UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

Proposal of Embedded Standalone and Distributed Genetic Algorithms for Low-Power, Low-Cost and Small-Sized Memory Devices

Denis Ricardo da Silva Medeiros

Supervisor: Prof. Dr. Marcelo Augusto Costa Fernandes

Masters Dissertation presented to the Graduate Program in Electrical and Computer Engineering of UFRN (concentration area: Computer Engineering) as part of the requirements for obtaining the degree of Master of Science.

PPgEEC Order Number: M614
Natal, RN, November 2020

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Central Zila Mamede

Medeiros, Denis Ricardo da Silva.
Proposta de algoritmos genéticos embarcados para dispositivos com baixo consumo de energia, baixo custo e tamanho de memória reduzido / Denis Ricardo da Silva Medeiros. - 2020.
100 f.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Norte, Centro de Tecnologia, Programa em Pós-Graduação em Engenharia Elétrica e de Computacção Natal, RN, 2020.
Orientador: Prof. Dr. Marcelo Augusto Costa Fernandes.

1. Algoritmos genéticos - Dissertação. 2. Microcontroladores - Dissertação. 3. 8-bits - Dissertação. 4. Sistemas embarcados - Dissertação. 5. Sistemas distribuídos - Dissertação. I. Fernandes, Marcelo Augusto Costa. II. Título.

RN/UF/BCZM CDU 004.021

Elaborado por Ana Cristina Cavalcanti Tinoco - CRB-15/262

Acknowledgements

I wish to express my sincere appreciation to my supervisor, Professor Marcelo Augusto Costa Fernandes, who guided and encouraged me to believe this work would be accomplished even with the adverse situation of me having to leave Brazil to live abroad. Without his persistent help and incentive, the goal of this work would not have been successfully finalized.

I wish to acknowledge the support and partnership of Matheus F. Torquato on the first part of this work, which resulted in the publication of an academic article to a journal.

I wish to thank Professor Ivanovitch Medeiros Dantas da Silva, for allowing me to take the Machine Learning course remotely, and for his patience and support during the course even with my adverse situation.

I wish to thank all the members of the examining board for their participation and valuable contributions.

Finally, I want to thank the support from my family and friends, who always motivated to keep dreaming big and pursuing my goals.

Abstract

This work proposes implementations of genetic algorithms targeting low-power, low-cost, and small-sized memory devices in two variants: a standalone version, to be used in one single device, and a distributed version, to be used in multiple devices simultaneously. The motivation is to adapt and optimize this important artificial intelligence (AI) technique so that it can be used in numerous applications where traditionally it was not feasible to be utilized, such as in microcontrollers. An investigation about how to optimize each separated segment of the algorithm is done and extensive details about both implementations are provided, including their source codes. Moreover, various experiments and simulations for multiple scenarios were done to validate their correct operation using Hardware-In-Loop technique, as well as to find out limitations for the algorithm parameters. The standalone implementation is compared to other work in the literature and it performs faster and using fewer resources. For the distributed version, an important analysis was done to verify that it can be faster than the standalone version but also more power-efficient when reducing the clock speed and voltage of those devices. Finally, this investigation could determine what is the threshold from where the distributed version, even with a high overhead due communication between the devices, become either faster or more power-efficient than the standalone one.

Keywords: Genetic algorithms, microcontrollers, 8-bit, embedded systems, distributed systems.

Resumo

Este trabalho propõe duas implementações de algoritmos genéticos voltadas para dispositivos com limitações em consumo de energia, baixo custo, e quantidade de memória: uma versão autônoma, para ser uttilizada em um único dispositivo, e uma versão distribuída, para ser usada em múltiplos dispositivos simultaneamente. A motivação deste trabalho é adaptar e otimizar essa importante técnica de inteligência artificial para que ela possa ser usada em inúmeras aplicações que tradicionalmente não era viável, como em microcontroladores. No trabalho, é feita uma investigação sobre como otimizar cada segmento do algoritmo e são fornecidos detalhes específicos sobre as duas implementações, incluindo seus códigos-fonte. Além disso, vários experimentos e simulações para diferentes cenários foram realizadas para validar seu funcionamento correto usando a técnica de Hardware-In-Loop, bem como para descobrir limitações nos parâmetros do algoritmo. A implementação autônoma é comparada a outro trabalho da literatura e ela é mais rápida e usa menos recursos. Para a versão distribuída, uma importante análise feita foi verificar se ela pode ser mais rápida que a versão autônoma, mas também mais eficiente em termos de consumo de energia ao se reduzir a frequência e a tensão de operação desses dispositivos. Por fim, esta investigação pôde determinar qual é o limite a partir de onde a versão distribuída, mesmo com uma alta sobrecarga devido à comunicação entre os dispositivos, se torna mais rápida ou mais eficiente em termos de consumo energia quando comparada à outra versão.

Palavras-chave: Algoritmos genéticos, microcontroladores, 8-bits, sistemas embarcados, sistemas distribuídos.

Contents

List of Figures

List of Tables

List of Symbols

µC	Microcontroller
CBIF	Collect Best Individual Function
CEVF	Collect Evaluation Value Function
CF	Crossover Function
CFM	Crossover Function Module
CIF	Collect Individual Function
COF	Continue Operations Function
CPF	Command Processing Function
DGA	Distributed Generic Algorithm
DVFS	Dynamic Voltage and Frequency Scaling
EF	Evaluation Function
EFM	Evaluation Function Module
FFM	Fitness Function Module
GA	Genetic Algorithm
GDP	Gross Domestic Product
GPU	Graphics Processing Unit
HIL	Hardware-In-Loop
IFM	Initialization Function Module
IoT	Internet of Things
LCG	Linear Congruential Generator
LFSRG	Linear Feedback Shift Register Generator
LUT	Lookup Table

M2M	Machine-to-Machine
MIPS	Millions Instructions per Second
MISO	Master Input, Slave Output
MOSI	Master Output, Slave Input
MPI	Message Passing Interface
MTG	Mersenne Twister Generator
MWCG	Multiply With Carry Generator
NFM	Normalization Function Module
NPFM	New Population Function Module
PRNG	Pseudo-random Number Generator
RNG	Random Number Generator
SBIF	Send Best Individual Function
SCFM	Selection and Crossover Function Module
SCLK	Serial Clock
SF	Selection Function
SFM	Selection Function Module
SIF	Send Individual Function
SMG	SplitMix Generator
SoC	System-on-Chip
SPI	Serial Peripheral Interface
SS	Slave Select
TF	Tournament Function
UFM	Update Function Module
USART	Universal Asynchronous Receiver/ Transmitter
XSG	XorShift Generator

Chapter 1

Introduction

Artificial intelligence (AI) has been one of the fields that have received a lot of attention from researchers and technology companies lately. Big companies and industries have invested heavily in AI and created numerous solutions in areas such as image and audio processing, cloud computing, autonomous driving, robotics, medicine, financial markets, and others (Lu 2019). In general, any activity that traditionally could be done by humans, such as classification, prediction, and control, but with the ability to adapt to changes in the environment can be potentially be done with by machines using artificial intelligence (Yampolskiy 2020).

The numbers involving AI are really impressive. One study estimates that artificial intelligence may contribute 15.7 trillion USD to the global economy by 2030, which is about 14% of the global GDP, and 50 trillion USD by 2050, which is nearly 20% of the expected global GDP (Abdulzaher 2019). In terms of the job market, Brazil is among the countries that experienced the fastest growth in AI hiring between 2015 and 2019 (*The 2019 AI Index Report* n.d.). This shows that emerging countries also have the potential to create AI innovation and boost their local economies.

Historically, it had been challenging to run some artificial intelligence algorithms due to the high and intensive processing power required, mainly when using machine learning techniques such as Deep Learning. In the last years, with the development of new graphics processing units (GPUs) and dedicated hardware to process AI algorithms, and even some cloud platforms with clusters dedicated to processing AI, the performance has increased a lot (Wang et al. 2019). Thus, several complex machine learning algorithms have become feasible to be used in numerous applications.

Even though AI has been used broadly in those high-performance platforms, it is still a challenge to run those algorithms in low-cost, low-power, and low-size-memory hardware. An example of these devices is the Systems on Chips (SoCs), which includes the microcontrollers (μCs). They are widely available and used in industry and household applications, such as in industrial automation and control systems, measurement equipment,

vehicles, robots, security cameras, household appliances, and in many other applications where some sort of computing is needed (Panda et al. 2019).

System on Chips are usually constituted of a general-purpose microprocessor (8, 16, or 32 bits), data and program memory, and several peripherals such as counters, signal generators, an analog-to-digital converter, communication interfaces, input and output pins, and others (Risset 2011). Generally, they are applied as an embedded system for specific applications, mainly for having low costs to not increase the overall price of the application itself, and for consuming low-power, which is ideal for applications that run on battery, for instance.

The use of SoCs has increased significantly in the last years, mainly in emerging areas such as the Internet of Things (IoT), Smart Grid, Machine-To-Machine (M2M), Industry 4.0, and others. Some studies predict a huge increase in these areas in the following years. For IoT, it is expected about 8.4 billion of connected things in 2020 and 20.4 billion by 2022. For M2M, the expectation is that by 2024 there will be about 27 billion machine-to-machine connections (Hassija et al. 2019). These numbers show that SoCs and microcontrollers will be even more used for those applications that open several opportunities for research and development using them.

Based on the current trends and upcoming scenarios, artificial intelligence applications will also be more present in those emerging areas, such as in IoT applications. One possible strategy that initially sounds feasible is to make the IoT applications to send the data to be processed in the cloud by some AI algorithm and then receive the results back. However, with the analysts predicting a huge increase in the number of these devices, this may create a communication problem due to an enormous volume of data being transmitted back and forth. Thus, an interesting approach is to make the embedded system to be able to process part of the AI solution, which is called edge computing (Elazhary 2019, Sittón-Candanedo et al. 2019).

Developing AI applications for low processing power devices, such as microcontrollers, has been studied in several works found in the literature. As mentioned before, some AI algorithms require a high computing effort to run the algorithm and this could be a problem for some embedded systems, mainly if they are applied in some real-time application that has some time restriction. Therefore, several researchers try to optimize artificial intelligence algorithms for these platforms to make them viable for several applications (Sakr et al. 2020, Burkert 2019).

One artificial intelligence technique that has gained space in microcontrolled-systems is the genetic algorithms (GAs). They are usually used in applications that require solving some optimization or search problem, such as finding the shortest path or finding the op-

timal parameter tuning for some application, for example, (Nazarahari et al. 2019, Chen & Gao 2019, dos Santos et al. 2019). Since GAs can require a lot of computing resources to run, then the researchers need to optimize its implementation when targeting low processing power devices such as μCs.

Most works in the literature that address GAs for microcontrollers try to solve specific problems, such as in (Zakaria et al. 2017, Megantoro et al. 2018, El-Saadawi et al. 2020). In (Zakaria et al. 2017), the authors try to create an application to optimize the navigation using the Stewart Platform model. In (Megantoro et al. 2018), they try to optimize a technique called MPPT to improve the efficiency of DC/DC buck converters in solar home systems. Finally, (El-Saadawi et al. 2020) proposes a GA implementation to find the optimal parameters for a Fractional-order PID control system to control the speed of DC motors. Thus, these works focus on how to solve those engineering problems and not in optimizing the GA implementation itself, besides not providing details about the GA implementation.

A few works try to focus on the genetic algorithm implementation and its optimizations for microcontrollers though. In (Santos et al. 2018), the authors propose a multi-objective GA implementation for microcontrollers and show some interesting results for different evaluation functions. They present clock-cycles, processing time, and energy results but do not analyze memory consumption or try multiple scenarios with different GA parameters. They also do not provide comprehensive details about the implementation or provide the source code to be compared to other works. Another work in this same direction is (Alves 2010), where the author proposes a GA implementation for Arduino Duemilanove, which uses the μATmega328P. That work is interesting for providing the source code of his implementation but the author does not analyze any aspect of resource consumption, such as processing time or memory usage.

Given this context, this work proposes an implementation of two variants of genetic algorithms for those low-power, low-cost, and low-size-memory devices: a standalone version, suitable to run in one single microcontroller, and one distributed version, intended to run in multiple microcontrollers. Both versions were designed aiming to reduce the processing time, energy, and memory consumption so that they can be integrated as part of a bigger project. Differently from most works found in the literature on this topic, another goal is to provide comprehensive details about the implementation and several results for different scenarios.

Both proposals have the same motivation, which is to make algorithms genetics feasible and efficient to run directly on microcontrollers, under the edge computing idea. There are numerous applications where they can be used, such as battery-powered drones

or robots that need to calculate the best path to a destination without collision, for instance (de Oliveira & Fernandes 2016, Silva et al. 2018, Lamini et al. 2018). Another example is when a vehicle has an embedded system that needs to find optimal parameters for car engines as presented in (Krope et al. 2017, Kalaivanan & Sakthivel 2017). Therefore, running an embedded GA on the edge is interesting because it does not need to access any other external systems, such as a cloud platform, which reduces costs and energy consumption since there is no need to use a communication device.

Finally, there is a special motivation for the distributed version of genetic algorithms. By having the GA running in a set of microcontrollers, it is possible to apply the Dynamic Voltage and Frequency Scaling (DVFS) technique, where the reduction of the clock speed and voltage of CMOS systems, like the one presented in microcontrollers, can reduce drastically the power-consumption (Le Sueur & Heiser 2010). Reducing power and energy consumption is an important goal for creating sustainable solutions, mainly due to the huge number of embedded systems expected for the following years.

1.1 Objectives

The objective of this dissertation is to contribute to the edge computing idea, by proposing an optimized implementation of an artificial intelligence technique, genetic algorithms, to be run directly on low-cost, low-power, and low-size-memory devices, such as microcontrollers.

The specific objectives are listed below:

- To propose an optimized implementation of standalone and distributed genetic algorithms for low-cost, low-power, and low-size-memory devices;
- To demonstrate through experiments the proper operation of both implementations for multiple scenarios, by using the Hardware-In-Loop technique;
- To evaluate through experiments the performance and limitations of both implementations and contrast each other to find out what is the best scenario for each one;
- To compare the current implementations with other work in the literature if there is any existent;
- To investigate how the distributed GA version can be more power-efficient by using the Dynamic Voltage and Frequency Scaling technique.

1.2 Submitted and Published Articles

The results obtained in this work generated two academic papers, where both accepted and published in international journals.

- MEDEIROS, Denis R. da S.; TORQUATO, Matheus F.; FERNANDES, Marcelo A. C. Embedded genetic algorithm for low-power, low-cost and low-size-memory devices. **Engineering Reports**, v. 2, n. 9. DOI: 10.1002/eng2.12231.
- MEDEIROS, Denis R. da S.; FERNANDES, Marcelo A. C. Distributed Genetic Algorithms for Low-Power, Low-Cost and Small-Sized Memory Devices. **MDPI Electronics**, v. 9, n. 11. DOI: 10.3390/electronics9111891.

1.3 Dissertation Outline

This dissertation is structured in 4 chapters, as described in the following paragraphs.

Chapter 1 is the introduction, where it starts by presenting the current context and future predictions for the fields of artificial intelligence and embedded systems, mainly due to the fast growth of areas such as the Internet of Things. It also explains the challenge of adapting AI techniques to be run in low-power devices and why this is necessary to move towards a scenario of edge computing. Moreover, it presents the motivation of this work and where it can be applied, and finally the objectives of this research.

Chapter 2 presents the proposal of a standalone implementation of genetic algorithms for low-power, low-cost, and low-size-memory devices. It provides a detailed explanation of how each section of the genetic algorithm was modified through the definition of several constraints and strategies to reduce the consumption of processing time and memory. Furthermore, numerous experiments and simulations were done to validate its proper operation and also to analyze the resource consumption for different setups and parameters. Finally, a comparison is made with another work in the literature and the results demonstrate this standalone version is much more efficient in terms of performance and resource consumption.

Chapter 3 presents the proposal of the distributed implementation of algorithms for low-power, low-cost, and low-size-memory devices. It is built taking the standalone version as a base and describes different strategies for distributed GA and why a different approach was created for this work. Moreover, it details the master-slave architecture for this project and how the implementation is done for each kind of node. Also, it discusses the limitations on the communication between microcontrollers and a small protocol cre-

ated to control the data transfer between the nodes by using the Serial Peripheral Interface (SPI) interface. Finally, several experiments were done to analyze processing time, memory usage, how the DVFS can make this implementation power-efficient by reducing clock speed and voltage, and in which conditions this version can be better or worse than the standalone one.

