents the main concepts related to the two application problems for which these models are applied in the present work: cryptography, which is approached with a cryptographic system based on cellular automata, and task scheduling, which is adopted as the application domain for the genetic algorithm.

2.1.1 Cryptography

Cryptography is the study of techniques related to secure communication in the presence of third-party subjects that may interfere in the information exchange. Generally, such secure communication is achieved through the encryption process, which involves the encoding of messages into unintelligible symbols such that its decoding can be performed only by its intended destination (STALLINGS, 2013).

In modern cryptography, encryption is performed through a cryptographic system, which includes a set of algorithms devised for such purpose (MENEZES; VANSTONE; OORSCHOT, 1996). A cryptographic system is mathematically defined as the tupĺe $(A, M, C, K, \mathcal{E}, D)$ detailed as follows.

- A is the set of symbols which are present in messages, e.g., latin alphabet, binary codes.

- M is the space of messages (plaintext), a set that represents every possible message that the system is capable of handling for encryption, e.g., text in English language, digital images.
- C is the space of ciphertext, a set of every string of symbols resulting from the encryption of the elements of M.
- K is the space of cryptographic keys such that each element represents a bijection $M \leftrightarrow C$.
- $\mathcal{E}$ is a set of encryption functions defined over $M \rightarrow C$. Each element of $\mathcal{E}$ has a key $k \in K$.
- D is a set of decryption functions defined over $C \rightarrow M$. Each element of D has a key $k \in K$.
- For each key $k1 \in K$, there exists a unique key $k2 \in K$ such that $D_{k2}(\mathcal{E}_{k1}(m)) = m$ for every message $m \in M$.

The earliest methods of cryptography were the substitution ciphers, which employ a direct substitution of symbols as encryption and decryption algorithm. In this case, the cryptographic key is defined as a specific bijection between the set of symbols used for plaintexts and the set of symbols used for ciphertext. A special case of substitution cipher is the shift cipher, where the bijection of symbols is based on a simple shift of the alphabet, e.g., a three-letter shift $[A, B, C, D, E, ..., X, Y, Z] \longleftrightarrow [D, E, F, G, H, ..., X, Y, Z, A, B, C]$ which maps A to D, B to E etc. With this mapping, the plaintext EXAMPLE would be encrypted into the ciphertext HADPSOH, which then could be decrypted by applying the opposite mapping.

Notably, the space of cryptographic keys in shift ciphers is considerably small, being limited to 25 possible crytographic keys representing effective shifts of the alphabet. This makes this model highly vulnerable to cryptanalysis, which is the act of investigating vulnerabilities in cryptographic system in order to determine the cryptographic key in use and/or the plaintext being communicated. The more general substitution ciphers provide a considerably greater key space, but their use for communication in natural language is vulnerable to cryptanalysis concerning the skewed frequency of occurrences of each letter that is found in natural language. For example, the letter e is the most frequent letter in the English language, and thus, the corresponding ciphertext symbol in a substitution cipher must also be most frequent symbol (STALLINGS, 2013).

The aforementioned vulnerabilities demonstrate some of the following desirable features of cryptographic systems. The inference of plaintext given a ciphertext and no cryptographic key should be a considerably difficult task, which is represented by the cardinality of the key space. The resulting mapping between plaintexts and ciphertexts

should be distributed as uniformly as possible. The encryption and decryption algorithms must also be efficient to make efficient secure communication possible. These features represent the objectives aimed by the development of other cryptographic systems (MENEZES; VANSTONE; OORSCHOT, 1996).

Systems that perform both encryption and decryption using the same key are called symmetric, exemplified by the substitution cipher and the DES (Data Encryption Standard) (STALLINGS, 2013). The DES is a cryptographic system for binary plaintext/ciphertext and it processes messages by subdividing them in 64-bit blocks and processing each block independently. Thus, the DES is further categorized as a block cipher, in contrast to stream ciphers, which process each symbol in sequence, like the substitution cipher. The DES performs encryption through an iterative process which, on each iteration, generates sub-keys from its 56-bit key, performs XOR operations between the sub-key and half of the text, and applies a substitution function. The DES has been withdrawn as a standard encryption model due to its relatively short key size.

Another example of symmetric cipher is the AES (Advanced Encryption Standard), which subdivides plaintext into 128-bit blocks and applies an iterative process including bit shifts and substitutions (DAEMEN; RIJMEN, 2013). The AES makes use of cryptographic keys with size 128, 192 or 256 bits. Its greater key space results in higher security of its encryption when compared to the DES. This, in addition to its higher performance, led to the adoption of the AES as the standard encryption model by multiple organizations and governments (STALLINGS, 2013).

The RSA (RIVEST; SHAMIR; ADLEMAN, 1978) is an example of asymmetric cryptographic system. In this model, instead of employing a secret key known to both communicating subjects, a subject holds a public key that enables anyone to encrypt messages. There is also a corresponding private key that allows the subject to decrypt messages. This cryptographic system makes use of Number Theory concepts in order to ensure the secure generation of corresponding public and private keys. Asymmetric systems are usually less time-efficient than symmetric systems, leading to hybrid applications by secure protocols such as the TLS (RESCORLA; MODADUGU, 2006), where asymmetric systems are used to share a key for a symmetric system. The usage of hybrid systems also provide a secure solution to the problem of the key exchange that is present in symmetric systems.

The HCA cryptographic system (OLIVEIRA; MACEDO, 2019), which is one of the bio-inspired models investigated in this work, is a symmetric cipher that employs blocks of 128 bits and 256-bit keys. Its cipher algorithms are based on the evolution of cellular automata, which is a process that can be executed by multiple processing units simultaneously, thus enabling its execution in parallel computers that can achieve potential gains in performance. This model is further detailed in the following sections.

2.1.2 Task Scheduling

The multiprocessor task scheduling problem is a computationally intractable combinatorial optimization problem consisting in the assignment of a set of tasks to a multiprocessor device in order to satisfy a given evaluation criterion (HOU; ANSARI; REN, 1994). Due to its intractability, heuristics, approximations, and metaheuristics have been applied to this problem (KWOK; AHMAD, 1999). The variation of the problem studied in this work assumes the following properties.

- ❏ In addition to computation costs associated to the tasks, the execution of interdependent tasks by distinct processors is subject to a communication cost associated to the dependency of the tasks.
- ❏ The scheduling is static and deterministic such that all computation and communication costs and task dependencies are fixed and fully known.
- ❏ Processors are non-preemptive, executing tasks to completion without interruptions.
- ❏ The set of processors is homogeneous, having the same relative performance and operating in parallel.
- ❏ All processors can communicate directly with each other.
- ❏ Every task is executed once by exactly one processor.

An instance of the task scheduling problem is defined as the tuple (G, P) where G is a directed acyclic graph corresponding the the tasks and their dependencies, and P is the set of processors. Since every processor of P is equivalent in this variation of the problem, it is sufficient for an instance of the problem to determine the number of processors to be considered for scheduling.

Every task (vertex) of G has an associated computation cost, and every dependency (edge) has an associated communication cost. Figure 1A exemplifies a task graph with tasks $T0, T1, T2, ..., T8$ having computation costs $80, 90, 90, ..., 80$ and dependencies having communication costs set to 40, which are omitted in the figure.

In a scheduling, communication costs are only applied when the two associated tasks are executed by distinct processors, implying in a necessary time interval between the conclusion of the precedent task and the start of the dependent task. Otherwise, the dependent task can be executed immediately if the assigned processor is the same. A task can only be executed by a processor after the conclusion of every precedent task, plus the associated communication time when required.

A solution to a problem instance with p processors can be represented by p sequences of tasks, which are associated to each processor. Each sequence determines the processor assigned to each task as well as the order of execution of each task by the processor. In

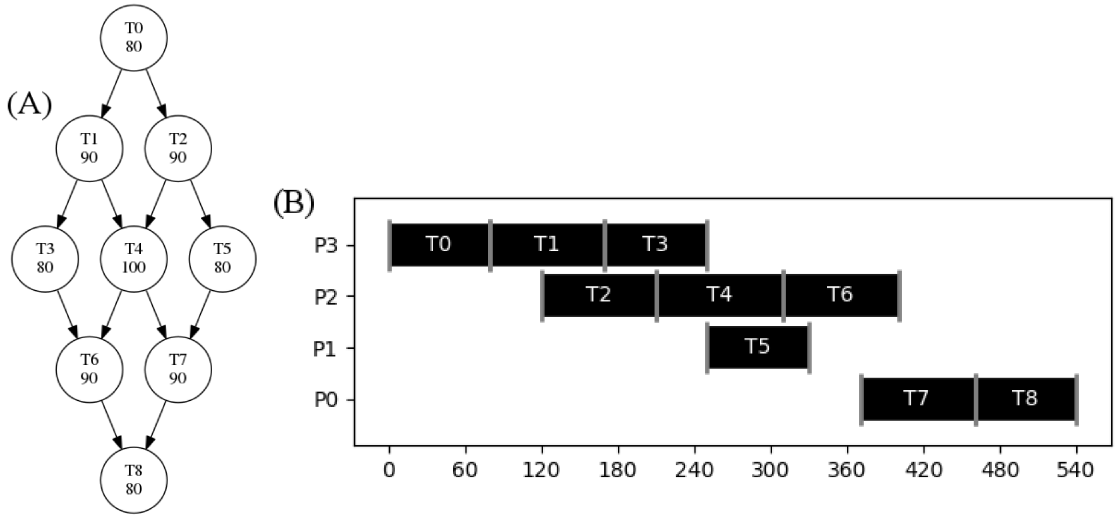

Figure 1 – **(A)**: example scheduling graph with computation costs indicated for each task and communication costs set as 40. **(B)**: Gantt diagram visualizing a scheduling solution and its makespan of 540 time units. Adapted from Morais, Oliveira e Carvalho (2019).

this representation, each sequence must follow the precedence constraints set by the task dependencies of the graph. For example, a solution to the problem instance composed of the graph from Figure 1A and 4 processors labeled $P0, P1, P2, P3$ is shown as follows.

$$P0 : [T7, T8], P1 : [T5], P2 : [T2, T4, T6], P3 : [T0, T1, T3]$$

Given a scheduling solution with tasks assigned to processors and their ordering, its execution timeline can be represented by a Gantt diagram, which visualizes the starting and conclusion points of each task. Figure 1B ilustrates this diagram representing the aforementioned solution over 4 processors, which results in a total execution time of 540 time units. The total execution time of a scheduling solution is called makespan, representing the total time duration taken to conclude all tasks. The makespan is usually adopted as an evaluation metric of a solution. Thus, an optimal scheduling features the lowest possible makespan for a given problem instance.

The communication costs of a scheduling instance can also be defined as a function of the computation costs by the computation-communication ratio (CCR). With this parameter, the communication costs y associated with edges with origin on a given vertex with computation cost x are determined as $y = x/CCR$. For example, CCR=0.5 defines communication costs as being 2 times higher than computation costs. Considering the graph from Figure 1A, the communication costs for CCR=0.5 would be 160 for each edge that originates from task T0, 180 for edges originating from task T1, 200 for edges originating from task T4 etc.

2.2 Bio-inspired models

This section presents the key concepts of the bio-inspired models investigated in this work, the cellular automaton and the genetic algorithm. The basic concepts of these models are defined, each followed by the description of the particular variations that are investigated in this work, being the hybrid cellular automata model (HCA) and the multipopulation genetic algorithm (MPGA).

2.2.1 Cellular Automaton

Cellular Automaton (CA) is a discrete dynamical model studied in different areas like Computer Science, Physics and Biology. It was introduced by John von Neumann and Stanislaw Ulam in the context of phenomena like crystal growth, self-replication and fluid dynamics (SARKAR, 2000). Applications of CA include simulations of epidemics (WHITE; REY; SÁNCHEZ, 2007), urban development (XIA et al., 2018), crowd dynamics (GIITSIDIS; DOURVAS; SIRAKOULIS, 2017), propagation of forest fires (NTINAS et al., 2016), image processing (LÓPEZ-FANDINO et al., 2016), and robotics (LIMA; OLIVEIRA, 2016; TINOCO; OLIVEIRA, 2019).

Despite their conceptual simplicity, CA are capable of producing a variety of dynamical behaviors. Even the simplest CA models feature behaviour patterns that are comparable to complex biological systems and can be applied to the modelling of a variety of natural phenomena (GREEN, 1990). For example, the CA model presented in this section is able to model natural processes such as the pigment patterns found in seashells in the genera *Conus* and *Cymbiola*, exemplified by Figure 2 (COOMBES, 2009). Figure 3 presents the evolution of a CA called rule 30, which demonstrates a similar dynamical process to the pigmentation pattern of Figure 2.

CA consist of a regular grid of cells that feature a state, and each one is iteratively updated according to their neighbor cells and a transition rule. CA are generally composed by the following attributes.

- ❏ Regular n-dimensional grid of cells (the CA lattice/grid).
- ❏ Cellular neighborhood, usually defined in terms of spatial distance, or radius.
- ❏ Set of states that can be assumed by each cell.
- ❏ Initial configuration, i.e., initial states of each cell.
- ❏ Transition rule, which updates a cell by mapping the states of its neighborhood to a new state.
- ❏ Number of steps or iterations where each cell will be updated.

Figure 2 – Pigment pattern generated by a process similar to cellular automata on the seashell of *Conus textile* (LING, 2005).

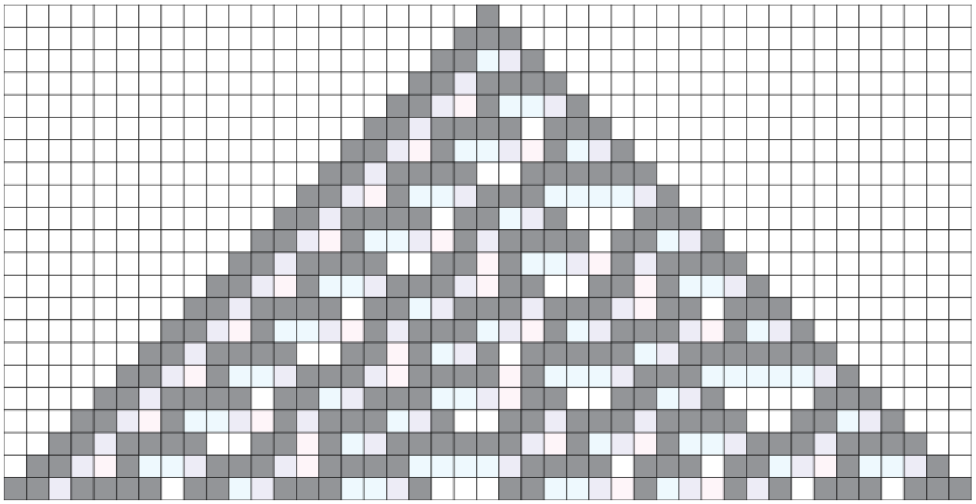

Figure 3 – Graphical representation of the evolution of the rule 30 CA. Adapted from (Wikipedia contributors, 2020).

CA grids are usually defined as a n-dimensional matrix of cells, with the one-dimensional grid being the simplest example. The set of states determines each state that each individual cell can assume, e.g., binary cells with states represented as 0 and 1. The neighborhood defines which cells are considered neighbors to each cell. In the simplest case, called radius-1 neighborhood, the neighbors of a cell are the cells adjacent to it. In the case of cells on the extremities of the grid, their missing neighbors must be accounted for. One approach for this is the definition of periodic neighborhood, where the cells at the extremities are considered adjacent as in a torus.

The initial configuration determines the states of each cell preceding the process of

evolution of the CA. The evolution of the CA consists of updating each cell through the transition rule, according to its neighborhood. Evolution occurs for a number of steps, where each cell is updated once.

The CA considered in this work assume one-dimensional grid of cells, radius-1 periodic neighborhood and two possible cell states. An example of transition rule for these CA is rule 30, which defines the following transitions of every possible neighborhood states into another state for the central cell.

$111 \mapsto 0, 110 \mapsto 0, 101 \mapsto 0, 100 \mapsto 1, 011 \mapsto 1, 010 \mapsto 1, 001 \mapsto 1, 000 \mapsto 0$

Figure 4 exemplifies the evolution of the aforementioned CA with unidimensional grid of 16 cells, radius-1 periodic neighborhood and two states. With initial configuration indicated as 0000000010000000, the CA is evolved for 8 steps, where each cell is updated according to its neighborhood. For instance, from the initial configuration to the resulting configuration of the first step, the cell with state 1 is updating according to the rule which maps its neighborhood as $010 \mapsto 1$. In that same step, cells to the left and right are updated according to the respective mappings $001 \mapsto 1$ and $100 \mapsto 1$. The remaining cells in this step are also updated accordingly, with $000 \mapsto 0$. After updating every cell, this process is repeated for each subsequent evolution step, finally reaching the configuration 0100100011100000 after step 8 in the example.

By means of their iterative mechanism, CA are capable of computation. CA are considered parallel decentralized computers, since the application of the transition rule on each cell is independent of the others. Therefore, CA applications in general have potential to benefit from parallel computing.

Usual CA models are constituted by the homogeneity and local connectivity between cells, deterministic transition rules, synchronous evolution steps and small neighborhood radius. Binary unidimensional CA with radius 1 are the simplest variation of CA, and can be extended with the following features.

- ❏ Higher grid dimensionality, such as 2D or 3D.
- ❏ Greater set of states, with three or more states.
- ❏ Higher neighborhood radius, considering farther neighbors for each cell besides its adjacent cells.
- ❏ Alternative definitions for the neighborhood of cells on the extremities of the grid.
- ❏ Non-deterministic transition rules.
- ❏ Non-homogeneous application of transition rules, where cells can be assigned distinct transition rules.
- ❏ Asynchronous evolution steps, making the evolution process non-deterministic by not performing synchronization between steps.

initial	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
step 1	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0
step 2	0	0	0	0	0	0	1	1	0	0	1	0	0	0	0	0
step 2	0	0	0	0	0	1	1	0	1	1	1	1	0	0	0	0
step 3	0	0	0	0	1	1	0	0	1	0	0	0	1	0	0	0
step 4	0	0	0	1	1	0	1	1	1	1	0	1	1	1	0	0
step 5	0	0	1	1	0	0	1	0	0	0	0	1	0	0	1	0
step 6	0	1	1	0	1	1	1	1	0	0	1	1	1	1	1	1
step 7	0	1	0	0	1	0	0	0	1	1	1	0	0	0	0	0
step 8	1	1	1	1	1	1	0	1	1	0	0	1	0	0	0	0

Figure 4 – An example of the evolution of the rule 30 CA over 8 time steps on a 16-cell grid. The first row contains its initial configuration, and each subsequent row contains the states resulting from the application of the transition rule to each cell of the previous row.

A CA is called homogeneous if it applies a single transition rule in its evolution. Differently, hybrid CA apply two or more transition rules in their evolution. The hybrid cellular automata model (HCA) investigated in this work is a unidimensional binary CA with radius 1 and non-homogeneous application of transition rules, employing two transition rules for different portions of the grid. This model is described in the detail in the next section.

Some transition rules of 1D CA exhibit a specific property that is important for the HCA model discussed in the next section. This feature, called toggle property, defines rules which outcome is completely dependent to the cell in the extremity of the neighborhood. Such transition rules are called toggle rules, denoting that they are sensitive to the leftmost cell or to the rightmost cell.

A transition rule is left-toggle if, considering every possible input neighborhood, the resulting state of the rule is dependent on the state of the leftmost cell of the neighborhood, i.e., changing the state of the leftmost cell in the neighborhood always changes the outcome of the rule. For example, the aforementioned rule 30 applied in the evolution of Figure 4 is a left-toggle rule since any pair of similar neighborhoods with changes only on the

leftmost cell results in a change in the resulting state, as shown in the following pairs of mappings.

- ❑ $111 \mapsto 0$ and $011 \mapsto 1$
- ❑ $110 \mapsto 0$ and $010 \mapsto 1$
- ❑ $101 \mapsto 0$ and $001 \mapsto 1$
- ❑ $100 \mapsto 1$ and $000 \mapsto 0$

The right-toggle property is analogous: a transition rule is right-toggle if, considering every possible input neighborhood, the resulting state of the rule is dependent on the state of the rightmost cell of the neighborhood.

2.2.2 Hybrid Cellular Automata model

Concerning the temporal evolution of CA previously discussed, some applications of CA also benefit from the inverse operation of the usual CA evolution. In this context, updating cells with the rule transition over a number of evolution steps is then referred as forward evolution, while the inverse operation is referred as backward evolution. The backward evolution of CA is defined such that: given a grid G in an initial configuration and the grid G' that is the result of the forward evolution of G for n steps, then the backward evolution of G' for n steps results in G. For example, an 8-step backward evolution of the lattice with states 0100100011100000, would result in the lattice 0000000010000000, which is the initial configuration in this figure. A backward evolution step is also called a pre-image computation.

The application of backward evolution is restricted to a class of CA called reversible CA. A CA is reversible if, according to its transition rule, every possible grid configuration has a single pre-image. Irreversible CA, on the other hand, might have zero or multiple pre-images for some configurations.

The HCA (Hybrid Cellular Automata) model developed by Oliveira e Macedo (2019) is composed of reversible CA as an application to cryptography such that encryption is performed via backward evolution and decryption is performed via forward evolution. This model applies two transition rules instead of one, called main rule and border rule. The rules must be both left-toggle or both right-toggle in order to ensure the existence of a unique pre-image for each possible grid configuration. Since it applies two transition rules, the HCA model is considered a non-homogeneous (or hybrid) CA model.

During steps of forward or backward evolution with the HCA model, each transition rule is applied to a region of the grid. First, the border transition rule is applied to a region called border, which is defined as the first and last cells of the grid in the first evolution step. The main rule is then applied to each of the remaining cells of the grid.

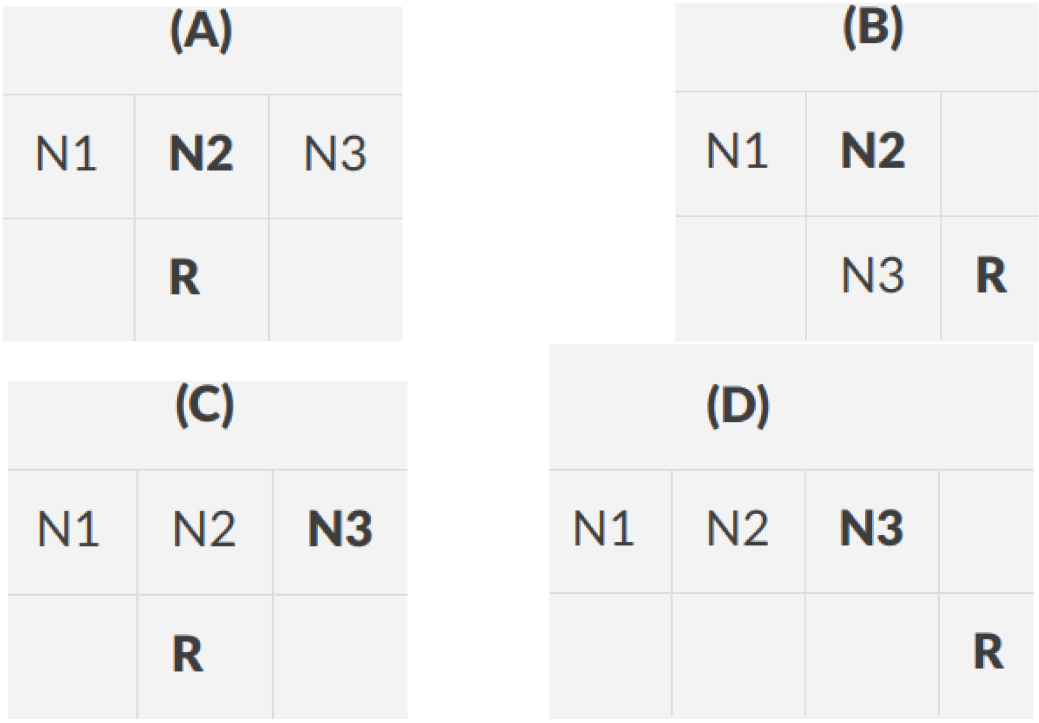

Figure 5 – Neighborhood definitions of the HCA evolution algorithms for right-toggle rules with neighbors denoted as N1, N2, N3, with main cell in bold text The cell to which a new state is assigned by the transition rule is denoted as R. **(A)**: neighborhood of the forward evolution with the main rule. **(B)**: neighborhood of the backward evolution with the main rule. **(C)**: neighborhood of the forward evolution with the border rule. **(D)**: neighborhood of the backward evolution with the border rule.

After updating the entire lattice, the border cells are shifted by one position (to the left or to the right depending on the rule sensitivity). This shift is applied after each forward/backward evolution step to enable a parallel implementation of the backward evolution algorithm.

The application of the border rule utilizes a distinct neighborhood definition when compared to the application of the main rule, as represented on Figure 5. For the application of the border rule in forward evolution, the neighbors of a cell with index i are the two cells to the left (i-1 and i-2) in the case of right-toggle rules, illustrated by Figure 5C), or to the right (i+1 and i+2) in the case of left-toggle rules, and the new state is assigned to the center cell.

When performing backward evolution with the border rule, the same neighborhood is applied, but the cell to which a state is assigned is shifted, such that the transition for the neighborhood (i-2,i-1,i) updates the cell i+1 in the case of right-toggle rules, illustrated by Figure 5D, or neighborhood (i,i+1,i+2) and updated cell i-1 for left-toggle rules. After each step, a shift is applied to the border region in order to make the application of parallel computing possible in the backward evolution across steps.

After applying the border transition rule to the border cells in an evolution step, the

main transition rule is applied to the remaining cells. Likewise, in forward evolution, the main rule expresses a transition state for a cell according to the states of its neighborhood, where the central cell of the neighborhood is updated with the resulting state, as illustrated by Figure 5A.

For backward evolution, transitions with the main rule are done sequentially due to the application of a distinct definition of neighborhood illustrated by 5B. It is composed, in the case of right-toggle rules, of the left and center cells of the previous step and the center cell of the current step, resulting in the assignment of a state to the the cell on the right. Or in the case of left-toggle rules, the neighborhood is flipped, being composed op the right and center cells of the previous step and the center cell of the current step, resulting in an update of the cell on the left. Notably, this neighborhood definition for backward evolution is the only case where information from the current step is needed to apply the transition rule, which results in the backward evolution step being a sequential procedure.

Figure 6 exemplifies the HCA forward evolution of a 8-cell grid with the right-toggle rules 101 and 170, defined in the following.

- Main rule (rule 101): $111 \mapsto 0, 110 \mapsto 1, 101 \mapsto 1, 100 \mapsto 0, 011 \mapsto 0, 010 \mapsto 1, 001 \mapsto 0, 000 \mapsto 1$
- Border rule (rule 170): $111 \mapsto 1, 110 \mapsto 0, 101 \mapsto 1, 100 \mapsto 0, 011 \mapsto 1, 010 \mapsto 0, 001 \mapsto 1, 000 \mapsto 0$

The forward evolution example from Figure 6 is detailed as follows. The border cells are highlighted with bold text on each step on Figure 6. The states of border cells are copied and shifted on each step by applying the border rule. For instance, the border cells in the initial configuration are defined as the first cell, having state 0, and the last cell, with state 1. For the first cell, the neighborhood defined for the border rule (Figure 5C) is 01**0**, corresponding respectively to the states of the seventh, eighth, and first cells, considering the periodic grid. According to this neighborhood and the corresponding border rule mapping $010 \mapsto 0$, the eighth cell is updated with the state 0. Likewise, applying the border rule to the eighth cell with state 1 and neighborhood 00**1** results in the assignment of the seventh cell to state 1. The border region for the next step is then defined as the seventh and eighth cells.

For the remaining cells with indexes $1, 2, 3, 4, 5, 6$, which were not yet updated, the main rule is applied according to the its neighborhood definition (Figure 5A). For instance, the first cell, with neighborhood 1**0**1 is updated according to the corresponding main rule mapping $101 \mapsto 1$, and the second cell, with neighborhood 0**1**0 is updated according to the corresponding main rule mapping $010 \mapsto 1$. This process continues until every cell has been updated.

The aforementioned forward evolution step procedure is then repeated for the subsequent steps of Figure 6.

Figure 7 exemplifies the HCA backward evolution of a 8-cell grid with the aforementioned right-toggle rules 101 and 170. The example is detailed in the following.

The border cells are highlighted with bold text on each step on Figure 7. Similarly to forward evolution, the states of border cells are copied and shifted on each step by applying the border rule, but the shift direction is reversed. The border cells in the initial configuration are likewise defined as the first cell, having state 0 in the example, and the last cell, with state 1. For the first cell, the neighborhood defined for the border rule (Figure 5D) is 11**0**, corresponding respectively to the states of the seventh, eighth, and first cells. According to this neighborhood and the corresponding border rule mapping $110 \mapsto 0$, the second cell grid is updated with the state 0. Likewise, applying the border rule to the eighth cell with state 1 and neighborhood 01**1** results in the assignment of the first cell to state 1. The border region for the next step is then defined as the first and second cells.

For the remaining cells with indexes $3, 4, 5, 6, 7, 8$, which were not updated yet, the main rule is applied according to the its neighborhood definition ((Figure 5B)). For instance, considering the state assignment for the third cell, with neighborhood states 0**1**0, from the respective states of the first and second cells of the initial configuration and the second cell of step 1, the main rule mapping $010 \mapsto 1$ is then applied to the third cell. Likewise, the forth cell, having neighborhood 1**0**1, is updated according to the corresponding main rule mapping $101 \mapsto 1$. This process continues until every cell has been updated.

The aforementioned backward evolution step procedure is then repeated for the subsequent steps of Figure 7.

As previously mentioned, the shift of the border region in both evolution algorithms enables the parallel execution of backward evolution. While a backward evolution step was shown to be a sequential process due to the dependency of its main rule neighborhood (corresponding to cell N3 on Figure 5B), the simultaneous processing of distinct steps is enabled by the shifting border region. Referring to the Figure 7 example, during step 1 of evolution, the processing of the border cells, which results on the updating the first cells, enables step 2 to start processing its border cells and assign the states of cells 2 and 3. In the same way, step 3 can start its processing of cells 3 and 4 after the border cells of step 2 have been assigned etc.

For the remaining cells, the processing of each cell also enables the processing of a cell on the next step. For instance, the processing of cell 3 on the first step enables the processing of cell 4 on step 2, which enables the processing of cell 5 on step 3. In this manner, multiple steps can be executed simultaneously, where periodic synchronizations are needed to ensure that the necessary data from one is step is available for the processing

initial	**0**	1	0	1	1	0	0	**1**
step 1	1	1	1	0	1	0	**1**	**0**
step 2	0	0	1	1	1	**1**	**0**	1
step 3	0	0	0	0	**1**	**0**	1	1
step 4	0	1	1	**1**	**0**	1	0	1

Figure 6 – An example of forward evolution of the HCA model using rule 101 as the main rule and rule 170 as the border rule. The two border cells in every step are in bold text, and their position shifts left on every step.

initial	**0**	1	0	1	1	0	1	**1**
step 1	**1**	**0**	1	1	0	1	1	0
step 2	1	**1**	**0**	1	0	0	0	0
step 3	1	0	**1**	**0**	1	1	1	1
step 4	0	1	1	**1**	**0**	1	0	1

Figure 7 – An example of backward evolution of the HCA model using rule 101 as the main rule and rule 170 as the border rule. The two border cells in every step are in bold text, and their position shifts left on every step.

of the next step.

2.2.3 Genetic Algorithm

The genetic algorithm (GA) is a search method proposed by John Holland, which is inspired by the evolution of the species and has been applied to problems related to search and optimization, including the scheduling problem (GOLDBERG, 1989). Being an optimization technique, the GA aims to generate a globally optimal solution according to the application and a given metric that quantifies the quality of a solution.

The GA performs its search by iteratively applying a series of stochastic operators to a set of solutions (population) and thus generating new solutions by combining and modifying them. Thus, the application of GA requires the formulation of a domain-specific data representation and genetic operators, described as follows.

The genetic representation is a data structure that is representative of the complete solution domain of the problem, i.e. it is capable of effectively codifying every single solution that belongs to the search space of the problem. An instance of this data structure is called an individual or chromosome, which compose the population of the GA. The classic genetic representation is a binary vector (GOLDBERG, 1989).

A procedure for generating valid individuals must be defined in order to compose the population during the initialization stage of the GA. Individual generation is usually done with random assignments, resulting in a random string of bits in the case of binary vector representation.

The chosen metric of the quality of an individual is called fitness, which is the value to be optimized by the GA. The fitness of an individual is calculated by an evaluation function.

The crossover operator is the procedure which generates new individuals from a given pair of parent individuals such that the children individuals are assigned features from both parents. A common method for crossover is the single-point crossover, which divides each parent's chromosome into two parts and recombines them together. Other approaches include the two-point crossover, which divides chromosomes into three parts, and PMX, an algorithm designed to handle permutation-based genetic representations.

The mutation operator is a procedure that modifies some of the children generated by crossover with a probability given as a parameter of the GA. A common method for mutation consists in toggling a random bit of the individual, in the case of binary genetic representations.

Besides the domain-specific genetic operators described, the GA also holds the following population operators, which are applied to the population iteratively.

The selection operator is used to define which individuals will reproduce with the crossover operator on each iteration of the algorithm. The likelihood of an individual to be selected for crossover is usually based on its fitness, such that better individuals have

higher probability of reproduction. A common method for selection in GA applications is the roulette, where the probability of each individual being selected is directly proportional to its fitness in comparison to the rest of the population. Another method is the binary tournament, where two randomly chosen individuals are compared and the fittest one is selected.

The reinsertion operator determines the individuals who will compose the population for the next iteration by combining the current population and the list of children. Methods for reinsertion include substitution, where the current population is fully replaced by the children generated, and elitism, where a portion of the best individuals of the current population is preserved while the remainder of the population is composed of the children.

Additionally, the operation of the GA is controlled by the following parameters.

- ❏ Population size: an integer number that defines the quantity of individuals to compose the population.
- ❏ Number of generations: an integer number that defines the number of iterations (generations) to be performed.
- ❏ Crossover rate: a percentage that indicates the number of child individuals to be generated in a generation, in relation to the size of the population.
- ❏ Mutation rate: a percentage that defines the probability of a child individual to be modified with the mutation operator.