Chapter 4, finally, brings up the final considerations and conclusions about the results, and discuss possible future works.

Chapter 2

Embedded Standalone Genetic Algorithm

This chapter proposes a strategy to create an embedded genetic algorithms (GAs) for low-power, low-cost and low-size-memory devices. This strategy aims to provide the means of GAs to run as a low cost and low power consumption embedded system, where microcontrollers (μCs) are commonly used. The implementation details are presented, emphasizing the limitations and restrictions imposed to turn it more compact and efficient. Also, data related to the algorithm effectiveness, processing time and memory consumption were obtained from simulations, oscilloscope measurements, and using the hardware-in-loop (HIL) technique. Lastly, this implementation is compared with other implementation from the literature and the results show that 8-bits μCs can run GAs for several practical applications.

2.1 Introduction

Artificial intelligence (AI) is a field that will offer many opportunities for emerging markets and services and will revolutionize almost every segment of society. AI covers a wide range of activities: Medical, multimedia, finance, or advertising, among others. The results are applied to the most different systems, such as clusters, mobile phones, or even smart sensors. However, AI requires intensive processing and high-performance hardware, and this can be a problem on IoT applications in which the target is hardware with low-power, low-cost and low-size-memory.

On the other hand, low-power, low-cost and low-size-memory devices, such as microcontrollers (μCs), have been used in applications. Microcontrollers have been used in several types of applications in areas such as industrial automation, control, measuring, consumer electronics, etc. Also, there is a growing demand for the use of these devices,

mainly in emerging markets such as Internet of Things (IoT), Smart Grid, and Machine to Machine (M2M), for instance. Hardware platforms such as μCs can be classified as digital systems called Systems on Chips (SoCs), which are commonly used as embedded systems for specific applications. Usually, this type of SoC is composed by a 8, 16 or 32-bits general purpose microprocessor, program and data memory, besides being coupled with other peripherals, such as counters, signal generators, analog-to-digital converters, among others. Even though μCs have a limited processing power, their main advantages are the low power consumption, reduced size and price, which makes them appropriate to be used in several situations where these characteristics are required, such as in IoT applications, for example (Qian et al. 2009, Gaikwad et al. 2015, Sawulski & Ławryńczuk 2019, Al-Kofahi et al. 2019).

The use of intelligent systems in embedded hardware with μCs have been studied and there are a few works in the literature on that subject. This can be explained by the fact that AI is currently one of the most studied areas in computing and most applications make use of it to solve different types of problems. Nevertheless, some AI techniques demand high computational power, especially when applications impose tight time constraint. Therefore, it has been a challenge to optimize and adapt AI algorithms to make them suitable to run on hardware platforms such as the μCs. As a result, several research groups have been working in this direction (Situmorang & Situmorang 2015, Ortega-Zamorano et al. 2016, Dong et al. 2017, Eldahab et al. 2017).

Among the AI techniques, the genetic algorithms are a kind of bio-inspired metaheuristics used to solve search and optimization problems in several areas of industry and engineering (Shelokar et al. 2014). The GAs are based on the Darwin's theory of evolution and they provide efficiency in solving numerous different kinds of problems (Holland 1992). However, they have a high computational complexity, which is affected by the population size and the evaluation function (also called fitness function).

There are some works in the literature that use genetic algorithms on μCs, such as (Garzia & Veca 2002, Alves 2010, Mamdoohi et al. 2012, Dendaluce et al. 2014). Nonetheless, the GA implementations presented on most of these works try to solve specific problems, hence their results do not focus on the genetic algorithm performance or on its implementation itself. For example, (Mamdoohi et al. 2012) and (Dendaluce et al. 2014) show the processing time only for a small number of individuals (or chromosomes) in their GA implementation. On the other hand, in (Garzia & Veca 2002) and (Alves 2010) the GA implementation is not optimized for low-power processors, as those that are commonly found on μCs. In fact most of these projects do not provide details about the implementation and do not analyze the processing time and memory consump-

tion used by the GA for different settings.

The GA algorithm encodes the information on a string of bits, and this codification permits to use a bitwise operation in must algorithms steps. The use of bitwise operations is used in dedicate hardware proposals (Torquato & Fernandes 2019) but it is have not been used to low-power, low-cost, and low-size-memory devices (Garzia & Veca 2002, Alves 2010, Mamdoohi et al. 2012, Dendaluce et al. 2014). Bitwise operations have been used in several works in the literature to reduce the algorithm processing time (Yordzhev 2012, Yordzhev 2013, Kim & Smaragdis 2016, Cislak & Grabowski 2017).

Therefore, the main goal of this present work is to present a genetic algorithm implementation for low-power, low-cost and low-size-memory devices, such as μCs, with the purpose of enabling its use in a generic way for different kinds of applications. The bitwise operations are used in all GA steps, and the proposal was optimized to 8-bits μCs architecture. This optimization allows the integration with other 16-bits and 32-bits embedded systems architectures.

An embedded genetic algorithm implementation with these characteristics is suitable for applications where there is a need to solve some non-linear optimization or search problem but with limitations in terms of costs and mainly power consumption. For example, a battery-powered drone or robot that needs to calculate the shortest path to reach a destination avoiding obstacles using genetic algorithms (de Oliveira & Fernandes 2016, Silva et al. 2018, Lamini et al. 2018). Another example is the use of GAs embedded in automobiles to calibrate car engine parameters as presented in (Krope et al. 2017, Kalaivanan & Sakthivel 2017). In (Kalaivanan & Sakthivel 2017), the authors were using even an 8-bit microcontroller to run the experiments. Finally, for these types of applications, the number of individuals and the number of generations do not need to be large – in (Silva et al. 2018), for instance, the number of individuals and the number of generations were at most 30. Therefore, a low-cost and low-power μC is feasible for these cases.

The implementation details, results concerning the processing time and memory consumption are analyzed for different parameters, such as population size and the number of bits needed to represent each individual from the population, the number of generations, and the type of evaluation function (or fitness function). Based on these results, it is possible to identify limits for each parameter in order to keep the GA feasible, as well as the parts of algorithm which affects more the performance and consume more resources. Furthermore, the hardware-in-loop technique is used in order to validate the correct operation of the GA. Finally, a comparison with other implementation available in the literature is also made to show how this one performs.

2.2 Genetic Algorithms

Genetic algorithms are iterative algorithms which start by randomly generating a population of N individuals (or chromosomes) that are mapped into M bits each. For each k-th iteration of the algorithm, called generation, the N individuals (or chromosomes) pass through the operations of evaluation, selection, crossover, and mutation. After these operations, a new population with the same size is generated and its individuals can replace all or some individuals from the previous generation and then this new created population becomes the starting point of the next generation. This cycle is repeated K times, where K is a GA parameter and represents the number of generations in which the algorithm will be executed.

The Algorithm 1 represents the pseudocode of the GA proposed in this work and it is inspired in (Holland 1992). The vector $\mathbf{x}_j(k)$ represents the j-th individual of the N-sized population $\mathbf{X}(k)$, on the k-th generation. Each j-th individual has dimension D, thus the element $\mathbf{x}_{j,i}[M](k)$ represents the i-th dimension of this individual, which is mapped into M bits. Therefore, the population $\mathbf{X}(k)$ can be expressed as

$$\mathbf{X}(k) = \begin{bmatrix} \mathbf{x}_0(k) & \dots & \mathbf{x}_{N-1}(k) \end{bmatrix} = \begin{bmatrix} \mathbf{x}_{0,0}[M](k) & \dots & \mathbf{x}_{0,D-1}[M](k) \\ \vdots & \ddots & \vdots \\ \mathbf{x}_{N-1,0}[M](k) & \dots & \mathbf{x}_{N-1,D-1}[M](k) \end{bmatrix}. \tag{2.1}$$

The first step of the algorithm is the generation of the first population (Line 1 of Algorithm 1). After this, the evaluation function (or fitness function), named EF (Line 4 of Algorithm 1), is applied over all N individuals $\mathbf{x}_j[M](k)$ and it calculates the fitness value for each one. The index of the best individual is stored in jb to be used in the elitism operation later. The fitness of the j-th individual with dimension D is stored in $y_j[B](k)$, where B is the number of bits required to represent the fitness. The better the fitness value $y_j[B](k)$ of the individual $\mathbf{x}_j[M](k)$, the greater the probability of this individual to be selected or forwarded to the next generation. These values of the N individuals are stored as follows

$$\mathbf{y}(k) = \begin{bmatrix} y_0[B](k) & \dots & y_{N-1}[B](k) \end{bmatrix}. \tag{2.2}$$

After the evaluation, the next operation is the selection, which aims to choose the individuals with best fitness value $y_j[B](k)$ in order to generate an improved population compared to the previous one. There are several selection methods described in the literature such as the roulette wheel selection, the stochastic universal sampling, the tournament selection, and the rank-based selection, for example (Talbi 2009). In this imple-

mentation, the tournament selection is used since it is one of the most used and efficient methods according to (Noraini & Geraghty 2011). This method compares two or more individuals randomly selected from $\mathbf{X}(k)$. In other words, this method compare the fitness value $y_j[B](k)$ of those chosen individuals and the best one, called the tournament winner, moves on to share its genes in the next step, that is, the crossover operation.

The selection function is represented in the pseudocode as SF (Line 10 of Algorithm 1). Its inputs are the vector $\mathbf{y}(k)$ and the matrix $\mathbf{X}(k)$ of the k-th generation and its output is the index of the j-th individual, $\mathbf{x}_j(k)$, that was selected. The elitism technique can also be applied, so that the best E individuals of the current population are passed directly to the new population. In this implementation, $E = 1$ and the best individual is placed on the first position of the new population (Line 16 of Algorithm 1).

The next operation is the crossover, where two or more individuals from the current population, $\mathbf{X}(k)$, are combined to generate new ones that will be inserted into the new population, $\mathbf{X}(k+1)$, after passing through the mutation operation. In the literature, there are several strategies for the crossover such as the one-point crossover, two-point, and uniform (Luke 2013). The implementation developed in this work allows the use of any of these three options by configuring a mask, as will be shown later in the Section 2.3.5.

The crossover function is defined as CF (Line 10 of Algorithm 1) and has as input two indices, which represents two individuals, and the matrix $\mathbf{X}(k)$ of the k-th generation, which represents the current population. As output, it returns two new individuals generated by the crossover, which will be part of the offspring of the k-th generation. This offspring is stored into the matrix $\mathbf{Z}(k)$, which is defined as

$$\mathbf{Z}(k) = \begin{bmatrix} \mathbf{z}_0(k) & \dots & \mathbf{z}_{N-1}(k) \end{bmatrix} = \begin{bmatrix} \mathbf{z}_{0,0}[M](k) & \dots & \mathbf{z}_{0,D-1}[M](k) \\ \vdots & \ddots & \vdots \\ \mathbf{z}_{N-1,0}[M](k) & \dots & \mathbf{z}_{N-1,D-1}[M](k) \end{bmatrix}. \tag{2.3}$$

With the matrix $\mathbf{Z}(k)$ filled with N individuals, the following operation is the mutation, where P individuals will have their information randomly modified. In this work, the mutation function is defined as MF (Line 13 of Algorithm 1) and receives as input a new individual $\mathbf{z}_v[M](k)$, from the k-th generation, and replaces it with its modified version. The mutation rate, called R_M, defines the proportion of individuals that suffer mutation, hence P can be specified as

$$P = \lceil R_M \times N \rceil. \tag{2.4}$$

The last operation of the GA is the population update. In the literature, there are different approaches in which the entire older population or only a part of it is replaced

Algorithm 1 Genetic Algorithm Pseudocode

▷ Generation of the initial population
1: Initialize($\mathbf{X}(0)$)
▷ Starts to process the generations
2: **for** $k \leftarrow 0$ **to** $K-1$ **do**
▷ Calculates the fitnesses and evaluates the individuals (or chromosomes)
3: **for** $j \leftarrow 0$ **to** $N-1$ **do**
4: $y_j[B](k) \leftarrow EF\left(\mathbf{x}_j(k)\right)$
5: **if** $y_j[B](k) < y_{jb}[B](k)$ **then**
6: $jb = j$
7: **end if**
8: **end for**
▷ Selection and crossover
9: **for** $i \leftarrow 0$ **to** $N-1$ **with step** 2 **do**
10: $\begin{bmatrix} \mathbf{z}_i(k) \\ \mathbf{z}_{i+1}(k) \end{bmatrix} \leftarrow CF\left(\begin{bmatrix} \mathrm{SF}\left(\mathbf{y}(k), \mathbf{X}(k)\right) \\ \mathrm{SF}\left(\mathbf{y}(k), \mathbf{X}(k)\right) \end{bmatrix}\right)$
11: **end for**
▷ Mutation
12: **for** $v \leftarrow 0$ **to** $P-1$ **do**
13: $\mathbf{z}_v(k) \leftarrow MF\left(\mathbf{z}_v(k)\right)$
14: **end for**
▷ Elitism
15: **for** $i \leftarrow 0$ **to** $D-1$ **do**
16: $x_{0,i}[M](k) \leftarrow x_{jb,i}[M](k)$
17: **end for**
▷ Updates the population
18: **for** $j \leftarrow 1$ **to** $N-1$ **do**
19: **for** $i \leftarrow 0$ **to** $D-1$ **do**
20: $x_{j,i}[M](k) \leftarrow z_{j,i}[M](k)$
21: **end for**
22: **end for**
23: **end for**

(Gendreau & Potvin 2010). In this implementation, the entire population $\mathbf{X}(k)$ is renewed, that is, each j-th individual of the k-th generation is replaced by a new individual, generating the population of the next generation, $\mathbf{X}(k+1)$. These new individuals can come from both the offspring of the k-th generation, stored in $\mathbf{Z}(k)$, or directly from the old population due the elitism technique (Lines 16 and 20 of Algorithm 1).

2.3 Implementation

The implementation proposed in this work aims to produce an efficient and optimized design to run on 8-bits microcontrollers. Thus, after its compilation, it should have small size, low memory occupation, efficient use of variables to save data memory, and optimized operations to decrease the processing time of the GA. Furthermore, most of the GA operations were structured as close as possible to a hardware level, with some of them implemented by using bit-wise operations, which increase the GA performance (Pedemonte et al. 2011). Therefore, to meet these requirements it was needed to define some development strategies and constraints that are going be to explained in the following sections.

2.3.1 Parameters Constraints

Based on the pseudocode presented on Algorithm 1 and in order to achieve the goal of saving memory and reducing processing time, several parameters constrains were defined. Moreover, it is important to mention that this work was implemented using the C programming language, but the same ideas can be applied to other programming languages as well.

Initially, as shown in the previous section, there are three main variables in the GA are $\mathbf{X}(k)$, $\mathbf{y}(k)$ and $\mathbf{Z}(k)$ and they are responsible for consuming most of data memory. Thus, the first established constraint was related to the representation of the individual, where the number of bits, M, needs be the same of the primitive data types `uint8_t`, `uint16_t` and `uint32_t`. In other words, the individuals have to be represented as unsigned integer numbers of 8, 16 or 32 bits respectively (Fre 2018*b*). In order to standardize the source code, it was created an alias responsible to represent the individuals data type. It is called `chromosome_t`, which is configured before the compilation and it helps the code to be more organized and legible.

The second constraint was to limit the population size N and the number of generations K. N has to be a power of 2, less or equal to 256 and K needs to be less or equal to 256. By using these limits, it was possible to optimize some bitwise operations during the selection operation, which will be detailed in Section 2.3.5, as well as to reduce the size of auxiliary variables that are used during the loops. The types of these auxilary variables were standardized by the aliases `popsize_t`, which indexes the size of the population, and `generationsize_t`, which indexes the generations. Both aliases are mapped into `uint8_t`, since 8-bits are sufficient to represent 256 values. Mathematically, these limits

are defined as

$$N = 2^{A_1}, \text{ where } 1 \leq A_1 \leq 8, \tag{2.5}$$

and

$$1 \leq K \leq 256. \tag{2.6}$$

Other 4 aliases were also created: `chromosomesize_t`, `dimensionsize_t`, `fitness_t`, and `normalization_t`. Variables of the type `chromosomesize_t` saves the size of the individual in bits, M, which can be 8, 16 or 32 bits long as explained before, and it is represented as `uint8_t`. The second alias, `dimensionsize_t`, saves the number of dimensions that an individual can have and it is represented as `uint8_t` as well. The last two aliases are related to the evaluation function, which is likely to deal with real numbers, and therefore can be represented as floating point numbers, such as IEEE-754 single or double precision, for example. Thus, `fitness_t` is represented with B bits and `normalization_t` with G bits, which can vary depending on the compiler implementation.