With the described operators and parameters, the general GA procedure is defined on Listing 2.1, with number of generations represented by g and mutation rate represented by m_rate. As its execution ends, the fittest individual of the population is returned, representing the best solution found with the given parameters.

Listing 2.1 – The genetic algorithm with parameters population size n, number of generations g, crossover rate c, mutation rate m.

```
1  procedure genetic_algorithm:
2      initialize population P with n individuals
3      evaluate each individual from P
4      for g generations:
5          select c parent individuals
6          generate c children via crossover
7          for each child generated:
8              mutate child with probability m
9              evaluate child
10         perform reinsertion between P and children
11     return fittest individual
```

The total individual evaluations performed by the GA represents the number of solutions explored in the search space of the corresponding problem instance. Evaluations occur during the initialization stage and after generating each child with crossover/mutation. The total number of evaluations is then given by $n + g * n * c$ where n is the population size, g is the number of generations and c is the crossover rate.

Since the GA operates with stochastic processes, its efficiency must be measured statistically with multiple executions for a given problem instance and set of parameters.

The lack of dependencies between the genetic operators, which are independently performed on multiple individuals, reveal a significant potential for the application of parallel computing to GA.

2.2.4 Multipopulation Genetic Algorithm

Considering the applicability of parallel computing, multiple approaches have been developed with the goal of obtaining high performance of its execution on parallel computers. These approaches can be categorized as follows (CANTÚ-PAZ, 1998).

- ❏ Global parallelism approaches employ worker processing units to perform genetic operations while the main algorithm is controlled by the main processing unit.
- ❏ Fine-grained approaches define limited neighborhoods for each individual, with whom they compete and/or reproduce instead of interacting with the entire population. These approaches are intended for execution on massively parallel computers with high numbers of processing units.
- ❏ Multipopulation approaches employ a number of semi-independent populations instead of one, with communications between populations occurring with some frequency.

In particular, the fine-grained and multipopulation approaches can be considered distinct algorithms from the GA, as they feature local interactions between subsets of the population, which can affect the quality of the solutions. For this reason, some authors suggest that the use of these approaches is beneficial even if their parallel execution is unavailable (ALBA; TROYA, 1999).

The multipopulation genetic algorithm (MPGA) is a popular variation of the GA model intended to achieve higher performance through the application of parallel computing. It extends the GA model by employing a number of independent populations that communicate through a procedure called migration, where individuals are exchanged between populations.

The definition of the MPGA makes it highly suitable for distributed memory implementations, since each population is evolved independently and the communication

mechanism between them is defined as the migration operator. In the distributed memory case, each process is comparable to a GA that evolves its single population and also communicates with other processes through migration.

The MPGA is also referred to as distributed GA, for being suitable for distributed memory implementations; coarse-grained GA, for having relatively infrequent communications compared to the fine-grained approach; and island-based GA, for its similarity with the island model from the field of population genetics. Each population of the MPGA is also referred to as a subpopulation or island.

Besides the aforementioned genetic operators and parameters, the MPGA is also composed of the following definitions.

The migration operator is defined as a population sending a portion of its current individuals to another population, such that the arriving individuals replace the departing ones in the destined population, resulting in unchanged population sizes.

The selection of individuals to depart for migration is defined as a portion of the fittest individuals of the population. The number of migrating individuals is expressed as a percentage of the population size by the parameter migration rate.

The number of migrations that occur during an execution of the MPGA is defined by the migration frequency parameter. This parameter defines which generations will feature migration. For example, it can denote that migration will occur on every 10 generations.

The populations that will serve as destination for each migration must be defined in a communication topology, which establishes every connection between populations. An usual approach is the circular/ring topology, where each population has a single destination for sending migrations as well as a single source to receive migrations.

The population quantity parameter denotes the number of populations that will compose the MPGA. The population size parameter is then divided between populations, e.g., 4-population MPGA with population size 100 has populations of 25 individuals each. This is done in order to maintain the same quantity of individual evaluations in comparison to the GA, thus enabling an appropriate performance comparison.

The particular case where the population quantity of the MPGA is 1 is equivalent to the execution of the corresponding sequential GA, without migration nor parallel computing.

The MPGA parameters can also be set to not perform migration. For instance, in the case of MPGA set to run for 100 generations, a migration frequency value of 100 or higher (migration occurring on every 100 generations) implies that migration will not be performed. This case, where populations are evolved independently, is equivalent to performing multiple runs of the corresponding GA with reduced population size. This approach is discouraged, since it degrades the quality of the solutions when compared to the other approaches (CANTÚ-PAZ; GOLDBERG, 2003).

2.3 Parallel Computing

Parallel computing refers to the simultaneous execution of computing tasks by a set of processor units, in contrast to the traditional paradigm of sequential execution of algorithms (QUINN; QUINN, 1994). The application of parallel computing aims at achieving higher efficiency of programs in terms of execution time, and is dependent on a series of resources and paradigms detailed in the following.

The primary resource for parallel computing is the hardware capable of simultaneous execution of tasks, generally called a parallel computer architecture. Modern parallel computing approaches are made possible by classes of architecture including the following (SCHMIDT et al., 2017).

- ❏ Multicore processors (CPU) are general-purpose processors that include multiple processing units (cores) on the same chip, such that each core has its own memory registers and cache as well as independence in the execution of instructions, i.e., each core performs as a complete independent processor. Currently, multicore processors are usually available in personal computers.

- ❏ Graphics processing units (GPU) are specialized processors developed for high performance of computer graphics processing, which are based on linear algebra and matrix operations. For this, GPUs are composed of numerous cores, but each one has less resources if compared to multicore processors. The usage of GPU for general purpose computing, besides graphics processing, is made possible by programming frameworks such as Nvidia CUDA (NVIDIA; VINGELMANN; FITZEK, 2020). GPUs are usually available in personal computers.

- ❏ Field-programmable gate arrays (FPGA) are specialized integrated circuits composed of numerous logic gates and memory blocks, which can be configured for highly efficient computing using hardware description languages such as VHDL. Algorithms can be implemented for execution in FPGA by expressing the algorithm in sequences of logic gate operations, thus presenting considerable difficulty to utilize. FPGA are not available in consumer-oriented computers, being used for specialized applications.

- ❏ Cluster computing is the association of multiple independent computers connected by a network and configured to function as a single parallel computer, e.g., multiple personal computers connected via a local area network. As such, the application of clusters can be relatively accessible according to the monetary resources available to projects.

Moreover, memory management in parallel computing can be achieved in two ways: distributed memory or shared memory (SCHMIDT et al., 2017).

In distributed memory applications, each processing unit operates mainly by using a reserved memory space, which is logic and/or physically distributed, and generally inaccessible to other processing units. A mechanism for communication between processing units is usually applied to enable information sharing when needed. This approach then provides an encapsulation of each parallel task, which is useful for applications where tasks are highly independent, requiring infrequent communications and/or synchronizations. Computer architectures based on highly independent processing units generally benefit the most from distributed memory approaches, including multicore processors and clusters.

Shared memory applications mainly employ a unified memory space to each processing unit, which can be accessed freely by each one. This approach facilitates the communication between processing units, while the lack of encapsulation may require additional memory management effort in programming. The shared memory is then more appropriate for applications with frequent communications and/or synchronizations. Computer architectures which provide shared memory modules to their processing units are the most benefited by this approach, including multicore processors and modern GPUs.

Software resources to achieve parallel computing implementations in multicore processors include the use processes and threads that are made available by the operating system (SCHMIDT et al., 2017).

Processes are independent execution instances of a program, and their use for parallel computing usually represent the application of the distributed memory paradigm. Communication between processes is achieved with specific operating system directives such as the ones specified by the MPI (message passing interface) standard. MPI defines directives such as process creation, sending/receiving data structures, and broadcasting information between processes.

Threads are simplified instances of execution that compose a process. Multiple threads in a process share its memory resources while being able to be executed simultaneously. Thus, the use of threads for parallel computing represents the application of the shared memory paradigm. Communication between threads is relatively trivial, as their memory resources are accessible to each thread.

In the case of graphics processing units following the CUDA framework, the concept of threads is employed with some differences. As GPUs are specialized on operations over matrix data structures, the CUDA framework provides the parameterization of a thread hierarchy based on the locality of the data according to the data structure to be processed. To this end, functions for execution on a CUDA GPU can be configured to employ either a 1D, 2D or 3D thread hierarchy such that this dimensionality corresponds to the input data structure and enables the mapping of each element of the data structure to a thread. CUDA threads can access the global memory of the GPU device as well as the device shared memory, which enables efficient communication. Each CUDA thread also holds a

limited space of local memory. Considering that each thread has limited resources, GPU applications usually reflect the shared memory paradigm (SCHMIDT et al., 2017).

The efficiency gain obtained by the application of parallel computing to a given program is constrained by the dependency of its subtasks. An application is said to exhibit fine-grained parallelism if its subtasks require frequent communication, or coarse-grained if communication is infrequent (Bernstein, 1966). Generally, coarse-grained applications benefit more from parallel computing, since inter-process communication represents added overhead when compared to serial implementations which do not require communication.

A third category of parallel programs is called embarrassingly parallel tasks, which have either zero or negligible amounts of subtask dependencies, thus barely requiring any communication (SCHMIDT et al., 2017). This is the case for algorithms like matrix multiplication, where the computation of each resulting matrix element is entirely independent of the others.

The speedup is a metric used to express the efficiency gain when comparing the execution times of a serial program and its parallel version. For example, the case of a serial program and a parallel program with respective execution times of 10 minutes and 5 minutes exhibit 2X speedup, as the complete execution occurred twice as fast in the second case. Speedup is limited to the number of processing units applied, e.g., an application that makes use of a 4-core processor would ideally exhibit 4X speedup when compared to a serial implementation. Speedup values are usually below the ideal value, since parallel implementations have the added overhead of communication, which is not present in serial implementations.

Throughput is a metric that expresses the number of operations per unit of time performed by different programs, thus providing a measure of performance according to input size. For example, a program that takes 10 seconds to process 2000 bytes of data has throughput of 200 bytes per second.

In this work, approaches with shared memory over threads with CPU and GPU were applied to the implementations of the HCA evolution algorithms, while approaches based on distributed memory with processes and shared memory with threads were applied to the implementations of the MPGA.

2.4 Multiobjective optimization

Multiobjective optimization is the area of study of optimization problems with multiple objective functions (MIETTINEN, 1999). In this context, the optimality of solutions to a problem is measured considering every given objective simultaneously, which might result in trade-offs between conflicting objectives. Solutions in the context multiobjective optimization usually are not optimal for every objective. Thus, the concept of Pareto dominance (MIETTINEN, 1999) is employed for the comparison of two solutions, defined

as follows.

Given a multiobjective problem with n objective functions $f_1, ..., f_n$ to be minimized and two solutions x_1, x_2, x_1 dominates x_2 if $f(x_1) \leq f(x_2)$ for every function and $f(x_1) < f(x_2)$ for at least one function, i.e., x_1 dominates x_2 if x_1 is better than x_2 in at least one metric and equivalent or better in the others. The same concept applies for maximization functions, which are equivalent to minimizing its negative.

Given the definition of Pareto dominance, a solution is defined as Pareto optimal (or non-dominated) if it is not dominated by any other solution. The set of non-dominated solutions is called Pareto frontier (MIETTINEN, 1999).

Solutions for multiobjective problems can be visualized with scatter plots as exemplified by Figure 8. Each axis of the chart represents an objective function, with functions f1 and f2 represented respectively by the axes y and x in the example. Solution C is shown to be dominated by both A and B for both functions. Solutions A and B are shown to be non-dominated, despite A being better at minimizing function f2 and B being being better at minimizing f1. Considering all the solutions presented in the example of Figure 8, the Pareto frontier is outlined, being composed by 9 solutions and including A and B.

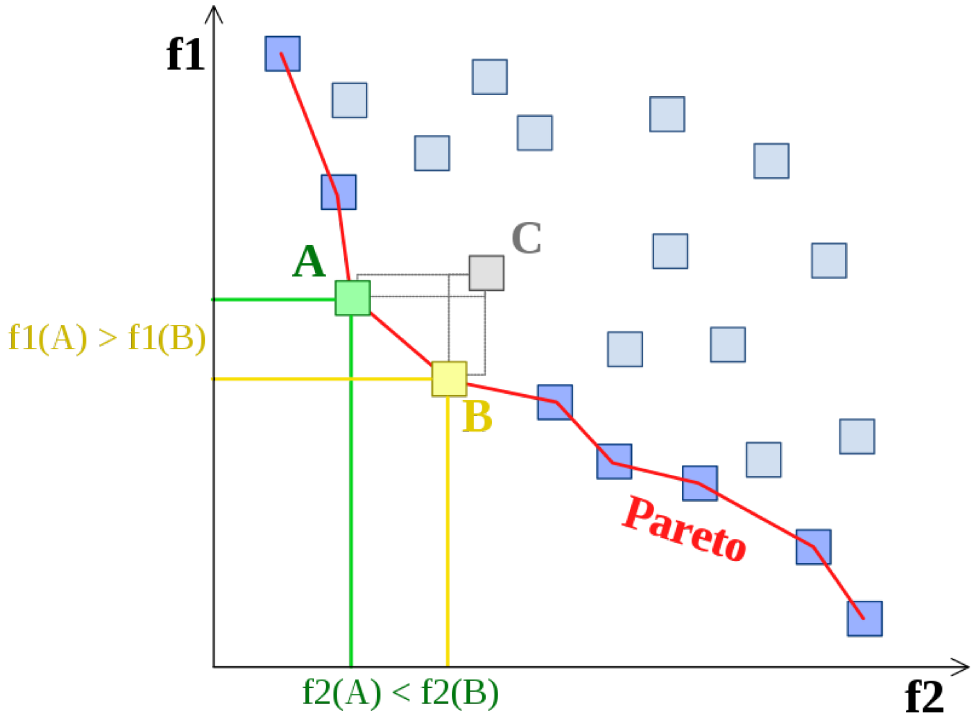

Figure 8 – Example of multiobjective chart with objective functions f1 and f2 represented respectively by the axes y and x. The Pareto frontier is outlined in red, and solution C is shown to be dominated by A and B. Source: (DRéO, 2006).

In this work, the concept of multiobjective optimization is applied as a methodology to analyze the results of the experiments with MPGA applied to task scheduling, where the variation of parameters are shown to influence in two performance metrics: execution time

of the algorithm and makespan of the solutions produced. However, the search performed with MPGA is applied for the optimization of a single objective, being the makespan.

Chapter 3

Review of the Literature

This chapter presents a review of related works that involve applications of CA to cryptography, parallel CA implementations, applications of GA to task scheduling and approaches to the implementation of MPGA.

3.1 Cellular Automata applied to Cryptography

The application of CA to cryptography started with Wolfram (1986). This model consists of a stream cipher that makes use of a 1D binary CA and a chaotic transition rule. With this CA, it is possible to generate an effectively random bit sequence, which is then used for encryption using the XOR operator. In this model, the cryptographic key is the initial configuration of the CA, which is evolved for a number of steps by the chaotic transition rule, resulting in an effectively random bit string. Notably, the utilization of CA evolution in this model is exclusively applied to the generation of the random bit string, while the encryption process is done with the XOR operator.

Later models explore the algebraic properties of additive CA, which involve multiple transition rules composed by XOR and XNOR operations (NANDI; KAR; CHAUDHURI, 1994), (SEN et al., 2002). However this kind of approach was show to be vulnerable to cryptanalysis via chosen-plaintext attack (BLACKBURN et al., 1997).

The method developed by Gutowitz (1993) is based on dynamical systems and performs block cipher in 384-bit blocks via the evolution of the CA itself. Its cryptographic key defines a transition rule with the toggle property, and the encryption algorithm is given by the backward evolution of the CA over the bits of the plaintext, while the decryption is done by the forward evolution of the CA. The encryption process is done with the use of toggle rules and the insertion of extra cells in each evolution step when doing the backward evolution. These extra bits are used to define a neighborhood for the cells in the lattice border. They are employed in the backward evolution to ensure that the pre-image computation is always possible for any lattice configuration. In the forward

evolution, the extra cells are removed on every evolution step. Thus, this method results in ciphertext that is significantly longer than the original plaintext.

Later CA-based methods were developed to address two problems that are present in the Gutowitz (1993) model, being the increased length of ciphertext and the unidirectional spread of plaintext disturbances. The work (OLIVEIRA; COELHO; MONTEIRO, 2004) solved the second flaw by using CA rules that have both right-toggle and left-toggle properties. A general method for pre-image computation was investigated by (WUENSCHE, 2008; OLIVEIRA et al., 2008), which were limited by the challenge of assuring the existence of a unique pre-image for every possible CA lattice. Further investigation included the application of wrap procedures that ensured the application of the encryption procedure to any given plaintext (OLIVEIRA et al., 2010; OLIVEIRA et al., 2010; OLIVEIRA; MARTINS; ALT, 2011). However, these approaches still resulted in ciphertext with increased length.

The HCA model (MACÊDO, 2007; OLIVEIRA; MACEDO, 2019) is based on the method by Gutowitz (1993). It applies a periodic neighborhood and two transition rules instead of one, the main rule and border rule, which are both left-toggle or both right-toggle in order to ensure the existence of a unique pre-image for each possible grid configuration. With these features, the HCA model achieves an encryption method through forward and backward evolution without an increase in the length of the ciphertext.

Recently, a model inspired by HCA was proposed, replacing the CA structure by complex networks (MACÊDO; OLIVEIRA; RIBEIRO, 2014). Although some advantages were observed in the fast propagation of information promoted by non-local connections, the intrinsic parallelism of CA models is not presented in this model.

3.2 Parallel Computing approaches to Cellular Automaton evolution

As previously discussed, the general definition of CA specifies an evolution algorithm where, on each evolution step, the next state of each cell can be computed independently via the application of the transition rule to its neighborhood. Some approaches to the parallel execution of CA evolution are presented in the following.

Execution of CA on GPU has been applied to domains such as the simulation of urban development (XIA et al., 2018), image segmentation (LÓPEZ-FANDINO et al., 2016), modeling of aircraft disembarking and emergency evacuation (GIITSIDIS; DOURVAS; SIRAKOULIS, 2017), simulation of wildfire spreading (NTINAS et al., 2016). The high number of processing threads that is present on GPU is shown to be pertinent to the parallel execution of CA, such that every thread is responsible for one cell of its grid.

Execution of CA on FPGA has been applied to domains such as random number generation (DEVI; CHITHRA; SETHUMADHAVAN, 2019), modeling of aircraft disembarking

and emergency evacuation (GIITSIDIS; DOURVAS; SIRAKOULIS, 2017), simulation of wildfire spreading (NTINAS et al., 2016). In this approach, the CA evolution process is programmed in hardware description languages such as VHDL as a parallel array of logic gates. Being dedicated hardware, FPGA is regarded as the highest performing approach for CA evolution, while also being the least accessible due to its financial cost and difficulty to program.

Another special-purpose parallel hardware is described by (TOFFOLI, 1984), which performs the simultaneous processing of every cell with the use of a parallel processor optimized to the execution of cellular automata and similar discrete systems.

3.3 Genetic Algorithm applied to Task Scheduling

A number of works were developed regarding the application of GA to task scheduling, while the works by (KWOK; AHMAD, 1997) and (MORADY; DAL, 2016) performed applications of MPGA to task scheduling. This section presents approaches adopted by these related works for the implementation of genetic operators in the domain of task scheduling.

The genetic representation employed by earlier works involves lists of tasks, where each list represents the tasks assigned to a corresponding processor (HOU; ANSARI; REN, 1994; CORREA; FERREIRA; REBREYEND, 1999; KAUR et al., 1999). More recent works utilize a simplified representation using two vectors that represent an ordering of tasks and their processor assignments (WANG et al., 1997; OMARA; ARAFA, 2010; CHITRA; RAJARAM; VENKATESH, 2011; MORADY; DAL, 2016).

The minimization of makespan is usually considered as the fitness metric in scheduling applications (HOU; ANSARI; REN, 1994; WANG et al., 1997; CORREA; FERREIRA; REBREYEND, 1999; HWANG; GEN; KATAYAMA, 2008; OMARA; ARAFA, 2010; KWOK; AHMAD, 1997; MORADY; DAL, 2016). Other metrics include flowtime (KAUR et al., 1999), the sum of conclusion times for each task, and reliability (CHITRA; RAJARAM; VENKATESH, 2011), which quantifies possible processor failure scenarios.

The generation of the initial population in these works is usually done with randomly generated solutions (HOU; ANSARI; REN, 1994; CORREA; FERREIRA; REBREYEND, 1999; HWANG; GEN; KATAYAMA, 2008; CHITRA; RAJARAM; VENKATESH, 2011; MORADY; DAL, 2016). Some works also apply heuristics to generate part of the initial population (WANG et al., 1997; KWOK; AHMAD, 1997; KAUR et al., 1999; OMARA; ARAFA, 2010).

A common method for crossover is the single-point crossover, which divides each parent's chromosome into two parts and recombines them together (WANG et al., 1997; KWOK; AHMAD, 1997; CHITRA; RAJARAM; VENKATESH, 2011; KAUR et al., 1999; OMARA; ARAFA, 2010; XU et al., 2014). Other approaches include the two-point

crossover, which divides chromosomes into three parts, and PMX, an algorithm designed to handle permutation-based genetic representations (HWANG; GEN; KATAYAMA, 2008; MORADY; DAL, 2016).

A common method for mutation in scheduling applications is swap, which performs an exchange in the execution sequence of two tasks, possibly implying in a switch of their processor assignments as well (HOU; ANSARI; REN, 1994; WANG et al., 1997; KWOK; AHMAD, 1999; HWANG; GEN; KATAYAMA, 2008; CHITRA; RAJARAM; VENKATESH, 2011; XU et al., 2014). Another approach is the switch of the processor assignment of a task (OMARA; ARAFA, 2010; MORADY; DAL, 2016).

A common method for the selection of parents for crossover is the roulette, where the probability of each individual being selected is directly proportional to its fitness in comparison to the rest of the population (HOU; ANSARI; REN, 1994; WANG et al., 1997; HWANG; GEN; KATAYAMA, 2008; KAUR et al., 1999; CHITRA; RAJARAM; VENKATESH, 2011; XU et al., 2014). Another method is the binary tournament, where two randomly chosen individuals are compared and the fittest one is selected (OMARA; ARAFA, 2010; MORADY; DAL, 2016).

Reinsertion is usually performed with the elitism method, where a portion of the best individuals of the current population is preserved while the remainder of the population is composed of the children (HOU; ANSARI; REN, 1994; WANG et al., 1997; KWOK; AHMAD, 1997; CORREA; FERREIRA; REBREYEND, 1999; HWANG; GEN; KATAYAMA, 2008; KAUR et al., 1999; CHITRA; RAJARAM; VENKATESH, 2011; XU et al., 2014; MORADY; DAL, 2016).

3.4 Approaches with the Multipopulation Genetic Algorithm

Related works employing the MPGA referenced in this section are applications to task scheduling (KWOK; AHMAD, 1997; MORADY; DAL, 2016), job-shop scheduling (QI; BURNS; HARRISON, 2000), data mining (SRINIVASA; VENUGOPAL; PATNAIK, 2007), container loading (GEHRING; BORTFELDT, 2002), function optimization (MüHLENBEIN; SCHOMISCH; BORN, 1991; YAO; KHARMA; GROGONO, 2010) and combinatorial optimization (HAN et al., 2001).

The communication topology defines the destinations considered for the migration operator for each population. Approaches for communication topology are described as follows. Fully connected populations denote that migrations can occur between any pair of populations (KWOK; AHMAD, 1997; QI; BURNS; HARRISON, 2000; SRINIVASA; VENUGOPAL; PATNAIK, 2007). Circular topologies define one set destination for each population (GEHRING; BORTFELDT, 2002; MORADY; DAL, 2016). Other approaches include a variation of the circular topology with multiple connections between populations

(MüHLENBEIN; SCHOMISCH; BORN, 1991), and a dynamical topology with merging and divisions of populations during execution (YAO; KHARMA; GROGONO, 2010).

The occurrences of migration are usually defined as the fixed migration frequency parameter of the algorithm, which determines the generations where migration will occurs (MüHLENBEIN; SCHOMISCH; BORN, 1991; QI; BURNS; HARRISON, 2000; HAN et al., 2001; GEHRING; BORTFELDT, 2002; SRINIVASA; VENUGOPAL; PATNAIK, 2007; MORADY; DAL, 2016). Alternatively, KWOK; AHMAD apply a varying migration frequency that increases exponentially during execution.

The selection of individuals for migration is usually defined as a portion of the fittest individuals of the population (MüHLENBEIN; SCHOMISCH; BORN, 1991; KWOK; AHMAD, 1997; QI; BURNS; HARRISON, 2000; HAN et al., 2001; GEHRING; BORTFELDT, 2002; SRINIVASA; VENUGOPAL; PATNAIK, 2007; MORADY; DAL, 2016).

The parallel computing approaches for the execution of MPGA usually define the assignment of each population to a processing unit and make use of distributed memory, including the following applications: massively parallel supercomputer (KWOK; AHMAD, 1997), clusters of personal computers (MORADY; DAL, 2016), multiple computers over a local network that communicate via a central database (GEHRING; BORTFELDT, 2002). For the other related works (MüHLENBEIN; SCHOMISCH; BORN, 1991; HAN et al., 2001; QI; BURNS; HARRISON, 2000; SRINIVASA; VENUGOPAL; PATNAIK, 2007; YAO; KHARMA; GROGONO, 2010), it was not possible to identify which parallel computing approach was used in the implementation of their respective MPGAs. Notwithstanding, it is assumed that they were implemented with some kind of parallelism, as it is one of the basic motivations for the adoption of MPGA models.

Chapter 4

Development

This chapter presents the major development activities of this work. The following sections describe the preliminary experiments with parallel computing in Python, which use the matrix multiplication algorithm as an example, the development of the sequential and parallel versions of the forward and backward evolution algorithms of the HCA model, and the development of the MPGA with the distributed memory and shared memory approaches.

4.1 Preliminary experiments with parallel computing in Python

Python has been one of the most popular languages in academia and in scientific computing in the recent years for its ease of use and great availability of libraries. This popularity occurs despite its weak performance, which motivated the efforts devoted by the Python community to improve it, with various solutions been developed. Each of these solutions adopt different approaches and various optimization techniques. Among them, Numba (LAM; PITROU; SEIBERT, 2015) is a library for optimization that provides high-performance and performance portability for Python applications. Python code acceleration tools, such as Numba, allow access to high-performance programming with a significantly lower technical barrier when compared to languages like C and FORTRAN (MAROWKA, 2018).

The majority of the experiments in this work were implemented in Python using Numba. Numba is a Python compiler that uses the LLVM platform to provide high performance serial execution as well as parallel execution for multi-core CPUs and also GPUs over the Nvidia CUDA platform (LAM; PITROU; SEIBERT, 2015).

The matrix multiplication algorithm is used in this section as a benchmark example for Numba's features and performance, demonstrating the ease of use of these tools and the speedup that can be achieved in a highly parallel application. The Python code provided

by (MAROWKA, 2018; Anaconda, Inc., 2012) is then employed to provide an overview of the methods employed for both serial parallel execution in this work. Each implementation is described as follows, and the corresponding code is found on the appendix A.

The pure Python implementation of matrix multiplication employs three nested for-loops, as shown on listing A.1 (MAROWKA, 2018).

The serial implementation using Numba consists of adding the decorator `@njit` to the previous Python implementation, as shown on listing A.2. This indicates the function is set be optimized by Numba's Just-in-Time compiler in *nopython* mode, which generates high-performance code while ensuring its execution does not fall back to the slower Python interpreter.

The CPU parallel implementation using Numba consists of using the decorator `@njit (parallel=True)` to set it up for optimization over multiple CPU cores. Then, loops can be set up for parallel execution by using Numba's `prange` function instead of the standard Python `range` function. For the matrix multiplication, every one of its three for-loops can be marked for parallel execution with `prange`, since every element of the resulting matrix can be calculated independently of the rest. The resulting code in shown on Listing A.3.

The GPU parallel implementation using Numba consists of the decorator `@cuda.jit`, which sets a function up for compilation for the Nvidia CUDA execution model. The function also needs to be adapted into a *kernel*, which is a function to be executed by each single CUDA thread. This kernel is presented on Listing A.4. Note that each thread computes the result for a single element of the output matrix C, and its 2D coordinates are given by the CUDA API itself via the function `cuda.grid`. This function handles spacial abstraction in CUDA programming, with support for 1, 2 or 3 dimensions. A user-defined thread hierarchy is also set by the parameters *threads per block* and *blocks per grid*, which defines the parallel computing model and can be fine-tuned *ad hoc* (NVIDIA Corporation, 2007). CUDA implementations can also be optimized by making use of its shared memory capabilities, and a second implementation using this feature is also included in the following benchmark, with code provided as an example in the Numba official documentation (Anaconda, Inc., 2012).

Preliminary experiments were conducted for each program implemented for matrix multiplication by computing the product of two random 320x320 matrices aiming to investigate the performance of each approach. The hardware employed was a Ubuntu 20.04 laptop with CPU Core i5-7300HQ @ 2.50GHz x 4 and GPU GeForce GTX 1050. Figure 9 shows the resulting mean execution times of these experiments, while omitting the result with the pure Python program for clarity in the scaling of the graph. The pure Python implementation had 22.4 seconds of execution time. The Numba serial implementation ran in 41.51 ms, a 542X speedup compared to the pure Python implementation. The parallel implementation on CPU ran for 19.39 ms, a further 2.1X speedup compared to the serial Numba, while the initial GPU implementation had 2.9X speedup (14.4 ms).

The shared memory CUDA implementation had a much more significant improvement at 15.2X speedup (2.72 ms) over the serial Numba. With this in mind, each of the following CUDA implementations in this work will make use of the shared memory feature.

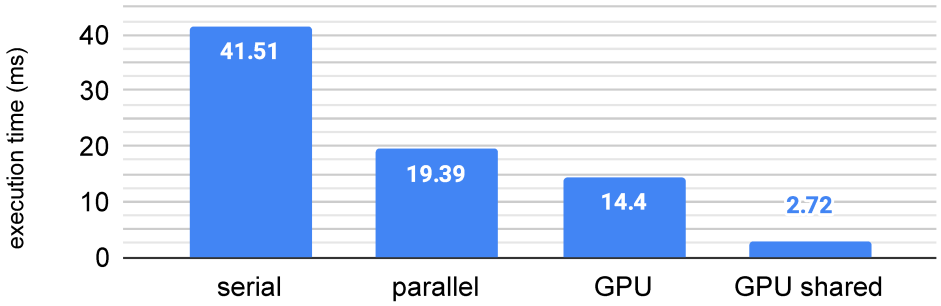

Figure 9 – Execution time in milliseconds of the matrix multiplication experiments with 320x320 matrices. The pure Python implementation was omitted, being 22490ms.

The experiments in this section demonstrate the effectiveness of the Python language for the application of parallel computing aimed at CPU and GPU platforms through the Numba library, besides the optimization that is provided to sequential programs by the Numba compiler. The accessibility of this technology is demonstrated by the simplicity of the code provided on Appendix A and the devices to which it can be applied, being consumer-focused CPU and GPU hardware. Therefore, the Python language is also employed is this for the development of the HCA and MPGA models, and the Numba library is employed for the implementations of the HCA evolution algorithms and the shared memory MPGA variation.

4.2 HCA evolution algorithms

The development and experiments regarding the HCA model in this work concern the investigation of empirical results related to the application of parallel computing to its two main algorithms: the forward evolution, which has relevance to most CA applications, and backward evolution, which has relevance to Cryptography and other fields that may benefit from reversible CA. In the following, the corresponding development is explained in detail, which employ the tools of the Numba library described in the previous section.

4.2.1 Forward evolution

The forward evolution is the main algorithm associated with CA in general, defined as the iterative application of the transition rules to every cell for n steps.

A Python implementation of the forward evolution takes the following arguments. Grid: an integer array representing the grid of cells to be evolved; main transition rule

and border transition rule: integer arrays representing the output values of a CA rule for inputs $000, 001, 010, ..., 111$ taken as the integer indexes $0, 1, 2, ..., 7$; steps: the number of steps to be evolved. The border region is defined as the first and last cells of the grid, and is shifted after each step. Each step starts by computing the border cells, and each of the remaining cells is computed by applying the main rule over the neighborhood. A second array is used in the program to store the new values of each cell and the two grids are swapped after each step. Listing 4.1 presents the forward evolution algorithm for right-toggle rules.

Listing 4.1 – The forward evolution algorithm for right-toggle rules, given an input grid, number of steps, the border transition rule and the main transition rule.

```
1 procedure forward_evolution:
2     initialize output grid
3     initialize the border as the first and last cells
4     for s steps:
5         apply the border rule to the border cells
6         apply the main rule to each remaining cell
7         swap input grid and output grid
8         shift the border to the left
9     return output grid
```

From the Python implementation, an optimized serial implementation of the forward evolution algorithm (FCA) is achieved by adding the `@njit` decorator to the declaration of the function, as explained in the previous section.

The parallel implementation is achieved by using Numba's `prange` function. However, unlike the matrix multiplication example, each step of the forward evolution depends on the previous step. Therefore, the computation of each evolution step cannot be executed in parallel. On the other hand, the inner for-loop of the this program can be set up with `prange`, since each cell can be computed independently in the same step.

The CUDA implementation of the algorithm is described on Listing 4.2, consisted of adapting the function such that each CUDA thread corresponds to one cell in the CA grid. A single array in shared memory is enough to store the grid state in every step, as each thread can store the rule output in its local memory until every thread is ready to update the grid. In order to update the shared grid effectively and avoid racing conditions, every thread must be synchronized using the CUDA `syncthreads` function. This resulted in two limitations for the thread hierarchy in this implementation: the only possible number for the blocks per grid parameter is 1, since the `syncthreads` function can only synchronize threads in the same block; the maximum number of cells in this implementation is 1024, which is the maximum number of threads per block on the current versions of the CUDA API. These limitations of thread hierarchy imply that the function can only utilize one of the GPU's multiprocessors, limiting its performance significantly when compared to

programs without loop dependencies, like the matrix multiplication example.

Listing 4.2 – The forward evolution algorithm for right-toggle rules applied to GPU, given an input grid, number of steps, the border transition rule and the main transition rule.

```
procedure fca_gpu:
    get the thread position in the grid with cuda.grid(1)
    initialize the border as the first and last cells
    initialize grid in shared memory
    synchronize threads
    for s steps:
        read the neighborhood states from the grid
        compute next state with the transition rule
        synchronize threads
        update state in the grid
        shift the border to the left
        synchronize threads
    return grid
```

A second CUDA implementation, described on Listing 4.3 was also developed in order to compute multiple CA cells per thread, making it possible to evolve CA with more than 1024 cells. It differs from the previous program by having an additional grid on shared memory for writing, while the first grid is exclusive for reading during the computation of a step. An additional nested loop is employed to compute n cells at each step instead of one, while another loop at the end of each step is needed to copy the new values into the first grid. Thus, by making each thread update more than one cell, it's possible to evolve grids with more than 1024 cells with this program, unlike the previous program. However, the other limitations in thread hierarchy previously discussed still apply to this program.

Listing 4.3 – The second variation of the forward evolution algorithm for right-toggle rules applied to GPU, given an input grid, number of steps, the border transition rule and the main transition rule.

```
procedure fca_gpu2:
    get the thread position in the grid with cuda.grid(1)
    initialize the border as the first and last cells
    initialize input grid and output grid in shared memory
    synchronize threads
    for s steps:
        for n consecutive cells:
            read the neighborhood states from the grid
            compute next state with the transition rule
```

```
10              update state in the output grid
11          synchronize threads
12          swap grids
13          shift the border to the left
14          synchronize threads
15      return output grid
```

Experiments and the analysis of the results involving each implementation of the forward evolution algorithm are presented on Section 5.1.1.

4.2.2 Backward evolution

As defined by the HCA model, the backward evolution with right-toggle rules of the radius-1 CA can be computed by, on each step, applying the border transition rule to the border cells and computing the new state for each subsequent cell with the main transition rule over the neighborhood defined for the backward evolution.

The Python implementation for the backward evolution algorithm is similar to the aforementioned forward evolution. The border is defined as the first and the last cells, and is shifted after each evolution step. Each step starts by computing the border, and each following cell is computed by applying the main rule over the backward neighborhood. Thus, this algorithm differs from Listing 4.1 by the neighborhood definitions shown on Figure 5 that imply in sequential evolution steps and by shifting the border to the right, considering right-toggle rules.

The high performance serial implementation is achieved by applying the `@njit` decorator to the Python implementation.

The previous parallel implementation approaches cannot be applied for the backward evolution, since the cells must be computed in a sequential manner. However, after each cell is computed on a given evolution step (pre-image), it becomes available for the computation of the next evolution step. Thus, the parallel algorithm was implemented with parallelism over evolution steps, such that multiple evolution steps are processed simultaneously. To achieve this, the program allocates multiple grids corresponding to the maximum number of steps to be computed in parallel, which is equal to the number of cells at most. A batch of n cells, defined by the parameter update rate, is computed across every active evolution step on a `prange` loop. Afterwards, the algorithm keeps track of which new step can be started, while the resources of finished steps can be reused by the new ones by the means of a circular buffer. This process repeats until every step has been computed. It is important to note that the additional complexity to manage this circular buffer implies significantly more processing and memory usage that is not present in the serial implementations nor in any of the aforementioned versions of the FCA and matrix multiplication. The resulting parallel algorithm is shown on Listing 4.4.

Listing 4.4 – The parallel backward evolution algorithm for right-toggle rules, given an input grid, number of steps, the border transition rule, the main transition rule, the number of threads and the update rate n.

```
1 procedure parallel_bca:
2     initialize a grid for each thread
3     flag the first evolution step as active
4     for each active evolution step (in parallel):
5         compute n cells
6         if any thread is idle:
7             flag the next evolution step as active
8         if the current evolution step is finished:
9             flag the current evolution step as inactive
10            free the current thread and grid resources to be reused
11    return last computed grid
```

The CUDA implementation described on Listing 4.5 is also based on parallelism over steps. To this end, each thread manages one grid in shared memory to compute a CA step, and another shared array is used as flags to control which threads are active. The threads are recycled in a similar circular manner: a thread flags the next step as active after computing its first few cells; a thread flags itself as inactive after completing a step, and is activated again by another thread in order to compute a later step. Again, on an iteration, each thread computes n cells given by the update rate parameter. In turn, the update rate is directly associated with the maximum concurrent number of threads, since it determines the rate of activation of threads. Then, the maximum number of concurrent threads is equal to the ratio: grid size $\div$ update rate. In addition to the extra overhead of managing the circular thread recycling, the same limitations to thread hierarchy from the previous section also apply to this implementation, as the threads must be synchronized on every iteration.

Listing 4.5 – The parallel backward evolution algorithm for right-toggle rules applied to GPU, given an input grid, number of steps, the border transition rule, the main transition rule, the number of threads and the update rate n.

```
1 procedure bca_gpu:
2     initialize a grid in shared memory for each thread
3     flag the first evolution step as active
4     for each active evolution step:
5         if the current thread is active:
6             compute n cells
7         if any thread is idle:
8             flag the next evolution step as active
9         if the current evolution step is finished:
10            flag the current evolution step as inactive
```

```
11              free the current thread and grid resources to be reused
12          synchronize threads
13      return last computed grid
```

Experiments and the analysis of the results involving each implementation of the backward evolution algorithm are presented on Section 5.1.2.

4.3 MPGA applied to task scheduling

This section describes the MPGA developed for the task scheduling problem. The first development stage of this MPGA model was performed at (MORAIS, 2017), where multiple variations of genetic operators from the literature were investigated, including methods for selection, crossover and mutation, and the performance of the MPGA was analyzed empirically in comparison to the sequential GA. This investigation allowed for the refinement of the MPGA and the publication of the research paper (MORAIS; OLIVEIRA; CARVALHO, 2019). A copy of this paper is presented in Appendix D. For simplification, this section presents the final MPGA specification that was employed in the present work, followed by the description of the parallel computing approaches applied to the MPGA, and the scheduling graph instances employed for performance evaluation in the experiments with MPGA.

4.3.1 MPGA specification

The genetic representation adopted for the task scheduling problem consists of two arrays S, P, where S is a sequence of tasks that represent a valid topological ordering of the given graph, and P is an indexed vector which maps tasks and processors such that $P[task] = processor$. Figure 10B exemplifies a valid individual for the scheduling of the graph ilustrated by Figure 10A and 4 processors, and the corresponding scheduling is shown on Figure 10C.

From the aforementioned arrays, the corresponding scheduling is evaluated by iterating the S array and allocating it to the processor defined by the P array. The execution of each task starts in the moment that its assigned processor is available and all of its predecessor tasks have been executed, with additional accounting for the corresponding communication costs in the case of distinct processors. At the end of this procedure, the makespan of the scheduling is given by the total time taken to complete the last task, being 540 in the example of Figure 10C. The procedure described is performed by the evaluation function of the MPGA, and the makespan is the fitness value which the algorithm will attempt to minimize.

The population size parameter is divided between populations, e.g., the MPGA with 4 populations and population size 400 results in 4 populations with size 100 each.

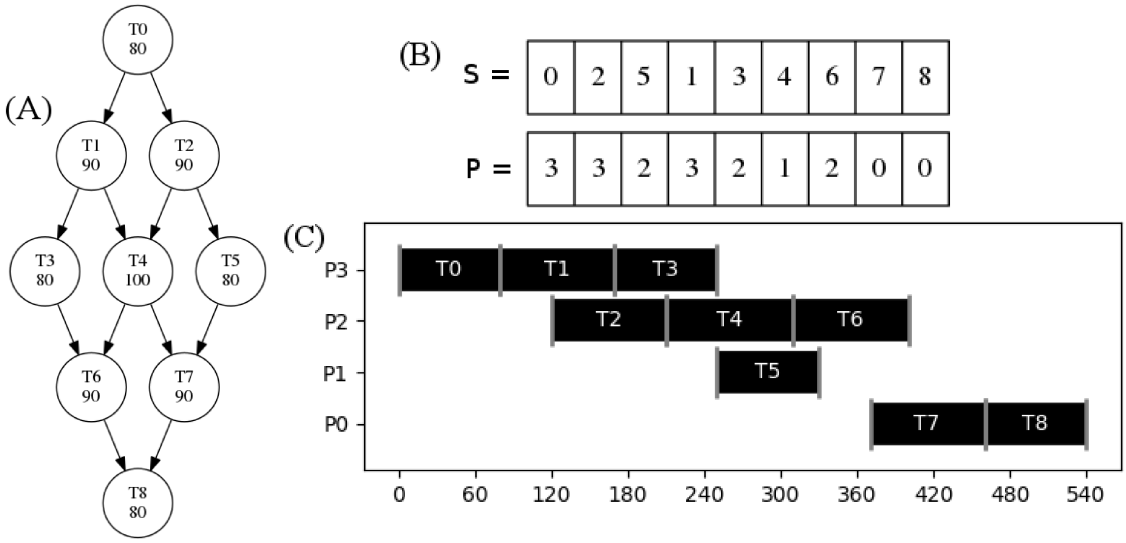

Figure 10 – **(A)**: example graph with computation costs indicated for each task and communication costs set as 40. **(B)**: example of an individual's arrays which represent a scheduling solution for this graph. **(C)**: Gantt diagram visualizing the corresponding scheduling and its makespan of 540 time units. Adapted from Morais, Oliveira e Carvalho (2019).

The initial populations of the MPGA are randomly generated with valid individuals, where the S array is a random topological ordering of the given graph, and the P array is an assignment of each task to a random processor.

The selection of parents for crossover is performed via binary tournament, while reinsertion after each generation is performed with elitism.

For crossover, a single point crossover operator is applied to both arrays of the parents, which is trivial for the P array, but demands adjustments for the S array to be representative of a topological ordering, and thus, a valid solution. The resulting algorithm for the S array, exemplified by Figure 11, consists of copying the first portion of the first parent's array and completing the child array with tasks following the ordering found in the second parent's array, without repetition (OMARA; ARAFA, 2010). The operator results in two children.

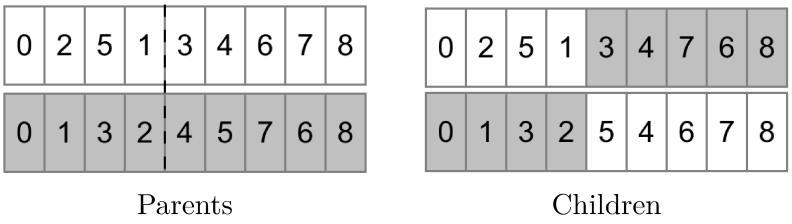

Figure 11 – Example of the single point crossover adapted to the vector S of task ordering. Adapted from Carvalho, Morais e Oliveira (2018).

Mutation is performed by making a small change in one of the arrays. In the case of the S array, two adjacent tasks without inter-dependency are swapped. In the case of the

P array, one the processor assignments is set to a random different processor.

The communication topology developed for the MPGA is the unidirectional ring, which has been shown to achieve high quality solutions (CANTÚ-PAZ, 1998). With this topology, every subpopulation has one single fixed destination for its migrant individuals and also receives migrant individuals from one single fixed subpopulation. For example, with 4 populations, each one is given an unique identifier between 0, 1, 2, 3, where population 0 will receive migrations from population 3 and send migrations to population 1 etc.

The migration frequency parameter defines the interval of generations where migration will occur. For example, migration frequency of 5 determines that migration will occur on generations 5, 10, 15, 20, 25 etc. In the occurrence of migration, the 10% best individuals of each subpopulation migrate to the next subpopulation. Migrations are not performed if the number of populations is set to 1 or if the migration frequency value is equal or greater than the number of generations.

4.3.2 Parallel approaches to MPGA

Using the final specification for the MPGA applied to task scheduling described in the previous section, the implementation of the MPGA was developed in two variations described in this section: the distributed memory MPGA (DM-MPGA) and the shared memory MPGA (SM-MPGA).

In the DM-MPGA, each subpopulation functions as a distinct process of the operational system, having their distinct resources and interacting in the event of migration. This implementation represents the usual application of MPGA as a relatively simple extension of the serial GA via the introduction of the migration operator. In this work, the DM-MPGA was implemented in Python using the Message Passing Interface (MPI) standard via the mpi4py library (DALCIN et al., 2011). The MPI standard defines a set of routines that enable the communication between processes in distributed memory applications, such as routines for sending or receiving instances of data structures. The MPI routines are utilized in the DM-MPGA to create a subprocess for each population, broadcast the parameters to all subprocesses, perform the migration operator, and retrieve the best solution found by each population. With this definition, a main program associated with the main process of the DM-MPGA and a subprogram associated with each population were developed. The main program is detailed as follows.

- During initialization, the main process of the DM-MPGA is given its numerical parameters and the scheduling problem instance, which is composed by the number of processors and a text file that describes the scheduling graph.

- The main process creates a subprocess for each population, as defined by the "number of populations" parameter. This is done via the MPI Spawn routine, which

creates a list of processes from the population program and a communication channel between the main process and its subprocesses.

- ❏ The main process broadcasts the parameters and the problem instance to each subprocess via the MPI Bcast routine.
- ❏ The main process waits for every subprocess to finish execution, then receiving the best solution found by each subprocess via the MPI Gather routine.
- ❏ The MPI execution environment is finished with the Finalize MPI routine, and the best solution is returned.

The subprogram of the DM-MPGA corresponding to each population is an extension of the genetic algorithm shown on Listing 2.1, detailed as follows.

- ❏ The population process retrieves the communication channel with the main MPGA process via the Get_parent MPI routine.
- ❏ The population process receives the parameters and the problem instance from the main process via the MPI Bcast routine.
- ❏ The population process retrieves its identifier considering the spawned processes using the MPI Get_rank routine.
- ❏ The population process reads the file containing the scheduling graph description and creates an adjacency list representation.
- ❏ The process initializes its population.
- ❏ For g generations, the process performs the operators for selection, crossover, mutation, reinsertion. Then, if migration is set to occur in a generation, each population sends individuals via the MPI Send routine and receives individuals via the MPI Recv routine.
- ❏ After g generations, the process sends the best solution from its population to the main MPGA process via the MPI Gather routine.

A second version of the MPGA was also implemented in order to exploit the optimization provided by the Numba library, which also enables the investigation of the performance of the MPGA in a different execution environment that maintains the resources of the algorithm in shared memory instead of distributed memory. The resulting shared memory MPGA (SM-MPGA) required significantly more adjustments in its code compared to the DM-MPGA. This program required a modification to the main loop of the algorithm in order to evolve multiple populations instead of one, taking one additional loop to iterate through each population. This loop over populations is then executed in

parallel with the application of Numba's `prange` operator. The use of the Numba library also required the application of static data structures, resulting in further increase of complexity in this implementation. As such, the DM-MPGA implementation consists of a single program where multiple threads evolve each population in parallel. This program is detailed as follows.

- ❏ The SM-MPGA process is given its numerical parameters and the scheduling problem instance, which is composed by the number of processors and a text file that describes the scheduling graph.
- ❏ The process reads the file containing the scheduling graph description and creates an adjacency list representation.
- ❏ An array of populations is instantiated, containing each population according to the "number of populations" parameter. Likewise, multiple arrays are instantiated to serve as buffers for crossover and migration for each population.
- ❏ Each population is initialized in a parallel loop using the Numba prange operator.
- ❏ The number of generations is subdivided into "runs" according to the migration frequency parameter. For example, for 100 generations and migration frequency being 5, each population is evolved in parallel in runs of 5 generations and migration is performed between each run.
- ❏ For each run, each population is processed in parallel using the Numba prange operator, performing the operators for selection, crossover, mutation, reinsertion. Migration is performed between each run.
- ❏ For migration, each migration buffer array is filled with the individuals that each population is sending to the next. Then, each population receives migrating individuals by copying them from each corresponding array.
- ❏ After g generations, the best solution accross every population is returned.

GA execution on GPU is out of the scope of this work, as it requires the investigation of different models that are appropriate for massive parallel processing, such as the fine-grained GA (CANTÚ-PAZ, 1998).

Experiments and the analysis of the results involving each implementation of the MPGA applied to task scheduling are presented on Section 5.2.

4.3.3 Task scheduling instances

The set of scheduling graphs used as benchmark is a subset from the work (CARVALHO; MORAIS; OLIVEIRA, 2018), which demonstrated that some of the scheduling

graphs with CCR=1 were inadequate for benchmark purposes, as they can be scheduled efficiently with a randomized algorithm. Thus, the benchmark set adopted in this work features the 8 graphs Laplace graphs (lap) and Fourier transform graphs (fft) with CCR=0.25 from that work, i.e., graphs where the communication costs are 4 times higher than the computation costs.

The lap graphs represent tasks associated with the Laplace equation solver algorithm, with a structure given by the input square matrix of the algorithm. The fft graphs are based on the execution of the Cooley-Tukey Fast Fourier Transform algorithm, which assumes an input with size of a power of 2, and represents a sequence of recursion steps doubling the number of tasks successively, followed by a series of "butterfly" steps that describes the patterns of communication of the algorithm. Figure 12 exemplifies both of the graph structures. The benchmark set includes the graphs lap36, lap64, lap100, lap144, fft39, fft95, fft223, fft511, where each associated number denotes the number of tasks.

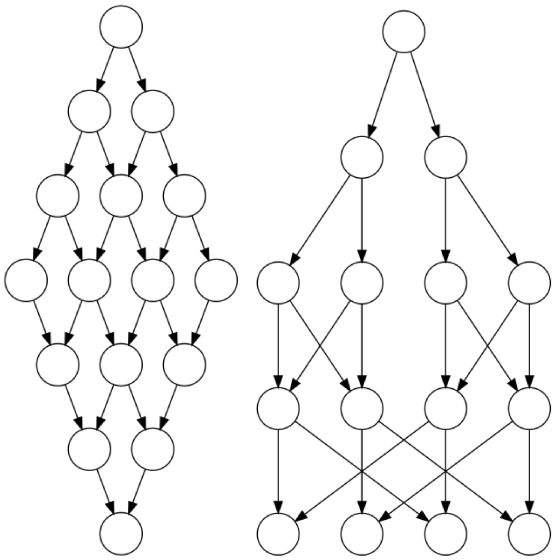

Figure 12 – Graphs lap16 and fft15 exemplify the structure of the lap and fft graphs. Adapted from Carvalho, Morais e Oliveira (2018).

Chapter 5

Experiments and Results

This chapter presents in detail the experiments that were conducted and the results observed. All experiments were performed on a Ubuntu 20.04 laptop with CPU Core i5-7300HQ @ 2.50GHz x 4 and GPU GeForce GTX 1050. The following sections discuss the experiments with the HCA algorithms and the experiments with the MPGA implementations.

5.1 Experiments with the HCA model

This section describes the experiments with the forward and backward evolution algorithms defined by the HCA model (MACÊDO, 2007). The following experiments employed varying block size and a 1D CA model with radius-1 neighborhood and right-toggle rules which defines a cryptographic model with 8-bit cryptographic key, given by the size of the main transition rule for radius-1 neighborhood. The HCA cryptographic system that recently received a patent (OLIVEIRA; MACEDO, 2019) employs toggle-rules with radius 4, defining 256-bit cryptographic key. Despite this difference on key size specification, HCA can be applied with any neighborhood size, and the present work is focused in the evaluation of the effects of the parallel execution of HCA, which is simpler to implement using radius-1 rules. All of the analyses performed with the radius-1 rule space can be extrapoletade for the HCA standard specification (OLIVEIRA; MACEDO, 2019). Moreover, a variation of the HCA model proposition is under investigation in an on-going PhD thesis, the VHCA model (Very Heterogeneous Cellular Automata), which employs 1D radius-1 CA rules (LIRA, 2020).

The following right-toggle transition rules are used in the experiments.

- Main rule (rule 101): $111 \mapsto 0, 110 \mapsto 1, 101 \mapsto 1, 100 \mapsto 0, 011 \mapsto 0, 010 \mapsto 1, 001 \mapsto 0, 000 \mapsto 1$
- Border rule (rule 170): $111 \mapsto 1, 110 \mapsto 0, 101 \mapsto 1, 100 \mapsto 0, 011 \mapsto 1, 010 \mapsto 0, 001 \mapsto 1, 000 \mapsto 0$

5.1.1 Forward evolution

Initial experiments were conducted by evolving 1024-cell lattices over 10^6 evolution steps, which is associated to the number of lattice configurations required to perform the decryption of a single block of the ciphertext. In fact, in the HCA specification (OLIVEIRA; MACEDO, 2019) only 1024 evolution steps are required to decipher a single 1024-bit block of ciphertext. However, this work employs a more robust number of evolution steps in the experiments in order to avoid oscillations in the execution time that could influence the comparative analysis regarding the deciphering of a single block.

The resulting execution times are found on Figure 13. The serial Numba implementation, which ran for 1.394 seconds, achieved a 952X speedup over the Python one (1327 seconds). For the parallel implementation on CPU, the overhead associated with thread and memory management was significantly greater than the performance gain of the multiple working CPU cores, resulting in an execution time of 3.624, which was 2.6 times higher than the serial Numba. The CUDA implementation, on the other hand, was able to achieve 3.3X speedup at 0.425 seconds despite being limited to one CUDA block, as discussed in the previous chapter.

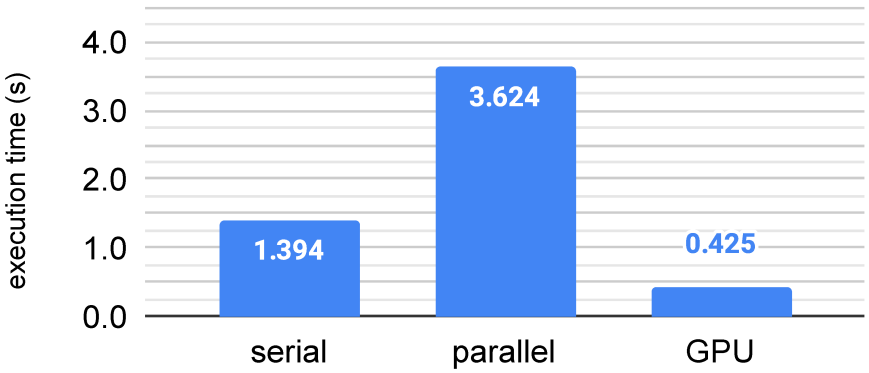

Figure 13 – Execution time in seconds of the initial forward evolution experiments (1024 cells and 10^6 steps) with the serial, parallel on CPU and parallel on GPU programs. The pure Python implementation was omitted, being 1327s.

Another experiment was conducted to measure the performance of the implementations with varying lattice sizes from 128 to 4096, which represents the block size used in the cryptographic model. The number of time steps, which is associated to the number of lattice configurations required to decipher a single block of the ciphertext, was fixed again at 10^6 steps for all the lattice sizes. As previously explained, this robust number of evolution steps was chosen to avoid oscillations in our analysis. Figure 14 shows the execution time obtained with the serial CPU program optimized using Numba (serial) and the two variations of parallel GPU implementations presented in the previous chapter (GPU1 and GPU2), while the pure Python and the CPU parallel implementation were omitted in this experiment due to their inferior performance that was observed in the previous

experiment. For the serial Numba implementation, execution time approximately doubles as the grid size doubles. For both CUDA implementations, execution time increases very slightly as the grid size doubles up to 512 cells, and increases significantly more at 1024 cells. This is presumably due to under-utilisation of the GPU resources when processing small amounts of data. The use of GPU, then, starts to be worthwhile at 256 cells for the first CUDA program, and at 1024 cells for the second. It is also notable the significant difference in execution time between the two CUDA programs, due to the overhead of the synchronization and memory access associated with the additional grid of the second program. Note that both programs computed one cell per thread in the experiments up to 1024 cells. With more than 1024 cells, the first CUDA program could not run because of the thread limit, as previously discussed. The second CUDA program was able to evolve the 2048 and 4096 cells by computing 2 and 4 cells per thread, respectively. From 1024 to 2048 cells, its execution time still had a relatively small 50% increase while its input doubled, indicating that the GPU resources were still under-utilised by this point. From 2048 to 4096 cells, its execution time had a much larger 260% increase, indicating the overhead that this large amount of data exerts over the single CUDA block.

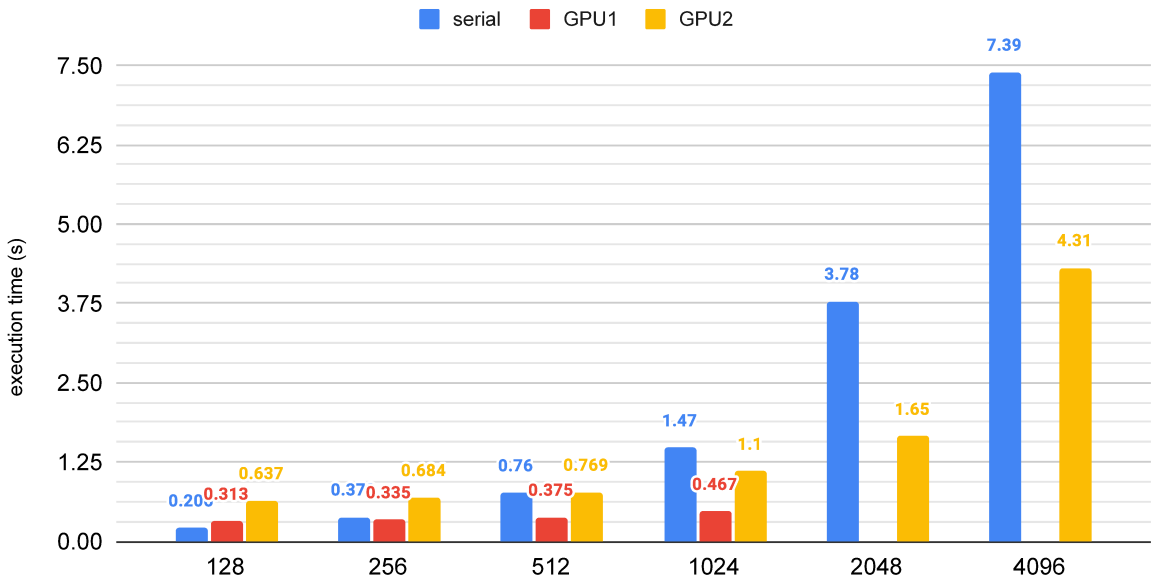

Figure 14 – Execution time in seconds of the second forward evolution experiments with the Numba serial CPU program and two variations of parallel GPU programs. Grid sizes vary from 128 to 4096 cells, and steps are fixed at 10^6.

Besides execution time, the throughput is an useful metric for the evaluation of these results by demonstrating the number of operations per second performed by each implementation by accounting for the grid size. For example, the evolution of 512 cells with the serial variation had 0.76 seconds of execution time, implying in a throughput of 637.7 bits per second, while the GPU1 variation had 1365.3 bits per second of throughput (512

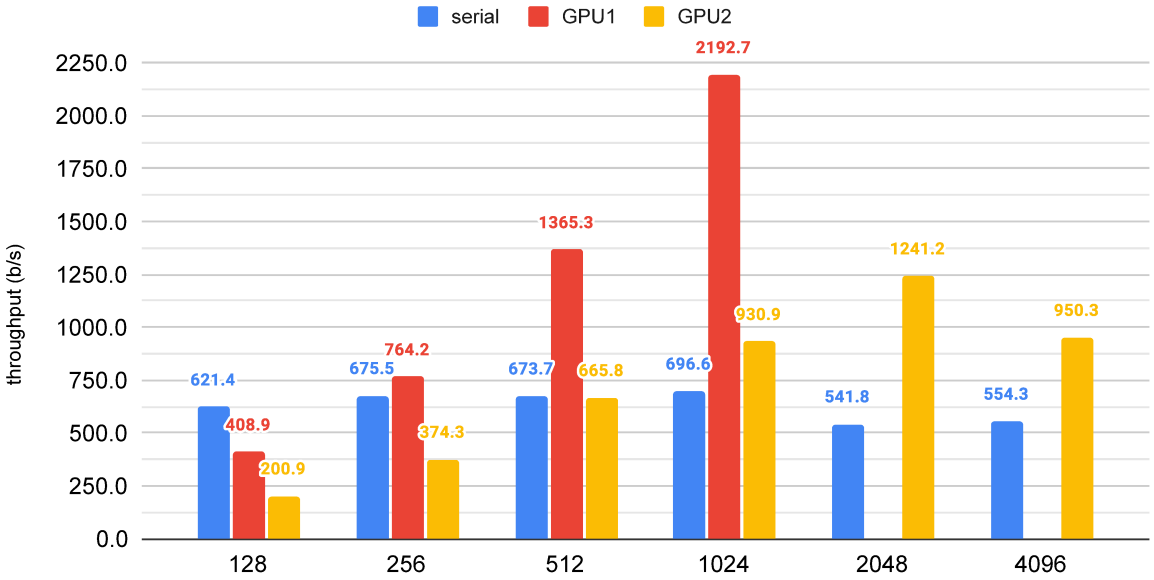

Figure 15 – Throughput in bits per second of the second forward evolution experiments with the Numba serial CPU program and two variations of parallel GPU programs. Grid sizes vary from 128 to 4096 cells, and steps are fixed at 10^6.

cells over 0.375 seconds). Figure 15 shows the corresponding throughput for each grid size and implied execution time from the previous results.

Since the execution time of the serial variation increases proportionally to the grid size, as previously discussed, its throughput has little variation between grid sizes 128 to 1024, while decreasing more significantly with 2048 cells. In contrast, the throughput of the first GPU variation increases with grid size up to 1024 cells, which is the block size limit to be used with this variation, since it was possible to observe in Figure 14 that the execution time increased insignificantly with the doubling of grid size. A similar behavior was observed for the second GPU variation. However in this case, the throughput increases with grid size up to 2048 cells. The highest throughput values were achieved by the GPU1 variation with 1024 cells (2192.7 b/s) and the GPU2 variation with 2048 cells (1241.2 b/s).

These results suggest that, for applications like cryptography, the most efficient implementation of forward evolution for the parallel devices considered in this work is the application of GPU and a grid size 1024 cells, which is notably equal to the number of available GPU threads in this implementations. Thus, in this scenario, the performance of encryption with the HCA model is benefited the most with the application of 1024-bit blocks. Considering 128-bit block size, which is more commonly adopted in modern cryptographic systems such as the AES, the results indicate higher throughput for the serial implementation. However, the GPU implementation might achieve higher throughput with 128-bit blocks in future experiments with parallel multi-block encryption, since the GPU resources are notably under-utilized for small block sizes in the current experiments.

5.1.2 Backward evolution

Initial experiments with backward evolution were conducted by evolving 1024-cell lattices over 10^6 evolution steps, which is associated to the number of pre-images required to perform the encryption of a single block of plaintext. The values chosen for lattice size and evolution steps are due to the same reasons explained in the forward experiments of the previous section.

Results from Figure 16, including the omitted Python execution time of 1490 seconds, show that the transition from Python to Numba (8 seconds) implied in a 186X speedup. Similarly to the previous section, the CPU parallel implementation had high overhead, resulting in 36.7 seconds of execution time, being 4.7 times higher compared to the serial program. In the same way, the GPU implementation ran for 18.1 seconds, resulting in a 2.26 times increase in execution time. In both cases, the overhead was accentuated by the management of the circular buffer.

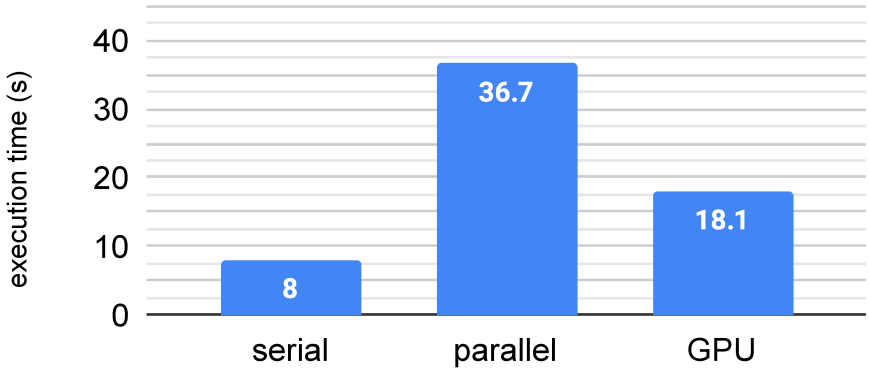

Figure 16 – Execution time in seconds of the initial CA backward evolution experiments (1024 cells over 10^6 steps) with the serial, parallel on CPU and parallel on GPU programs. The pure Python implementation was omitted, being 1490ms.

More experiments were conducted to measure performance with varying input sizes, i.e., the number of cells in the lattices representing the HCA block size. Due to memory restrictions of the CUDA program that are detailed in the following, the lattice size was set from 64 to 512 cells, which are lower values than the 128 to 1024 interval for the experiments with forward evolution, and number of evolution steps steps was fixed at 10^6. Figure 17 presents the results for the serial Numba program, showing an approximate 2 times increase as the input increases, as expected. Table 1 presents the results for the CUDA program with multiple values for its update rate parameter, which controls the maximum number of threads employed and the memory usage. Missing values denote configurations that were unable to run due to memory limitations of the GPU used in the experiments, which has available shared memory of 49152 bytes. The memory requirements for these experiments are detailed in Table 2. It is noticeable that lower update

rates had significantly higher performance, as they were able to use the most threads possible. However, using more threads in this program also implies using more shared memory to allocate more grids, reaching the GPU memory limitations in some cases. It is also noticeable that the execution times had either small increases or decreases as the input size increased, which again imply under-utilisation of the GPU resources. However, its performance only surpassed the serial program in the case of 256 cells and update rate = 2, with 1.34X Speedup. Therefore, while most results of CUDA program were lacking in relative performance, the results suggest that the CUDA program would be superior for larger amounts of data if its memory issues were resolved. That is, using a GPU board configuration with more available memory will enable the execution of the experiments with the missing parameter values, and higher speedups would be achieved, surpassing the serial Numba program for the input sizes starting from 256. As such, to application of backward evolution to cryptography is benefited by sequential implementations concerning encryption of a single block on such limited hardware. However, future experiments with more robust hardware or parallel multi-block encryption might result in higher performance of the GPU implementation.