From the constraints presented above, it can be calculated the number of bytes that the main variables of the program will need from the data memory during the execution. Hence, the amount of consumed memory by the matrix $\mathbf{X}(k)$, the vector $\mathbf{y}(k)$ and the matrix $\mathbf{Z}(k)$ is defined for each one, respectively, as

$$n_{RAM}^{\mathbf{X}} = \frac{M \times N \times D}{8} \; bytes, \tag{2.7}$$

$$n_{RAM}^{\mathbf{y}} = \frac{B \times N \times D}{8} \; bytes, \tag{2.8}$$

and

$$n_{RAM}^{\mathbf{Z}} = \frac{M \times N \times D}{8} \; bytes. \tag{2.9}$$

Finally, based on Equations 2.7-2.9, it is possible to estimate the total data memory consumed by the entire genetic algorithm. Given that n_{RAM}^{aux} is the amount of memory in bytes consumed by auxiliary variables, such as those used inside of the scope of loops, for instance, this amount can be expressed as

$$n_{RAM}^{AG} = n_{RAM}^{\mathbf{X}} + n_{RAM}^{\mathbf{y}} + n_{RAM}^{\mathbf{Z}} + n_{RAM}^{aux} = \frac{2(M \times N \times D) + (B \times N \times D)}{8} + n_{RAM}^{aux}. \tag{2.10}$$

2.3.2 GA Parameters Representation

One important characteristic of the Genetic Algorithms is their parameters, such as number of individuals, maximum number of generations, mutation rate, etc, which must be chosen as per the problem where the GA is being applied. Since all these parameters are fixed values, the idea in this implementation is to avoid wasting memory with variables to store them, thus the idea is to insert them directly in the processor instructions.

An automatic and straightforward way to perform this insertion is to use the pre-processing directives from the C programming language. Essentially, these directives are expressions evaluated during the pre-compilation stage and occurs, as its name suggests, before the compilation of the program itself (Fre 2018*a*). A common directive is the macro, which is a type of key that is replaced by some code fragment in the source code and that is created using the reserved word `define`. Other useful directives are the conditionals, as with them it is possible to evaluate expressions and define parts of the source code that can be enabled or disabled, for example. Among them can be cited the `if`, `elif` and the `else`, for example.

Based on these concepts, the GA parameters were defined as macros in the source code. Due to the existence of constraints that require some numbers to be less than 256, depending on the value, it can be represented as an immediate in the assembly instruction during the compilation step. Moreover, the implementation was configured in such a way that some operations are processed differently according to those parameters defined previously. An example of how this strategy works is shown in Algorithm 2.

From now on, in the following algorithms it must be considered that the GA parameters were already defined as macros. They are the following ones:

- M - Size of the individual (number of bits);
- D - Number of dimensions of the individuals;
- N - Population size (number of individuals);
- K - Maximum number of generations;
- P - Number of individuals that will mutate;
- L_{min} - Minimum value used in the normalization;
- L_{max} - Maximum value used in the normalization.

2.3.3 Pseudo-random Number Generator

During the first experiments using this implementation, it was noticed that disregarding the evaluation function, which varies according to the problem, the parts of the algo-

Algorithm 2 Example of the use of directives

```
    ▷ Examples of macros
 1: (DIRECTIVE) #define M 16
 2: (DIRECTIVE) #define N 32
 3: (DIRECTIVE) #define D 2
 4: (DIRECTIVE) #define K 32
 5: (DIRECTIVE) #define P 2
    ▷ Examples of conditionals
 6: (DIRECTIVE) IF M == 32
 7:     typedef uint32_t chromosome_t;
 8:     (DIRECTIVE) #define MASK 0xFFFF0000
 9: (DIRECTIVE) ELSE IF M == 16
10:     typedef uint16_t chromosome_t;
11:     (DIRECTIVE) #define MASK 0xFF00
12: (DIRECTIVE) ELSE
13:     typedef uint8_t chromosome_t;
14:     (DIRECTIVE) #define MASK 0xF0
15: (DIRECTIVE) END IF
    ▷ Defition of the matrix X(k)
16: chromosome_t X[N][D];
    ▷ Loop for processing the generations of the GA
17: for k ← 0 to K − 1 do
18:     [...]
19: end for
```

rithm that were more time consuming were those ones that needed to use a function to generate a pseudo-random number, a Pseudo-random Number Generator (PRNG). In the first implementation of this work, the PRNG was implemented using the function `rand`, present in the `stdlib` C standard library, and which generates a pseudo-random integer number (Fre 2018*b*).

The function `rand` is a general-purpose function that can be used for most applications but is not optimized to run on μCs. In most implementations of the C compiler, this function works internally with 16 or 32-bits numbers and it consumes excessive clock cycles on a 8-bits processor, for example. Thus, in order improve this pseudo-random number generation, other alternatives described in the literature were considered and tested for this implementation (DiCarlo 2012, Hissoiny et al. 2011, Steele Jr et al. 2014).

The goal of testing different PRNGs was to find one that could generate a long enough sequence of pseudo-random numbers consuming the less clock cycles as possible on an 8-bits μC. The following PRNG algorithms were experimentally tested using a microcon-

troller Microchip ATmega328P:

- Linear Congruential Generator (LCG);
- Linear Feedback Shift Register Generator (LFSRG);
- Mersenne Twister Generator (MTG);
- Multiply With Carry Generator (MWCG);
- SplitMix Generator (SMG);
- XorShift Generator(XSG).

The results for all PRNG implementations listed above are shown in Table 2.1. After analyzing the results and making some experiments, the most efficient algorithm in the results was the Linear Feedback Shift Register Generator (LFSRG), which consumed fewer clock cycles and less program memory than all other implementations, including the default `rand` function.

Table 2.1: Comparison of different PRNG implementations tested on a microcontroller Microchip ATmega328P to generate a 32-bit pseudo-random number.

RNG	**Clock Cycles**	**Program Memory (bytes)**	**Data Memory (bytes)**
LCG	117	360	4
LFSRG	56	316	4
MTG	10689	1390	256
MWCG	650	910	264
SMG	628	462	4
XSG	295	374	4
stdlib	809	606	4

In order to make the LFSRG more suitable for this work, since the individuals can assume sizes of 8, 16 or 32 bits, it was implemented three optimized versions of this generator. The function `lfsr_rand32()` is used in the case `chromosome_t` needs 32 bits during the population initialization, for example. In the rest of the program it is only used the `lfsr_rand8`, since the constraints ensure the population size is not bigger than 256. Finally, each function was defined with the modifier *inline* so that in exchange of consuming more program memory, where it is possible to save some clock cycles by avoiding the function calling.

- `inline uint32_t lfsr_rand32(void)`, returns a 32-bits pseudo-random number;
- `inline uint16_t lfsr_rand16(void)`, returns a 16-bits pseudo-random number;

- `inline uint8_t lfsr_rand8(void)`, returns a 8-bits pseudo-random number.

Furthermore, it was also implemented three functions responsible for receiving the initial number of the PRNG, usually refereed as the PRNG seed in the literature. This seed can come from an external device such as a sensor, for example, to make sure there is real entropy added to the generator. These functions are shown below and each one must be used together with its respective function previously listed.

- `void_t lfsr_srand32(uint32_t seed)`, receives a 32-bits seed;
- `void_t lfsr_srand16(uint16_t seed)`, receives a 16-bits seed;
- `void_t lfsr_srand8(uint8_t seed)`, receives a 8-bits seed.

2.3.4 Logical Shift

Another part of the implementation that was identified spending a significant amount of clock cycles was the logical shift operation that is used in the mutation operation, as will be explained in Section 2.3.5. During the first tests, the logical shift with 16 or 32-bits numbers was requiring a high number of clock cycles. By definition, the instructions and buses present in an 8-bits µCs are able to deal with 8-bits data. That means that operations with 16 or 32-bits variables takes more than one clock cycle to be completed because they need to be divided into several steps.

These details are relevant because, as explained in the parameters constrains section, the individuals in this GA implementation can be represented in 8, 16 or 32 bits. With that in mind, the assembly code generated after the compilation was analyzed and it was noticed that the logical shift works recursively. This means that when the program needs to shift positions of a 32-bits number, for example, it needs to shift one bit per time successively until it reaches the correct position.

Thus, in order to avoid logical shift operations with 16 or 32-bits numbers, a different strategy was adopted based on the characteristics of the bit shift that appears in the mutation operation, which consists in moving the number 1 to a specific position. That consists oin identifying the byte or octet where the final position of the number 1 will appear and then, to do a simple 8-bits logical shift inside of that byte. For example, by admitting the number 1 needs to be shifted 27 bits to the left, then it would be located in the 4th byte. This is equivalent to shift this number 3 positions inside the 4th byte, for example, as shown in Figure 2.1.

The Algorithm 3 shows how this strategy was implemented. As it can be seen, conditionals directives are used in such way that this strategy comes in only when the GA is

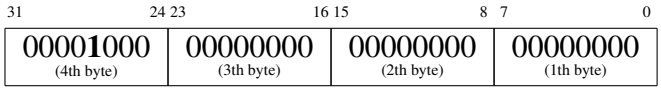

Figure 2.1: Example of logical shift to the left with a 32-bits number.

working with 16 or 32-bits individuals, that is, when $M = 16$ or $M = 32$. Even though this idea uses more instructions and consumes more program memory, it was able to reduce the clock cycles in about 65% in the worst case, that is, when it needs to shift the number 1 to the 31th position.

Algorithm 3 Binary Shift Algorithm

1: **(DIRECTIVE)** IF M == 32
2: **if** $(position < 8)$ **then**
3: $result \leftarrow (0x00000001 \ll position)$
4: **else if** $(position < 16)$ **then**
5: $result \leftarrow (0x00000100 \ll (position - 8))$
6: **else if** $(position < 24)$ **then**
7: $result \leftarrow (0x00010000 \ll (position - 16))$
8: **else**
9: $result \leftarrow (0x01000000 \ll (position - 24))$
10: **end if**
11: **(DIRECTIVE)** ELSE IF M == 16
12: **if** $(position < 8)$ **then**
13: $result \leftarrow (0x0001 \ll position)$
14: **else**
15: $result \leftarrow (0x0100 \ll (position - 8))$
16: **end if**
17: **(DIRECTIVE)** ELSE
18: $result \leftarrow (0x01 \ll position)$
19: **(DIRECTIVE)** END IF

2.3.5 Modularization

In order to make the implementation proposed in this work easy to read and modify, the different steps of the GA were divided into several modules. Additionally, other auxiliary modules were defined as well and each one of them is named as follows:

- Initialization Function Module (IFM);
- Normalization Function Module (NFM);

- Evaluation Function Module (EFM);
- Fitness Function Module (FFM);
- Selection Function Module (SFM);
- Crossover Function Module (CFM);
- Selection and Crossover Function Module (SCFM);
- Mutation Function Module (MFM);
- Update Function Module (UFM);
- New Population Function Module (NPFM).

The architecture that describes the whole GA and how those modules are connected is shown in Fig. 2.2. In this figure, it can be seen the chronological order wherein each module comes in and where the variables $\mathbf{X}(k)$, $\mathbf{y}(k)$, and $\mathbf{Z}(k)$ are used. It is important pointing out that only the reference to these variables are passed through the modules to avoid memory waste. In the next subsections the implementation of each module will be detailed as well as their processing time will be informed.

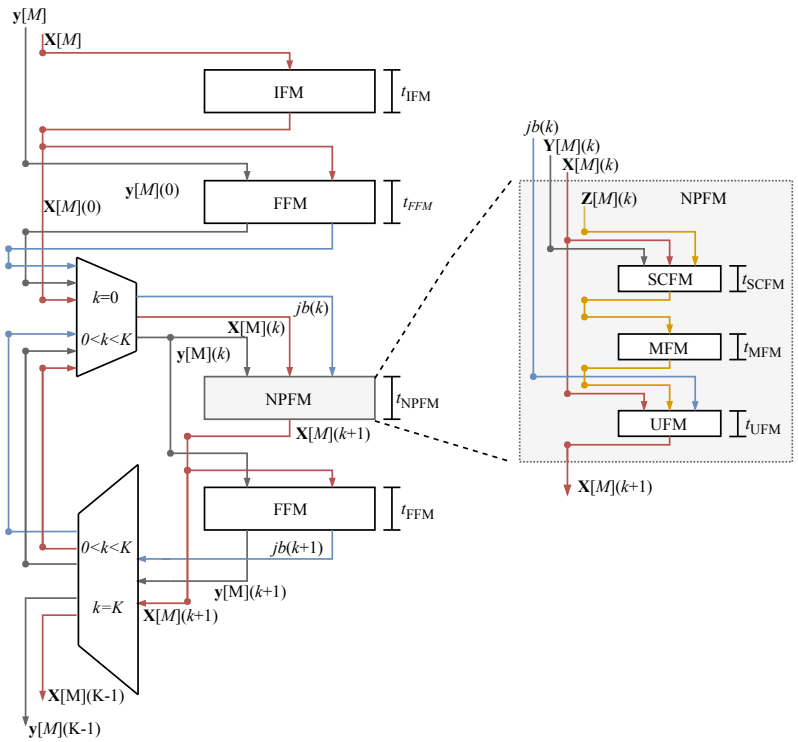

Figure 2.2: Illustration of the GA architecture and its modules proposed in this work.

Initialization Function Module (IFM)

The first module of the program represents the initialization of the population. Its only input is the reference for the matrix $\mathbf{X}(0)$ and its values are initialized with random numbers which follow a uniform distribution. The function RNG-IFM() represents a pseudo-random number generator which produces a M-bits number. The structure of this module is show in Algorithm 4.

Algorithm 4 IFM Algorithm

```
1: function IFM(X(0))
2:     popsize_t j
3:     dimensionsize_t i
4:     for j ← 0 to N − 1 do
5:         for i ← 0 to D − 1 do
6:             x_{j,i}[M](0) ← RNG-IFM( )
7:         end for
8:     end for
9: end function
```

$c_{\text{CLK}}^{\text{atr}M}$ is the number of clock cycles needed by the processor to make an M-bits attribution, $c_{\text{CLK}}^{\text{for}}$ to make an iteration of the for loop, and $c_{\text{CLK}}^{\text{RNG-IFM}}$ for the function generate a M-bits pseudo-random number. Therefore, the total processing time of this module measured in seconds, t_{IFM}, can be defined as

$$t_{IFM} = N \times \left[c_{\text{CLK}}^{\text{for}} + D \times \left(c_{\text{CLK}}^{\text{for}} + c_{\text{CLK}}^{\text{atr}M} + c_{\text{CLK}}^{\text{RNG-IFM}} \right) \right] \times \frac{1}{\text{CLK}}, \tag{2.11}$$

where N is the population size, D the number of dimensions of an individual and CLK is the frequency of operation of the processor measured in Hz.

Normalization Function Module (NFM)

This second module is where the values of the individuals, $x_{j,i}[M]$, are normalized to a real number between the boundaries, $L_{\min}$ and $L_{\max}$, in order to limit the search region. These limits are defined before compiling the program and the normalization can be described as

$$\bar{x}_i(k) = L_{\min} + \left(\frac{L_{\max} - L_{\min}}{2^M - 1} \right) \times x_i(k), \tag{2.12}$$

where $\bar{x}_i(k)$ represents the normalized individual, from the type `normalization_t`, and $x_i(k)$ the original individual, which is defined as a M-bits integer number.

In the Algorithm 5, it can be seen how the normalization module works, so that it is applied to all individual dimensions. Furthermore, it is possible to calculate the processing time for this module as

$$t_{NFM} = D \times \left(c_{\text{CLK}}^{\text{for}} + c_{\text{CLK}}^{\text{atr}G} + c_{\text{CLK}}^{\text{CalcNorm}} \right) \times \frac{1}{\text{CLK}}, \tag{2.13}$$

which is proportional to the number of dimensions, D, the number of clock cycles for a loop iteration, $c_{\text{CLK}}^{\text{for}}$, an attribution of G bits, and the calculus of the normalization, represented by $CalcNorm$. Again, CLK represents the processor frequency.