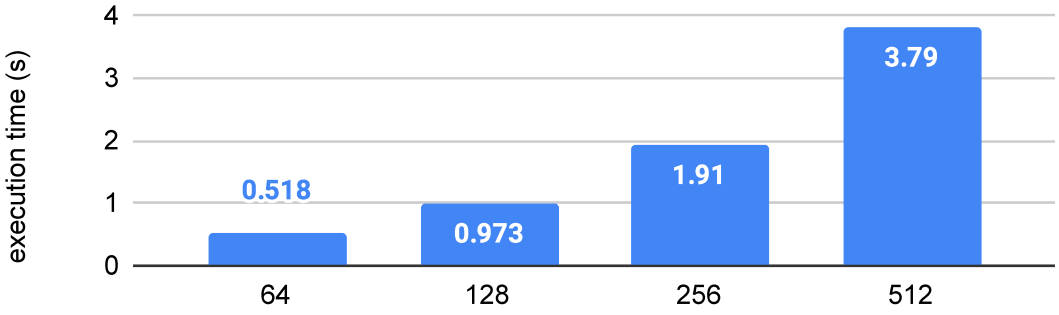

Figure 17 – Execution time in seconds of the Numba serial backward evolution experiments, with number of cells varying from 64 to 512 and steps fixed as 10^6.

Table 1 – Execution time in seconds of the CUDA backward evolution experiments, with update rate varying from 1 to 8 and grid size from 64 to 512 and 10^6 steps.

update rate	64 cells	128 cells	256 cells	512 cells
1	1.38	1.4	-	-
2	1.83	1.58	1.42	-
4	2.87	3.07	2.43	-
8	4.84	5.18	5.31	3.84

5.2 Experiments with the MPGA model

This section describes the experiments with the MPGA and their results by analyzing the performance of the algorithm while varying the parameters that indicate the number

Table 2 – Shared memory consumption in bytes by the grids of each configuration of the BCA CUDA program, varying the update rate and grid size. This program allocates a grid for each thread to compute a step of the evolution as a circular buffer, plus one extra grid to prevent premature overwriting. The total number of threads employed is given by size ÷ update rate, while each grid is an array of bytes. This results, for example in 256 cells times 129 grids for update rate = 2, a consumption of 33024 bytes. Numbers in red denote shared memory requirements higher than the GPU capacity (49152 bytes).

update rate	64 cells	128 cells	256 cells	512 cells
1	4160	16512	65792	262656
2	2112	8320	33024	132608
4	1088	4224	16640	66048
8	576	2176	8448	33280

of populations and the migration frequency.

For the following experiments, the other parameters of the algorithm were fixed as follows, as determined empirically in the preliminary stage of this work as presented in (MORAIS; OLIVEIRA; CARVALHO, 2019). With this parameterization, the algorithm performs 52200 fitness evaluations.

- ❏ Population size: 400.
- ❏ Number of generations: 200.
- ❏ Crossover rate: 65%.
- ❏ Mutation rate: 35%.
- ❏ Migration rate: 10%.

The number of populations was varied in the experiments between the values: $1, 2, 4, 8, 16$. Notably, the MPGA with 1 population works exactly like the regular serial GA and the migration procedure is not executed in this case. For each of these population quantity parameters, the population size is divided between each subpopulation, resulting in subpopulations with sizes $400, 200, 100, 50, 25$, respectively.

The selected values for migration frequency in the experiments were: $1, 5, 10, 20, 50, 100, 200$. That is, in each case, migration is performed $200, 40, 20, 10, 1, 0$ times, respectively. Thus, the first case results in migration after every single generation, while the latter case is equivalent to multiple independent executions of a GA with reduced population size.

The benchmark set for the experiments consists of the eight graphs mentioned in the previous chapter: fft39, fft95, fft223, fft511, lap36, lap64, lap100, lap144. The number of processors for scheduling was fixed as 4.

5.2.1 Execution time

The initial experiment with MPGA provides an assessment about the influence of the aforementioned parameters over the execution time of the algorithm considering a large input, being the fft223 graph. Tables 3 and 4 respectively show the mean execution times of 10 executions for the distributed memory implementation (DM-MPGA) and the optimized shared memory implementation (SM-MPGA) applying every pair of parameters. In the case of 1 population, its result is presented only once, as it does not perform migration. Comparing both tables, it is noticeable that the optimization provided by the Numba library on the SM-MPGA achieved much lower execution times than the DM-MPGA, with the results for the SM-MPGA varying between 0.141 and 0.531 seconds and the DM-MPGA times varying between 11.7 and 32.8 seconds.

The DM-MPGA gained performance with the increase of the population quantity up to 8 populations, where it presented the shortest execution time in six of the seven cases. The increase of migration frequency also generally implied in the decrease of execution time, especially between the values of 1 and 20. At higher values, the performance gain was not as significant, and the total number of migrations performed has little change. Some variance observed in these values can be attributed to the synchronization between the population processes that is needed to perform migration. In these results, the configurations with 16 pops/200 migf and 8 pops/20 migf presented the shortest execution time, at 11.7 seconds, a 2.8X speedup from the serial case.

The results for the SM-MPGA had similar patterns to the DM-MPGA, with the 8-population configuration being the fastest in four of the seven cases and generally gaining performance with higher migration frequency. The shortest execution time was 0.141 seconds with 8 pops/200 migf, a 3.76X speedup from the serial case.

Table 3 – Mean execution time in seconds of the DM-MPGA applied to the fft223 graph, with columns representing population quantity and rows representing migration frequency.

migf / pops	**1**	**2**	**4**	**8**	**16**
1	32.8	17.7	18.9	18.6	39.2
5		17	15.1	15	17.2
10		17.9	13.9	12.3	13.2
20		16.8	14.1	11.7	12
50		17.9	14.1	12.6	13
100		17.5	13.8	12.6	12.6
200		17.7	14	12.5	11.7

Since these parameters also affect the search process of the MPGA besides the execution time, the next experiment is meant to assess their influence over the quality of the evolved solutions in terms of makespan.

Table 4 – Mean execution time in seconds of the SM-MPGA applied to the fft223 graph, with columns representing population quantity and rows representing migration frequency.

migf / pops	1	2	4	8	16
1	0.531	0.297	0.241	0.183	0.195
5		0.26	0.161	0.177	0.144
10		0.232	0.177	0.142	0.165
20		0.255	0.167	0.16	0.15
50		0.243	0.165	0.146	0.149
100		0.242	0.165	0.152	0.147
200		0.236	0.157	0.141	0.147

5.2.2 Makespan

In order to evaluate the influence of the MPGA parameters over the fitness (makespan) of the solutions, the following experiment was conducted.

The SM-MPGA was executed 100 times for each of the eight benchmark graphs and each parameter combination to provide statistically significant results, due to the MPGA nondeterministic processes. For each configuration of parameters, its performance was determined by the metrics of the mean makespan and minimum makespan obtained across the 100 executions. These results can be found in the Appendix B on Table 8.

In contrast to the execution times obtained in the previous experiment, the minimization of makespan was generally improved in configurations with small values of migration frequency. Considering the 10 best configurations regarding mean makespan for each graph, the most frequent value for migration frequency was 1 in every case, which denotes that migration is set to occur in every generation.

Similarly, small values for population quantities were the most featured in the 10 best results for each graph, where the serial 1-population appeared in all cases, and had the 2-population configurations as the most frequent.

This dominance of small values for both parameters was more pronounced in the larger graphs, while in contrast, the 10 best configurations for the lap36 graph featured 6 total occurrences of 8 and 16 population quantities and 4 total occurrences of migration frequencies of 10 or higher. This suggests that the search for larger graphs benefited more from the slower convergence that occurs in configurations with fewer/larger populations and frequent migrations, also meaning that the population size of 400 was possibly too small when subdividing it to the subpopulations, leading to premature convergence in the case of numerous/small populations.

The contrast found in the comparison of the fitness results and the results regarding execution time highlights a clear trade-off between these two performance metrics.

Table 5 shows the bounding values of the results from Table 8, i.e., the best and worst values found for each metric. Notably, the most pronounced differences between best

case and worst case were 4464.6 / 5689.2 (27%) and 8940 / 11520 (28%) for mean and minimum makespan, which correspond to graphs fft223 and fft511, respectively. When compared to the speedup achieved in the previous experiment (2.8X and 3.76X), these bounding values indicate that the potential gains in speedup when employing parameters of larger values can overcome the losses in the resulting makespan of the solutions.

Table 5 – Bounding values for mean makespan and minimum makespan found in the makespan experiments.

graph	**mean makespan**		**minimum makespan**	
	lower bound	upper bound	lower bound	upper bound
fft39	1344.6	1468.2	1080	1320
fft95	2371.2	2744.4	2160	2580
fft223	4464.6	5689.2	4080	5220
fft511	9633	12181.2	8940	11520
lap36	2854.6	3125.7	2320	2840
lap64	4646.8	5167.1	3920	4800
lap100	6686.8	7577.1	5890	6990
lap144	8999.4	10542.3	8110	10050

Thus, the next experiment is aimed at visualizing the trade-off between the execution time and makespan and reveal the most efficient parameters when considering both performance metrics.

5.2.3 Multiobjective analysis of performance

The following experiment is aimed at the investigation of the most efficient parameters for the MPGA in the presence of the trade-off that was verified in the previous experiment. The experiment is detailed as follows.

Besides the parameters for population quantity and migration frequency, new results were obtained for the population sizes 1000 and 2000, in addition to the value of 400, in the interest of reducing the effect of premature convergence. The benchmark set for this experiment was reduced to the four lap graphs. A multiobjective analysis is employed in order to elucidate the efficient configurations when considering execution time and makespan (mean and minimum) as objectives. Efficient (non-dominated) configurations are the ones that are not outperformed in both metrics by any other configuration. Non-dominated configurations compose the Pareto frontier.

Each configuration of parameters was given an alphanumeric label, shown on Table 6, in order to facilitate the visualization of the following charts.

The first step of the analysis consisted of organizing the results for each graph in a multiobjective perspective, considering both execution time and makespan metrics. Figure 18 exemplifies a multiobjective chart, containing the two scatter plots representing mean makespan and minimum makespan versus execution time, obtained by applying the

Table 6 – Labels used to denote parameter configurations in the results of the multiobjective analysis.

label	pop qty	migf
1	1	-
2	2	1
3	2	5
4	2	10
5	2	20
6	2	50
7	2	100
8	2	200
9	4	1
A	4	5
B	4	10
C	4	20
D	4	50
E	4	100
F	4	200

label	pop qty	migf
G	8	1
H	8	5
I	8	10
J	8	20
K	8	50
L	8	100
M	8	200
N	16	1
O	16	5
P	16	10
Q	16	20
R	16	50
S	16	100
T	16	200

DM-MPGA with population size 1000 to the lap144 graph. This chart condenses both mean and minimum makespan plots using blue circles and red triangles to differentiate them. The Pareto frontier found for each plot is also emphasized using different colors: yellow circles and green triangles for the mean and minimum makespan, respectively. Similar multiobjective charts were generated for each of the four investigated graphs (lap36, lap64, lap100, lap144) for each value of population size (400, 1000, 2000) and for both of the implementations of the MPGA. Therefore, 24 multiobjective charts similar to the one in Figure 18 were generated, each one presenting two scatter plots representing mean and minimum makespan versus execution time. These initial charts were omitted in this dissertation for the sake of brevity and simplicity. The x axis in these multiobjective charts, which corresponds to execution time, is displayed in logarithmic scale to facilitate the visualization of the data points. From this data, the following steps aim to summarize it and derive highly efficient configurations.

The second step of the analysis consisted of the selection of highly efficient points from the Pareto frontiers of the previous step, considering the population sizes evaluated in our experiments: 400, 1000 and 2000 individuals. These selected data points are then aggregated into a new chart. For example, Figure 19 represents the data that was selected and aggregated from the resulting Pareto frontiers for the three evaluated population sizes, obtained from the application of the DM-MPGA to the lap144 graph for both mean and minimum makespan plots. These non-dominated points were extracted from the charts in Figure 18, having population 1000, and from two other similar charts generated for population 400 and 2000 for lap144 graph. A similar analysis was performed for each one of the graphs and for both of the algorithm implementations, with results presented

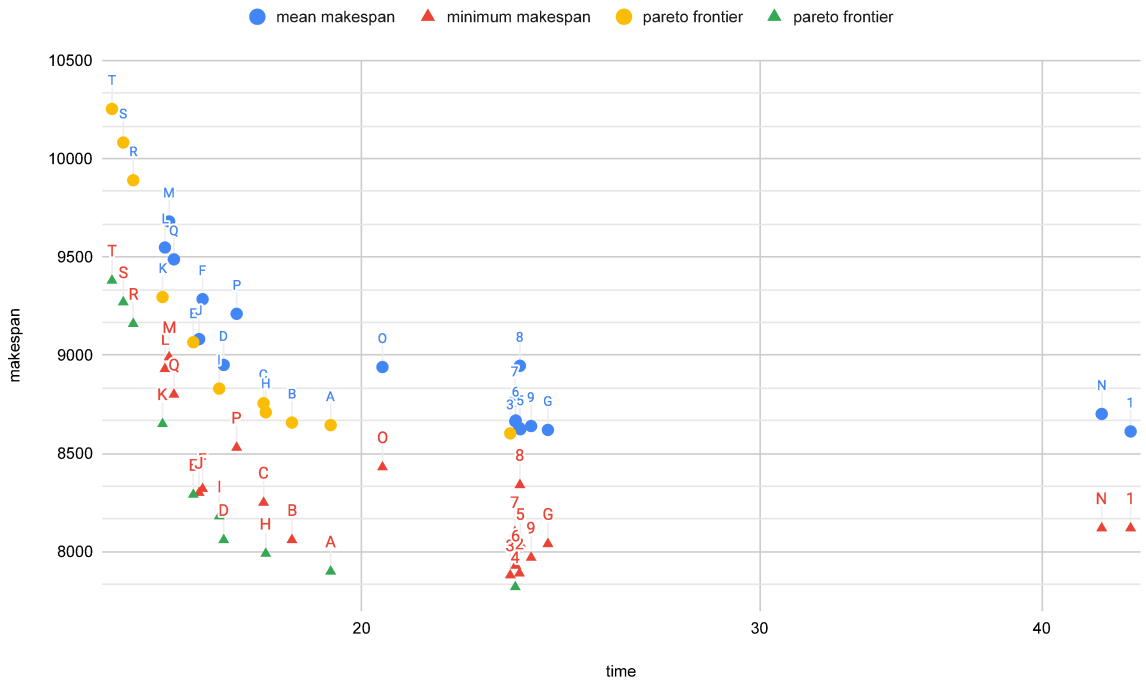

Figure 18 – Multiobjective chart of mean makespan and minimum makespan versus execution time for the graph lap144 and the DM-MPGA with population size 1000.

on Appendix B, where Figures 21-24 present the results for the DM-MPGA and Figures 25-28 present the results for the SM-MPGA.

The selection criterion for the data points in step 2 consists of the two extreme points of the Pareto frontier, being the most efficient at their respective metrics, as well as intermediate points with significantly beneficial trade-offs. For example, considering the points T and E in yellow from Figure 19, it becomes clear that the configuration E provides a significantly high improvement in minimizing the mean makespan metric when compared to the configuration T, while presenting a relatively small increase in execution time.

The third step of the analysis consisted of further summarizing by selecting non-dominated data points from the charts from the previous step, independently of the population size used to obtain each point. It results in new simplified charts containing the aggregated highly efficient configurations for each graph and MPGA implementation. For example, Figure 20 shows the resulting Pareto frontiers (mean and minimum makespan) corresponding to the non-dominated data points from the data shown in Figure 19, representing the results for the DM-MPGA applied to the lap144 graph. This analysis was performed for each one of the graphs and for both of the algorithm implementations, with results presented on Appendix B, where Figures 29-32 present the results for the DM-MPGA and Figures 33-36 present the results for the SM-MPGA.

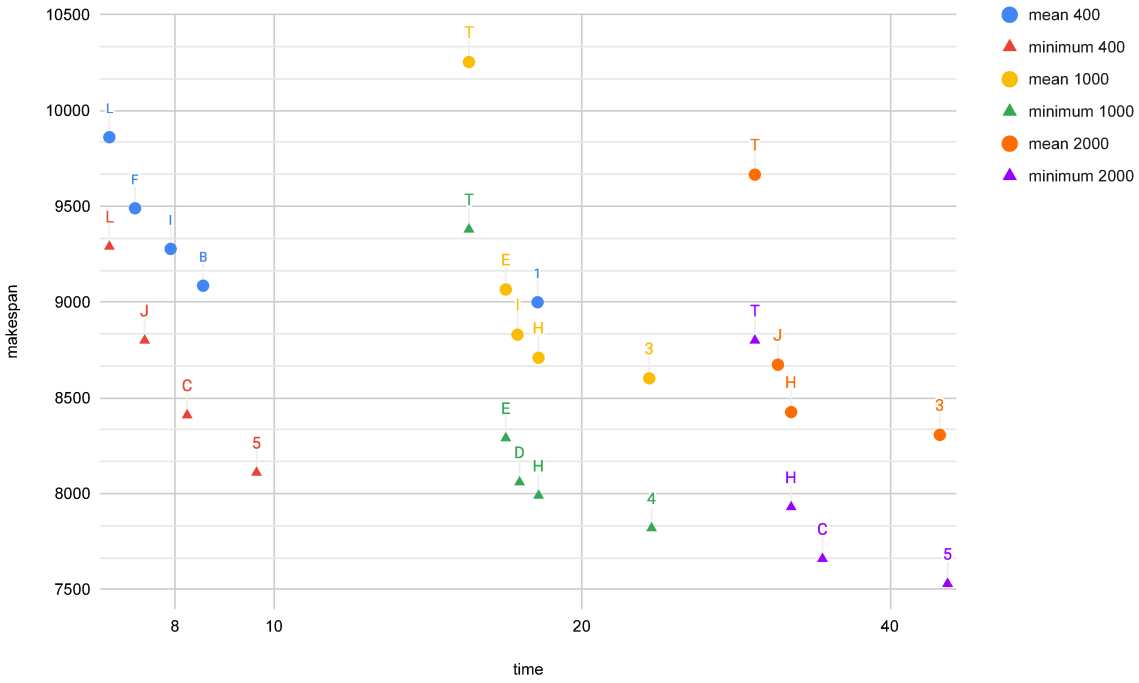

Figure 19 – Highly efficient configurations selected from the Pareto frontiers of the results for the graph lap144 and the DM-MPGA with population sizes 400, 1000, 2000.

The remaining data points from step 3 thus represent highly efficient configurations considering the larger context of varying population sizes. By using these resultant data processed from a multiobjective view for each input program graph, it is then possible to determine the most efficient configurations for each implementation of the MPGA as follows.

Table 7 shows the five most occurring configurations resulting from the third step of the analysis, considering each of the data points from both mean makespan and minimum makespan shown on Figures 29-36.

In the case of the DM-MPGA, three of its configurations with the highest efficiency involve 4 populations: D, C, F. Notably, this number is equal to the number of processor cores in the machine where the experiments were conducted, suggesting that the added overhead of employing more populations and processes than processor cores generally did not result in favorable trade-offs in the performance metrics. Regarding migration frequency, the three best configurations employ the values 50, 20, 10, demonstrating that the efficiency of the distributed-memory application of MPGA is dependent on a relatively infrequent communication between processes.

For the SM-MPGA, the efficient trade-offs were found more often with 8 populations on configurations I and M, employing a number of threads that is double the number of processor cores, demonstrating their lower overhead when compared to the processes

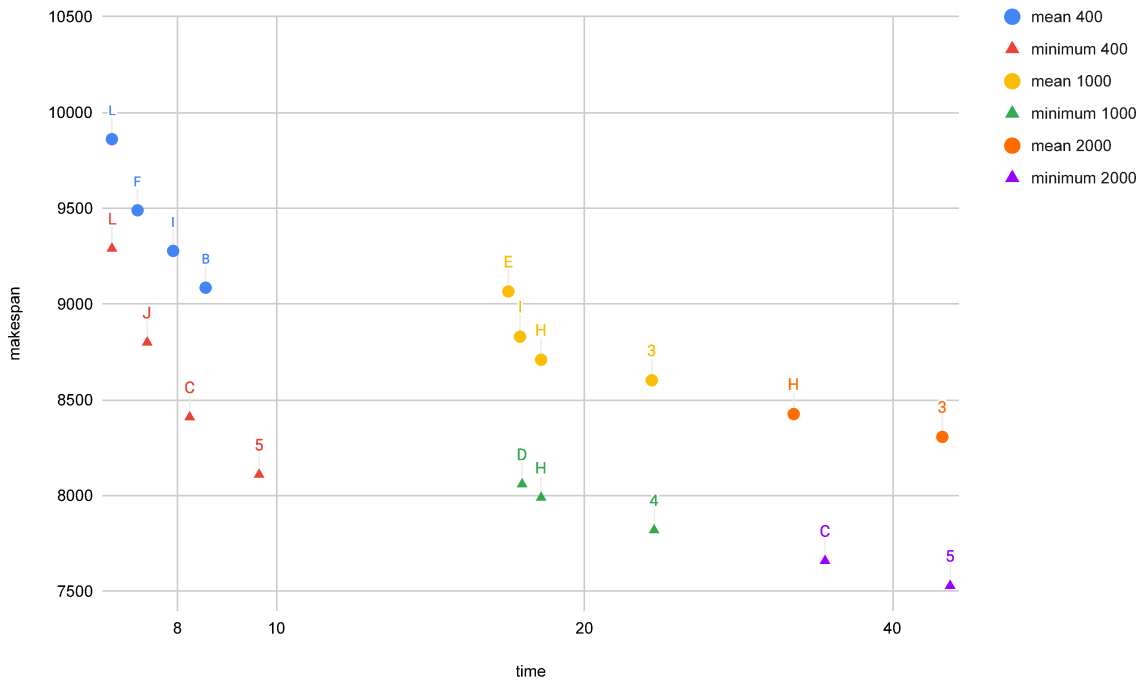

Figure 20 – Non-dominated data points selected from the highly efficient configurations of the previous step of analysis for the graph lap144 and the DM-MPGA with population sizes 400, 1000, 2000.

utilized in the case of distributed memory. Similarly, the smaller overhead associated with migration in shared memory resulted in the top 3 configurations employing migration frequency of 10, 5, 20.

In the general case for both implementations, configurations I and C hold the largest amount of combined occurrences. That is, for a general conclusion of the experiments presented in this section involving the scheduling graphs lap36, lap64, lap100, lap144, the configurations that have shown a good trade-off between makespan and execution most frequently were: 8 populations with migration frequency of 10 generations and 4 populations with migration frequency of 20 generations.

Table 7 – Most occurring configurations in the final analysis.

DM-MPGA		**SM-MPGA**	
config	occurrences	config	occurrences
D	11	I	12
C	8	O	6
I	8	C	5
3	5	M	5
F	4	B	4

Chapter 6

Conclusion

In this work, the development of two high performance bio-inspired models, the HCA cryptographic model and the MPGA applied to task scheduling, was investigated with the use of the high level Python language.

In the case of the HCA, the application of serial and parallel computing was achieved with Python and Numba targeting both CPU and GPU architectures. This choice of programming language was proven useful, as it allowed the efficient implementation of the algorithms while bypassing the significant technical barrier commonly associated with high-performance computing, especially GPU programming, as demonstrated by the concise resulting code. Firstly, Numba has shown to be an efficient solution to provide high-performance for Python applications executed over the CPU architecture both for single core and multi-core approaches. Moreover, the relative ease of use of the language then makes Python+Numba a valuable and accessible point of entry for GPU programming, which could then be preferred over more specialized options like FPGA programming, as the latter are more inaccessible in terms of hardware availability and difficulty of programming.

With the corresponding experiments, new empirical results about the parallel execution of the HCA algorithms were obtained. For both CA evolution methods, the results demonstrated high speedup values for the optimized serial code with Numba in comparison to the Python-interpreted implementations, while the CPU-parallel versions were lacking in comparison to the serial optimized due to overhead, but still outperformed the interpreted programs. The forward evolution algorithm benefited significantly from the GPU implementation at higher input sizes, despite its limitations in parallelism by reason of synchronization over steps. The backward evolution algorithm also performed better with higher input sizes, but its increased overhead needed for memory management made it slower than the CPU optimized code in most test cases. However, both algorithms generally demonstrated relative small performance drops with the increase of input size compared to the CPU program. This indicates further potential for speedup when applying these algorithms to parallel architectures with more computer resources. As such,

the research hypothesis established on section 1.3 was only met for the forward evolution algorithm on GPU, which provided significant speedup, according to the experiments conducted and the limitations of the hardware.

Future work concerning cellular automata include investigating the parallel evolution of CA with different properties, such as lattices of higher dimensionality, alternative neighborhood definitions, and asynchronous evolution steps. Also, applications that require the evolution of multiple independent CA could be able to bypass the limitation to a single CUDA block that was observed in this work, and possibly obtain higher performance with GPU, such as the parallel encryption of multiple blocks using the HCA model. A preliminary experiment with parallel processing of multiple CA is presented on Appendix C. Future work concerning the parallel evolution of the HCA model include the application of alternative parallel computing resources such as FPGA, and the investigation of alternative implementations of backward evolution that require less memory resources and management, which were prominent sources of overhead for the parallel implementations of this algorithm in this work. Future work concerning the performance of the HCA model include the comparison of its execution time to other cryptographic systems, including sequential and parallel encryption algorithms.

In the case of MPGA, the empirical investigation of its operators concerning the scheduling problem was performed and published as the research paper (MORAIS; OLIVEIRA; CARVALHO, 2019). From the resulting MPGA specification, two approaches were developed in Python by applying the resources of distributed memory (DM-MPGA) and shared memory (SM-MPGA) using the mpi4py and Numba libraries, respectively. The choice of Python was again beneficial for the efficient implementation of the algorithm, especially in the case of the DM-MPGA, which encapsulates each subpopulation as a semi-independent genetic algorithm, with the relatively simple addition of the migration operator. In the case of the SM-MPGA, further changes were needed in the main algorithm to account for the evolution of multiple subpopulations instead of one, making it a more complex program. On the other hand, the use of the Numba compiler to LLVM instructions made the latter implementation more efficient in comparison.

Experiments were conducted to measure the influence of the MPGA parameters of population quantity and migration frequency over its performance. For this algorithm, these parameters were shown to affect the execution time as well as the fitness of its solutions, which is the minimization of the makespan of the corresponding scheduling. Furthermore, the initial experiments revealed a clear trade-off between these two performance metrics, where the minimization of execution time was generally improved by higher values for both parameters, while the minimization of makespan was generally improved by lower values. Thus, in order to assess both metrics in combination, a multiobjective analysis was employed to investigate highly efficient parameter sets for the algorithm. After aggregating and pruning the results, the most occurring configurations

of population quantity and migration frequency were, respectively, (4,50) for the DM-MPGA and (8,10) for the SM-MPGA. As such, the research hypothesis established on section 1.3 was met by both parallel MPGA implementations, as they provided more favorable trade-offs than the sequential algorithm. Considering that the DM-MPGA had more benefit by employing comparably less populations and less occurrence of migration, the results demonstrate the performance advantages of the use of shared memory approaches, which employ lightweight threads, over distributed memory approaches, that are more resource-intensive with the use of processes.

Future work concerning the MPGA include the investigation of the influence of other parameters such as the communication topology, the application of alternative parallel computing resources such as clusters, and the comparison of its performance with other search algorithms such as fine-grained parallel genetic algorithm models that employ execution on GPU. Concerning the application to task scheduling, future work includes experiments with greater datasets of scheduling graphs and processors quantities, the analysis of alternative genetic representations for scheduling solutions and alternative operators for crossover and mutation, the investigation of variations of the scheduling problem with features such as heterogeneous processors, and the comparison with other bio-inspired optimization algorithms such as BOA(ARORA; SINGH, 2019).

Bibliography

ALBA, E.; TROYA, J. M. A survey of parallel distributed genetic algorithms. **Complex.**, John Wiley & Sons, Inc., New York, NY, USA, v. 4, n. 4, p. 31–52, mar. 1999. ISSN 1076-2787.

Anaconda, Inc. **Numba for CUDA GPUs: Examples**. 2012. Https://numba.pydata.org/numba-doc/dev/cuda/examples.html. Accessed: 14-05-2020.

ARORA, S.; SINGH, S. Butterfly optimization algorithm: a novel approach for global optimization. **Soft Computing**, Springer, v. 23, n. 3, p. 715–734, 2019.

Bernstein, A. J. Analysis of programs for parallel processing. **IEEE Transactions on Electronic Computers**, EC-15, n. 5, p. 757–763, 1966.

BLACKBURN, S. R. et al. Comments on" theory and applications of cellular automata in cryptography"[with reply]. **IEEE Transactions on Computers**, IEEE, v. 46, n. 5, p. 637–639, 1997.

CANTÚ-PAZ, E. A survey of parallel genetic algorithms. **Calculateurs paralleles, reseaux et systems repartis**, v. 10, n. 2, p. 141–171, 1998.

CANTÚ-PAZ, E.; GOLDBERG, D. E. Are multiple runs of genetic algorithms better than one? In: CANTÚ-PAZ, E. et al. (Ed.). **Genetic and Evolutionary Computation — GECCO 2003**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 801–812. ISBN 978-3-540-45105-1.

CARVALHO, T. I.; MORAIS, B. W. D.; OLIVEIRA, G. M. B. Bio-inspired and heuristic methods applied to a benchmark of the task scheduling problem. In: IEEE. **2018 7th Brazilian Conference on Intelligent Systems (BRACIS)**. [S.l.], 2018. p. 516–521.

CHITRA, P.; RAJARAM, R.; VENKATESH, P. Application and comparison of hybrid evolutionary multiobjective optimization algorithms for solving task scheduling problem on heterogeneous systems. **Applied Soft Computing**, v. 11, n. 2, p. 2725 – 2734, 2011. ISSN 1568-4946. The Impact of Soft Computing for the Progress of Artificial Intelligence.

COOMBES, S. The geometry and pigmentation of seashells s coombes. In: . [S.l.: s.n.], 2009.

CORREA, R. C.; FERREIRA, A.; REBREYEND, P. Scheduling multiprocessor tasks with genetic algorithms. **IEEE Transactions on Parallel and Distributed Systems**, v. 10, n. 8, p. 825–837, Aug 1999. ISSN 1045-9219.

DAEMEN, J.; RIJMEN, V. **The design of Rijndael: AES-the advanced encryption standard**. [S.l.]: Springer Science & Business Media, 2013.

DALCIN, L. D. et al. Parallel distributed computing using python. **Advances in Water Resources**, v. 34, n. 9, p. 1124 – 1139, 2011. ISSN 0309-1708. New Computational Methods and Software Tools. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0309170811000777>.

DEVI, D. I.; CHITHRA, S.; SETHUMADHAVAN, M. Hardware random number generator using fpga. **Journal of Cyber Security and Mobility**, River Publishers, v. 8, n. 4, p. 409–418, 2019.

DRéO, J. **Optimum de Pareto**. 2006. [Accessed 01-10-2020]. Disponível em: <https://en.wikipedia.org/wiki/File:Front_pareto.svg>.

GEHRING, H.; BORTFELDT, A. A parallel genetic algorithm for solving the container loading problem. v. 9, p. 497 – 511, 07 2002.

GIITSIDIS, T.; DOURVAS, N. I.; SIRAKOULIS, G. C. Parallel implementation of aircraft disembarking and emergency evacuation based on cellular automata. **The International Journal of High Performance Computing Applications**, SAGE Publications Sage UK: London, England, v. 31, n. 2, p. 134–151, 2017.

GOLDBERG, D. E. **Genetic Algorithms in Search, Optimization and Machine Learning**. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN 0201157675.

GREEN, D. Cellular automata models in biology. **Mathematical and Computer Modelling**, Elsevier, v. 13, n. 6, p. 69–74, 1990.

GUTOWITZ, H. Cryptography with dynamical systems. In: **Cellular Automata and Cooperative Systems**. [S.l.]: Springer, 1993. p. 237–274.

HAN, K.-H. et al. Parallel quantum-inspired genetic algorithm for combinatorial optimization problem. In: **Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)**. [S.l.: s.n.], 2001. v. 2, p. 1422–1429 vol. 2.

HOU, E. S. H.; ANSARI, N.; REN, H. A genetic algorithm for multiprocessor scheduling. **IEEE Transactions on Parallel and Distributed Systems**, v. 5, n. 2, p. 113–120, Feb 1994. ISSN 1045-9219.

HWANG, R.; GEN, M.; KATAYAMA, H. A comparison of multiprocessor task scheduling algorithms with communication costs. **Computers & Operations Research**, v. 35, n. 3, p. 976 – 993, 2008. ISSN 0305-0548. Part Special Issue: New Trends in Locational Analysis.

KAUR, K. et al. Heuristics based genetic algorithm for scheduling static tasks in homogeneous parallel system. **International Journal of Computer Science and Security (IJCSS)**, v. 4, n. 2, 1999.

KWOK, Y.-K.; AHMAD, I. Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. **J. Parallel Distrib. Comput.**, Academic Press, Inc., Orlando, FL, USA, v. 47, n. 1, p. 58–77, nov. 1997. ISSN 0743-7315.

_____. Static scheduling algorithms for allocating directed task graphs to multiprocessors. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 31, n. 4, p. 406–471, dez. 1999. ISSN 0360-0300.

LAM, S. K.; PITROU, A.; SEIBERT, S. Numba: A llvm-based python jit compiler. In: **Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC**. New York, NY, USA: Association for Computing Machinery, 2015. (LLVM '15). ISBN 9781450340052.

LIMA, D.; OLIVEIRA, G. A probabilistic cellular automata ant memory model for a swarm of foraging robots. In: . [S.l.: s.n.], 2016. p. 1–6.

LING, R. **A textile cone snail**. 2005. [Accessed 01-10-2020]. Disponível em: <https://commons.wikimedia.org/wiki/File:Textile_cone.JPG>.