Algorithm 5 NFM Algorithm

1: **function** NFM($\mathbf{x}_j(k)$, $\bar{\mathbf{x}}(k)$)
2: dimensionsize_t i
3: **for** $i \leftarrow 0$ **to** $D-1$ **do**
4: $\bar{x}_i(k) \leftarrow L_{\min} + (L_{\max} - L_{\min}) \times x_{j,i}(k)/(2^M - 1)$
5: **end for**
6: **end function**

Evaluation Function Module (EFM)

This module represents the evaluation function of the GA, which is the main link between this metaheuristic and the problem which the GA is being applied to. It consists of a function $f(\cdot)$, the evaluation function of an optimization problem, and has as input a normalized individual, $\bar{x}_i(k)$, and produces as output a value of type `fitness_t`. This value represents the fitness degree which an individual (a potential solution) has to the problem which is trying to be optimized. Once it is not possible to know the nature of that function, the Algorithm 6 generically represents it with multiple dimensions.

Algorithm 6 EFM Algorithm

1: **function** EFM($\bar{\mathbf{x}}(k)$)
2: **return** $f(\bar{x}_0[M](k), \ldots, \bar{x}_i[M](k), \ldots, \bar{x}_{D-1}[M](k))$
3: **end function**

The processing time of this module depends solely on the function $f(\cdot)$, and as will be shown in the results, it can consume a lot more clock cycles than any other module. Therefore, the processing time can be specified as

$$t_{EFM} = c_{\text{CLK}}^{f(\cdot)} \times \frac{1}{\text{CLK}}. \tag{2.14}$$

Algorithm 7 FFM Algorithm

```
1:  function FFM(X(k), y(k))
2:      popsize_t j, jb = 0
3:      normalization_t x̄(k)
4:      for j ← 0 to N − 1 do
5:          NFM(x_j(k), x̄(k))
6:          y_j[B](k) ← EFM(x̄(k))
7:          if (y_j[B](k) < y_jb[B](k)) then
8:              jb ← j
9:          end if
10:     end for
11:     return jb
12: end function
```

Fitness Function Module (FFM)

This module is where the evaluation of the entire population is performed by using the problem's evaluation function. In order to do that, it utilizes the two previous modules, NFM and EFM, and its operation is described in Algorithm 7. Essentially, each individual is firstly normalized (Line 5 of Algorithm 7), later evaluated (Line 6 of Algorithm 7), and finally a verification is carried out in order to find the best individual in the population (Line 7 of Algorithm 7). In the end, the index of the best individual, jb, is returned to be used in the elitism.

The processing time of this module depends on several operations, including the processing time of the modules NFM and EFM, t_{NFM} and t_{EFM}, previously defined in Equations 2.13 and 2.14. Thus, it can be defined as

$$t_{FFM} = N \times (c_{\text{CLK}}^{\text{for}} + c_{\text{CLK}}^{\text{atr}B} + c_{\text{CLK}}^{\text{if}B} + c_{\text{CLK}}^{\text{atr8}}) \times \frac{1}{\text{CLK}} + N \times (t_{NFM} + t_{EFM}), \tag{2.15}$$

where N is the population size, CLK is the processor frequency and $c_{\text{CLK}}^{\text{for}}$, $c_{\text{CLK}}^{\text{atr}B}$, $c_{\text{CLK}}^{\text{if}B}$ and $c_{\text{CLK}}^{\text{atr8}}$ are, respectively, the number of clock cycles needed for: a loop iteration; a B-bits attribution; a B-bits conditional evaluation; an 8-bits attribution, which is being considered even tough it happens only when the condition is true.

Selection Function Module (SFM)

As its name suggests, this module is where the selection step occurs. The selection strategy adopted was the tournament selection with two individuals competing between themselves. With this purpose, the module receives as inputs the fitness values of the en-

tire population of the k-th generation, $\mathbf{y}(k)$, and after randomly choosing two individuals, it returns the index of the winner. In other words, it returns the index of the individual with best fitness value and this procedure is presented in Algorithm 8.

The Lines 2 and 3 of the Algorithm 8 select two random indices of individuals (or chromosomes) from the population. These indices must be between 0 and $N-1$, where N is the size of population. The smaller PRNG fuction as discussed in Section 2.3.3 generates an 8-bits number, that is, a number between 0 and 255. Thus, the easiest way to ensure that the produced number remains between 0 and $N-1$ is by using the modulus operator, represented by the symbol %. In other words, the index can be calculated as

$$ich = RNG() \ \% \ (N-1). \tag{2.16}$$

However, since the size of the population N is always a power of 2, the modulus operation, which is computationally costly and needs a significant amount of clock cycles, can be replaced by a bitwise AND operation. This logical operation is represented in Algorithm 8 by the symbol $\wedge$, which is faster to a microcontroller to carry out when compared to the modulus operation.

Algorithm 8 SFM Algorithm

```
1: function SFM(y(k))
2:     popsize_t ichx ← RNG-SFM-1() ∧ (N − 1)
3:     popsize_t ichy ← RNG-SFM-2() ∧ (N − 1)
4:     if (y_ichx[M](k) ≤ y_ichy[M](k)) then
5:         return ichx
6:     else
7:         return ichy
8:     end if
9: end function
```

Due to the constraint of the population size N being less than 256, as described in Section 2.3.1, both functions RNG-SFM-1() and RNG-SFM-2() were implemented using the function `lfsr_rand8()`, since 8-bits are enough to represent all the possible sizes. Therefore, the processing time of this module can be calculated as

$$t_{SFM} = \left[2 \times (c_{\text{CLK}}^{\text{RNG-SFM}} + c_{\text{CLK}}^{\text{atr8}} + c_{\text{CLK}}^{\text{AND}}) + c_{\text{CLK}}^{\text{if}B}\right] \times \frac{1}{\text{CLK}}. \tag{2.17}$$

t_{SFM} is proportional to: the number of clock cycles needed to generate an 8-bits random number, $c_{\text{CLK}}^{\text{RNG-SFM}}$; the clock cycles for an 8-bits attribution, $c_{\text{CLK}}^{\text{atr8}}$; the clock cycles needed for an 8-bits bitwise AND, $c_{\text{CLK}}^{\text{AND}}$; and the clock cycles needed for an 8-bits conditional

operation, $c_{\text{CLK}}^{\text{if}B}$. CLK represents the processor frequency.

Crossover Function Module (CFM)

This module performs the crossover operation, where two individuals previously selected are combined together to generate two new individuals, as can be seen in Algorithm 9. The function receives as parameters the index i from the matrix $\mathbf{Z}(k)$, which represents the new population, so that the newly created two individuals are stored in positions i and $i+1$ from that population. It also receives as parameters the reference for the current population of the k-th generation, $\mathbf{X}(k)$, the reference for the new population, $\mathbf{Z}(k)$ and the indexes of the two individuals from the current population which will be used in the crossover operation, *ichA* and *ichB*.

The crossover is performed using a mask, which is a 8, 16 or 32-bits numerical constant, defined as a macro before the compilation. It is named $mask[M]$ in Algorithm 9, where M is the number of bits to represent a dimension of an individual and it is defined as

$$mask[M] = [mask_0, mask_1, \ldots, mask_{M-1}]. \tag{2.18}$$

Depending on how the mask is configured, it is possible to do different types of crossovers such as one-point, two-point, and uniform. These three possibilities are shown below:

- One-point: $mask_j = 1$ for the interval $j = 0$ to $M/2-1$ and $mask_j = 0$ for the interval $j = M/2$ to $M-1$;
- Two-point: $mask_j = 1$ for the interval $j = 0$ to $M/4-1$ and $j = 3M/4$ to $M-1$; and $mask_j = 0$ for the interval $j = M/4$ to $3M/4-1$;
- Uniform: $mask[M]$ defined as desired.

Algorithm 9 CFM Algorithm

```
1: function CFM(i, X(k), Z(k), ichA, ichB)
2:     dimensionsize_t t
3:     for t ← 0 to D − 1 do
4:         z_{i,t}[M](k) ← (x_{ichA,t}[M](k) ∧ mask[M]) ∨ (x_{ichB,t}[M](k) ∧ ¬mask[M])
5:         z_{i+1,t}[M](k) ← (x_{ichA,t}[M](k) ∧ ¬mask[M]) ∨ (x_{ichB,t}[M](k) ∧ mask[M])
6:     end for
7: end function
```

The processing time of this module is

$$t_{CFM} = D \times \left[c_{\text{CLK}}^{\text{for}} + 2 \times \left(c_{\text{CLK}}^{\text{atr}M} + c_{\text{CLK}}^{\text{OR}M} + 2 \times c_{\text{CLK}}^{\text{AND}M} + c_{\text{CLK}}^{\text{Neg}M} \right) \right] \times \frac{1}{\text{CLK}}. \tag{2.19}$$

This time is influenced by the number of clock cycles needed for a loop iteration, $c_{\text{CLK}}^{\text{for}}$, by the clock cycles needed to perform an M-bits attribution, $c_{\text{CLK}}^{\text{atr}M}$, by the clock cycles for an M-bits bitwise OR operation, $c_{\text{CLK}}^{\text{OR}M}$, which is represented by the symbol $\vee$, by the clock cycles required for an M-bits bitwise AND operation, $c_{\text{CLK}}^{\text{AND}M}$, which is represented by the symbol $\wedge$ and by the clock cycles for necessary for an M-bits bitwise NOT operation, $c_{\text{CLK}}^{\text{Neg}M}$, represented by the symbol $\neg$. As before, CLK represents the processor frequency.

Selection and Crossover Function Module (SCFM)

The purpose of this module is to produce a new offspring to substitute the previous population. For each iteration, it performs a crossover of two individuals previously selected using the two preceding modules, SFM and CFM. As shown in the Algorithm 10, the module receives as inputs: the fitness values of the population, $\mathbf{y}(k)$, at the k-th generation; the current population, $\mathbf{X}(k)$; and the new population to be generated, $\mathbf{Z}(k)$. It can be noticed that the loop runs only $\frac{N}{2}$ times (the loop uses step 2), since the crossover generates two individuals at every execution.

Algorithm 10 SCFM Algorithm

```
1: function SCFM(y(k), X(k), Z(k))
2:     popsize_t i
3:     for i ← 0 to N − 1 with step 2 do
4:         CFM(i, X(k), Z(k), SFM(y(k)), SMF(y(k)))
5:     end for
6: end function
```

The processing time of this module is defined as

$$t_{\text{SCFM}} = \frac{N}{2} \times \left[\left(c_{\text{CLK}}^{\text{for}} \times \frac{1}{\text{CLK}} \right) + t_{\text{CFM}} + 2 \times t_{\text{SFM}} \right], \tag{2.20}$$

which depends mainly on the processing time of the modules SFM and CFM, t_{CFM} and t_{SFM}, and the clock cycles needed for a loop iteration. CLK represents the processor frequency.

Mutation Function Module (MFM)

This is the module in which the mutation operation occurs as described in Algorithm 11. In this operation, P individuals mutate as explained in Section 2.2. The module has as input the matrix that represents the new population, $\mathbf{Z}(k)$, which is filled with individuals generated after the selection and crossover operations. In order to simplify and optimize

the mutation, it was defined that the individuals $\mathbf{z}_j[M](k)$ from $j = 1$ to $j = P$ are submitted to this transformation. This is possible since P can be calculated from the mutation rate, defined in Equation 2.4.

The chosen strategy for the mutation process was to change of one bit in all dimensions of all P individuals. The position of this bit, o, is randomly chosen by the function RNG-MFM (Line 5 of the Algorithm 11), which is implemented using the function `lfsr_rand8()` because the highest possible position is 31, since M must be 8, 16 or 32. Again, for the same reason explained in Section 2.3.5, the modulus operation was replaced by a bitwise AND. Finally, the value modification of the chosen bit is performed with a logical shift and the bitwise XOR operation (Line 7 of the Algorithm 11).

Algorithm 11 MFM Algorithm

```
1:  function MFM(Z(k))
2:      popsize_t j
3:      dimensionsize_t i
4:      for j ← 1 to P do
5:          chromosomesize_t o ← RNG-MFM() ∧ (M − 1)
6:          for i ← 0 to D − 1 do
7:              z_{j,i}[M](k) ← z_{j,i}[M](k) ⊕ (1 ≪ o)
8:          end for
9:      end for
10: end function
```

The calculation of the processing time of this module is presented as

$$t_{\text{MFM}} = P \times \left[c_{\text{CLK}}^{\text{for}} + c_{\text{CLK}}^{\text{atr8}} + c_{\text{CLK}}^{\text{RNG-MFM}} + c^{\text{AND8}} + D \times \left(c_{\text{CLK}}^{\text{for}} + c_{\text{CLK}}^{\text{atr}M} + c_{\text{CLK}}^{\text{XOR}M} + c_{\text{CLK}}^{\text{Shift}M} \right) \right] \times \frac{1}{CLK}, \tag{2.21}$$

where P is the number of mutated individuals, D is the number of dimensions of the individual and CLK is the processor clock. Additionally, the following variables represent that amount of clocks needed to: a loop iteration ($c_{\text{CLK}}^{\text{for}}$); an 8-bits attribution ($c_{\text{CLK}}^{\text{atr8}}$); generate an 8-bits random number ($c_{\text{CLK}}^{\text{RNG-MFM}}$); to carry out an 8-bits bitwise AND (c^{AND8}); to a M-bits attribution ($c_{\text{CLK}}^{\text{atr}M}$); to carry out a M-bits bitwise XOR operation ($c_{\text{CLK}}^{\text{XOR}M}$); and to a M-bits logical shift ($c_{\text{CLK}}^{\text{Shift}M}$).

Update Function Module (UFM)

This module represents the last stage of the genetic algorithm. It is in this module that the old population of the k-th generation, $\mathbf{X}(k)$, is updated with new individuals stored in $\mathbf{Z}(k)$ in order to create the population of the $k+1$ generation, $\mathbf{X}(k+1)$. This procedure is

presented in Algorithm 12.

In addition to the references for the matrices $\mathbf{X}(k)$ and $\mathbf{Z}(k)$, this module takes also as input the index of the best individual of the current generation, jb. This last input is used by the elitism feature, where the individual $x_{jb,i}[M](k)$ is forwarded directly to the first position of the population of next generation, as can be seen in Line 5 of the Algorithm 12. In order to complete the population of the $k+1$ generation the new individuals stored in $\mathbf{Z}(k)$ are copied to the the matrix $\mathbf{X}(k)$.

Algorithm 12 UFM Algorithm

```
1:  function UFM(jb, X(k), Z(k))
2:      popsize_t j
3:      dimensionsize_t i
4:      for i ← 0 to D − 1 do
5:          x_{0,i}[M](k) ← x_{jb,i}[M](k)
6:      end for
7:      for j ← 1 to N − 1 do
8:          for i ← 0 to D − 1 do
9:              x_{j,i}[M](k) ← z_{j,i}[M](k)
10:         end for
11:     end for
12: end function
```

The processing time of this module is calculated as

$$t_{UFM} = \{D \times (c_{\text{CLK}}^{\text{for}} + c_{\text{CLK}}^{\text{atr}M}) + N \times [c_{\text{CLK}}^{\text{for}} + D \times \left(c_{\text{CLK}}^{\text{for}} + c_{\text{CLK}}^{\text{atr}M}\right)]\} \times \frac{1}{\text{CLK}}. \quad (2.22)$$

t_{UFM} is proportional to: the size of the population, N; the number of dimensions of the individuals (or chromosomes), D; and the clock cycles for a loop iteration, $c_{\text{CLK}}^{\text{for}}$; the clocks cycles for a M-bits attribution, $c_{\text{CLK}}^{\text{atr}M}$. CLK is the clock of the processor.

New Population Function Module (NPFM)

This is the last module proposed in this work and its purpose is to encapsulate the three previous modules, SCFM, MFM and UFM. It is also here that the life cycle of the matrix $\mathbf{Z}[M](k)$ occurs, which stores temporarily the individuals for the new population. In other words, it is where that matrix is allocated in memory and later freed. The structure of this module is presented in Algorithm 13 and its inputs are: a reference to the vector $\mathbf{y}(k)$, which stores the fitnesses of the individuals; a reference to the matrix $\mathbf{X}[M](k)$, which stores the current population and the index of the best individual of the current generation, $jb(k)$.