LIRA, E. **Criptografia Simétrica com Autômatos Celulares Altamente Híbridos**. Monografia (Qualificação de Doutorado) — Universidade Federal de Uberlândia, Uberlândia, 2020.

LÓPEZ-FANDINO, J. et al. Gpu projection of ecas-ii segmenter for hyperspectral images based on cellular automata. **IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing**, IEEE, v. 10, n. 1, p. 20–28, 2016.

MACÊDO, H. B. de. **A new cryptography method based on the pre-image calculus of chaotic, non-homogeneous and non-additive cellular automata**. Dissertação (Mestrado) — Universidade Federal de Uberlândia, Uberlândia, 2007.

MACÊDO, H. B. de; OLIVEIRA, G. M. B. de; RIBEIRO, C. H. C. Dynamic behaviour of network cellular automata with non-chaotic standard rules. In: IEEE. **2014 Second World Conference on Complex Systems (WCCS)**. [S.l.], 2014. p. 451–456.

MAROWKA, A. Python accelerators for high-performance computing. **J. Supercomput.**, Kluwer Academic Publishers, USA, v. 74, n. 4, p. 1449–1460, abr. 2018. ISSN 0920-8542.

MENEZES, A. J.; VANSTONE, S. A.; OORSCHOT, P. C. V. **Handbook of Applied Cryptography**. 1st. ed. Boca Raton, FL, USA: CRC Press, Inc., 1996. ISBN 0849385237.

MIETTINEN, K. **Nonlinear Multiobjective Optimization**. Springer US, 1999. (International Series in Operations Research & Management Science). ISBN 9780792382782. Disponível em: <https://books.google.com.br/books?id=ha_zLdNtXSMC>.

MITCHELL, M. **Complexity: A guided tour**. [S.l.]: Oxford University Press, 2009.

MITCHELL, M. et al. Computation in cellular automata: A selected review. **Nonstandard Computation**, Citeseer, p. 95–140, 1996.

MITCHUM, R. **The Parallel Path to Computing Present and Future**. 2013. [Accessed 01-10-2020]. Disponível em: <https://voices.uchicago.edu/compinst/blog/parallel-path-computing-present-and-future>.

MORADY, R.; DAL, D. A multi-population based parallel genetic algorithm for multiprocessor task scheduling with communication costs. In: **2016 IEEE Symposium on Computers and Communication (ISCC)**. [S.l.: s.n.], 2016. p. 766–772.

MORAIS, B. W. D. **Modelos evolutivos aplicados ao escalonamento de tarefas em sistemas multiprocessados: algoritmo genético serial e multipopulação**. 36 p. Monografia (Trabalho de Conclusão de Curso: Graduação em Ciência da Computação) — Universidade Federal de Uberlândia, Uberlândia, 2017.

MORAIS, B. W. D.; OLIVEIRA, G. M. B. de; CARVALHO, T. I. de. Evolutionary models applied to multiprocessor task scheduling: Serial and multipopulation genetic algorithm. **Revista de Informática Teórica e Aplicada**, v. 26, n. 1, p. 11–25, 2019.

MüHLENBEIN, H.; SCHOMISCH, M.; BORN, J. The parallel genetic algorithm as function optimizer. **Parallel Comput.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 17, n. 6-7, p. 619–632, set. 1991. ISSN 0167-8191.

NANDI, S.; KAR, B.; CHAUDHURI, P. P. Theory and applications of cellular automata in cryptography. **IEEE Transactions on computers**, IEEE, v. 43, n. 12, p. 1346–1357, 1994.

NTINAS, V. G. et al. Gpu and fpga parallelization of fuzzy cellular automata for the simulation of wildfire spreading. In: WYRZYKOWSKI, R. et al. (Ed.). **Parallel Processing and Applied Mathematics**. Cham: Springer International Publishing, 2016. p. 560–569. ISBN 978-3-319-32152-3.

NVIDIA; VINGELMANN, P.; FITZEK, F. H. **CUDA, release: 10.2.89**. 2020. Disponível em: <https://developer.nvidia.com/cuda-toolkit>.

NVIDIA Corporation. **CUDA Toolkit Documentation**. 2007. Https://docs.nvidia.com/cuda/cuda-c-programming-guide.

OLIVEIRA, G.; COELHO, A.; MONTEIRO, L. Cellular automata cryptographic model based on bi-directional toggle rules. **International Journal of Modern Physics C**, World Scientific, v. 15, n. 08, p. 1061–1068, 2004.

OLIVEIRA, G. et al. **A cryptographic model based on the preimage computation of cellular automata. Automata-2008: Theory and Applications of Cellular Automata**. [S.l.]: Luniver Press, Bristol, UK, 2008.

_____. A cellular automata-based cryptographic model with a variable-length ciphertext. In: **CSC 2010: proceedings of the 2010 international conference on scientific computing (Las Vegas NV, July 12-15, 2010)**. [S.l.: s.n.], 2010. p. 19–25.

OLIVEIRA, G. M.; MARTINS, L. G.; ALT, L. S. Deeper investigating adequate secret ey specifications for a ariable ength cryptographic cellular automata based model. **Cellular Automata: Innovative Modelling for Science and Engineering**, BoD–Books on Demand, p. 265, 2011.

OLIVEIRA, G. M. et al. Secret key specification for a variable-length cryptographic cellular automata model. In: SPRINGER. **International Conference on Parallel Problem Solving from Nature**. [S.l.], 2010. p. 381–390.

G. M. B. Oliveira e H. Macedo. **Sistema criptográfico baseado no cálculo de pré-imagem em autômatos celulares não-homogêneos, não-aditivos e com dinâmica caótica**. 2019. Número do registro: PI0703188-2.

OMARA, F. A.; ARAFA, M. M. Genetic algorithms for task scheduling problem. **Journal of Parallel and Distributed Computing**, v. 70, n. 1, p. 13 – 22, 2010. ISSN 0743-7315.

QI, J. G.; BURNS, G. R.; HARRISON, D. K. The application of parallel multipopulation genetic algorithms to dynamic job-shop scheduling. **The International Journal of Advanced Manufacturing Technology**, v. 16, n. 8, p. 609–615, Jul 2000. ISSN 1433-3015.

QUINN, M.; QUINN. **Parallel Computing: Theory and Practice**. McGraw-Hill, 1994. (Instructor's manual to accompany). ISBN 9780070512955. Disponível em: <https://books.google.com.br/books?id=RDraAAAACAAJ>.

RESCORLA, E.; MODADUGU, N. The transport layer security. In: **TLS) Protocol Version 1.1. Internet RFC 4346**. [S.l.: s.n.], 2006.

RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. **Communications of the ACM**, 1978.

SARKAR, P. A brief history of cellular automata. **Acm computing surveys (csur)**, ACM New York, NY, USA, v. 32, n. 1, p. 80–107, 2000.

SCHMIDT, B. et al. **Parallel Programming**. [S.l.]: Elsevier, 2017. ISBN 9780128044865.

SEN, S. et al. Cellular automata based cryptosystem (cac). In: DENG, R. et al. (Ed.). **Information and Communications Security**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. p. 303–314. ISBN 978-3-540-36159-6.

SRINIVASA, K. G.; VENUGOPAL, K. R.; PATNAIK, L. M. A self-adaptive migration model genetic algorithm for data mining applications. **Inf. Sci.**, Elsevier Science Inc., New York, NY, USA, v. 177, n. 20, p. 4295–4313, out. 2007. ISSN 0020-0255.

STALLINGS, W. **Cryptography and Network Security: Principles and Practice**. 6th. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2013. ISBN 0133354695, 9780133354690.

TINOCO, C. R.; OLIVEIRA, G. M. Heterogeneous teams of robots using a coordinating model for surveillance task based on cellular automata and repulsive pheromone. In: IEEE. **2019 IEEE Congress on Evolutionary Computation (CEC)**. [S.l.], 2019. p. 747–754.

TOFFOLI, T. Cam: A high-performance cellular-automaton machine. **Physica D: Nonlinear Phenomena**, Elsevier, v. 10, n. 1-2, p. 195–204, 1984.

WANG, L. et al. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. **Journal of Parallel and Distributed Computing**, v. 47, n. 1, p. 8 – 22, 1997. ISSN 0743-7315.

WHITE, S. H.; REY, A. M. D.; SÁNCHEZ, G. R. Modeling epidemics using cellular automata. **Applied Mathematics and Computation**, Elsevier, v. 186, n. 1, p. 193–202, 2007.

Wikipedia contributors. **Cellular Automaton Wolfram-rule-30 colored based on previous state**. 2020. [Online; accessed 01-November-2020]. Disponível em: <https://en.wikipedia.org/wiki/File:Cellular_Automata_running_Wolfram-rule-30.svg>.

WOLFRAM, S. Cryptography with cellular automata. In: WILLIAMS, H. C. (Ed.). **Advances in Cryptology — CRYPTO '85 Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986. p. 429–432. ISBN 978-3-540-39799-1.

WUENSCHE, A. Encryption using cellular automata chain-rules. In: **Automata**. [S.l.: s.n.], 2008. p. 126–138.

XIA, C. et al. A high-performance cellular automata model for urban simulation based on vectorization and parallel computing technology. **International Journal of Geographical Information Science**, Taylor Francis, v. 32, n. 2, p. 399–424, 2018.

XU, Y. et al. A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues. **Information Sciences**, v. 270, p. 255 – 287, 2014. ISSN 0020-0255.

YAO, J.; KHARMA, N.; GROGONO, P. Bi-objective multipopulation genetic algorithm for multimodal function optimization. **IEEE Transactions on Evolutionary Computation**, v. 14, n. 1, p. 80–102, Feb 2010. ISSN 1089-778X.

Appendix

APPENDIX **A**

Matrix Multiplication implementations

Listing A.1 – The matrix multiplication algorithm in Python. Source: (MAROWKA, 2018).

```
def matmul(A, B, C) :
    m, n = A.shape
    n, p = B.shape
    for i in range(m) :
        for j in range(p) :
            C[i, j] = 0
            for k in range(n) :
                C[i, j] += A[i, k] * B[k, j]
```

Listing A.2 – The matrix multiplication algorithm in Python set for optimization with Numba.

```
@njit
def matmul_numba(A, B, C) :
    m, n = A.shape
    n, p = B.shape
    for i in range(m) :
        for j in range(p) :
            C[i, j] = 0
            for k in range(n) :
                C[i, j] += A[i, k] * B[k, j]
```

Listing A.3 – The parallel matrix multiplication algorithm in Python using Numba's `prange` function.

```
@njit(parallel=True)
def matmul_parallel(A, B, C) :
    m, n = A.shape
    n, p = B.shape
```

```
5       for i in prange(m) :
6           for j in prange(p) :
7               C[i, j] = 0
8               for k in prange(n) :
9                   C[i, j] += A[i, k] * B[k, j]
```

Listing A.4 – The parallel matrix multiplication algorithm for GPU using the Numba CUDA API. Source: (MAROWKA, 2018).

```
1 @cuda.jit
2 def cuda_matmul(a, b, c, n):
3     x, y = cuda.grid (2)
4     if (x >= n) or (y >= n):
5         return
6     c[x, y] = 0
7     for i in range(n) :
8         c[x, y] += a[x, i] * b[i, y]
9
10 device = cuda.get_current_device()
11 tpb = device.WARP_SIZE
12 n = 320
13 bpg = (n+tpb-1)//tpb
14 grid_dim = (bpg, bpg)
15 block_dim = (tpb , tpb)
```

APPENDIX B

Results of the experiments with MPGA

This appendix contains the results of the experiments with makespan and the the multiobjective analysis with the MPGA results.

B.1 Makespan

Table 8 presents the results of the experiments with makespan.

Table 8 – Results of the experiments with makespan for each graph, population quantity and migration frequency parameters, sorted by mean makespan.

graph	pop qty	migf	mean makespan	min makespan
fft39	8	1	1344.6	1200
	2	20	1347	1200
	2	100	1353.6	1260
	4	1	1354.8	1140
	16	1	1357.2	1140
	1	-	1357.8	1080
	4	5	1357.8	1200
	2	1	1358.4	1200
	2	5	1358.4	1200
	2	10	1360.2	1140
	16	5	1363.2	1140
	2	50	1366.2	1200
	4	10	1366.2	1200
	8	5	1366.2	1200
	2	200	1369.2	1140
	4	20	1370.4	1200
	4	50	1372.2	1260

	8	10	1373.4	1200
	8	20	1374.6	1260
	4	100	1380.6	1140
	16	10	1386.6	1260
	4	200	1395	1260
	8	50	1405.2	1260
	16	20	1410.6	1260
	8	100	1414.8	1260
	8	200	1422.6	1260
	16	50	1434.6	1260
	16	100	1450.8	1320
	16	200	1468.2	1320

fft95	2	10	2371.2	2220
	4	1	2375.4	2220
	2	5	2376.6	2220
	4	5	2381.4	2220
	8	1	2383.2	2220
	2	1	2385	2220
	16	1	2388.6	2160
	1	-	2397	2220
	2	20	2397	2160
	2	50	2404.2	2220
	4	10	2408.4	2220
	2	200	2425.2	2220
	8	5	2431.8	2280
	2	100	2436	2220
	4	20	2437.8	2220
	8	10	2455.2	2280
	16	5	2468.4	2280
	4	50	2470.8	2280
	4	100	2481	2280
	8	20	2508	2280
	4	200	2509.2	2220
	16	10	2526.6	2340
	8	50	2557.2	2400
	8	100	2594.4	2460

	8	200	2597.4	2460
	16	20	2599.8	2340
	16	50	2658	2400
	16	100	2721	2520
	16	200	2744.4	2580

fft223	1	-	4464.6	4080
	2	5	4473.6	4140
	8	1	4473.6	4080
	2	10	4483.8	4080
	2	1	4490.4	4200
	4	1	4493.4	4080
	16	1	4531.2	4200
	4	5	4547.4	4200
	2	20	4553.4	4200
	4	10	4611	4260
	2	50	4624.8	4260
	2	200	4642.8	4320
	2	100	4647	4260
	8	5	4648.8	4260
	4	20	4715.4	4380
	8	10	4804.2	4380
	4	50	4807.8	4500
	16	5	4867.8	4500
	4	100	4896	4620
	4	200	4897.2	4560
	8	20	4943.4	4500
	16	10	5066.4	4680
	8	50	5121	4740
	8	100	5199	4800
	16	20	5254.2	4920
	8	200	5275.8	4680
	16	50	5492.4	4980
	16	100	5625	5220
	16	200	5689.2	5160

fft511	1	-	9633	9000

	8	1	9654	9000
	2	10	9675	8940
	4	1	9696.6	9000
	2	1	9700.8	9060
	2	5	9717	9000
	2	20	9773.4	9240
	16	1	9839.4	9000
	4	5	9877.2	9000
	4	10	9913.2	9300
	2	50	9934.2	9300
	2	100	10001.4	9300
	8	5	10107.6	9120
	2	200	10131	9540
	4	20	10163.4	9480
	8	10	10438.8	9720
	4	50	10447.8	9720
	4	100	10549.8	9720
	16	5	10567.2	9960
	4	200	10657.8	9960
	8	20	10756.8	9960
	16	10	10917	10260
	8	50	11116.8	10380
	8	100	11274.6	10740
	8	200	11340.6	10620
	16	20	11355.6	10620
	16	50	11810.4	11160
	16	100	12068.4	11520
	16	200	12181.2	11520

lap36	16	5	2854.6	2570
	2	5	2866	2580
	8	10	2871.4	2510
	16	10	2873.7	2650
	8	5	2875.6	2320
	16	1	2888.9	2460
	2	100	2889.7	2500
	2	10	2889.8	2320

	8	1	2896.5	2500
	1	-	2900.4	2570
	4	1	2901.5	2620
	4	10	2902.8	2540
	8	20	2911.6	2490
	2	1	2918.2	2540
	2	50	2918.7	2490
	16	20	2920.6	2620
	4	20	2920.7	2490
	4	5	2922.1	2430
	2	20	2926.1	2530
	4	50	2928.7	2560
	2	200	2940	2620
	4	100	2949.4	2630
	8	50	2956.2	2620
	4	200	2970.8	2500
	8	100	2990.7	2580
	16	50	3015	2710
	8	200	3053.3	2790
	16	100	3075.3	2780
	16	200	3125.7	2840

lap64	2	5	4646.8	4200
	1	-	4647.2	4300
	8	1	4647.5	3920
	4	1	4654.2	4140
	16	1	4659.7	4240
	4	10	4663.9	4200
	4	20	4670.4	4290
	4	5	4675.4	4280
	2	1	4696.3	4320
	8	10	4696.4	4100
	2	50	4696.8	4150
	8	5	4699.1	4310
	4	50	4709.5	4250
	16	5	4712.4	4360
	2	10	4712.6	4300

	2	20	4714.8	4230
	2	100	4717.9	4180
	16	10	4745	4190
	8	20	4753.1	4380
	2	200	4759	4180
	4	100	4776.8	4300
	8	50	4824.7	4350
	4	200	4851.3	4320
	16	20	4873.9	4380
	8	100	4918.8	4330
	8	200	4969.1	4400
	16	50	4992.4	4520
	16	100	5097.1	4800
	16	200	5167.1	4740

lap100	1	-	6686.8	5890
	8	1	6695.9	6200
	2	1	6706.2	6160
	2	5	6709.8	6120
	4	5	6714.2	6140
	16	1	6719.1	6100
	2	20	6727	6140
	4	10	6733.1	6120
	2	50	6740.1	6220
	8	5	6744.8	6250
	4	1	6752.3	6070
	4	20	6753	6140
	2	100	6757.6	6200
	8	10	6777.4	6320
	2	10	6801.6	6220
	2	200	6832.4	6190
	4	50	6857	6140
	16	5	6863.4	6320
	4	100	6929.1	6480
	8	20	6931.1	6430
	4	200	6995.7	6440
	16	10	7004.1	6430

	8	50	7071.8	6570
	8	100	7166.8	6610
	16	20	7219	6860
	8	200	7252.8	6670
	16	50	7417.2	6910
	16	100	7513.4	6990
	16	200	7577.1	6990

lap144	1	-	8999.4	8360
	2	1	9002.3	8330
	4	1	9028.8	8300
	2	10	9032.3	8420
	2	5	9046.8	8470
	2	20	9050.4	8110
	4	5	9051.6	8520
	8	1	9059.5	8390
	4	10	9085.9	8460
	8	5	9105.1	8360
	16	1	9108.2	8520
	2	50	9123.9	8430
	2	100	9142.6	8480
	4	20	9146.7	8410
	2	200	9204.7	8570
	8	10	9278	8790
	4	50	9345.7	8770
	16	5	9368.6	8620
	4	100	9449.1	8960
	4	200	9489.8	8930
	8	20	9496.9	8800
	16	10	9652.4	9130
	8	50	9727.4	9220
	8	100	9861.1	9290
	16	20	9894.6	9350
	8	200	9926	9200
	16	50	10194.7	9720
	16	100	10413.2	9990
	16	200	10542.3	10050

B.2 Multiobjective analysis

This section presents the data resulting from the multiobjective analysis with both MPGA implementations.

Figures 21-24 present the data resulting from step 2 of the multiobjective analysis with the DM-MPGA. Figures 25-28 present the data resulting from step 2 of the multiobjective analysis with the SM-MPGA.

Figures 29-32 present the data resulting from step 3 of the multiobjective analysis with the DM-MPGA. Figures 33-36 present the data resulting from step 3 of the multiobjective analysis with the SM-MPGA.

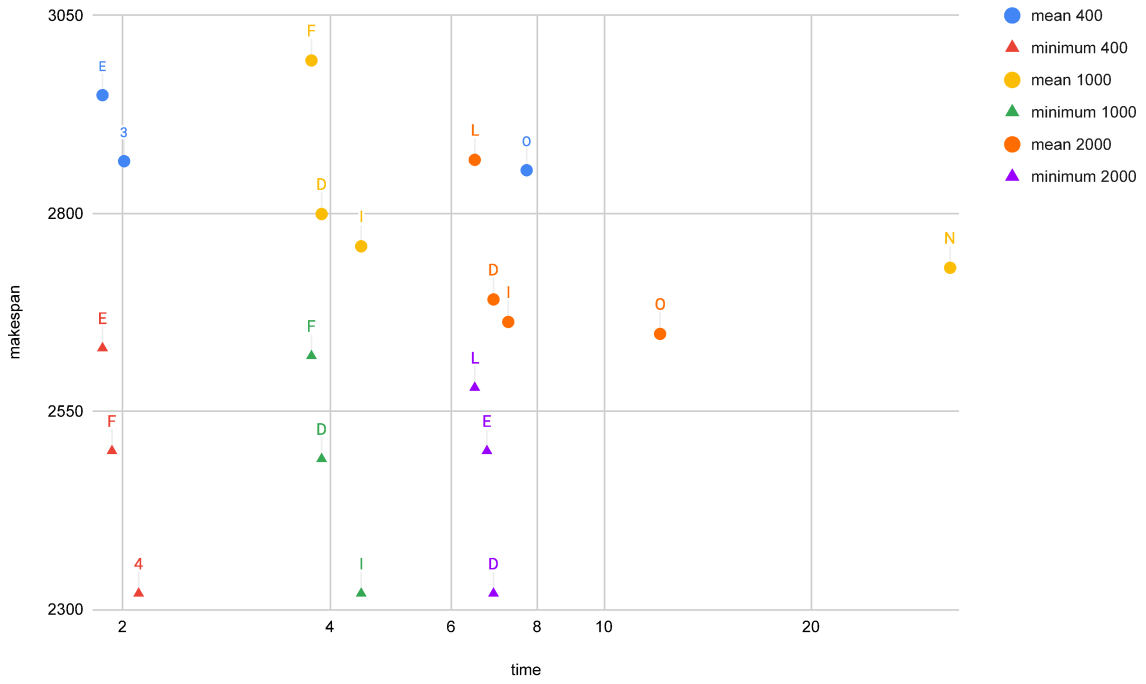

Figure 21 – Highly efficient configurations selected from the Pareto frontiers of the results for the graph lap36 and the DM-MPGA with population sizes 400, 1000, 2000.

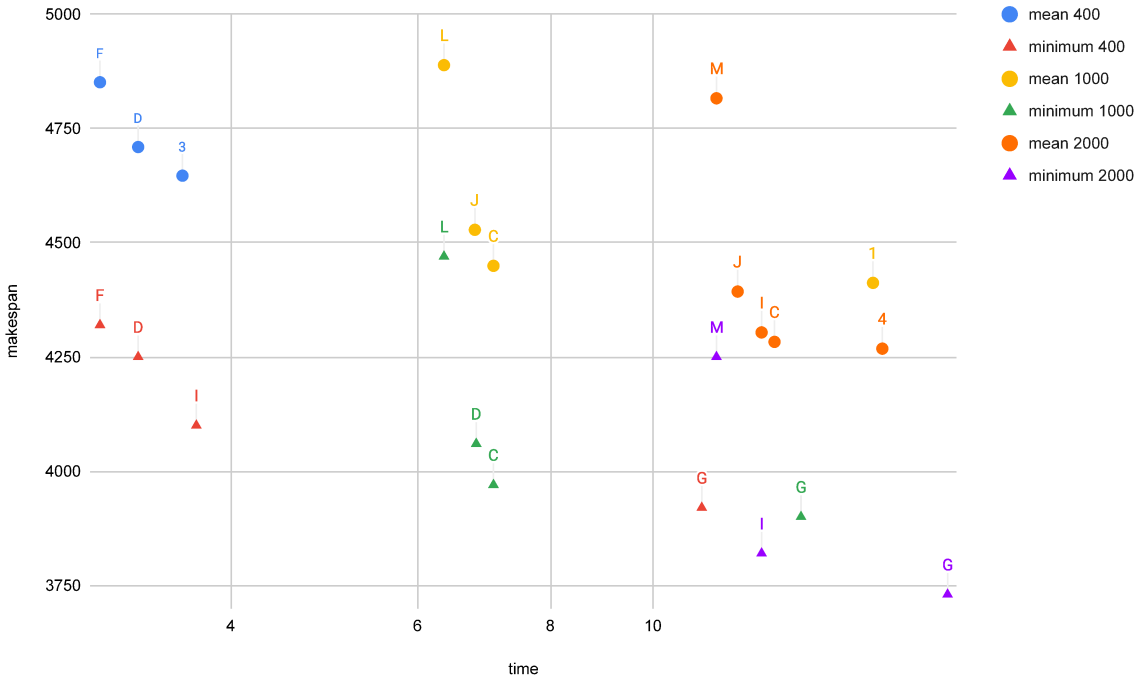

Figure 22 – Highly efficient configurations selected from the Pareto frontiers of the results for the graph lap64 and the DM-MPGA with population sizes 400, 1000, 2000.

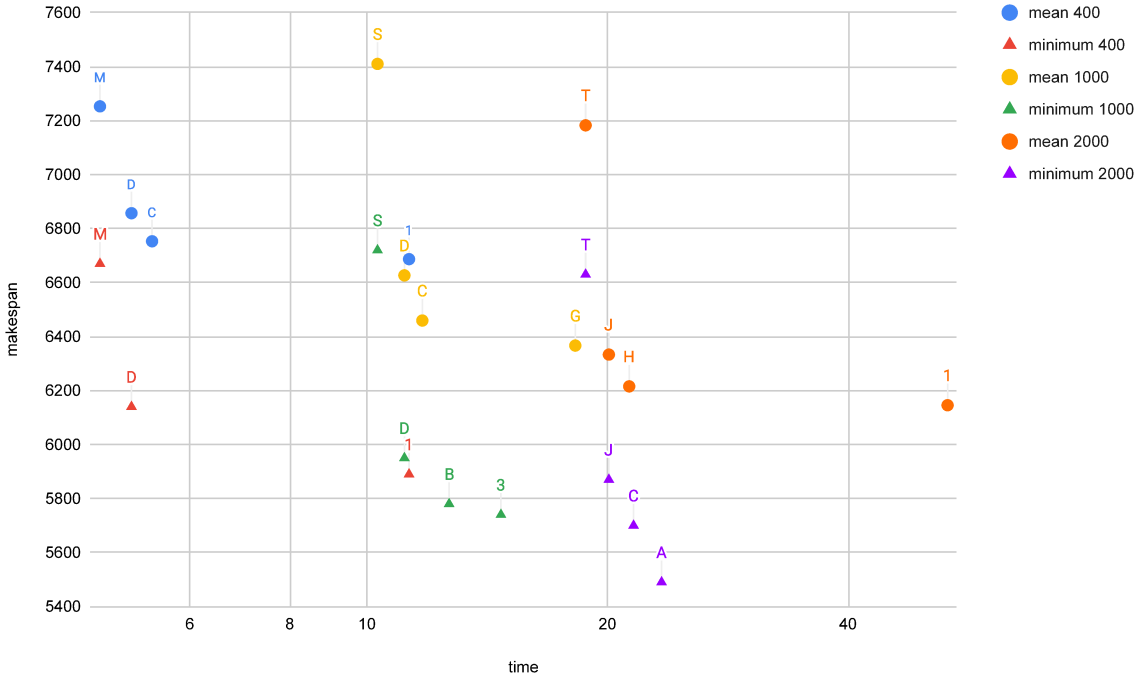

Figure 23 – Highly efficient configurations selected from the Pareto frontiers of the results for the graph lap100 and the DM-MPGA with population sizes 400, 1000, 2000.

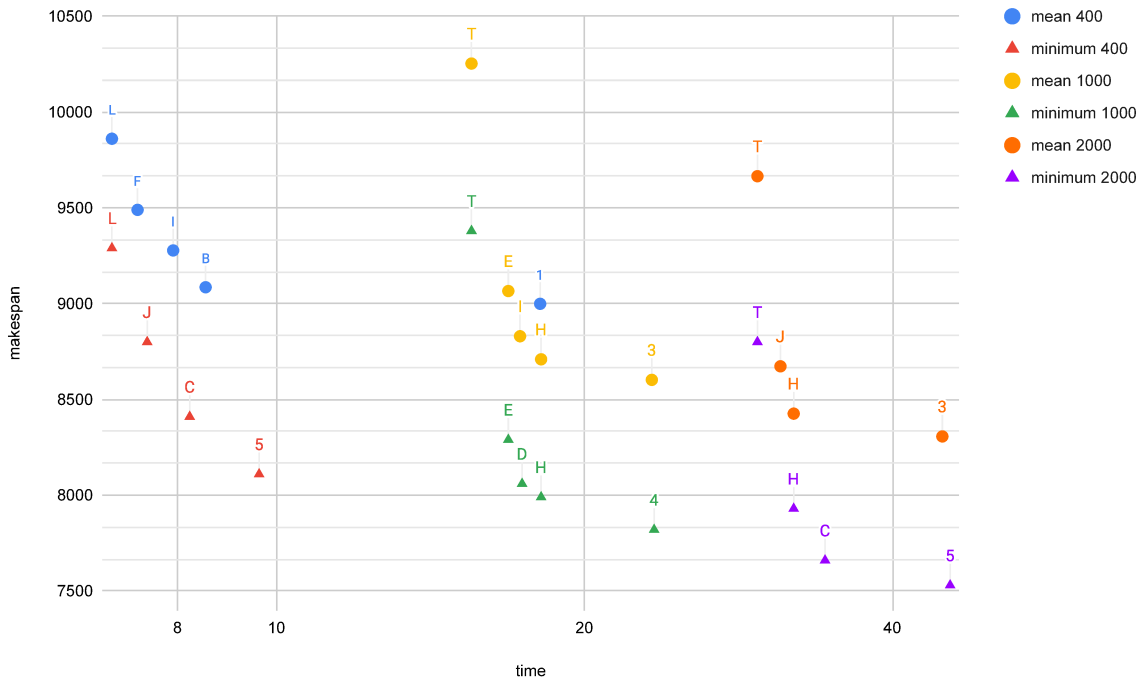

Figure 24 – Highly efficient configurations selected from the Pareto frontiers of the results for the graph lap144 and the DM-MPGA with population sizes 400, 1000, 2000.

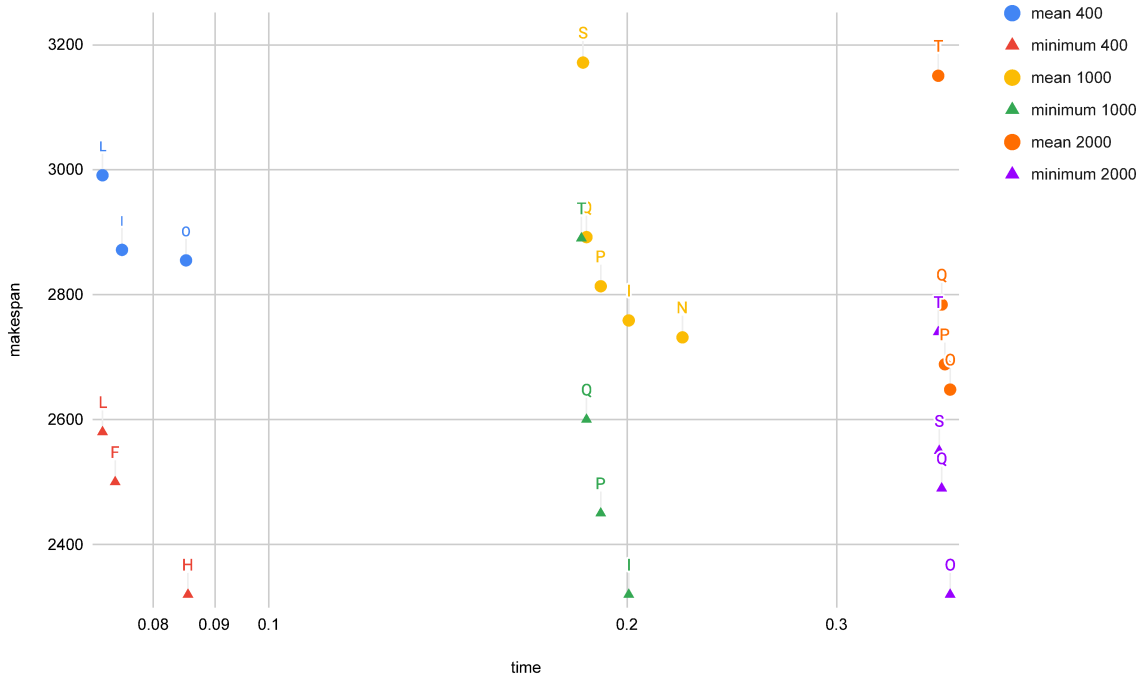

Figure 25 – Highly efficient configurations selected from the Pareto frontiers of the results for the graph lap36 and the SM-MPGA with population sizes 400, 1000, 2000.

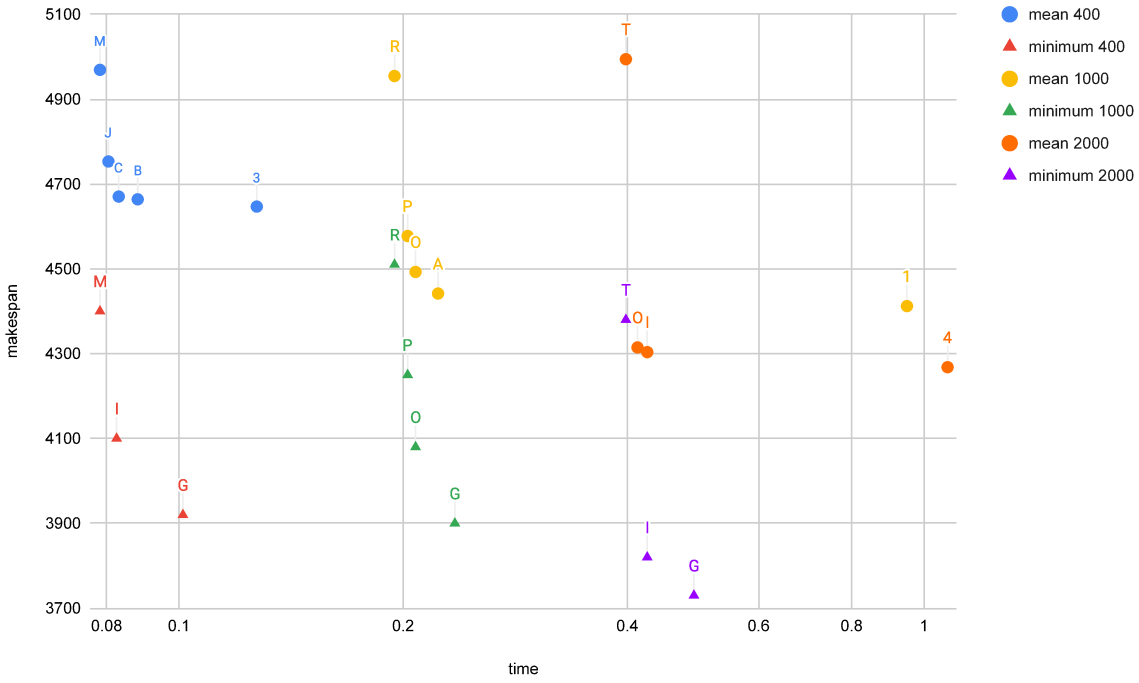

Figure 26 – Highly efficient configurations selected from the Pareto frontiers of the results for the graph lap64 and the SM-MPGA with population sizes 400, 1000, 2000.

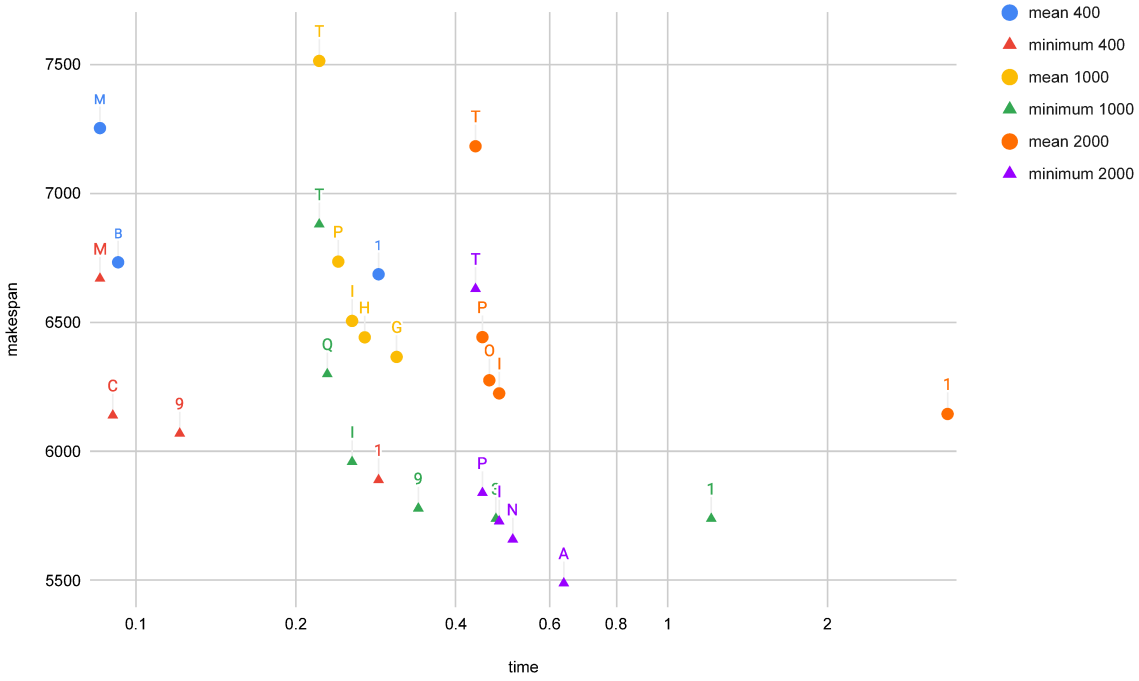

Figure 27 – Highly efficient configurations selected from the Pareto frontiers of the results for the graph lap100 and the SM-MPGA with population sizes 400, 1000, 2000.

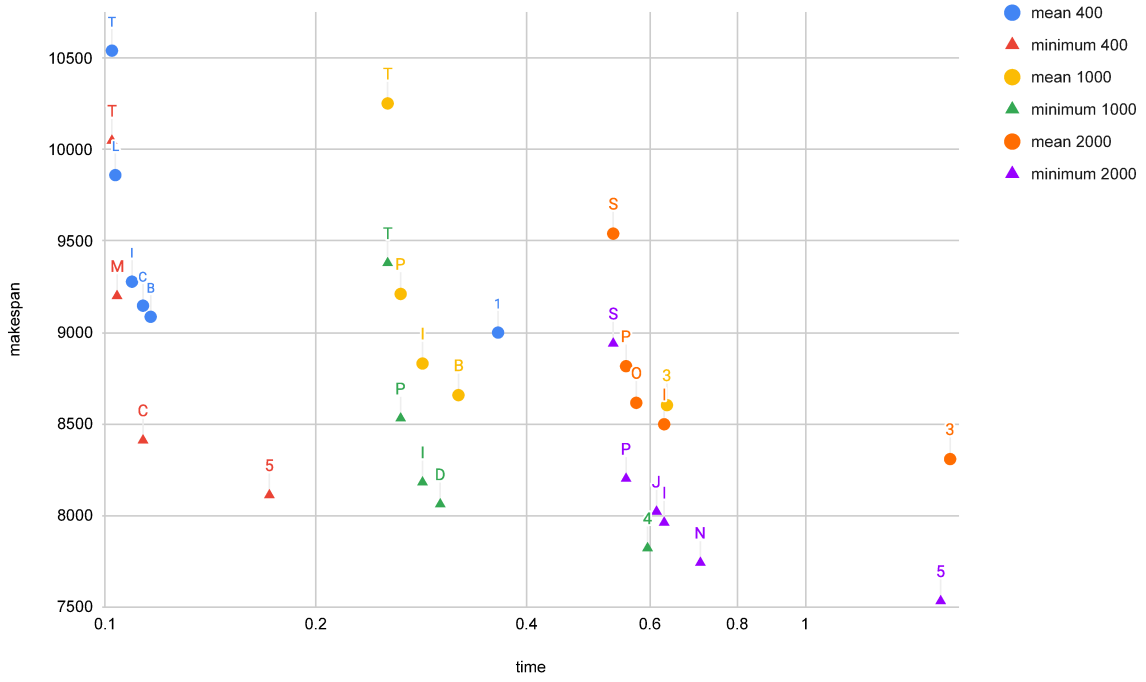

Figure 28 – Highly efficient configurations selected from the Pareto frontiers of the results for the graph lap144 and the SM-MPGA with population sizes 400, 1000, 2000.

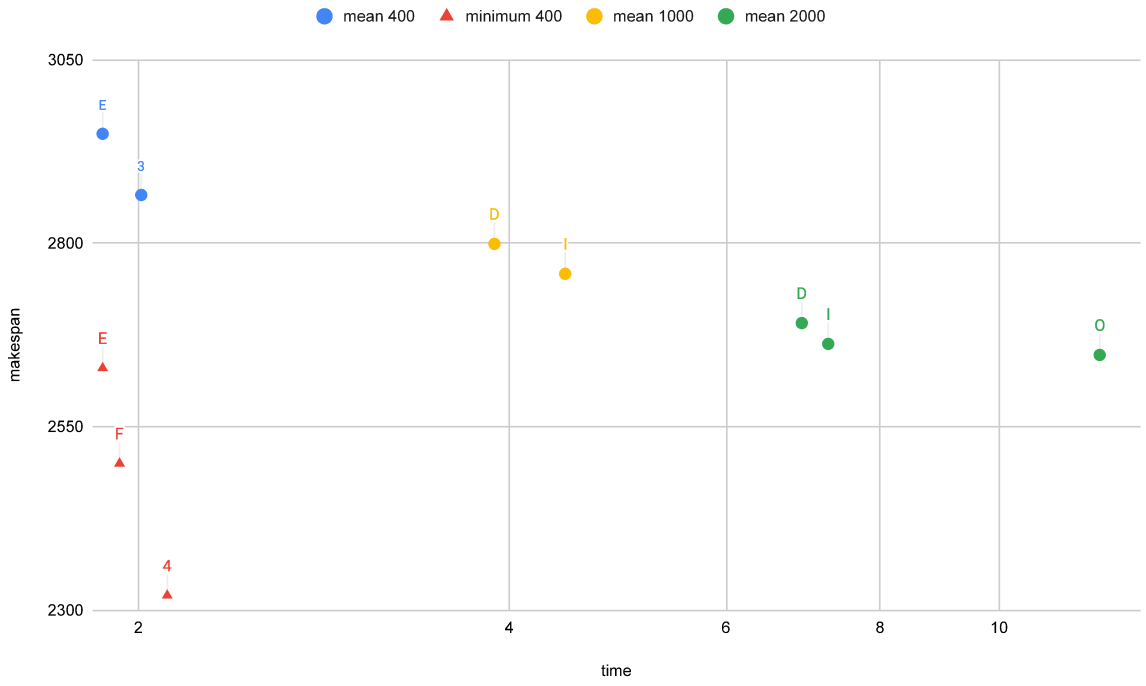

Figure 29 – Non-dominated data points selected from the highly efficient configurations from step 2 for the graph lap36 and the DM-MPGA with population sizes 400, 1000, 2000.

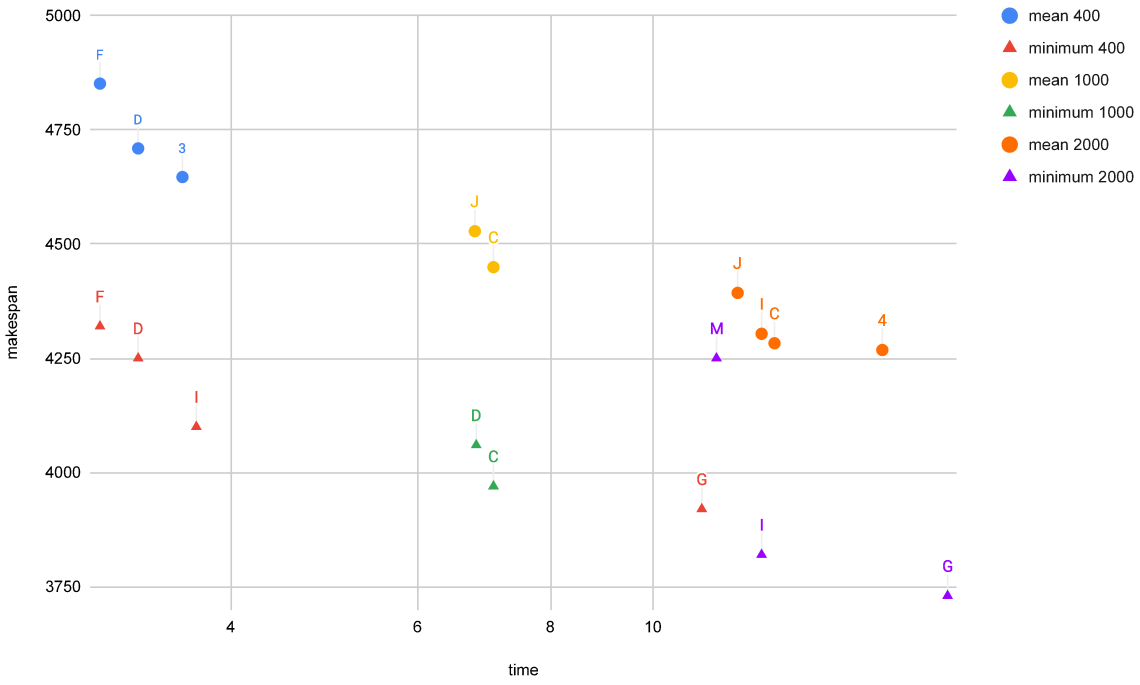

Figure 30 – Non-dominated data points selected from the highly efficient configurations from step 2 for the graph lap64 and the DM-MPGA with population sizes 400, 1000, 2000.

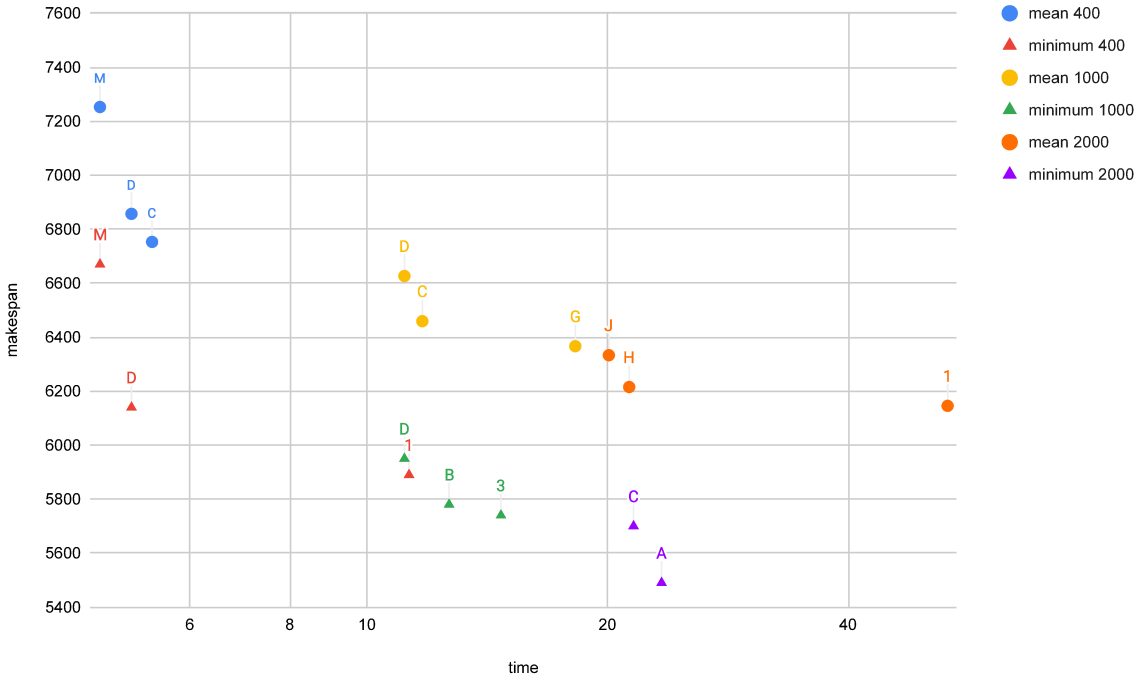

Figure 31 – Non-dominated data points selected from the highly efficient configurations from step 2 for the graph lap100 and the DM-MPGA with population sizes 400, 1000, 2000.

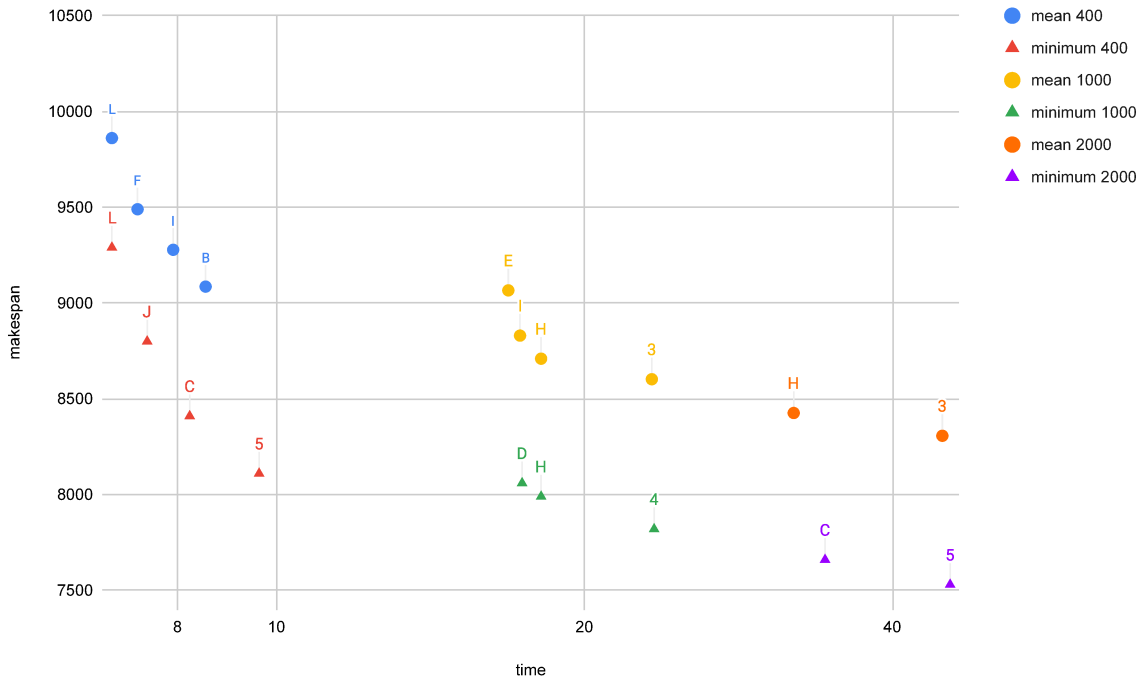

Figure 32 – Non-dominated data points selected from the highly efficient configurations from step 2 for the graph lap144 and the DM-MPGA with population sizes 400, 1000, 2000.

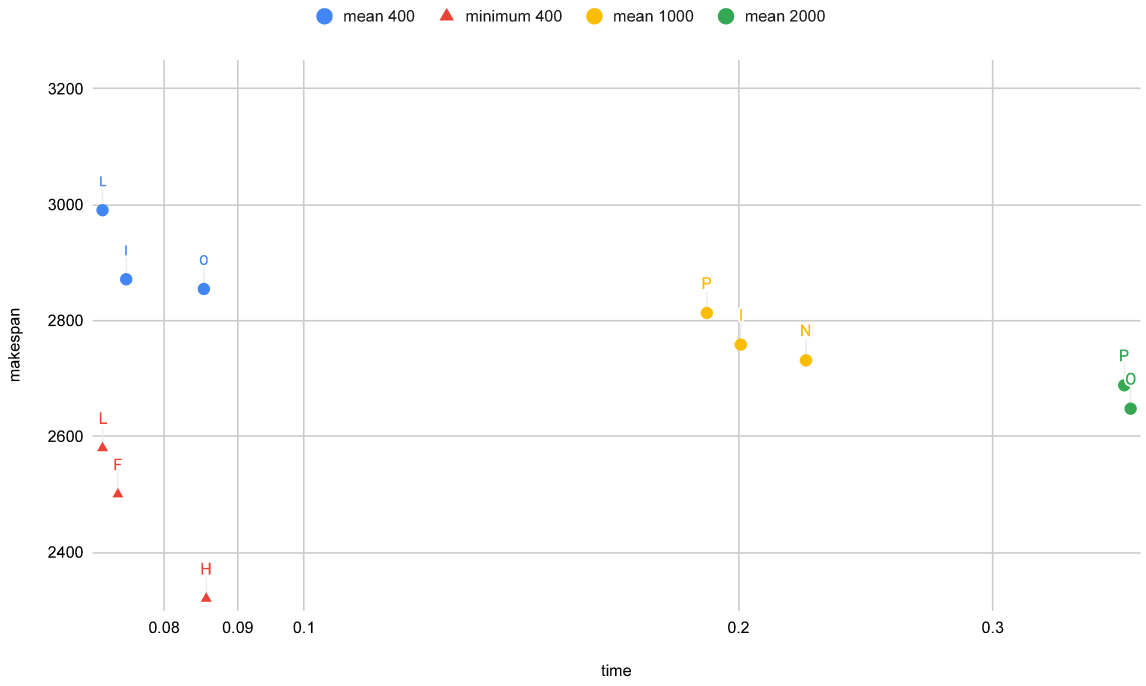

Figure 33 – Non-dominated data points selected from the highly efficient configurations from step 2 for the graph lap36 and the SM-MPGA with population sizes 400, 1000, 2000.

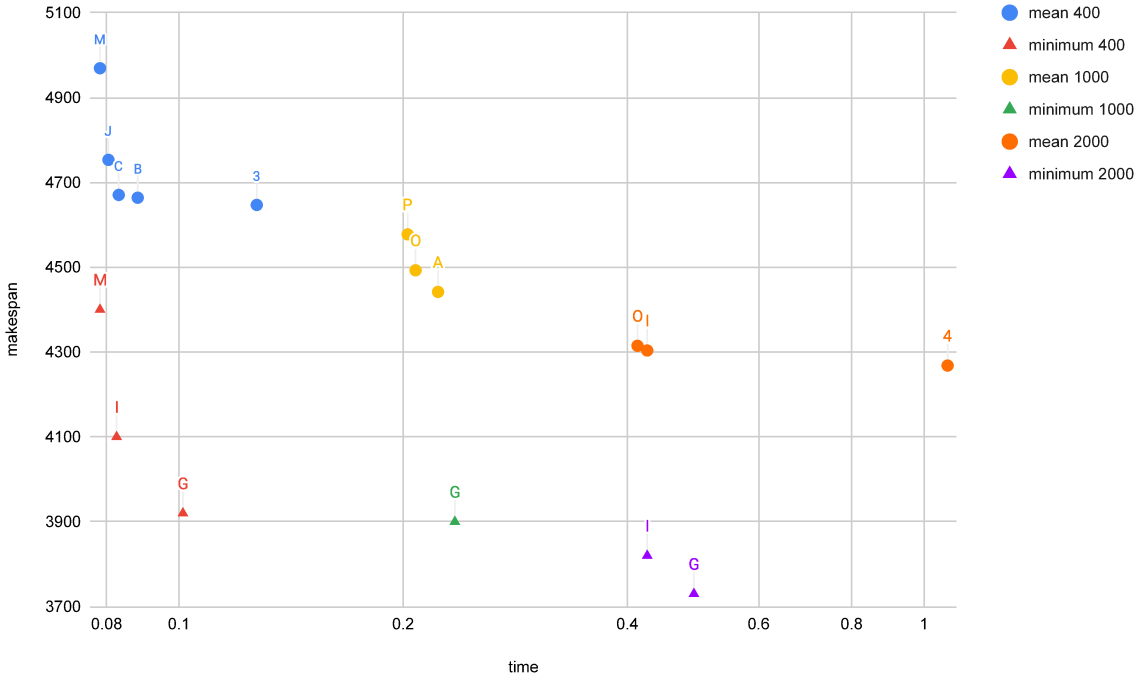

Figure 34 – Non-dominated data points selected from the highly efficient configurations from step 2 for the graph lap64 and the SM-MPGA with population sizes 400, 1000, 2000.

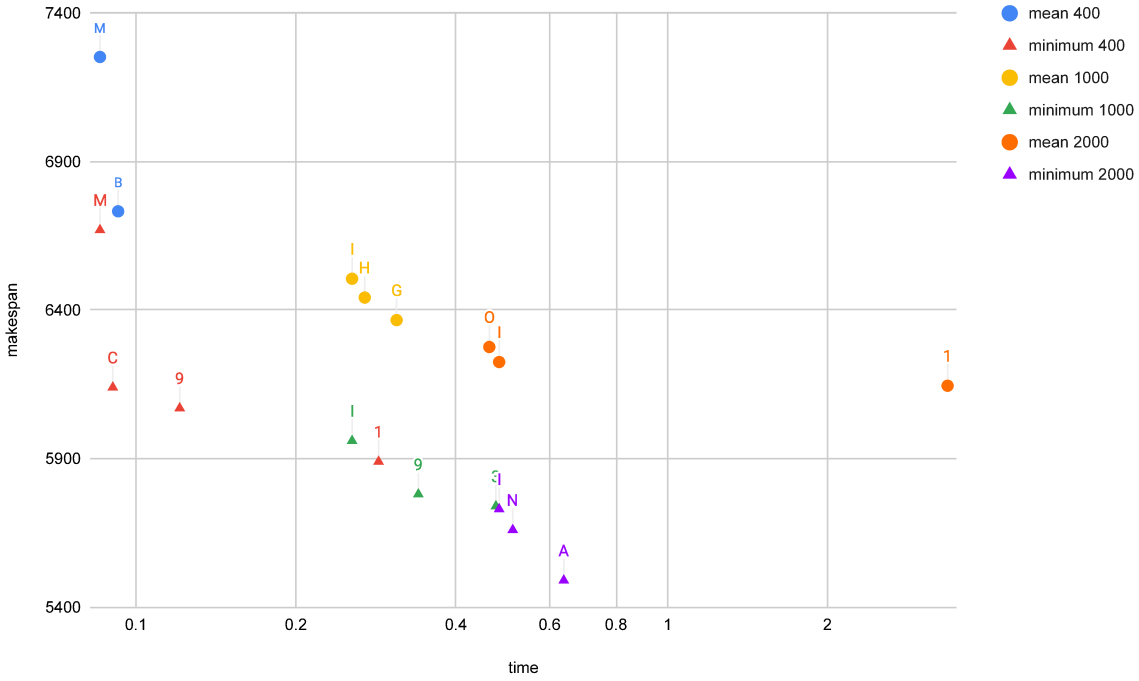

Figure 35 – Non-dominated data points selected from the highly efficient configurations from step 2 for the graph lap100 and the SM-MPGA with population sizes 400, 1000, 2000.

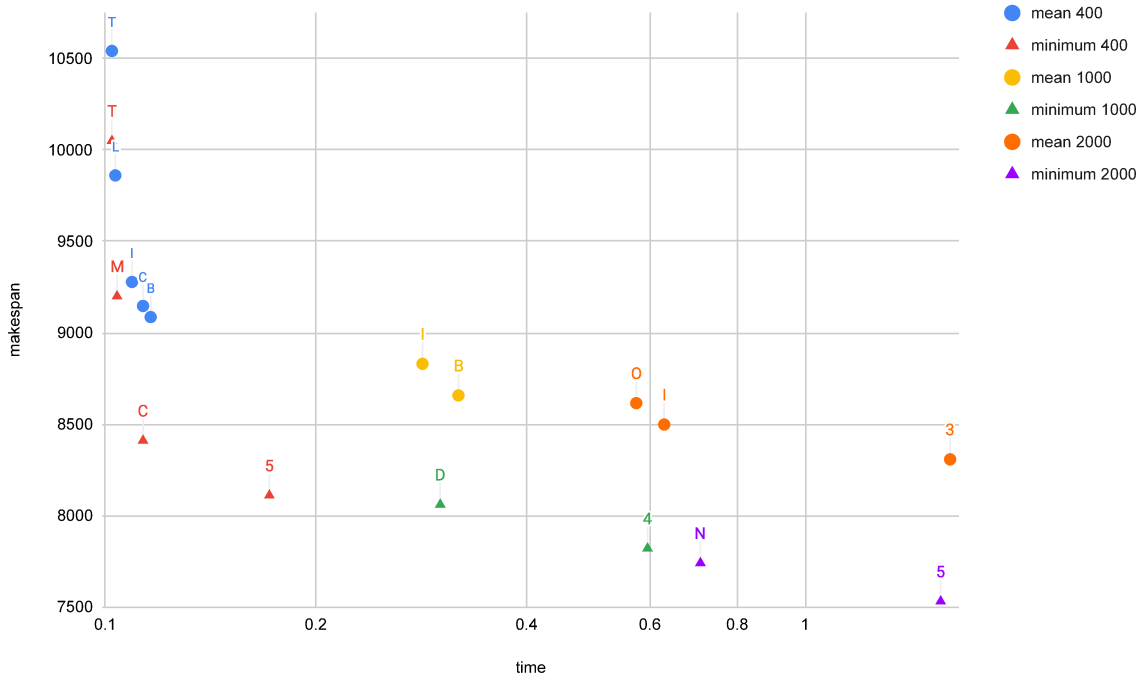

Figure 36 – Non-dominated data points selected from the highly efficient configurations from step 2 for the graph lap144 and the SM-MPGA with population sizes 400, 1000, 2000.

APPENDIX C

Preliminary experiment with parallel processing of multiple blocks

This appendix contains results of a preliminary experiment with parallel execution of the HCA forward evolution for text composed of multiple blocks. The text for this experiment has total length of 1Mb (2^{20} bits), and the block size for the experiment is varied between 128, 256, 512, 1024 bits. On each execution, the forward evolution is applied for a number of steps equal to the block size. With these parameters, each variation of the forward evolution program is described as follows.

The *serial* program is equivalent to the optimized serial implementation described on section 4.2.1. With this program, one block is processed at a time, while each cell of the CA is also processed in serial.

The *par1* program is a modification of the aforementioned *serial* program where multiple blocks are processed in parallel with Numba's prange function, while each cell of the CA is still processed in serial. Thus, multiple CA are processed in parallel, while the cells of each CA are processed sequentially.

The *par2* program is a modification of the *par1* program where Numba's prange function is applied to the CA as well. As such, it is equivalent to the parallel implementation on CPU described on section 4.2.1, with the addition of the parallel processing of multiple blocks. Thus, the program is set to process multiple CA in parallel, while the cells of each CA are also set for parallel processing.

The *GPU1* program is equivalent to the first CUDA implementation described on Listing 4.2. With this program, one block is processed at a time, while each cell of the CA is processed in parallel by one CUDA thread.

The *GPU2* program is a modification of the *GPU1* implementation where each text block is assigned to a CUDA block for parallel execution. Thus, the program is set to process multiple CA in parallel, while the cells of each CA are also set for parallel processing. Notably, since every CA is independent from each other, they can be executed in parallel by different CUDA blocks without synchronization between them, thus exceeding

the limitations on thread hierarchy described on section 4.2.1.

The results of this experiment are shown on Figure 37. Regarding block size, the CPU programs generally achieved slightly lower execution time with larger block sizes, while the opposite is true for the GPU programs. The par1 and par2 programs had approximately equivalent performance, with both achieving between 2.78X to 3.19X speedup for the executions with block size 128 to 1024. The GPU1 program had 5.44X speedup for block size 128, decreasing on each increment down to 3.96X at block size 1024. Similarly, the GPU2 program achieved 58.45X speedup at 128 block size, decreasing until 25.62X at 1024 block size.

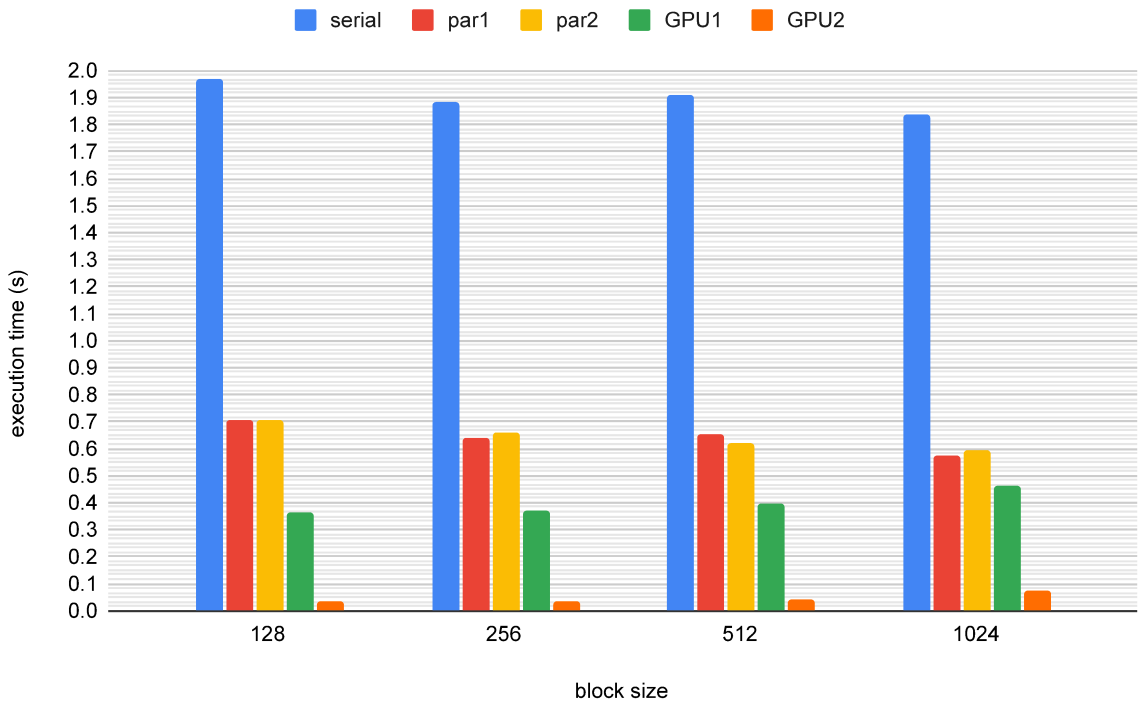

Figure 37 – Execution time (seconds) of the experiment with each variation of the forward evolution program for text with length 1Mb, varying block size, and number of steps equal to block size.

Thus, the results indicate that significant performance increases can be achieved on the CPU and GPU implementations by processing multiple text blocks in parallel.

APPENDIX **D**

Published research paper

Revista de Informática Teórica e Aplicada - RITA - ISSN 2175-2745
Vol. 26, Num. 1 (2019) 11-25

RESEARCH ARTICLE

Evolutionary Models applied to Multiprocessor Task Scheduling: Serial and Multipopulation Genetic Algorithm

Modelos Evolutivos aplicados ao Escalonamento de Tarefas em Sistemas Multiprocessados: Algoritmo Genético Serial e Multipopulação

Bruno W. Dantas Morais, Gina M. Barbosa de Oliveira, Tiago Ismailer de Carvalho

Resumo: This work presents the development of a multipopulation genetic algorithm for the task scheduling problem with communication costs, aiming to compare its performance with the serial genetic algorithm. For this purpose, a set of instances was developed and different approaches for genetic operations were compared. Experiments were conducted varying the number of populations and the number of processors available for scheduling. Solution quality and execution time were analyzed, and results show that the AGMP with adjusted parameters generally produces better solutions while requiring less execution time.
Keywords: multipopulation genetic algorithm — multiprocessor task scheduling

Resumo: Neste trabalho foi desenvolvido um Algoritmo Genético multipopulação para o problema de escalonamento de tarefas com custos de comunicação, com o objetivo de comparar seu desempenho com o Algoritmo Genético serial. Para isto, um conjunto de instâncias do problema foi elaborado e abordagens de operações genéticas foram comparadas. Experimentos foram conduzidos com variação do parâmetro de número de populações e do número de processadores utilizados no escalonamento. Foram avaliados a qualidade das soluções produzidas e o tempo de execução, e concluiu-se que o AGMP com seus parâmetros ajustados em geral obtém soluções melhores demandando um menor tempo de execução.
Palavras-Chave: algoritmo genético multipopulação — escalonamento de tarefas em multiprocessadores

Faculdade de Computação, Universidade Federal de Uberlândia, Brasil
*__Corresponding author__: brunowelldm@gmail.com
DOI: https://doi.org/10.22456/2175-2745.82412 • **Received:** 30/04/2018 • **Accepted:** 11/01/2019

1. Introdução

A crescente utilização de sistemas multiprocessados e o desenvolvimento de programas paralelos e distribuídos implicam no aumento da relevância de métodos que buscam a eficiência em sua execução. Entre eles, o problema de escalonamento envolve alocar as subtarefas de um programa paralelo em cada processador de uma arquitetura multiprocessada, de modo a reduzir o tempo de processamento total. Porém, o espaço de busca relacionado ao problema do escalonamento geralmente apresenta uma alta cardinalidade, dificultando a obtenção da solução ótima. Assim, têm sido utilizadas heurísticas e aproximações para se obter soluções aceitáveis para esse problema. [1]

Algoritmo Genético (AG) é uma técnica bio-inspirada e baseada em processos estocásticos que é utilizada para busca e otimização em problemas que envolvem espaços de busca de alta cardinalidade, incluindo o escalonamento [2]. O AG mantém um conjunto de soluções que são evoluídas iterativamente, a fim de explorar melhor o espaço de busca. Isso é feito por meio de operações que geram novas soluções a partir das soluções já exploradas, e as integram no conjunto de maneira semelhante ao processo da seleção natural: boas soluções tem maior probabilidade de ”sobreviver”e ”reproduzir”. No AG serial, estas operações são executadas sequencialmente.