Algorithm 13 NPFM Algorithm

1: **function** NPFM($\mathbf{y}(k)$, $\mathbf{X}[M](k)$, $jb(k)$)
2: chromosome_t $\mathbf{Z}[M](k)$
3: SCFM $(\mathbf{y}(k), \mathbf{X}(k), \mathbf{Z}(k))$
4: MFM $(\mathbf{Z}(k))$
5: UFM $(\mathbf{X}(k), \mathbf{Z}(k), jb(k))$
6: **end function**

The processing time consists in the sum of the times spent by the modules SCFM, MFM and UFM, that is, t_{SCFM}, t_{MFM}, and t_{UFM}. The expressions for calculating each one of these was previously defined. The final expression can be calculated as

$$t_{NPFM} = t_{SCFM} + t_{MFM} + t_{UFM}. \tag{2.23}$$

2.4 Results

To validate the genetic algorithm implementation proposed in Section 2.3 and evaluate its operation as well as its performance and resources consumption, a program was developed using the C programing language targeting Atmel microcontrollers. The whole application was written completely in a high-level language to aim portability because the use of assembly instructions are architecture dependant. Hence, this implementation can be compiled for several different targets as long as there is a C compiler available for that architecture.

The development was performed using the software Atmel Studio 7, provided by Atmel. The code strictly followed the algorithms presented for each module, so that each one was defined as a C function. In addition, programming recommendations provided by Atmel had also been taken into account in order to generate a satisfactory code (Mic 2018*b*). The source code used in this project to generate the results can be accessed in this public repository: `https://github.com/DenisMedeiros/EmbeddedGeneticAlgorithms` (Medeiros et al. 2020*a*).

The microcontroller used to validate this implementation was the ATmega328P, which uses AVR architecture. According to the manufacturer, the chip is an 8-bits µC and has the potential of 1 Millions Instructions per Second (MIPS) per MHz and can reach a peak of 20 MIPS running with a clock of 20 MHz. Moreover, it has 32 KB of program memory and 2 KB of data memory (Mic 2018*a*).

Even though µCs have become robust, many of them running 32-bits instructions in modern architectures such as ARM and with more program and data memories, the choice

of the Atmega328p was intentional. The aim is to use the genetic algorithm proposed in this work on a simple and light microcontroller, which usually has lower price and consumes less power. This is more suitable for many applications, such as those applied in the Internet of Things, for example. Once it is validated in a simple and limited platform such as this μC, it can be configured to run in more complex ones as well.

It is necessary to mention that the genetic algorithm was implemented as an embedded system using the development kit Arduino Uno, which uses the ATmega328P. Besides the microcontroller, the Arduino platform comes with a built-in programming interface to write the program into the flash memory and with other interfaces which help the development and test of the system (Ard 2018).

Finally, all the charts presented in this work were generated using the Python library Matplotlib (Hunter 2007).

2.4.1 Resources Consumption

The data collection for the information presented in this section was performed in the following way:

- For program and data memory consumption, it was used reports generated by the Atmel Studio 7 during the compilation and during the upload of the program into the microcontroller;
- For processing time, it was used the Atmel Studio 7 which can measure the clock cycles during the debugging as well as oscilloscope observations while the μC was running the code.

For all measurements, the system was compiled using the optimization flag -O2, which makes the compiler optimize its processing time. On the other hand, the user can change this flag to -Os in order to optimize the size of the program, although, as will be shown later, the program size was already small enough.

Memory Consumption

The first results were the program and data memory consumption. The program memory is the non-volatile memory where the compiled program is stored to be executed by the microcontroller. The data memory is the the dynamic and volatile memory where the variables are stored during when the program is running. The way how this data memory is used depends on how the variables are defined:

- Static memory: the memory consumed by global and static variables and it is allocated during the whole program execution. That means this section of the memory cannot be freed and used by other variables.
- Stack memory: the memory used by local variables and that can be allocated and freed according to the lifetime of those variabes (for example, a local variable defined inside of a function will be freed after the function is finished).

After the program compilation, the Atmel Studio 7 reports the program memory necessary to store it as well as the initial data memory used by global and static variables, that is, the static memory. Nevertheless, to measure the maximum stack memory consumed by the program during its execution (the peak of consumption), it was used a different methodology since this memory is used in a dynamic way. Since it is known that the word size of the ATmega328P architecture is 1 byte, then the maximum data memory consumption can be calculated as follows:

1. Find the start and end address of the stack in memory;
2. Before the execution of the program, mark all stack addresses with a default mask (0xC5, for example);
3. Carry out the program;
4. Count the amount of bytes which are still with that mask and, based on this, calculate the maximum number of bytes used during the running.

Therefore, the program and data memory consumption are presented in Tables 2.2-2.4 and Figures 2.3-2.5 for different configurations of the GA, respectively. In order to simplify the measurements, all simulations were performed with a fixed number of generations $K = 64$, since this affects only the processing time. Also, the evaluation function used was $f_1(\bar{\mathbf{x}}) = (x-2) \times (x-4)$, with dimension $D = 1$, to avoid the use of external libraries, such as `libm`, which may be necessary for some mathematical operations such as trigonometric functions. Finally, the crossover was configured as one-point and the number of mutated individuals was $P = 1$.

As presented in Table 2.2, the program memory was significantly low. The compiled code averaged about 3.5 KB. This represents approximately 11% of the ATmega328P capacity and was considerably smaller than the results presented by (Dendaluce et al. 2014), where their implementation consumed, in the best case, 78.4 KB.

Regarding the Table 2.3, the initial data memory consumption was almost null. This can be explained by the avoidance of the use of global and static variables, since the memory used by them is retained during all execution, even if these variables are barely used. Thus, in this work the whole data memory is available to be used as stack.

Table 2.2: Program memory consumption (in bytes), of a total of 32 KB.

Individual Size (M)	Population Size (N)			
	16	32	64	128
8-bits	3212	3212	3216	3216
16-bits	3412	3416	3416	3410
32-bits	3678	3678	3670	3670

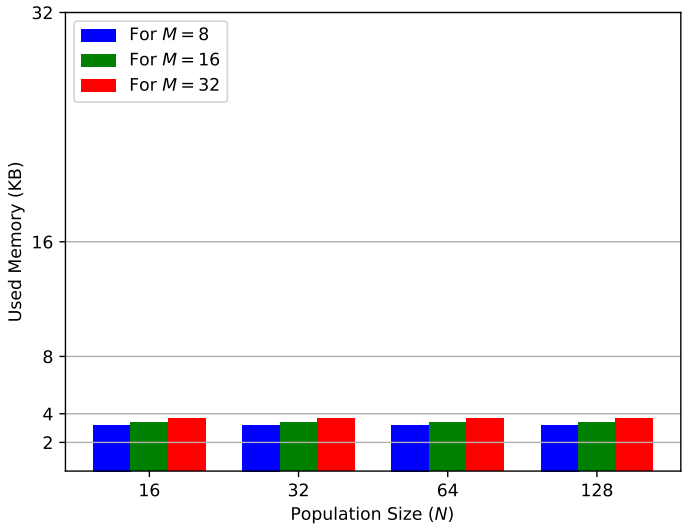

Figure 2.3: Comparison of program memory consumption (in bytes) for different values of population size, N.

Table 2.3: Static memory consumption (in bytes), of a total of 2 KB.

Individual Size (M)	Population Size (N)			
	16	32	64	128
8-bits	4	4	4	4
16-bits	4	4	4	4
32-bits	8	8	8	8

The most important data memory consumption analysis is presented in Table 3.3, plotted in Figure 2.6, since it shows how much it is consumed from the stack after executing the genetic algorithm. In the simplest case, with 8 individuals represented in 8-bits, the program consumed 143 bytes or about 7% of the data memory of the ATmega328P. In a practical situation with 64 individuals represented in 16-bits, it was consumed 558 bytes

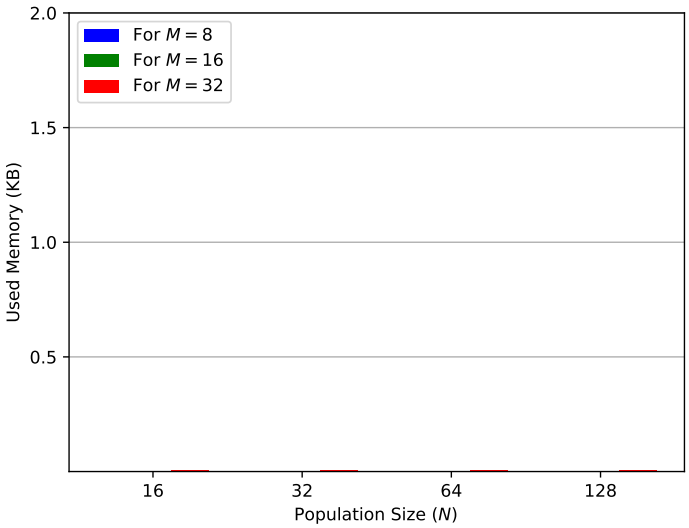

Figure 2.4: Comparison of static memory consumption (in bytes) for different values of population size, N.

or about 27% of the memory. Finally, with 128 individuals represented in 32-bits, about 80% of the whole memory was consumed.

Table 2.4: Stack memory consumption (in bytes), of a total of 2 KB.

Individual Size (M)	Population Size (N)			
	16	32	64	128
8-bits	143	239	431	815
16-bits	174	302	558	1070
32-bits	242	434	818	1586

Looking at these results, both the sizes of population and individuals increase the consumption of data memory, but increasing N is more memory consuming, since its growth is almost linear. Furthermore, it can be inferred that the number of dimensions D must produces a growth on the memory consumption analogous to N, because it is equivalent to multiply the population size by D.

Therefore, it is important to choose the correct genetic algorithm configuration and take into consideration the memory limitations of the microcontroller. The parameter that affects the most the memory consumption is the population size N. Also, most applications will require more or less precision, which affects the size of M. Hence, both parameters must be adjusted according to the problem requirements in order to make the

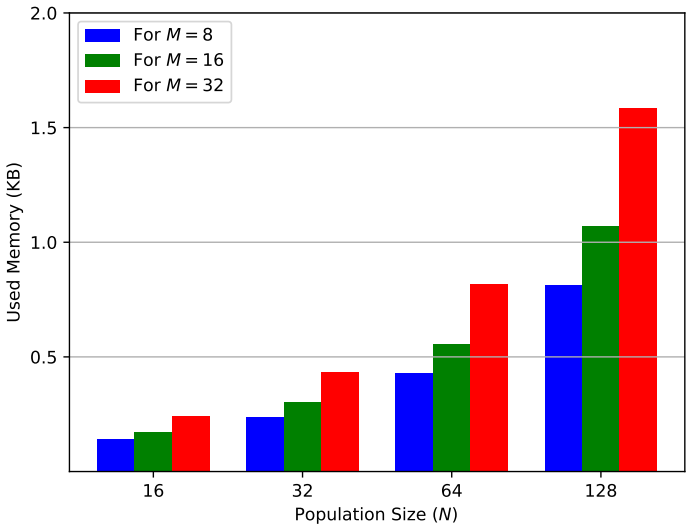

Figure 2.5: Comparison of stack memory consumption (in bytes) for different values of population size, N, with linear interpolated functions for the values of memory.

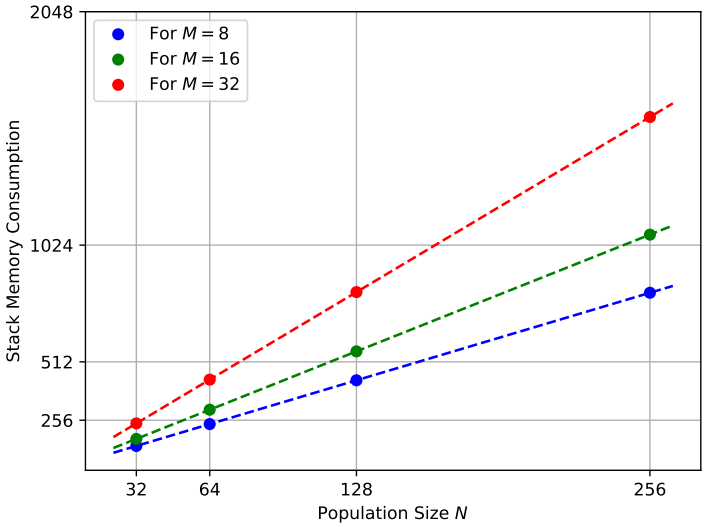

Figure 2.6: Stack memory consumption (in bytes) for different values of population size, N.

GA implementation feasible.

Processing Time

The first part of the processing time results was collected from the reports generated by the Atmel Studio 7 during the debugging process. The objective was to analyze the number of clock cycles consumed by each module from this implementation in order to identify critical parts of the program, that is, which sections are more costly. With the purpose of calculating the processing time, t, in seconds, one must divide the number of clock cycles, c_{CLK}, by the processor clock, which in the Arduino Uno development board is 16 MHz. Hence, this is calculated as

$$t(s) = \frac{c_{\text{CLK}}}{16 \times 10^6}. \tag{2.24}$$

The first measurements were performed with the simplest possible configuration of the GA: the population size was $N = 2$; number of generations $K = 1$; number of dimensions $D = 1$; number of mutated individuals (or chromosomes) $P = 1$; the EFM was set as the same function used in the previous section, that is, $f_1(\bar{\mathbf{x}})$. Moreover, it was not considered any processing time of a module inside another (such as SFM and CFM inside of NPFM, for instance), since each one was measured separately. Finally, the only changed parameter was the individual size, M, since it affects the pseudo-random number generator.

The Table 2.5 displays the clock cycles and processing time consumed by each module, respectively. Although the EFM used a simple function, it was the module responsible for the longest processing time. This can be justified by the fact that the evaluation function makes mathematical operations using float point numbers. This explains the reason why the NFM also spends numerous clock cycles, as it needs to evaluate the expression shown in Equation 2.12 with float point numbers as well. Finally, other modules that also stand out are the IFM and SFM, due to their need to use the PRNG, as explained in Algorithms 4 and 8.

The processing time of the entire GA, presented in Table 2.6, is greater than the sum of the times of all above modules. This can be explained by the existence of an extra overhead while invoking each function and also caused by the memory allocation of local variables in the stack. Furthermore, some modules are executed more than once, such as EFM and NFM, as they are applied to the whole population.

In addition to the previous result, it was also measured the number of clock cycles for other evaluations functions, which are commonly used in the literature to evaluate metaheuristics. The purpose is to show how costly these functions can be in comparison to the entire GA, even in a simple simulation as performed before. These functions are

Table 2.5: Clock cycles consumed by each module of the GA.

Module	Individual Size (M) 8-bits		16-bits		32-bits	
	$c_{\mathbf{CLK}}$	$t(\mu s)$	$c_{\mathbf{CLK}}$	$t(\mu s)$	$c_{\mathbf{CLK}}$	$t(\mu s)$
IFM	56	3.50	74	4.62	128	8.00
NFM	305	19.0	297	18.6	353	22.1
EFM	386	24.1	379	23.7	373	23.31
FFM	222	13.9	252	15.8	253	15.81
SFM	142	8.88	146	9.12	141	8.81
CFM	59	3.69	74	4.62	127	7.94
SCFM	13	0.812	31	1.94	36	2.25
MFM	63	3.94	72	4.50	84	5.25
UFM	29	1.81	40	2.50	65	4.06
NPFM	37	2.31	42	2.63	78	4.88

Table 2.6: Clock cycles consumed by the entire GA.

Individual Size (M) 8-bits		16-bits		32-bits	
$c_{\mathbf{CLK}}$	$t(\mu s)$	$c_{\mathbf{CLK}}$	$t(\mu s)$	$c_{\mathbf{CLK}}$	$t(\mu s)$
3766	235	3845	240	4202	263

listed below and it is important to mention that only $f_4(\bar{\mathbf{x}})$ has one dimension while $f_2(\bar{\mathbf{x}})$, $f_3(\bar{\mathbf{x}})$ and $f_5(\bar{\mathbf{x}})$ have two. The reason to choose these functions is that they are usually used to test metaheuristics and optimization algorithms (Pohlheim 2007).

- $f_2(\bar{\mathbf{x}}) = \bar{x}_0^2 + \bar{x}_1^2$
- $f_3(\bar{\mathbf{x}}) = 2\bar{x}_0 + 3\bar{x}_1 + 5$
- $f_4(\bar{\mathbf{x}}) = 21.5 + \bar{x}_0 \left(\sin\left(40\pi\bar{x}_0\right) + \cos\left(20\pi\bar{x}_0\right)\right)$
- $f_5(\bar{\mathbf{x}}) = 10 + \bar{x}_0^2 - 10\cos\left(2\pi\bar{x}_0\right) + \bar{x}_1^2 - 10\cos\left(2\pi\bar{x}_1\right)$

The Table 2.7 shows the number of clock cycles and time required by each evaluation function (or fitness function). The execution of either of these last two functions, regardless of having one or two dimensions, consume more processing time that the whole genetic algorithm, as shown in Table 2.6. In fact, for most metaheuristics the evaluation function is the most critical part. In these particular examples, the need of calculating trigonometric functions was sufficient to make them overly costly.

In order to validate the time estimations based on the number of clock cycles, an oscilloscope was used to measure the real processing time. Hence, the implementation

Table 2.7: Clock cycles requires by other evaluation functions (or fitness functions).

$f(\bar{\mathbf{x}})$	**Chromosome Size (M)**					
	8-bits		**16-bits**		**32-bits**	
	$c_{\mathbf{CLK}}$	$t(\mu s)$	$c_{\mathbf{CLK}}$	$t(\mu s)$	$c_{\mathbf{CLK}}$	$t(\mu s)$
$f_2(\bar{\mathbf{x}})$	392	24.5	366	23.8	365	22.8
$f_3(\bar{\mathbf{x}})$	428	26.8	441	27.6	433	27.1
$f_4(\bar{\mathbf{x}})$	4259	266	4409	276	4377	274
$f_5(\bar{\mathbf{x}})$	4824	302	4960	310	4717	294

was slightly modified so that for each execution of the GA or of some specific part of the program, the microcontroller generated an oscillating digital output signal with low and high values on a GPIO pin. In other words, it generated a square wave in which the half of the period represented the total time needed to execute that part of the program. This idea is presented in Algorithm 14.

Algorithm 14 Algorithm used to generate the signal measured by oscilloscope.

```
1: while true do
2:     Execute the GA or part of the program.
3:     Toggle the output of a specific GPIO pin.
4: end while
```

The Tables 2.8 and 2.9 show the measured time for the entire GA and for the evaluation functions presented before, respectively. They show the processing time, t, in microseconds, and the relative error comparing to the estimations using clock cycles and the Equation 2.24. Looking at the presented values, the relative error is low for almost all cases, even when considering the oscilloscope imprecision and the signal toggle settling time. Therefore, this proves that those estimations are fair and reasonable.

Table 2.8: Processing time of the GA using oscilloscope and relative error in comparison with Table 2.6.

Individual Size (M)					
8-bits		**16-bits**		**32-bits**	
$t(\mu s)$	**error (%)**	$t(\mu s)$	**error (%)**	$t(\mu s)$	**error (%)**
240	1.75	248	3.20	270	2.81

In order to conclude the processing time investigation, a few simulations were conducted with the purpose of understanding the time complexity of the implemented GA,

Table 2.9: Processing time of different evaluations functions (or fitness functions) using oscilloscope and relative error in comparison with Table 2.7.

$f(\bar{\mathbf{x}})$	**Individual Size (M)**					
	8-bits		**16-bits**		**32-bits**	
	$t(\mu s)$	**error (%)**	$t(\mu s)$	**error (%)**	$t(\mu s)$	**error (%)**
$f_2(\bar{\mathbf{x}})$	24.8	1.22	23.2	1.42	23.6	3.45
$f_3(\bar{\mathbf{x}})$	27.9	4.49	28.2	2.13	26.9	0.60
$f_4(\bar{\mathbf{x}})$	270	1.43	278	0.88	276	0.89
$f_5(\bar{\mathbf{x}})$	306	1.49	314	1.29	298	1.08

according to the parameters configuration. For these results, the analyzed parameters were the population size, N, and the number of generations, K. The individuals were set with number of dimensions $D = 1$, represented in $M = 16$ bits; the number of mutated individuals was fixed as $P = 2$; and the evaluation function was kept as $f_1(\bar{\mathbf{x}})$. These results were obtained using the Atmel Studio 7 and are shown in Table 2.10.

Table 2.10: GA processing time in *ms* for different configurations of N and K.

Number of Generations (K)	**Population Size (N)**			
	16	**32**	**64**	**128**
16	16.5	32.2	65.0	129
32	32.0	62.7	126	252
64	63.2	123	249	496
128	126	246	494	986

To illustrate the results shown in Table 2.10, the data was plotted on a chart in Fig. 2.7 and Fig. 2.8. The points represent the measured values and the dashed lines represent the best polynomial approximation. Therefore, as for the growth of the population size N and as for the growth of the number of generations K the processing time complexity was observed approximately linear.

From the processing time results, it can be inferred that all modules are optimized and the EFM, which represents the evaluation function and can be quite complex, is the critical part of the GA. Also, the NFM must be considered a secondary critical part, since it performs mathematical operations with float point numbers and it is applied to the entire population in all generations, similarly to the EFM. Specifically, both EFM and NFM are executed $K \times N$ times during the GA. One option to reduce the processing time of those modules would be to use fixed-point arithmetic or to spend more memory to use lookup

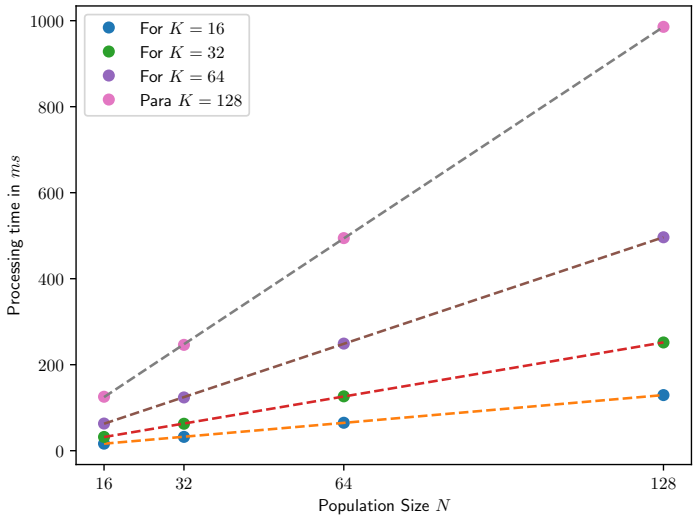

Figure 2.7: GA processing time for different values of population size, N.

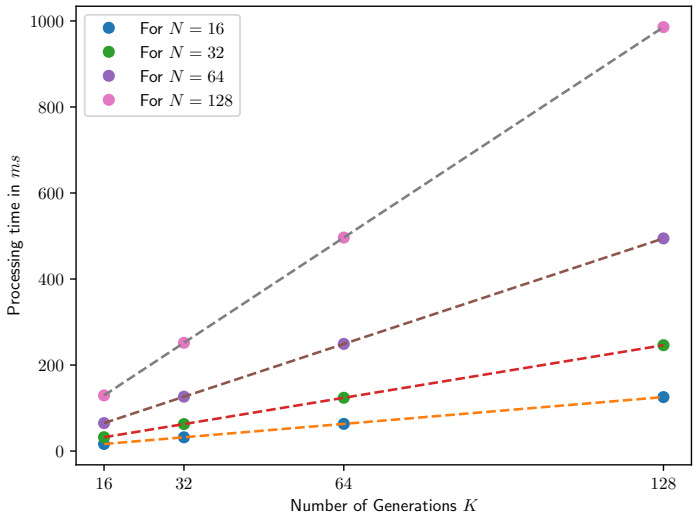

Figure 2.8: GA processing time for different value of number of generations, K.

tables (LUTs) instead to represent both functions or, at least, parts of them.

Finally, in most simulations the whole GA could be executed in few milliseconds. This makes it feasible to be used in several problems such as real-time applications or in areas such as robotics, industrial automation, automotive applications, etc. However, to make this possible, the GA needs to be properly tuned regarding its parameters so that it

can meet the time requirements and memory limitations.

2.4.2 Validation with hardware-in-loop

In order to verify whether the genetic algorithm was working as expected, experiments using the Hardware-in-the-loop technique were performed. In this model, the microcontroller is linked to the computer so that the program is executed on the µC and they can share data, such as parameter values and results. During the experiments, the communication between both devices was set via Universal Asynchronous Receiver/ Transmitter (USART) using a Python script, which was used to receive data (the best individual for each generation) from the µC and to generate the charts.

The first experiment consisted in finding the global minimum of the function $f_2(\bar{\mathbf{x}})$ and the second one focused on finding a local maximum, between 0.8 and 1.0, of $f_4(\bar{\mathbf{x}})$. The charts of both functions are presented in Fig. 2.9 and Fig. 2.10, respectively. Looking at them, it can be noticed that the global minimum of $f_2(\bar{\mathbf{x}})$ is in the point $(x_0 = 0, x_1 = 0)$ and the local maximum of $f_4(\bar{\mathbf{x}})$ is located around $x = 0,910$.

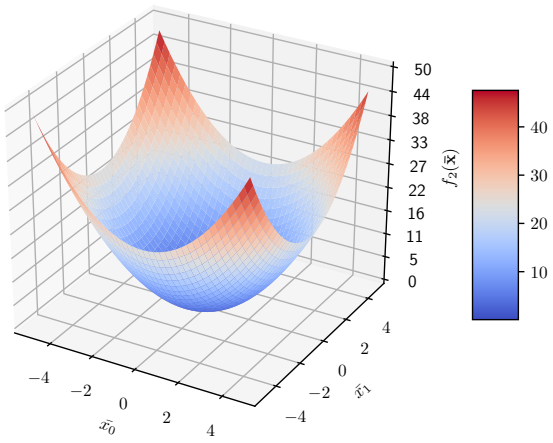

Figure 2.9: Chart showing values of $f_2(\bar{\mathbf{x}})$, with x_0 and x_1 from -5 to 5.

For the function $f_2(\bar{\mathbf{x}})$, the GA was configured as follows: $N = 32$ individuals, $D = 2$ dimensions, $M = 16$ bits, number of generations $K = 32$, $P = 2$ mutated individuals and normalization limits $L_{\text{min}} = -4.0$ and $L_{\text{max}} = 4.0$. The results of this experiment are shown in Fig. 2.11 and Fig. 2.12. Each dimension converged at different moments to the

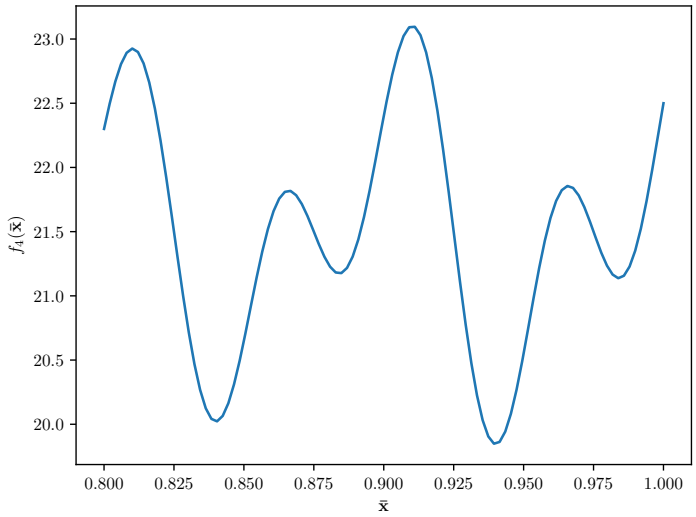

Figure 2.10: Chart showing values of $f_4(\bar{\mathbf{x}})$, with x from 0.8 to 1.0.

expected result, but x_0 needed slightly more iterations to accomplish it.

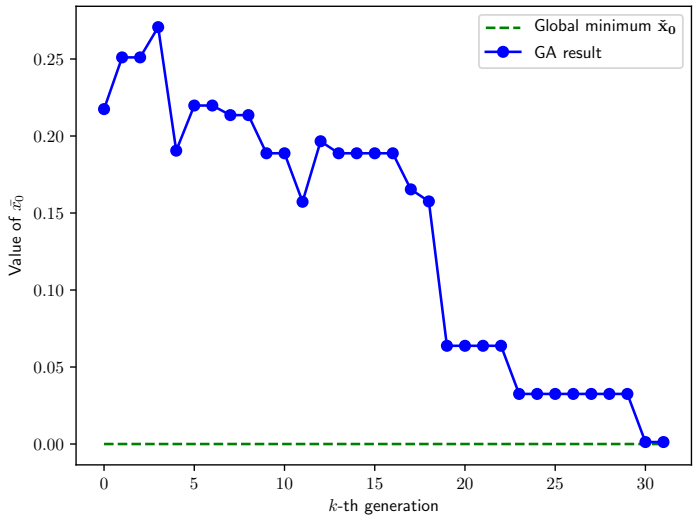

Figure 2.11: HIL result showing the convergence of coordinate x_0 of $f_2(\bar{\mathbf{x}})$.

For the function $f_4(\bar{\mathbf{x}})$, the GA was configured as follows: $N = 16$ individuals, $D = 1$ dimensions, $M = 16$ bits, number of generations $K = 32$, $P = 2$ mutated individuals and normalization limits $L_{\text{min}} = 0.8$ and $L_{\text{max}} = 1.0$. The result for this function is presented

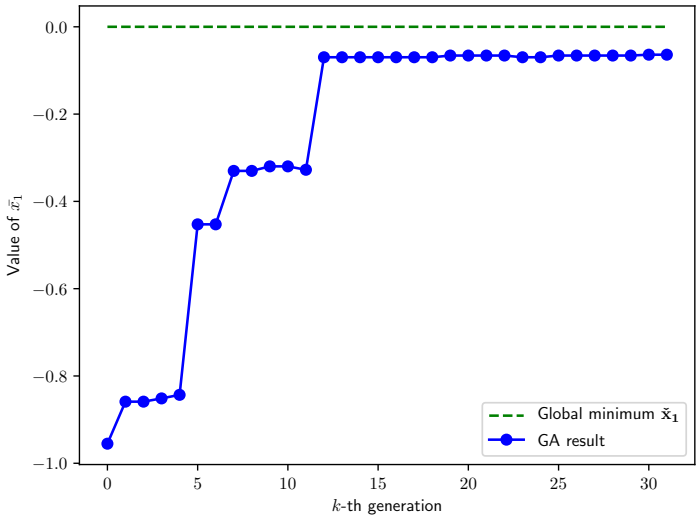

Figure 2.12: HIL result showing the convergence of the coordinate x_1 of $f_2(\bar{\mathbf{x}})$.

in Fig. 2.13. As can be seen in it, after few generations the GA converged to the correct result.

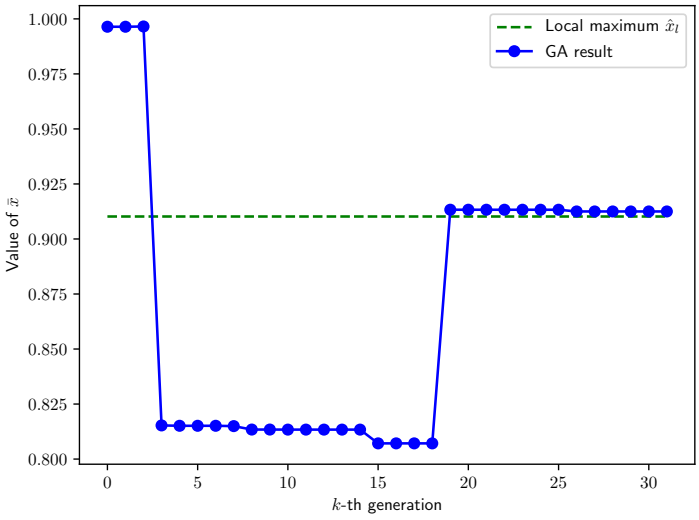

Figure 2.13: HIL result showing the convergence of x for $f_4(\bar{\mathbf{x}})$.

2.4.3 Comparison with other implementation

As was stated in the introduction, there are only few works in the literature with a similar objective to the one presented in this work, that is, to propose a GA implementation targeting 8-bits microcontrollers. Also, the majority of those works did not present detailed results about memory consumption and processing time nor explained how the GA was implemented. Thus, the only comparison that could be conducted was with the implementation proposed by (Alves 2010), since the source code of the implementation was provided.

The implementation presented by (Alves 2010) used the C++ programing language and the Arduino IDE. The author used several libraries provided by the development platform to implement his project. This publication did not present results about memory consumption and processing time, but the author pointed that his implementation had the following constraints:

- The population size is less than 100;
- The individuals are represented in $M = 32$-bits;
- The fitness value is represented between 0 and 100.