Neste trabalho foi desenvolvido um Algoritmo Genético Multipopulação (AGMP) para o problema de escalonamento estático de tarefas com custos de comunicação. Tal técnica estende o modelo do AG serial mantendo subconjuntos de soluções que são evoluídos em processos paralelos de maneira semi-independente, com “migrações” periódicas. Essa abordagem é útil por explorar o paralelismo de sistemas multiprocessados, demandando um menor tempo de processamento sem que haja degradação das soluções encontradas [3].

O objetivo deste trabalho é comparar os resultados dos dois algoritmos em termos de qualidade das soluções obtidas e do tempo de execução. É esperado que ambos produzam soluções satisfatórias e que o AGMP apresente um ganho no tempo de processamento em relação ao AG serial, de acordo com o número de processos e processadores utilizados.

Este texto é organizado da seguinte forma. A seção 2 apresenta uma revisão de conceitos de escalonamento, AG, e AGMP. A seção 3 apresenta abordagens exploradas em trabalhos correlatos. A seção 3 contém descrições do conjunto de grafos de *benchmark* e do desenvolvimento e implementação do AG e do AGMP. A seção 5 descreve os parâmetros utilizados, as configurações avaliadas, o modelo de experimentos, e apresenta as comparações feitas entre o AG e o AGMP. A seção 6 descreve as conclusões e trabalhos futuros.

2. Revisão Bibliográfica

Esta seção descreve os principais conceitos utilizados e as abordagens de trabalhos correlatos.

2.1 Escalonamento de tarefas

Escalonamento de tarefas é um problema clássico de otimização combinatória considerado computacionalmente intratável [1]. Ele consiste no mapeamento de um conjunto de tarefas a um conjunto de processadores de modo a satisfazer um critério de avaliação. Assim, algoritmos baseados em heurísticas, aproximações e meta-heurísticas foram desenvolvidos para abordar o problema [4].

O problema tratado neste trabalho apresenta as seguintes propriedades [1].

- Escalonamento estático e determinístico: os tempos de execução, comunicação e as relações de precedência entre tarefas são conhecidos e não se alteram.
- Processadores não-preemptivos: um processador executa uma tarefa até que ela se encerre, sem que haja trocas ou interrupções.
- Os processadores são idênticos para fins de eficiência relativa entre processadores. Ou seja, todo processador leva x unidades de tempo para executar tarefas de custo x.
- Os processadores são totalmente conectados, i.e., se comunicam diretamente com cada um dos demais processadores.
- Um escalonamento não possui duplicação da execução de uma tarefa, i.e., cada tarefa é executada exatamente uma vez por um único processador.

Uma instância do problema de escalonamento é representada por uma tupla (G,P), em que P é o conjunto de processadores que executarão as tarefas, $G = \{V,E\}$ é um grafo direcionado acíclico cujos vértices de V representam um conjunto de tarefas, e arestas de E representam relações de precedência entre as tarefas. Para cada tarefa $T_i \in V$ existe um peso w_i que indica o tempo de processamento necessário para executar a tarefa correspondente. Para cada aresta $e_{ij} \in E$ existe um custo c_{ij} que indica o tempo necessário de comunicação entre o término da execução de T_i e o começo da execução de T_j, no caso dessas tarefas serem executadas em processadores diferentes. Caso sejam executadas no mesmo processador, não há custo de comunicação. Na Figura 1A é apresentado o grafo de tarefas laplace9, que representa o algoritmo para resolução de equações de Laplace, com os custos de comunicação omitidos, uma vez que são iguais a 40 para todas as arestas.

Uma tarefa T_i só pode ser escalonada por um processador P_j após o término da execução de todas as tarefas que precedem T_i, levando em conta os custos de comunicação para predecessoras executadas em processadores diferentes de P_j. Por exemplo, considerando o grafo da Figura 1A, o início do processamento da tarefa $T4$ deve ser posterior ao término de ambas as tarefas $T1$ e $T2$.

Dado um conjunto de p processadores, um escalonamento pode ser representado por p sequências de tarefas, cada uma associada a um processador. Cada sequência é parcialmente ordenada, obedecendo as precedências do grafo. A sequência de tarefas alocada a um processador indica as tarefas que ele executará e a ordem.

Um processador fica ocioso caso a tarefa a ser executada dependa de um resultado de outra tarefa que ainda não foi terminada ou comunicada por outro processador.

Dada a representação de um escalonamento, é possível calcular uma linha do tempo de execução, com pontos de início e término de cada tarefa. Essa linha do tempo pode ser ilustrada por um diagrama de Gantt. A Figura 1B apresenta o diagrama de Gantt para uma possível solução de escalonamento do grafo laplace9 em uma arquitetura de 4 processadores, e que demanda um tempo total de execução igual a 540.

O tempo do escalonamento ou *makespan* corresponde ao ponto da linha do tempo em que a execução de todas as tarefas foi terminada. Isto corresponde à duração total do escalonamento. Um escalonamento ótimo é aquele que apresenta um *makespan* mínimo.

2.2 Algoritmo Genético

Algoritmo Genético (AG) é uma meta-heurística proposta por John Holland que se inspira na evolução das espécies e tem sido usada para tratar diversos tipos de problemas relacionados a busca e otimização, incluindo o problema de escalonamento de tarefas [2].

Um AG opera de maneira estocástica para evoluir um conjunto de soluções com objetivo de encontrar uma solução globalmente satisfatória no espaço de busca de um problema. Para isso, deve ser definida uma forma de representação de uma solução e uma função para avaliação da mesma. Depois é escolhida um modo de inicializar uma população de soluções como ponto inicial da busca, os operadores que modificarão tal população iterativamente, e parâmetros numéricos que controlam seu comportamento. Tais componentes são detalhados a seguir.

Representação genética – Dado um problema a ser resolvido pelo Algoritmo Genético, deve ser definida uma estrutura de dados que codifique uma solução pertencente ao espaço de busca do problema. Uma instância dessa estrutura de dados

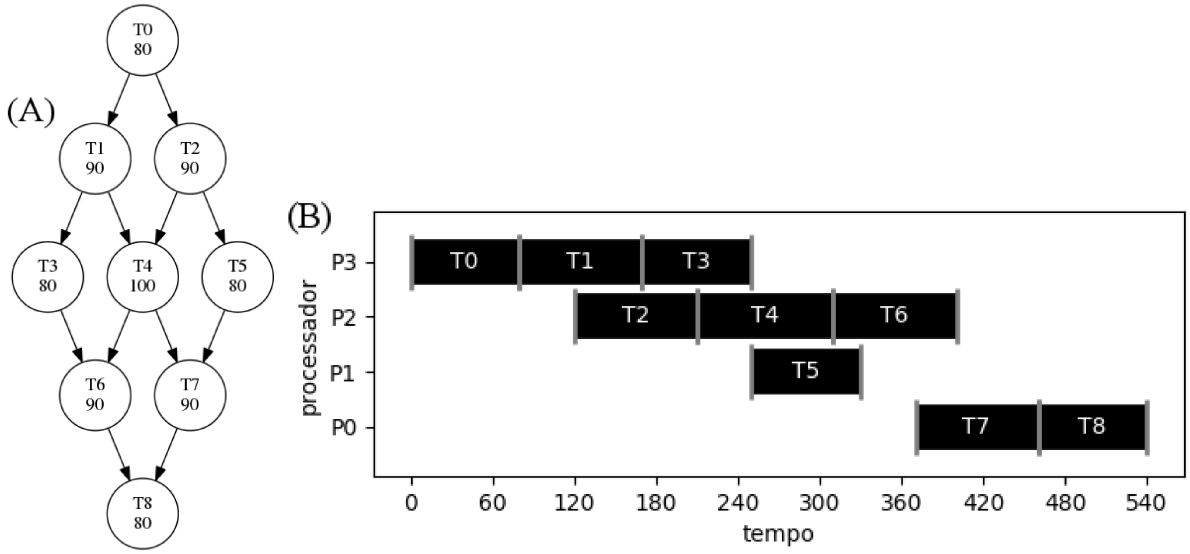

Figura 1. **(A)**: grafo laplace9 com custos de comunicação omitidos, que são iguais a 40 para todas as arestas. **(B)**: diagrama de Gantt que representa um possível escalonamento para 4 processadores com makespan de 540 unidades de tempo.

constitui um indivíduo (ou "cromossomo") que fará parte da população do AG. A representação genética clássica é um vetor binário. [2].

Fitness – Uma função de *fitness* (aptidão) é escolhida para a avaliação da qualidade de um indivíduo. Em geral, todo indivíduo explorado pelo AG é avaliado.

População inicial – Gerar uma população inicial de indivíduos constitui a etapa de inicialização do AG. Dado um parâmetro N_{pop} que indica o tamanho da população, são gerados N_{pop} indivíduos, de forma estocástica, que representam soluções válidas e compõem a população a ser evoluída pelo algoritmo. Geralmente, a população de um AG é uma lista de indivíduos ordenada pelo *fitness*.

Seleção – A cada iteração (ou geração) do AG, pares de indivíduos são selecionados da população para serem recombinados, gerando novas soluções (filhos). Em geral, indivíduos são selecionados de acordo com seu *fitness*, tal que indivíduos melhores terão maior probabilidade de se reproduzir. O número de pares a serem selecionados é obtido a partir da taxa de *crossover* (T_{cross}), que é dada como parâmetro do AG. O método clássico para seleção é a roleta, que dá a cada indivíduo uma probabilidade de ser selecionado que é diretamente proporcional ao seu *fitness* relativo aos demais indivíduos da população. [2]

Crossover – O operador de *crossover* é o método para gerar novos indivíduos a partir de um par de indivíduos pais. Seu objetivo é que os filhos possuam características de ambos os pais combinando suas representações. Por exemplo, *crossover* de um ou mais pontos consiste em dividir dois indivíduos (representados por vetores) em várias secções delimitadas por pontos que são alternadamente permutadas para gerar dois filhos. [2]

Mutação – Após o processo de *crossover*, alguns dos indivíduos filhos podem sofrer uma modificação pelo operador de mutação. A probabilidade de haver mutação em um filho definida pela taxa de mutação (T_{mut}), a qual é dada como parâmetro do AG. Quando necessário, correções e adaptações são feitas para que o novo indivíduo represente uma solução válida, pertencente ao espaço de busca. Para a representação genética clássica de vetor binário, o método de mutação consiste em negar um de seus bits. [2]

Reinserção – Dada uma lista de indivíduos pais e filhos, a operação de reinserção produz uma lista de indivíduos que constituirá a população na iteração seguinte combinando a população atual e a lista de filhos. [2]

2.3 Algoritmo Genético Multipopulação

Um Algoritmo Genético paralelo é um modelo de AG para execução em sistemas multiprocessados visando obter um tempo de processamento menor em relação a execução serial. Segundo Cantú-Paz[5], os vários tipos de AGs paralelos podem ser categorizados em:

- Paralelização global ou mestre-escravo: utiliza uma população única controlada por um programa mestre enquanto avaliações e/ou operadores genéticos são feitas por programas escravos.
- Paralelização de fina granularidade: voltada para sistemas massivamente paralelos. Utiliza uma população única cujos indivíduos competem e reproduzem apenas com uma vizinhança limitada.
- Paralelização multipopulação: utiliza várias populações semi-isoladas que se comunicam com uma certa frequência.

Alba e Troya[3] consideram AGs paralelos como uma classe distinta de meta-heurísticas por apresentarem a característica da interação local em subgrupos da população (com exceção da abordagem de paralelismo global), o que causa um impacto na qualidade das soluções. São citados casos em que AGs paralelos encontram soluções melhores do que as de AGs seriais. Os autores argumentam que a abordagem paralela é benéfica mesmo em sistemas que não possuam múltiplos processadores.

Algoritmo Genético multipopulação é a forma mais popular de AG paralelo [5]. Um AGMP pode ser encarado como uma extensão da implementação de AG, em que vários AGs independentes são executados, trocando alguns indivíduos entre si ocasionalmente. Assim, é pode-se converter um AG serial em AGMP adicionando o procedimento de migração, e é possível executá-lo em vários tipos de sistemas diferentes, seja uma máquina com um número qualquer de processadores ou uma rede de computadores [5].

AGMP é também conhecido como: AG distribuído, por sua abordagem relacionada a paralelismo MIMD (múltiplas instruções, múltiplos dados); AG paralelo de grossa granularidade, por sua alta taxa de computação por comunicação; e AG baseado em ilhas, devido a sua semelhança com o modelo de ilhas em genética populacional. O AGMP se caracteriza por usar poucas populações de tamanho relativamente grande, em contraste com as pequenas e numerosas vizinhanças mantidas na abordagem de fina granularidade [5].

O funcionamento do AGMP também requer parâmetros adicionais como número de populações, frequência de migração (F_{mig}) e taxa de migração (T_{mig}). A seguir, são descritos os conceitos adicionais que caracterizem o AGMP.

Topologia de comunicação – Estabelece as ligações entre populações, definindo as populações de destino a serem consideradas por cada população durante migrações. Topologias são geralmente especificadas *a priori* e imutáveis ao longo da execução. [5]

Frequência de migração – Define a periodicidade de trocas de indivíduos entre populações. Migrações podem ocorrer de modo síncrono num intervalo predeterminado de gerações, ou assíncrono sob satisfação de uma condição [5]. Para o caso especial em que a frequência de migração é igual a zero, o AGMP é equivalente a uma sequência de execuções independentes de AG serial com população reduzida. Tal abordagem é desencorajada por CantÚ-Paz et al.[6] por, geralmente, produzir soluções de baixa qualidade.

Seleção de migrantes – O critério de escolha de quais indivíduos serão removidos de sua população de origem para serem inseridos em outra. Em geral, imigrantes substituem indivíduos emigrantes ou os de pior *fitness*. [5]

3. Trabalhos correlatos

Trabalhos correlatos podem ser divididos entre aplicações de AG para o problema do escalonamento de tarefas e abordagens para desenvolvimento de AGMP de modo geral.

3.1 Algoritmo Genético aplicado a escalonamento de tarefas

A maior parte dos trabalhos consultados formula suas representações considerando a ordenação topológica do grafo direcionado acíclico dado pelo problema de escalonamento, o que permite a disposição dos vértices em uma lista ordenada pela precedência. Assim, para toda aresta e_{ij}, a posição da tarefa T_i na lista é anterior à posição de T_j.

A representação genética de Hou, Ansari e Ren[1], Correa, Ferreira e Rebreyend[7], e Kaur et al.[8] mantém, para cada processador, uma lista de tarefas ordenada pela precedência. Wang et al.[9], Omara e Arafa[10], Chitra, Rajaram e Venkatesh[11], e Morady e Dal[12] empregam dois vetores de inteiros: um representa uma ordenação topológica de tarefas e o outro representa o processador correspondente a cada tarefa. Kwok e Ahmad[13] utilizam uma única lista de tarefas ordenada por precedência, cujos processadores são atribuídos durante a computação do *fitness*. Hwang, Gen e Katayama[14] codificam a sequência de tarefas usando uma lista de prioridades numéricas, na qual os processadores associados correspondem ao valor da prioridade módulo o número de processadores. Xu et al.[15] fazem uso de uma fila de prioridade que indica a ordem de execução da tarefas.

Como função de *fitness*, Hou, Ansari e Ren[1], Wang et al.[9], Correa, Ferreira e Rebreyend[7], Hwang, Gen e Katayama[14], Omara e Arafa[10], Morady e Dal[12] avaliam o *makespan*. Kwok e Ahmad[13] usam o *makespan* normalizado entre 0 e 1. Kaur et al.[8] consideram o *makespan* e o "*flowtime*", que foi definido como soma dos tempos de término de cada tarefa. Chitra, Rajaram e Venkatesh[11] fazem uma soma ponderada entre *makespan* e confiabilidade, obtida ao contabilizar possíveis cenários de falha de processadores. Xu et al.[15] calculam o *fitness* subtraindo o *makespan* de um indivíduo do maior *makespan* presente na população.

As populações iniciais de Hou, Ansari e Ren[1], Correa, Ferreira e Rebreyend[7], Hwang, Gen e Katayama[14], Chitra, Rajaram e Venkatesh[11], Morady e Dal[12] são formadas por indivíduos gerados aleatoriamente. Wang et al.[9], Kwok e Ahmad[13], Kaur et al.[8], Omara e Arafa[10], Xu et al.[15] geram uma parte da população aleatoriamente e obtém a outra por heurísticas de ranqueamento de prioridade das tarefas.

Na literatura existem vários métodos para seleção. Hou, Ansari e Ren[1], Wang et al.[9], Hwang, Gen e Katayama[14], Kaur et al.[8], Chitra, Rajaram e Venkatesh[11], Xu et al.[15] utilizam o método da roleta, que associa a cada indivíduo uma probabilidade de ser selecionado diretamente relacionada a seu *fitness*. Kwok e Ahmad[13] avaliam o *fitness* de um indivíduo em relação ao *fitness* médio da população para definir o número de vezes que cada indivíduo se reproduzirá. Omara e Arafa[10] utilizam o método do torneio binário em que o melhor entre dois indivíduos selecionados aleatoriamente é escolhido. Morady e Dal[12] fazem parte da seleção com torneio e parte com a escolha direta de uma porção dos melhores indivíduos da população.

Para *crossover*, Wang et al.[9], Kwok e Ahmad[13], Chi-

tra, Rajaram e Venkatesh[11], Kaur et al.[8], Omara e Arafa[10], Xu et al.[15] se baseiam em *crossover* de um ponto, com estratégias para reordenar ou recombinar metade do cromossomo conforme necessário. Hou, Ansari e Ren[1] escolhem pontos de *crossover* durante iterações sobre as listas de tarefas de cada processador. Correa, Ferreira e Rebreyend[7] aplicam *crossover* uniforme: mantém-se as tarefas que coincidem nos pais e sorteia-se uma das alternativas caso contrário. Hwang, Gen e Katayama[14] desenvolvem um método baseado em *crossover* de dois pontos. Morady e Dal[12] combinam *crossover* de dois pontos e PMX, um algoritmo utilizado para vetores que envolvem permutação, que mantém uma subsequência do vetor inalterada e elimina elementos repetidos.

Swap é o operador de mutação mais usado na literatura [1, 9, 4, 14, 11, 15], o qual consiste em trocar duas tarefas de lugar na sequência de execução, podendo implicar uma mudança de processador. O operador implementado por Omara e Arafa[10] muda o processador em que uma tarefa é executada. Correa, Ferreira e Rebreyend[7] fazem a remoção de uma tarefa e sua reinserção num ponto diferente. Kaur et al.[8] e Morady e Dal[12] usam duas operações: *swap* sem mudança de processador ou apenas mudança de processador.

A reinserção nos trabalhos de Hou, Ansari e Ren[1], Wang et al.[9], Kwok e Ahmad[13], Correa, Ferreira e Rebreyend[7], Hwang, Gen e Katayama[14], Kaur et al.[8], Chitra, Rajaram e Venkatesh[11], Xu et al.[15], e Morady e Dal[12] é feita por elitismo: uma porção dos melhores indivíduos permanece na população e o restante é preenchido pelos filhos gerados.

3.2 Abordagens em AGMP

Os trabalhos [13, 12] apresentam aplicações de AGMP para escalonamento de tarefas. Os demais trabalhos consultados usam o AGMP para outras aplicações: escalonamento job-shop [16], *data mining* [17], carregamento de contêiners [18], otimização de funções [19, 20], e otimização combinatória [21].

A respeito de topologia de comunicação, as populações de Kwok e Ahmad[13], Qi, Burns e Harrison[16], e Srinivasa, Venugopal e Patnaik[17] são totalmente conectadas. Gehring e Bortfeldt[18] e Morady e Dal[12] usam topologia em anel, na qual cada população possui uma única população de destino, descrevendo um anel conectado. MÜhlenbein, Schomisch e Born[19] dispõem uma estrutura de "escada circular", com múltiplas conexões em cada população. Han et al.[21] aplicam, uma estrutura de agrupamento voltada para a redução da carga de comunicação entre processadores. Yao, Kharma e Grogono[20] utilizam uma topologia dinâmica em que populações podem se unir ou dividir durante a execução.

A frequência de migração é um parâmetro fixo nos trabalhos [19, 16, 21, 18, 17, 12], enquanto Kwok e Ahmad[13] empregam uma frequência exponencialmente crescente ao longo da execução.

Em [19, 13, 16, 21, 18, 17, 12], os melhores indivíduos são selecionados para migração. Migrações são determinadas por meio de uma análise de agrupamento de indivíduos em [20].

4. Desenvolvimento

O trabalho desenvolvido consistiu em quatro etapas: selecionar um conjunto de grafos para *benchmark*, a ser utilizado nas avaliações comparativas dos algoritmos implementados; implementar o AG e suas diferentes configurações de *crossover*; implementar o AGMP; comparar as melhores configurações do AG e do AGMP.

4.1 Grafos para *benchmark*

Para fins de comparação, foi elaborado um conjunto de grafos referentes ao problema de escalonamento com custos de comunicação em algoritmos reais.

Compondo esse conjunto, foram utilizados grafos obtidos diretamente de outros trabalhos e também famílias de grafos baseadas em instâncias usadas na literatura.

4.1.1 Grafos relacionados a problemas paralelos reais

Com o objetivo de incrementar o conjunto com grafos relevantes e com espaços de busca de alta cardinalidade, foram implementados três scripts em linguagem Python que geram grafos pertencentes a "famílias" correspondentes a algoritmos paralelos mencionados na literatura: eliminação gaussiana, algoritmo de Laplace e transformada rápida de Fourier [22]. Assim, um grafo pertencente a uma dessas famílias corresponde à execução do algoritmo para um dado tamanho de entrada. A metodologia adotada na geração desses grafos foi inspirada no trabalho descrito em [22].

Os scripts implementados geram representações de grafos em arquivos de texto e também suas visualizações com a ferramenta Graphviz. Quatro grafos de cada família foram selecionados para compor o conjunto de *benchmark*, num total de 12 grafos baseados em programas paralelos reais.

A família "gauss" é definida pelo algoritmo da Decomposição LU, uma forma matricial do método da eliminação gaussiana. Esse algoritmo que realiza fatoração de matrizes quadradas em matrizes triangulares, o que se reflete na estrutura desses grafos. Seu número de tarefas depende diretamente do tamanho da matriz de entrada, e seus pesos e custos de comunicação obedecem um padrão regular [23]. A Figura 2 mostra duas instâncias desta família, nas quais as arestas contínuas representam custo de comunicação igual a 12 e as arestas tracejadas, custo igual a 8 [24].

A família "laplace" é dada pelo algoritmo para resolução de equações de Laplace, composta por uma estrutura de tarefas que reflete diretamente o seu tamanho de entrada, que é uma matriz quadrada. Nesta família, ilustrada na Figura 3, as dependências entre tarefas são dadas pela maneira em que a matriz é percorrida no algoritmo [25]. Nesta família, os pesos das tarefas obedecem um padrão regular e todos os custos de comunicação são iguais a 40. Os pesos de cada tarefa correspondem à posição do elemento correspondente na matriz, sendo 80 para as tarefas nas 4 extremidades do grafo (acima, abaixo, à direita e à esquerda), 90 para as tarefas nas laterais, e 100 para as tarefas mais internas.

A família "fft" é baseada no algoritmo de Cooley–Tukey

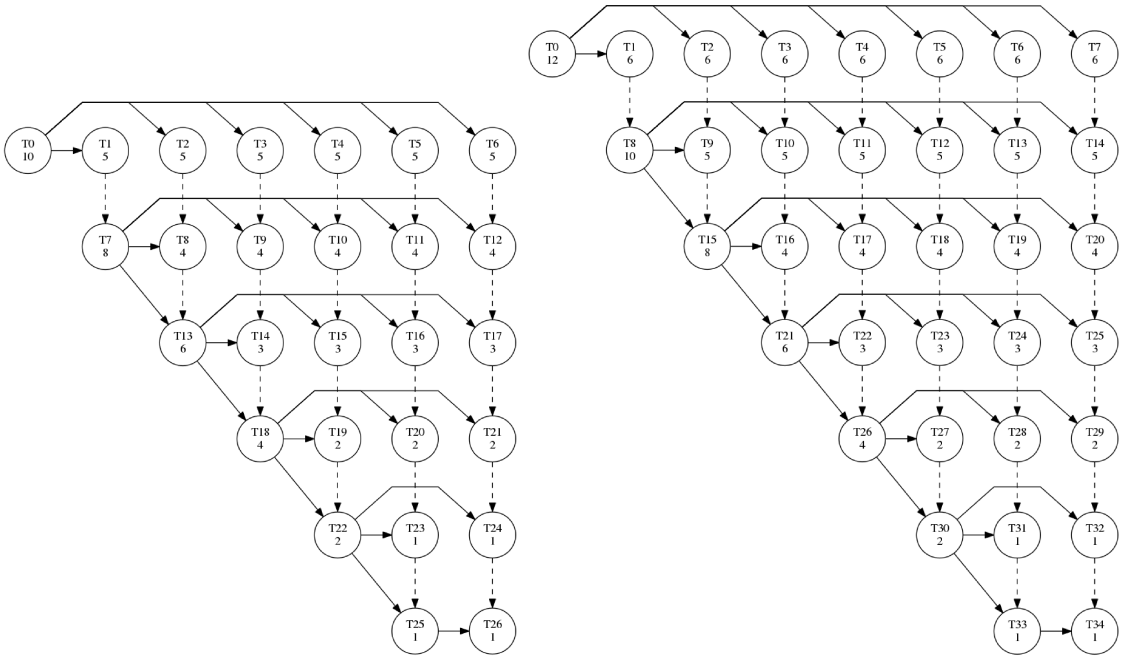

Figura 2. Grafos gauss27 e gauss35. Arestas contínuas representam custo de comunicação igual a 12 e arestas tracejadas, custo igual a 8. Fonte: elaborada pelos autores.

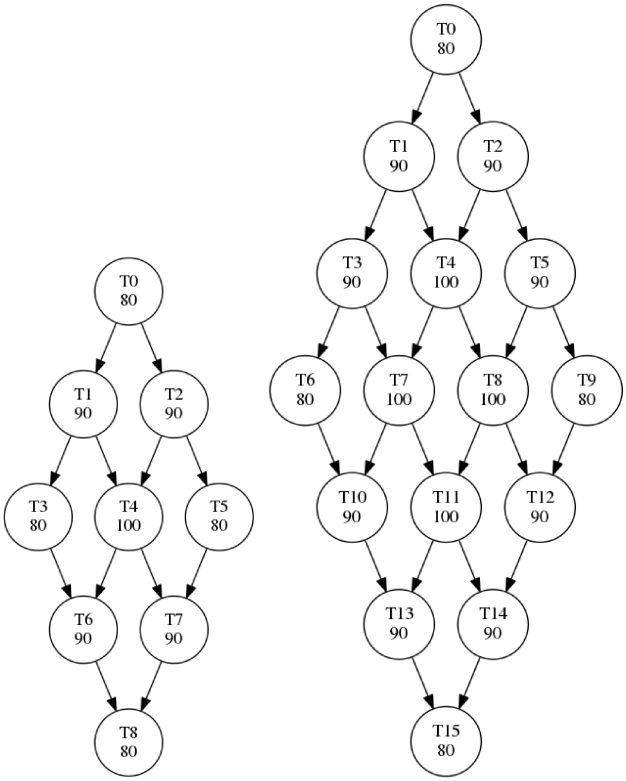

Figura 3. Grafos laplace9 e laplace16. Todos os custos de comunicação nesta família são iguais a 40. Fonte: elaborada pelos autores.

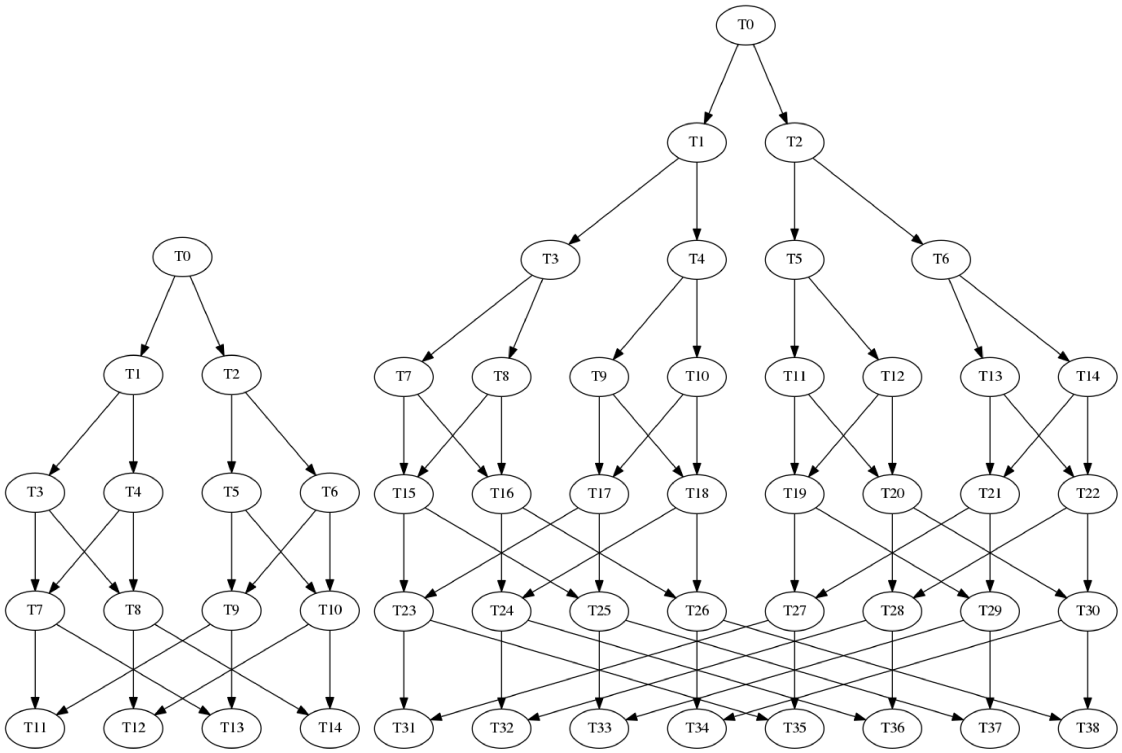

Figura 4. Grafos fft15 e fft39. Nesta família, os vértices possuem pesos iguais a 60 e os custos de comunicação são iguais a 80. Fonte: elaborada pelos autores.

para a transformada rápida de Fourier. Esta família de grafos assume como entrada uma lista de tamanho igual a uma potência de 2. Seus grafos assumem uma estrutura com uma etapa recursiva e uma etapa de operação "borboleta" característica do algoritmo [26]. Duas instâncias são mostradas na Figura 4. Os pesos e custos de comunicação nesta família são iguais a 60 e 80, respectivamente.

4.1.2 Outros grafos

Em [12] foram disponibilizados onze grafos com número de vértices entre 4 e 18, obtidos de diversos trabalhos da literatura. Aqui, eles são referenciados com prefixo "TG". Os autores não mencionam correspondência entre esses grafos e algoritmos paralelos reais.

Em [27] foi utilizada uma ferramenta para criação de três grafos aleatórios com 30, 40 e 50 vértices, identificados pelo prefixo "random".

No total, o conjunto do *benchmark* é composto por 26 grafos, sendo 12 gerados a partir de informações de programas paralelos reais e 14 extraídos da literatura sem conexão com problemas reais.

4.2 Algoritmo Genético Serial

O primeiro algoritmo evolutivo construído foi um AG serial para o problema do escalonamento de tarefas. Nessa etapa, algumas configurações relevantes do AG foram definidas e comparadas a fim de obter uma boa opção de operadores e parâmetros. O propósito deste trabalho foi explorar abordagens de AG "puro", em contraste com a tendência recente de desenvolver AGs híbridos incorporando diversas técnicas e heurísticas específicas do problema [8, 9, 10, 15, 13].

A representação do indivíduo utilizada é a mesma de [9], [10], [11], e [12]: dois vetores de inteiros S, P tal que S é uma sequência de tarefas que obedece às restrições de ordenação topológica do grafo dado e P é um vetor que mapeia os vínculos tarefa/processador de forma que $P[tarefa] = processador$.

A Figura 5B ilustra um indivíduo válido para o grafo laplace9 (Figura 5A), considerando-se alocação para 4 processadores. Para representar seu escalonamento correspondente, ilustrado na Figura 5C, o vetor $S = [0,2,5,1,3,4,6,7,8]$ é particionado em uma sequência para cada processador, obedecendo as associações dadas pelo vetor $P = [3,3,2,3,2,1,2,0,0]$. Por exemplo, as tarefas executadas pelo processador P3 são T0, T1 e T3, pois $P[0] = P[1] = P[3] = 3$. O processamento de cada tarefa é iniciado após a conclusão de todas as suas predecessoras, somado ao custo de comunicação em caso de serem executadas em processadores diferentes. Por exemplo, a tarefa T5 é iniciada após a conclusão e comunicação de T2. Com isso, T5 é executada após o início de T4, embora T5 seja anterior a T4 no vetor S.