In order to validate that implementation, the source code had to be slightly changed in order to be compiled and debugged on Atmel Studio 7. A fuction `main`, which is not required by the Arduino IDE, had to be created with the purpose to call the functions `setup` and `loop` that exist by default in programs developed using that tool. Moreover, the population size, N, and the number of generations, K, were configured to 64 so a fair comparison with the present work could be performed. Regarding the mutation operation, the compared work used a mutation rate of 0.001 and the original goal is to maximize the function $f(x) = x$, where x is a 32-bits unsigned integer number. With this purpose of making this evaluation, the respective function from (Alves 2010) counted the amount of bits with value 1 in an individual, which means that the one with the most bits 1 is considered the best. Finally, all the commands from the analyzed source code that generated reports via USART, such as in the piece containing `ga.reportStatistics(generation,0)`, were disabled in an effort to produce fair results.

After making these adjustments, both implementations were configured with the exact same parameters and several simulations using the Atmel Studio 7 were performed for the comparison of both. In order to illustrate that the present implementation was able to maximize the target function, an experiment using the HIL approach was performed and

the result is shown in Fig. 2.14. As can be seen, after approximately 50 generations the GA converged to close to the correct answer, which is 2^{32} or 4,294,967,296.

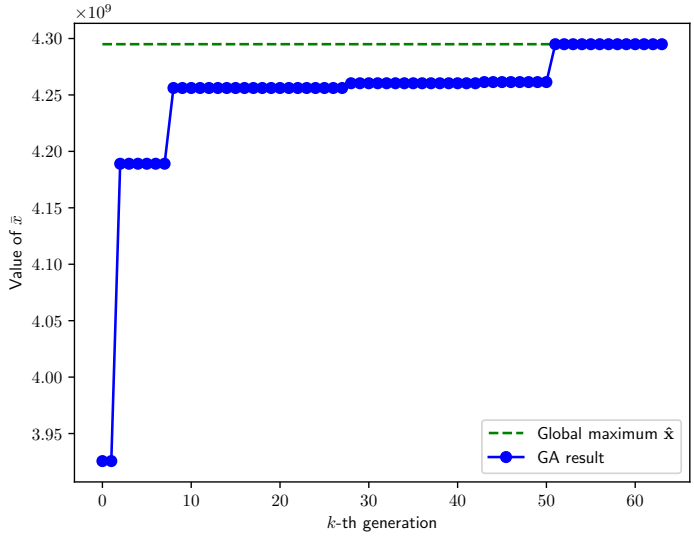

Figure 2.14: HIL result showing the convergence for $f(x) = x$.

The results for both implementations are presented in Table 2.11 and Figures 2.15 and 2.16. The results were obtained for $M = 32$, $N = 64$ and $K = 64$ on ATmega328P with 16 MHz. As depicted, the implementation here proposed achieved a better performance in all aspects. The work presented by (Alves 2010) spent more memory and processing time because it used object-oriented programming for implementing the GA, which produced an extra overhead due the abstraction provided by it. Furthermore, that implementation had numerous global variables, which retained data program from the beginning of the execution. Hence, it is evidenced that the implementation proposed in the present work is well optimized and more efficient than other found in the literature. The memory reduction was about $\frac{4514}{4044} \approx 1.11\times$, $\frac{1426}{6} \approx 237.67\times$ and $\frac{1527}{620} \approx 2.46\times$ for program memory, static memory and stack memory, respectively. The speedup about the processing time was about $\frac{2.66}{0.15} \approx 17.73\times$.

Based in Equation 2.24 and Table 2.11, Table 2.12 shows the processing time for several clock frequency can be used on ATmega328P microcontroller (Mic 2018*a*). The speedup were $\frac{2.66}{0.60} \approx 4.43\times$ and $\frac{2.66}{0.30} \approx 48.86\times$ for 4 MHz and 8 MHz, respectively. Even for small clock frequency, the results show speedup gains over the GA2 proposal (Alves 2010).

Table 2.13 shows safe operating voltage, V_{cc} and the typical current consumption,

Table 2.11: Comparison between the proposed implementation (GA1) and the work presented in (Alves 2010) (GA2). The results were obtained for $M = 32$, $N = 64$ and $K = 64$ on ATmega328P with 16 MHz.

	GA1	**GA2**	
			Memory reduction
Program memory (bytes)	4044	4514	$\approx 1.11\times$
Static memory (bytes)	6	1426	$\approx 237.67\times$
Stack memory (bytes)	620	1527	$\approx 2.46\times$
			Speedup
Processing time (s)	0.15	2.66	$\approx 17.73\times$

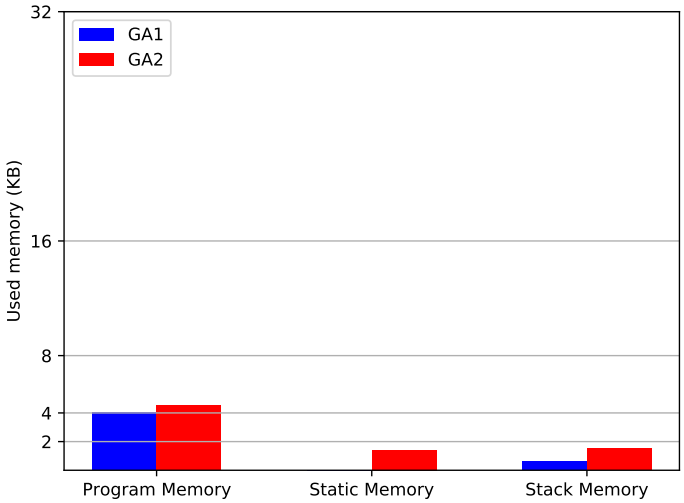

Figure 2.15: Comparison between GA1 and GA2 in terms of memory consumption. The results were obtained for $M = 32$, $N = 64$ and $K = 64$ on ATmega328P with 16 MHz.

I_{cc}, used with different operating clock frequency on ATmega328P microcontroller. This values was obtained from ATmega328P datasheet (Mic 2018*a*). In additional, Table 2.13 shows the power consumption and the power-saving for several values of clock, V_{cc} and I_{cc}. The power-saving was calculated regards the proposal GA2 (Alves 2010) and the power-saving was about $\frac{46}{6} \approx 7.67\times$ and $\frac{46}{26} \approx 1.77\times$ for 4 MHz and 8 MHz, respectively. That means this implementation can be configured to work with lower clock speed and voltage, reduce drastically the power consumption, and still have good performance when compared to other implementation in the literature.

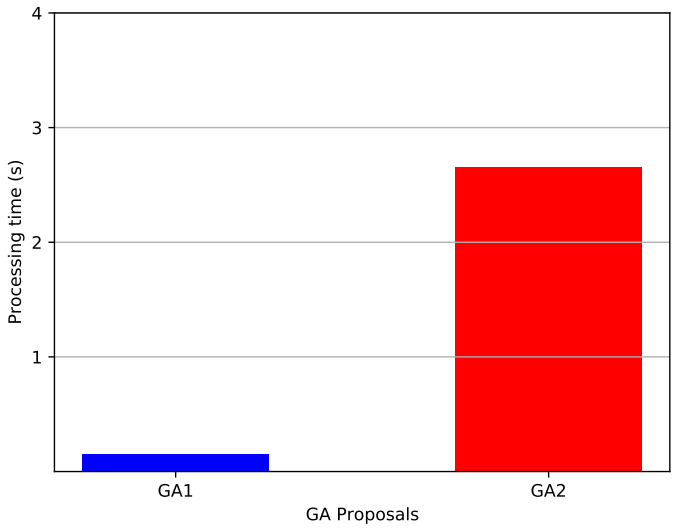

Figure 2.16: Comparison between GA1 and GA2 in terms of processing time. The results were obtained for $M = 32$, $N = 64$ and $K = 64$ on ATmega328P with 16 MHz.

Table 2.12: Speedup comparison between the proposed implementation (GA1) and the work presented in (Alves 2010) (GA2). The results were obtained for $M = 32$, $N = 64$ and $K = 64$ on ATmega328P.

Clock frequency (MHz)	**Processing time (s)**	**Speedup**
4	0.60	$\approx 4.43\times$
8	0.30	$\approx 8.86\times$
16	0.15	$\approx 17.73\times$

Table 2.13: Power-saving comparison between the proposed implementation (GA1) and the work presented in (Alves 2010) (GA2). The results were obtained for $M = 32$, $N = 64$ and $K = 64$ on ATmega328P.

Clock frequency (MHz)	**V_{cc} (V)**	**I_{cc} (mA)**	**Power consumption (mW)**	**Power-saving**
4	3	1.5	6	$\approx 7.67\times$
8	5	5.2	26	$\approx 1.77\times$
16	5	9.2	46	$1\times$

2.5 Conclusions

This work presented an implementation proposal of genetic algorithms targeting low-power, low-cost and low-size-memory devices devices, such as microcontrollers, which is optimized in terms of memory consumption and processing time. The details concerning the implementation and the strategies used to make the algorithm faster and more compact, such as the parameters constraints and the generation of pseudo-random numbers, were provided. In the end, it was also presented the results for the μC ATmega328P with respect to resources consumption, a validation of the GA using the HIL approach and a comparison with another implementation from the literature, which demonstrated this work to have a superior optimization. Hence, it can be concluded that this implementation has proven its feasibility to run on microcontrollers, even in applications where the time constraints are in the order of hundreds of milliseconds. Thus, the genetic algorithm proposed in this work has potential to be applied in different types of applications such as IoT, for example. However, as discussed in the results, the GA parameters need to be well configured according to each particular problem so the implementation can moderately consume data memory and the processing time can meet the defined requirements.

Chapter 3

Embedded Distributed Genetic Algorithm

This chapter presents a strategy to implement a distributed form of genetic algorithm (GA) on low power, low cost, and small-sized memory aiming for increased performance and reduction of energy consumption when compared to standalone GAs. This strategy focuses on making a distributed version of GA feasible to run as a low cost and a low power consumption embedded system utilizing devices such as 8-bit microcontrollers (μCs) and Serial Peripheral Interface (SPI) for data transmission between those devices. Over the SPI, a simple communication is proposed to abstract the tramissions of complex data structures. Also, details about how the distributed GA was designed from a previous standalone implementation made by the authors and how the project is structured are presented. Furthermore, this work investigates the implementation limitations and shows results about its proper operation, most of them collected with the Hardware-In-Loop (HIL) technique, and resource consumption such as memory and processing time. Finally, some scenarios are analyzed to identify where this distributed version can be utilized and how it is compared to the single-node standalone implementation in terms of performance and energy consumption.

3.1 Introduction

Distributed systems are present in our lives every day. They can be simple or complex such as the ones found in the World Wide Web, social networks, e-commerce, and others. A distributed system can be any system in which hardware or software components are separated and able to communicate between themselves by passing messages through some sort of network. The main motivation for constructing these distributed systems is resource sharing, that is, the system can use resources that are not in the same location

and it can be eventually scalable. However, distributed systems usually run concurrently on devices that do not share a global clock and memory, which requires some sort of synchronization, besides the fact that individual devices may present independent failure as well (Coulouris et al. 2011). Thus, that explains why this area is challenging and has been studied for decades.

Traditionally, most algorithms were created and implemented to run on a single machine. Over time, with the development of multiple-core devices and faster networks, several of those algorithms were reinvented to work in a distributed way, so that they could use more resources and be accelerated, for instance (van Steen & Tanenbaum 2016). An example of an algorithm that gained a distributed version years later after its first implementation was the Genetic Algorithms. They are a type of metaheuristics inspired by Darwin's theory of evolution and are an efficient method to solve numerous types of problems, mainly related to search and optimization in different areas (Eiben & Smith 2015). Some researchers already proposed different strategies to implement GAs in a parallel or distributed way, that is, by using multiple devices to share the GA workload (Tangen 2016, Guo et al. 2016, Abdelhafez et al. 2019, Harada & Alba 2020).

Most of the distributed systems depend on some communication system to send and receive data, no matter if it runs on regular computers or simpler devices, such as microcontrollers. Microcontrollers can be defined as System on Chip (SoC), which is commonly used as embedded systems for specific applications. They have limited processing power and usually contain an 8, 16, or 32-bit general-purpose microprocessor, program and data memory, input and output, and other peripherals (Deshmukh 2005). In terms of communication, microcontrollers send and receive data from other devices through simple low-level interfaces, such as SPI, I2C, UART, etc. These interfaces are commonly present in most μCs despite the manufacturer, therefore they are good choices for data transmission when developing a distributed system for those devices.

Some works in the literature propose some kind of distributed system using low-power, low-cost and small-sized memory devices, such as microcontrollers. In (Girolami et al. 2017), the authors developed a distributed system using microcontrollers to monitor the health of structures, with the devices sharing data using wireless communication and sending data to the cloud. In (Sumalan et al. 2020), the authors propose a distributed system to monitor a greenhouse using microcontrollers—most of them working as a smart sensor, that is, collecting, processing, and transmitting data. Finally, in (Wu et al. 2020) the authors propose a distributed microcontroller architecture for the Internet of Things (IoT)—which is also one of the areas where the implementation proposed in this work is expected to be used.

Following the same direction of the works cited above, this work introduces a proposal for the implementation of distributed genetic algorithms targeting low-power, low-cost and small-sized memory devices. There are not so many works that explore this same topic. The closest one is proposed by (Morell & Alba 2017), where the authors propose a distributed genetic algorithm implementation on portable devices for smart cities. Their implementation has a different focus from this work because they expect geographically distributed devices - this work expect multiple devices under the same embedded system.

The implementation of traditional GAs for a single-unit low-power, low-cost and small-sized memory device has been already proposed by the authors of this work in Chapter 2. While the results were satisfactory, the implementation imposes several constraints and has some limitations. For some scenarios, the proposed genetic algorithm can consume all available memory, and, depending on the problem, it may require a lot of processing time. Therefore, to increase the performance and provide more resources for the GA running on those limited devices, this work proposes an implementation of a distributed version of the genetic algorithms using as a base the that model presented before.

The main goal of a distributed genetic algorithm for low-power, low-cost and small-sized memory devices is to combine multiple nodes so that the whole algorithm can take advantage of all available shared resources. That means the GA will be able to store more content in memory, as well as carry out some routines in parallel by using multiple processors, which can reduce the processing time. Another advantage of using numerous devices is that it is possible to reduce the clock of all of them and have reasonable performance, but reducing power consumption (Zomaya & Lee 2012). By achieving these goals, this implementation can be used in emerging areas such as the Internet of Things (IoT), Smart Grid, and Machine to Machine (M2M), where those devices are commonly used exactly for having a low cost, reduced size and restricted consumption of power.

This implementation can be applied in situations where genetic algorithms or other optimization metaheuristics are necessary to solve non-linear optimization problems but with limitations on power consumption. This proposal has an advantage over the traditional GAs because it admits multiple devices and potentially a higher performance or lower energy consumption. Applications that can take advantage of it are, for instance, battery-powered drones or robots that need to calculate the shortest path to reach a destination avoiding obstacles using genetic algorithms (de Oliveira & Fernandes 2016, Silva et al. 2018, Lamini et al. 2018). Furthermore, the use of GAs embedded in automobiles to calibrate car engine parameters as presented in (Millo et al. 2018, Kalaivanan & Sakthivel 2017). Therefore, existing embedded solutions already solved by the use of

GAs can be improved in performance or energy consumption by using distributed GAs.

Another relevant aspect to be clarified is that this project is envisioned to be used in a limited number of devices, possibly two or four microcontrollers soldered and connected on a single board to reduce loss. The reason for this constraint is to not use all available pins of the devices just for communication (the SPI protocol uses several pins, for example) because the embedded system may be connected to other elements such as sensors and actuators. Moreover, a big number of µCs would increase the overall cost of a project and for a high cost, other platforms could be considered instead.

In the following sections, this work will explain the different approaches to implement the distributed genetic algorithm in Section 3.2. Inspired on the existing architectures, this works also proposes a new one for distributed GAs, which aims to keep as much operations in parallel as possible but without reducing the algorithm entropy to search for more solutions as explain in Section 3.3. Finally, in Section 3.4 results are presented showing how this new architecture performs in terms of memory, processing time, and power consumption.

3.2 Distributed Genetic Algorithms

The implementation of distributed genetic algorithms (DGAs) follows the same general idea of its traditional version as described in Algorithm 1, but the difference is that the workload is divided between multiple nodes. There are several possible architectures for DGAs as described in (Gong et al. 2015) and some of them will be presented below. The main advantage of those distributed architectures is that more resources can be used by the GA and hence it can work with larger populations, more bits to represent each individual and increase the precision, and even reduce the processing time by running simultaneous tasks using multiple processors. The architecture proposed in this work is inspired by the ones presented below but will be better explained in Section 3.3.