A função de *fitness* utilizada é dada pelo *makespan* associado ao escalonamento. Por exemplo, o indivíduo representado na Figura 5B tem *fitness* igual a 540, que corresponde ao

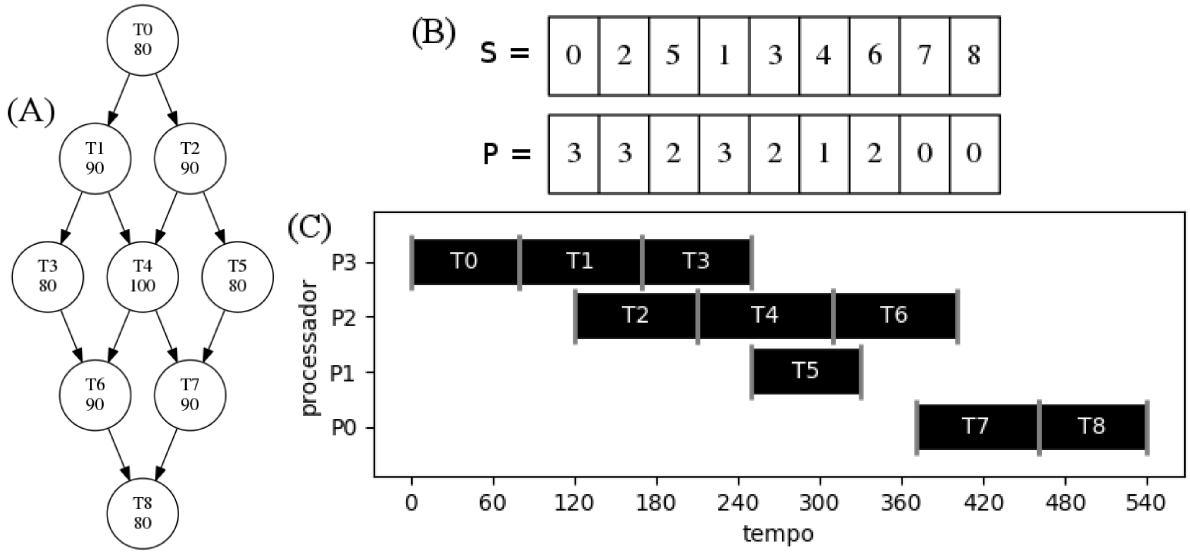

Figura 5. **(A)**: grafo laplace9 com custos de comunicação omitidos, que são iguais a 40 para todas as arestas. **(B)**: exemplo de indivíduo para o grafo laplace9 e 4 processadores. **(C)**: diagrama de Gantt que representa o escalonamento correspondente ao indivíduo de (B), com *makespan* de 540 unidades de tempo.

tempo de finalização do processador P0, o mais tardio na Figura 5C. A população inicial é gerada aleatoriamente com indivíduos válidos: vetor S sem repetição das tarefas e respeitando as restrições de ordenação topológica. A seleção dos pais para o *crossover* é feita com torneio binário. A reinserção da população no final de cada geração se dá por elitismo. A mutação é uma operação pontual em um dos dois vetores do indivíduo, mudando o processador de uma tarefa ou trocando duas tarefas de lugar na sequência caso não haja dependência entre elas. Essas duas operações de mutação são referidas aqui como mutação "proc" e mutação "seq", respectivamente.

Como a representação adotada é formada por duas partes distintas (S e P), existe uma variedade de opções diferentes para a aplicação do operador de *crossover*. Três abordagens foram observadas na bibliografia, aqui nomeadas como *carry*, *choose* e *combine*:

- *Carry*: o *crossover* é aplicado no vetor de tarefas "carregando" o vínculo de cada tarefa com o processador definido no pai correspondente. Assim, tarefas do indivíduo filho vindas do pai1 possuem o mesmo processador do pai1. Do mesmo modo, tarefas vindas do pai2 são executadas nos processadores definidos no pai2. Esta abordagem é análoga a métodos usados em trabalhos com representação de listas de tarefas por processador [1], [7], [8].

- *Choose*: é definida uma operação de *crossover* para o vetor de tarefas e uma para o vetor de processadores. Durante a reprodução de indivíduos, um dos dois tipos de *crossover* é sorteado e aplicado. O outro vetor é copiado do primeiro pai para o primeiro filho e vice-versa [10].

- *Combine*: é definido um *crossover* para o vetor de tarefas e um para o vetor de processadores. Durante a reprodução de indivíduos, ambos *crossovers* são aplicados [12].

Para avaliar estes métodos, foram implementados os seguintes algoritmos clássicos de *crossover*, que foram adaptados para evitar duplicação de tarefas e obedecer as restrições do vetor de tarefas (S):

- 1-ponto: sorteia-se um ponto de corte na sequência. Copia-se a primeira parte do pai1 e a segunda parte é preenchida, sem repetição, com tarefas seguindo a ordem em que se elas estão dispostas no pai2 [10].

- 2-pontos: análogo ao anterior; a sequência é dividida em 3 partes. A primeira é uma cópia do pai1, a segunda é preenchida com tarefas na ordem do pai2, e a terceira é preenchida com tarefas restantes do pai1, também obedecendo sua ordem.

- Uniforme: a sequência filha é composta percorrendo os pais e escolhendo aleatoriamente de qual pai será herdada a tarefa seguinte, ignorando repetições.

- Cíclico: uma posição p é sorteada e os p-ésimos elementos das sequências são trocados. Enquanto houver uma duplicata na primeira sequência, o elemento repetido é trocado pelo elemento na mesma posição na outra sequência. É necessária validação posterior para garantir as restrições de precedência.

- PMX: mantém-se uma subsequência do pai1, troca-se o restante com elementos do pai2 e faz-se trocas até eliminar duplicatas. Também é necessária validação do filho.

- OX: mantém-se uma subsequência do pai1 e preenche-se o restante seguindo a ordem do pai2. A versão implementada gera um filho válido por garantir que todas as tarefas predecessoras da subsequência estejam à esquerda dela.

Alguns desses métodos de *crossover* são exemplificados para o grafo laplace9 na Figura 6, em que filhos $F1$, $F2$, $F3$ e $F4$ foram produzidos respectivamente por *crossover* 1-ponto, 2-pontos, uniforme, OX, a partir dos pais $P1$ e $P2$. Além deles, foram implementados *crossovers* 1-ponto, 2-pontos e uniforme para o vetor de processadores, que são análogos aos descritos e sem restrição sobre repetição e ordenação. A implementação foi feita em linguagem C e compilada e otimizada com GCC. Os resultados dos experimentos com o algoritmo evolutivo serial serão apresentados na Seção 5.

4.3 Algoritmo Genético Multipopulação

O AGMP desenvolvido nesse trabalho emprega topologia em anel unidirecional para comunicação entre as populações, que é um modelo de simples implementação e que produziu resultados de boa qualidade em trabalhos correlatos [5]. No anel unidirecional, toda população tem uma vizinha para onde envia alguns indivíduos, e uma segunda vizinha da qual recebe outros indivíduos em um processo conhecido por migração. Os melhores indivíduos da população são os selecionados para a migração. Os parâmetros de migração foram definidos experimentalmente: $T_{mig} = 10\%$ e $F_{mig} = 5$. Demais parâmetros são dados pela *config6* (Tabela 1), escolhida para o AG serial.

Note que a quantidade de avaliações de *fitness* realizadas pelo AGMP é igual à do AG, pois o número total de indivíduos é distribuído igualmente entre cada população. Por exemplo, para 400 indivíduos e 4 populações, o tamanho de cada uma é igual a 100.

A taxa de migração é relativa ao tamanho das populações. No exemplo de 4 populações com 100 indivíduos cada, os parâmetros de migração ditam que os 10 melhores indivíduos (10%) de cada população migrarão.

A implementação foi feita estendendo o AG com procedimentos de inicialização das populações e migração, que foram feitos em memória distribuída com a biblioteca OpenMPI, que implementa o padrão *Message Passing Interface* para execução paralela de programas com múltiplos processos. Desse modo, cada população é executada em um processo diferente que se comunica com populações vizinhas de acordo com a frequência de migração. Por fim, os melhores indivíduos de cada população são enviados para uma mesma população, e o melhor deles é retornado pelo programa. Os demais operadores (seleção, *crossover*, mutação e reinserção) foram implementados da mesma forma descrita para o AG Serial na seção anterior. Uma execução do AG serial é equivalente a executar o AGMP com número de populações igual a 1.

5. Experimentos e Resultados

Nesta seção são definidos os parâmetros utilizados nos algoritmos evolutivos e o método adotado nos experimentos. Em seguida, o AG e o AGMP são comparados em termos de qualidade das soluções produzidas e tempo de execução.

5.1 Definições iniciais de parâmetros e configurações

Para escolha do método de aplicação de *crossover*, foram elaboradas execuções do AG serial com configurações baseadas na literatura, apresentadas na Tabela 1, que contém duplas de operações de *crossover* para vetor de tarefas e de processadores em casos do tipo *choose* ou *combine* e apenas um tipo de *crossover* para o vetor de tarefas no caso *carry*.

Config0 é uma configuração sugerida nas fases iniciais deste trabalho a partir de experiências anteriores do grupo com o problema. *Config1* é baseada na descrição de Omara e Arafa[10]. *Config2* é igual à anterior, com a adição de mutação no vetor de tarefas (mutação "seq"). *Config3* é baseada em [7], com a representação genética de dois vetores aqui descrita em vez de listas de tarefas para cada processador. *Config4* é igual à anterior com seleção via torneio em vez de roleta. *Config12* segue o trabalho de Morady e Dal[12]. As restantes são algumas variações das configurações anteriores.

Tabela 1. Configurações de AG

config	seleção	método	*crossover*	mutação
0	torneio	*carry*	cíclico	seq/proc
1	torneio	*choose*	1-point, 1-point	proc
2	torneio	*choose*	1-point, 1-point	seq/proc
3	roleta	*carry*	uniforme	seq/proc
4	torneio	*carry*	uniforme	seq/proc
5	torneio	*choose*	2-point, 2-point	seq/proc
6	torneio	*combine*	1-point, 1-point	seq/proc
7	torneio	*combine*	uniforme, uniforme	seq/proc
8	torneio	*combine*	2-point, 2-point	seq/proc
9	torneio	*choose*	uniforme, uniforme	seq/proc
10	torneio	*carry*	pmx	seq/proc
11	torneio	*carry*	ox	seq/proc
12	torneio	*combine*	pmx, 2-point	seq/proc
13	torneio	*combine*	cíclico, 2-point	seq/proc
14	torneio	*combine*	ox, 2-point	seq/proc

O número de avaliações de *fitness* feitas em uma execução do AG representa o número aproximado de pontos explorados no espaço de busca. Esse número decorre dos parâmetros adotados no AG, e pode conter imprecisão devido à possibilidade da avaliação de um indivíduo repetido. Os parâmetros usados por Morady e Dal[12] foram tomados como ponto de partida neste trabalho: $N_{pop} = 200$, $N_{ger} = 400$, $T_{cross} = 65\%$ e $T_{mut} = 35\%$. Desse modo, os parâmetros iniciais totalizam 52.200 avaliações de *fitness*, de acordo com o cálculo a seguir, baseado na atual implementação:

$$avaliacoes = N_{pop} + N_{ger} * N_{pop} * T_{cross}$$

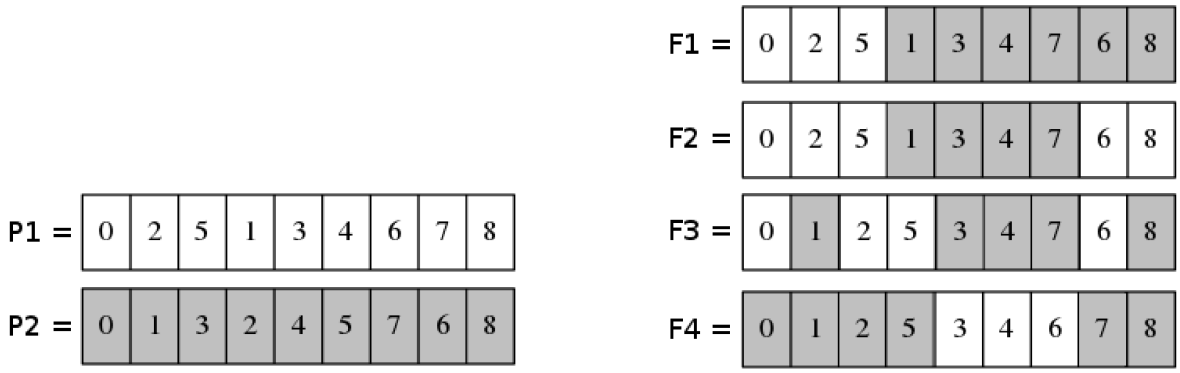

Figura 6. Exemplos de *crossover* sobre vetores de tarefa para o grafo laplace9, omitindo vetores de processadores (P) dos indivíduos. À esquerda, as tarefas dos indivíduos pais. À direita, tarefas de filhos gerados por *crossovers* 1-point ($F1$), 2-point ($F2$), uniforme ($F3$) e OX ($F4$).

$$avaliacoes = 200 + 400 * 200 * 0,65 = 52.200$$

Porém, foi observado que as configurações implementadas produziam resultados melhores se a população fosse maior em relação ao número de gerações (N_{ger}). Então, os parâmetros adotados foram: $N_{pop} = 400$, $N_{ger} = 198$, $T_{cross} = 65\%$ e $T_{mut} = 35\%$, totalizando 51.880 avaliações de *fitness*, ou seja, aproximadamente o mesmo número de avaliações adotado em [12].

5.2 Metodologia

Para avaliar a qualidade das soluções de cada configuração e levando-se em conta o comportamento estocástico dos AGs, os experimentos consistem em 100 execuções para cada grafo do conjunto de *benchmark* em cada uma das configurações da Tabela 1. Os dados avaliados incluem: convergência (número de vezes em que foi encontrada uma solução ótima); melhor *fitness* obtido; *fitness* médio; pior *fitness*.

Entretanto, as soluções ótimas não são conhecidas para a maioria dos grafos do *benchmark* elaborado, que foi descrito na Seção 4.1. Desse modo, os melhores *fitness* encontrados nos experimentos foram tomados como valores pseudo-ótimos, que são suficientes para realizar os experimentos comparativos. Estes valores são apresentados na Tabela 2.

Para analisar e comparar os resultados das configurações, foram usadas quatro métricas: média das convergências e média dos desvios do melhor, médio e pior *fitness* em relação ao *fitness* ótimo, tal que o desvio de um valor de *fitness* x é dado por [1]:

$$desvio(x) = \frac{x - otimo}{otimo}$$

Assim, uma boa configuração é a que obtém alta convergência e baixos desvios, o que significa que suas soluções são próximas da solução ótima.

5.3 Experimentos preliminares com o AG serial

Conforme relatado na seção anteriores, os experimentos com AGs no escalonamento de tarefas encontrados na literatura

Tabela 2. Valores pseudo-ótimos de *makespan* para 4 processadores.

grafo	*makespan*
fft15	560
fft39	820
fft95	1620
fft223	3540
gauss27	67
gauss35	91
gauss44	116
gauss65	175
laplace9	520
laplace16	760
laplace25	1000
laplace36	1290
random30	751
random40	551
random50	496
TG4	34
TG9	21
TG9b	16
TG10	83
TG11	20
TG11b	49
TG11c	12
TG12	27
TG14	35
TG17	37
TG18	440

variam sensivelmente em relação aos métodos de seleção, *crossover* e mutação empregados. Assim, os primeiros experimentos com o AG serial com parâmetros especificados para uma operação mais próxima de aplicações reais, ou seja, com $T_{pop} = 400$ e $N_{ger} = 198$, foram realizados com o objetivo de avaliar qual das configurações apresentadas na Tabela 1 seria a mais adequada para os algoritmos evolutivos. O número de processadores foi fixado em 4 nesses experimentos preliminares, sendo este um ponto de partida que deve possibilitar uma melhor generalização dos resultados para diferentes quantidades de processadores, se comparado com o caso mais trivial de 2 processadores.

A Figura 7 representa graficamente os resultados obtidos de acordo com cada métrica avaliada (convergência média para o ótimo estimado na Tabela 2 e os desvios médios do melhor, médio e pior *fitness*/*makespan*) para cada configuração. É possível verificar que *config6* obteve o melhor resultado em 3 das 4 métricas. Essa configuração utiliza torneio binário na seleção dos pais, método *combine* no *crossover*, com *crossover* 1-ponto em ambos os vetores, ambas mutações "seq" e "proc" e reinserção por elitismo. Além disso, mesmo na única métrica que não foi a melhor, a config6 retornou resultados próximos da melhor configuração (*config10*). Assim, concluise que *config6* produziu boas soluções de maneira mais consistente que as outras no caso geral. A configuração *config6* foi adotada nos experimentos com o AG serial que se seguiram e também com nos experimentos com o AGMP. Ou seja, os ambientes evolutivos empregam: (i) método de seleção por torneio binário, (ii) *crossover* com abordagem *combine* que realiza 2 recombinações independentes para cada vetor que compõe o indivíduo (processadores e tarefas), sendo que em cada vetor foi aplicado o *crossover* de 1 ponto e (iii) mutação mista que aplicam alterações pontuais tanto na sequência dentro de cada processador, quanto nos processadores onde as tarefas estão alocadas.

5.4 Experimentos comparativos entre o AG Serial e o AGMP

Após a realização dos experimentos preliminares com o AG serial, que nos permitiram estabelecer os principais parâmetros e a melhor configuração dos operadores de seleção, *crossover* e mutação, iniciamos os experimentos com o AGMP utilizando os resultados do AG serial como referência. Desse modo, a configuração *config6* foi também aplicada ao AGMP. Inicialmente, foram realizados experimentos para estabelecer qual o melhor valor para o parâmetro número de populações do AGMP. Além disso, foram realizadas comparações entre o AG serial e o AGMP em relação à qualidade das soluções encontradas e ao tempo de execução.

5.4.1 Análise da qualidade das soluções

Seguindo o modelo de experimentos definido para o AG serial, foram feitos testes comparativos variando-se o número de populações do AGMP de 1 a 10, sendo que as execuções com uma população são equivalentes ao AG serial.

Os resultados da Figura 8 mostram que a qualidade das soluções do AG serial foi superada em todas as métricas por pelo menos uma configuração do AGMP. A configuração do AGMP com 7 populações foi a melhor em convergência e obteve o desvio mais baixo de *fitness* médio, mas com um alto desvio de melhor *fitness*. Por outro lado, o AGMP com 8 populações obteve valores baixos e consistentes de desvio, possuindo o menor desvio de pior *fitness*, e alta convergência (a segunda melhor). Por isso, o AGMP com 8 populações foi escolhido como a melhor configuração, e adotada como padrão nos experimentos seguintes.

Em alguns dos experimentos, também nota-se que o AGMP encontrou soluções melhores do que as do AG com a configuração 6, como nos grafos fft39, fft95, laplace36 e random40, que são apresentados na Tabela 3.

Tabela 3. Melhor *fitness* encontrado pelo AG e pelo AGMP com número de populações indicado, para 4 processadores.

grafo	AG	AGMP	populações
fft39	840	820	8
fft95	1620	1600	4
laplace36	1300	1270	8
random40	555	551	6

A Tabela 4 mostra os resultados do teste de hipótese Wilcoxon Rank-Sum com significância de 95%, que compara as medianas dos resultados obtidos pelo AGMP de 8 populações e pelo AG. Cada amostra é composta por 1000 execuções. Foram considerados todos os grafos e, para avaliar se o desempenho relativo dos algoritmos se mantém para instâncias do problema com número diferente de processadores, foram consideradas as arquiteturas com 2, 4, 8 e 16 processadores, totalizando 104 cenários. Nota-se que o AGMP foi mais consistente para a minimização do *fitness* em 44 cenários, se destacando para os grafos da família gauss, laplace16, laplace25, TG14, TG17 e TG18. O AG teve os melhores resultados em 19 cenários, com destaque para os grafos fft39, fft95, fft223 e random40. De modo geral, também pode-se observar que o desempenho relativo dos algoritmos se mantém para as instâncias do problema com número diferente de processadores.

Ao observar que o AGMP superou ou se equiparou ao AG em 85 dos 104 cenários, consideramos que os resultados suportam a hipótese de que o AGMP produz soluções melhores ou equivalentes às do AG no caso geral.

5.4.2 Tempo de execução

Para avaliação de tempo de execução, foram usados os tempos gastos pelos experimentos da seção 5.4.1. Os algoritmos evolutivos, implementados em linguagem C e otimizados pelo GCC, foram executados em um laptop com sistema operacional Ubuntu 16.04 e processador Intel Core i7-6500U, que possui 4 núcleos e clock de 2,50GHz. Tal implementação proporcionou os tempos de execução médios dados na Tabela 5, de acordo com o número de populações. Assim, foi calculado o *Speedup* médio das configurações de AGMP, apresentado na Figura 9, que mensura os desempenhos relativos

Figura 7. Resultados agregados dos testes de configurações do AG para escalonamento de todos os grafos em 4 processadores. Os melhores resultados estão em destaque.

Figura 8. Resultados agregados dos experimentos variando número de populações do AGMP para escalonamento de todos os grafos em 4 processadores. Os melhores resultados estão em destaque.

Tabela 4. Testes de hipótese para as medianas dos resultados do AGMP de 8 populações e do AG. Os símbolos $<$, $=$ e $>$ indicam que a mediana do AGMP foi menor, igual ou maior que a do AG, respectivamente.

	processadores			
grafo	**2**	**4**	**8**	**16**
fft15	=	=	<	=
fft39	=	>	>	>
fft95	<	>	>	>
fft223	=	>	>	>
gauss27	<	<	<	<
gauss35	<	<	<	<
gauss44	<	<	<	<
gauss65	<	<	<	<
laplace9	=	=	=	=
laplace16	=	<	<	<
laplace25	=	<	<	<
laplace36	>	<	<	>
random30	>	>	<	=
random40	>	=	>	>
random50	=	<	<	=
TG4	=	=	=	=
TG9	=	=	<	=
TG9b	=	=	=	=
TG10	>	=	=	>
TG11	=	=	=	<
TG11b	=	=	=	=
TG11c	=	=	=	=
TG12	=	>	<	<
TG14	<	<	<	=
TG17	<	<	<	<
TG18	<	<	<	<

ao tempo de execução do AG serial. O gráfico tem pico em 4 populações e degrada em seguida, devido ao número de núcleos do processador utilizado, o que é previsto pela lei de Amdahl.

Considerando que o valor ideal de *Speedup* é igual ao número de processos utilizado por um programa paralelo, nota-se que o *overhead* relacionado ao procedimento de migração do AGMP teve impacto considerável sobre o seu tempo de execução. Por exemplo, o AGMP de 4 populações obteve *Speedup* de apenas 2,6, sendo que, idealmente, seria 4. Neste ponto, a configuração mais eficiente do AGMP foi a de 2 populações, que obteve o maior *Speedup* por número de processos.

Dado que o AGMP de 8 populações foi a configuração que obteve as soluções de melhor qualidade nos experimentos anteriores, nota-se que há um *trade-off* entre qualidade das soluções e tempo de execução, uma vez que o *Speedup* desta configuração foi inferior ao obtido pelas configurações supracitadas.

É importante observar que os baixos tempos de execução apresentados na Tabela 5 não refletem o espaço de busca

Tabela 5. Tempo de execução médio dos experimentos, em milissegundos.

populações	1	2	3	4	5
tempo (ms)	40,33	20,52	19,93	15,59	23,30
populações	6	7	8	9	10
tempo (ms)	21,93	23,67	23,00	22,89	23,26

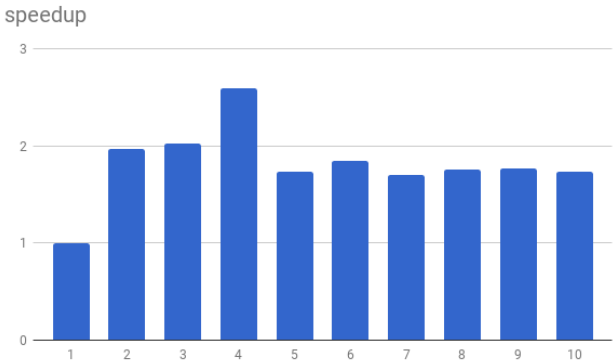

Figura 9. *Speedup* por número de populações.

do problema. Na realidade, esses tempos são resultado da exploração de um número fixo de pontos de busca dado pelos parâmetros dos algoritmos evolutivos, como definido na seção 5.1. Por exemplo, sendo n o número de tarefas de um grafo, a cardinalidade do espaço de busca para o grafo laplace9 e 8 processadores é limitado entre [12]: $P^n = 8^9 = 134.217.728$ e $n!P^n = 9! \cdot 8^9 = 48.704.929.136.640$, ou seja, este limite inferior é 2.571 vezes superior à quantidade de soluções avaliadas pelos algoritmos evolutivos neste trabalho. Para demonstrar experimentalmente a complexidade de se obter uma solução ótima para esta instância relativamente trivial, implementamos o algoritmo *Branch and Bound*, que aplica uma árvore de busca e uma poda baseada no custo das tarefas restantes dividido pelo número de processadores. Ele foi implementado em linguagem Python, por proporcionar maior simplicidade de programação apesar de menor desempenho. Com este programa, após 76 minutos de execução e 414.473.362 avaliações de *makespan*, foi obtido um *makespan* de 520, o mesmo resultado obtido pelos algoritmos evolutivos com apenas 51.880 avaliações. Assim, verifica-se a capacidade destes de obter soluções ótimas mesmo com recursos consideravelmente reduzidos. Também pode-se verificar a relativa complexidade de se obter uma solução ótima mesmo para instâncias menores do problema e a necessidade do uso de métodos aproximados devido ao crescimento exponencial do espaço de busca.

6. Conclusões e Trabalhos Futuros

Neste estudo, o desempenho do AGMP foi comparado com o do AG serial em relação ao problema de escalonamento de tarefas. Ambos foram desenvolvidos com a representação genética de dois vetores, que tem sido utilizada em trabalhos recentes. Foram implementados diversos métodos de

crossover adequados para cada vetor e, para a forma de sua aplicação, foram identificadas na literatura três abordagens diferentes, que foram implementadas e comparadas.

O desempenho do AGMP foi medido variando-se o número de populações e comparado com o desempenho do AG serial em termos de qualidade agregada das soluções e tempo de execução.

Foi composto um conjunto de grafos para *benchmark* dos algoritmos, dividido entre grafos retirados da literatura e famílias de grafos baseadas em algoritmos reais. Para estas famílias, foram implementados scripts que geram grafos correspondentes à execução do algoritmo para diferentes tamanhos de entrada.

Os resultados suportam a hipótese de que AGMP é uma técnica que, dados parâmetros adequados, faz um melhor aproveitamento dos recursos computacionais disponíveis em comparação com o AG serial. Isto se dá de duas formas:

- Produção de soluções de mesma ou melhor qualidade para uma dada quantidade de avaliações de *fitness*.
- Redução do tempo de execução pelo uso de múltiplos núcleos dos processadores atuais.

Como trabalhos futuros, identificamos: realizar experimentos complementares com arquiteturas e grafos diferentes, e com variações do problema de escalonamento. Comparar o desempenho com outras abordagens de AG paralelo e outras meta-heurísticas, e.g. busca tabu [28], GRASP [29] e VNS [30]. Analisar a representatividade das formas de representação genética em relação às soluções alcançáveis no espaço de busca.

7. Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001. Agradecemos a CAPES, CNPq e FAPEMIG pelo apoio financeiro que possibilitou esta pesquisa.

8. Contribuição dos Autores

- Bruno W. Dantas Morais: desenvolvimento geral do trabalho.
- Gina M. Barbosa de Oliveira: orientação.
- Tiago Ismailer de Carvalho: co-orientação e estudo sobre as famílias de grafos.

Referências

[1] HOU, E. S. H.; ANSARI, N.; REN, H. A genetic algorithm for multiprocessor scheduling. *IEEE Trans. Parallel Distrib. Syst.*, v. 5, n. 2, p. 113–120, 2 1994.

[2] GOLDBERG, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989. v. 1.

[3] ALBA, E.; TROYA, J. M. A survey of parallel distributed genetic algorithms. *Complex,*, v. 4, n. 4, p. 31–52, 3 1999.

[4] KWOK, Y.-K.; AHMAD, I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput, Surv,*, v. 31, n. 4, p. 406–471, 12 1999.

[5] CANTú-PAZ, E. A survey of parallel genetic algorithms. *Calc. paralleles reseaux syst. repartis*, v. 10, n. 2, p. 141–171, 1998.

[6] CANTÚ-PAZ, E. et al. Are multiple runs of genetic algorithms better than one? In: *Genetic and Evolutionary Computation — GECCO 2003*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 801–812.

[7] CORREA, R. C.; FERREIRA, A.; REBREYEND, P. Scheduling multiprocessor tasks with genetic algorithms. *IEEE Trans. Parallel Distrib. Syst.*, v. 10, n. 8, p. 825–837, 8 1999.

[8] KAUR, K. et al. Heuristics based genetic algorithm for scheduling static tasks in homogeneous parallel system. *Int. J. Comput. Sci. Secur. (IJCSS)*, v. 4, n. 2, p. 183–198, 1999.

[9] WANG, L. et al. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *J. Parallel Distrib. Comput.*, v. 47, n. 1, p. 8 – 22, 1997.

[10] OMARA, F. A.; ARAFA, M. M. Genetic algorithms for task scheduling problem. *J. Parallel Distrib. Comput.*, v. 70, n. 1, p. 13 – 22, 2010.

[11] CHITRA, P.; RAJARAM, R.; VENKATESH, P. Application and comparison of hybrid evolutionary multiobjective optimization algorithms for solving task scheduling problem on heterogeneous systems. *Appl. Soft Comput.*, v. 11, n. 2, p. 2725 – 2734, 2011.

[12] MORADY, R.; DAL, D. A multi-population based parallel genetic algorithm for multiprocessor task scheduling with communication costs. In: . Messina, Italy: IEEE, 2016. (ISCC, '16), p. 766–772.

[13] KWOK, Y.-K.; AHMAD, I. Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. *J. Parallel Distrib, Comput,*, v. 47, n. 1, p. 58–77, 11 1997.

[14] HWANG, R.; GEN, M.; KATAYAMA, H. A comparison of multiprocessor task scheduling algorithms with communication costs. *Comput. Oper. Res.*, v. 35, n. 3, p. 976 – 993, 2008.

[15] XU, Y. et al. A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues. *Inf. Sci.*, v. 270, n. 1, p. 255 – 287, 2014.

[16] QI, J. G.; BURNS, G. R.; HARRISON, D. K. The application of parallel multipopulation genetic algorithms to dynamic job-shop scheduling. *Int. J. Adv. Manuf. Technol.*, v. 16, n. 8, p. 609–615, 7 2000.

[17] SRINIVASA, K. G.; VENUGOPAL, K. R.; PATNAIK, L. M. A self-adaptive migration model genetic algorithm for data mining applications. *Inf, Sci,*, v. 177, n. 20, p. 4295–4313, 10 2007.

[18] GEHRING, H.; BORTFELDT, A. A parallel genetic algorithm for solving the container loading problem. *Int. Trans. Oper. Res.*, v. 9, n. 5-6, p. 497 – 511, 7 2002.

[19] MÜHLENBEIN, H.; SCHOMISCH, M.; BORN, J. The parallel genetic algorithm as function optimizer. *Parallel Comput,*, v. 17, n. 6-7, p. 619–632, 9 1991.

[20] YAO, J.; KHARMA, N.; GROGONO, P. Bi-objective multipopulation genetic algorithm for multimodal function optimization. *IEEE Trans. Evol. Comput.*, v. 14, n. 1, p. 80–102, 2 2010.

[21] HAN, K.-H. et al. Parallel quantum-inspired genetic algorithm for combinatorial optimization problem. In: . Seoul, South Korea: IEEE, 2001. (CEC2001, v. 2), p. 1422–1429 vol. 2.

[22] OLTEANU, A.; MARIN, A. Generation and evaluation of scheduling dags: How to provide similar evaluation conditions. *Comput. Sci. Master Res.*, v. 1, n. 1, p. 57, 2011.

[23] JIANG, Y.; SHAO, Z.; GUO, Y. A dag scheduling scheme on heterogeneous computing systems using tuple-based chemical reaction optimization. *Sci. World J.*, v. 2014, n. 1, p. 404375, 6 2014.

[24] COSNARD, M. et al. Parallel gaussian elimination on an mimd computer. *Parallel Comput.*, v. 6, n. 3, p. 275 – 296, 1988.

[25] WU, M. Y.; GAJSKI, D. D. Hypertool: a programming aid for message-passing systems. *IEEE Trans. Parallel Distrib. Syst.*, v. 1, n. 3, p. 330–343, 7 1990.

[26] XU, Y.; LI, K.; HU, J. A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues. *Inf. Sci.*, v. 270, n. 1, p. 255–257, 6 2014.

[27] CARNEIRO, M. G. *Abordagens baseadas em autômatos celulares síncronos para o escalonamento estático de tarefas em multiprocessadores*. Tese (Doutorado), Uberlândia, Brasil, 2012.

[28] GLOVER, F.; LAGUNA, M. Tabu search. In: *Handbook of Combinatorial Optimization*. 1. ed. Boston, MA: Springer US, 1999. Volume 1–3, p. 2093–2229.

[29] FEO, T. A.; RESENDE, M. G. C. Greedy randomized adaptive search procedures. *J. Glob. Optim.*, v. 6, n. 2, p. 109–133, 3 1995.

[30] MLADENOVIC, N.; HANSEN, P. Variable neighborhood search. *Comput. Oper. Res.*, v. 24, n. 11, p. 1097 – 1100, 1997.