The most traditional architecture for distributed systems probably is the master-slave, wherein the case of genetic algorithms one of the Q nodes will process most of the operations and sends individuals to be evaluated by the other nodes. While this approach does not sound so efficient at first, the evaluation function is where usually most of the computing load is done for most search and optimization problems. Consequently, by adopting this strategy it is possible to accelerate the evaluation of several individuals in parallel because these evaluations are mutually independent. However, there is a cost to transfer all the individuals during every generation and if the evaluation function is not too costly to cover the communication overhead, then it will not be efficient enough. This

architecture is shown in Figure 3.1.

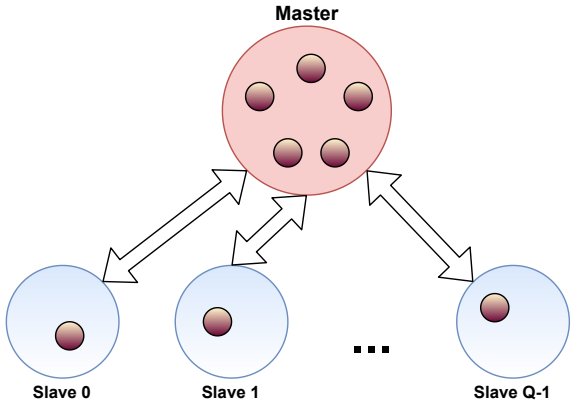

Figure 3.1: Master-slave model for distributed genetic algorithms. The top light-brown circle represents the master node, the light-blue nodes the slave nodes, and the small brown nodes the individuals. In this architecture, individuals can move only from a slave to a master and vice-versa.

Two other options for distributed genetic algorithms is the island and cellular models. By using them, the main population of N individuals is divided into sub-populations that are scattered between the Q nodes, which are spatially distributed. That means each node will be responsible for V individuals, wherein this project N must be divisible by Q. Hence V can be defined as

$$V = \frac{N}{Q}. \tag{3.1}$$

Thus, a node Q will have a sub-population $\mathbf{X}'_{\mathbf{q}}(k)$ of V individuals mapped into M bits and with D dimensions, and can be expressed as

$$\begin{aligned} \mathbf{X'}_{\mathbf{q}}(k) &= \begin{bmatrix} \mathbf{x}_{q\times V}(k) & \dots & \mathbf{x}_{(q+1)\times V-1}(k) \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{x}_{q\times V,0}[M](k) & \dots & \mathbf{x}_{q\times V,D-1}[M](k) \\ \vdots & \ddots & \vdots \\ \mathbf{x}_{(q+1)\times V-1,0}[M](k) & \dots & \mathbf{x}_{(q+1)\times V-1,D-1}[M](k) \end{bmatrix}. \end{aligned} \tag{3.2}$$

In both island and cellular models, all nodes process all the operations of the GA but there is also an extra stage where individuals from an island or cell can migrate to another one as a way to increase the diversity of the global population and avoid a local and premature convergence. That means the nodes can communicate between themselves differently from the master-slave model, where the communication happens only between

the master and slaves but not between slaves. The island model and the cellular model are presented in Figures 3.2 and 3.3, respectively.

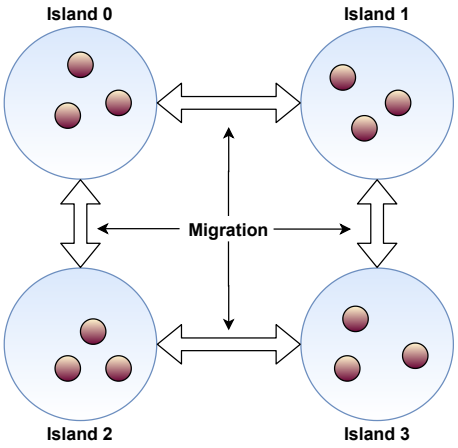

Figure 3.2: Islands model for genetic algorithms. The light-blue circles represent nodes (islands) and the small brown circles represent individuals. In this architecture, individuals can migrate from an island to another in any direction—the islands do not need to be neighbors.

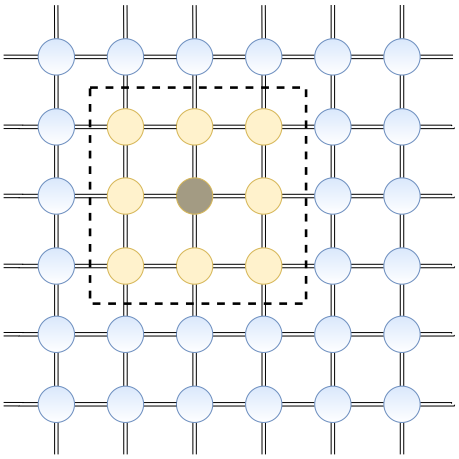

Figure 3.3: Cellular model for distributed genetic algorithms. All the circles represent nodes and in this architecture, differently from the island model, individuals can migrate from a node only to its neighbors. In this example, individuals from the central green node can migrate only to the adjacent yellow nodes—the dashed square region limits the nodes that exchange individuals with the green one.

One last model among numerous that exist for DGAs is the pool model. In this form, the population of individuals is put in a sort of shared global array where various autonomous nodes can access them. That array is then split in U segments so that each node is responsible for the group of individuals in that segment. Finally, each processor can

read individuals from any segment but can overwrite only individuals in its reserved segment. One advantage of this model when compared to the previous ones is that it can handle well asynchronous tasks and heterogeneity, while the others need to have some kind of synchronization between the nodes, mainly during the communication. This model can be seen in Figure 3.4.

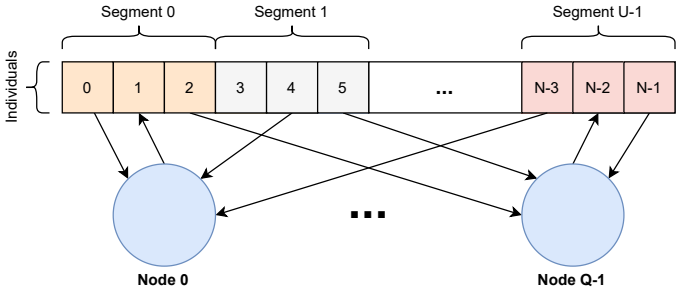

Figure 3.4: Pool model for distributed genetic algorithms.

3.3 Implementation

3.3.1 Algorithm

Based on the several available models of distributed genetic algorithms, the one implemented in this work is mostly inspired by the master-slave model but with some characteristics of the island model. The focus was to keep as many operations as possible in parallel and asynchronous instead of running only the evaluation concurrently to improve the overall performance. Furthermore, the global population is divided into sub-populations of size V between the Q nodes aiming to take advantage of the total memory available. Thus, after analyzing all the operations described in Section 2.2, it was noticed that most of the GA operations are independent and can be done in parallel and asynchronously, except the selection and crossover.

The decision of keeping the GA operations of selection and crossover synchronous between all nodes and coordinated by the master node is to allow the selection and combination of individuals from any sub-population, which may be stored in different microcontrollers. In the traditional island model, only individuals from the same sub-population, that is, from the same node can cross. Because of this, some individuals of different nodes that eventually would generate a good result would never have the change to cross their contents. To address this limitation, this implementation centralizes both operations of selection and crossover in the master node, with the slave nodes working synchronized with

the master during this stage, so that individuals of any microcontroller can be collected, combined, and new individuals sent back. This idea is presented in Figure 3.5.

Once these operations are done, all μCs can follow their run independently, and then they will synchronize again only during the selection and crossover of the next generation. After K generations, there is one extra step where the master needs to synchronize again all slaves to collect the best individual of all sub-populations. Finally, the master will compare Q best individuals and the best one will be the final result. This whole process is presented in Figure 3.6.

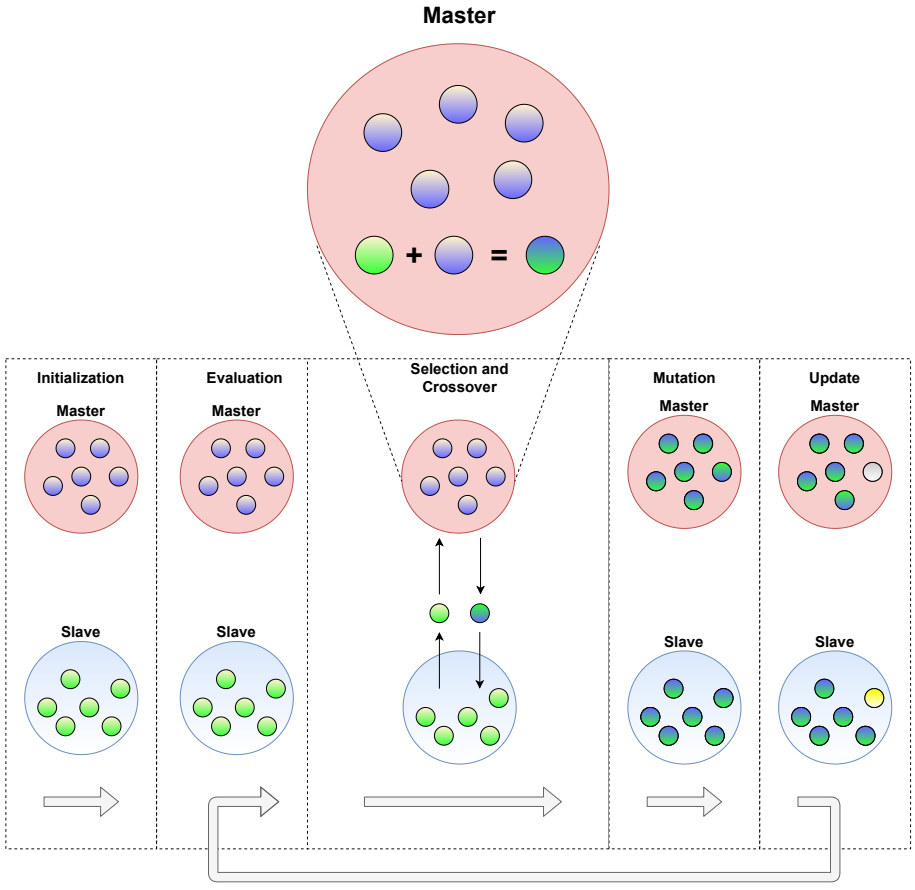

Figure 3.5: Master-slave distributed genetic algorithm (GA) model proposed by this work for distributed genetic algorithms showing that the selection and crossover are executed by the master node after collecting individuals from the slaves.

As stated before, this implementation is a modification of the work presented in (Medeiros et al. 2020*b*) and then it uses the same base structure and has the same constraints and limitations described there. To conform with those limitations, the number of

nodes, Q, must be a power of 2, that is

$$Q = 2^{A_1}, \text{ where } A_1 \in \mathbb{N}^*. \tag{3.3}$$

Since the resources are shared, then the size limit for the global population size, N', will be in function of Q as follows

$$N' = Q \times 2^{A_2}, \text{ where } A_2 \in \mathbb{N}^* \text{ and } A_2 \leq 8. \tag{3.4}$$

Ultimately, another consequence of this new N' is that it can be bigger than 256. Therefore, the type `popsize_t`, which is used in the implementation in variables that store the population size, now cannot be stored as an 8-bit unsigned int anymore, thus it may need a 16-bit variable instead.

Almost all operations, called function modules in that work, are still the same except the selection and crossover, which needed to be modified in this project. The selection and crossover functions have a different implementation for the master and the node. Also, at the end of the GA, the master node needs to collect the best individuals of all nodes and then select the best one as the final result. For that reason, the pseudocode for the master and the nodes are shown separated in Algorithms 15 and 16. In both Algorithms 15 and 16, there are new functions in comparison to the original Algorithm 1 and they are described below:

- Master Implementation
  - Collect Evaluation Value Function (CEVF)—Master collects an evaluation value from a node;
  - Tournament Function (TF)—Master runs the tournament method using 2 individuals;
  - Collect Individual Function (CIF)—Master collects an individual from a node;
  - Send Individual Function (SIF)—Master sends an individual to a node;
  - Continue Operations Function (CIF)—Master tell a node to continue the remaining operations;
  - Collect Best Individual Function (CBIF)—Master collect the best individual from a node;
- Slave Implementation
  - Command Processing Function (CPF)—Slave receives a command from the master.

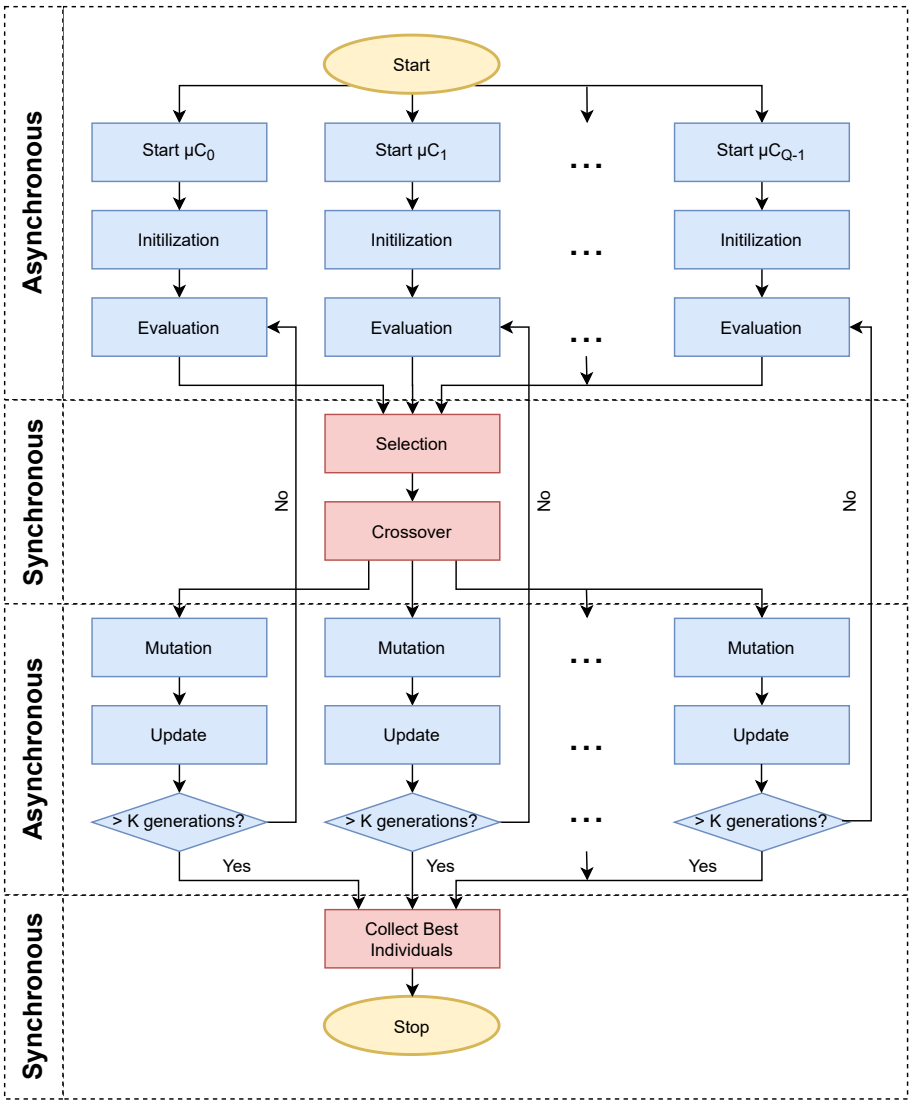

Figure 3.6: Master-slave distributed GA model proposed by this work for distributed genetic algorithms showing asynchronous and synchronous operations.

– Send Best Individual Function (SBIF)—Slave sends its best individual to master.

While the proposed implementation provides some benefits discussed previously, it also brings some drawbacks. The first one is the large time consumption during the selection and crossover because while the master is running the tournament method and crossing individuals, all slaves are idle and waiting for commands from the master. Only when the master finishes the processing of all sub-populations, then the slave nodes can continue the other operations. Furthermore, the communication method between the nodes

Algorithm 15 Distributed Genetic Algorithm Pseudocode-Master

▷ Generation of the initial population
1: Same block in Algorithm 1.
▷ Starts to process the generations
2: Same block in Algorithm 1.
▷ Calculates the fitnesses and evaluates the individuals (or chromosomes)
3: Same block in Algorithm 1.
▷ Selection and crossover
4: **for** $i \leftarrow 0$ **to** $N-1$ **with step** 2 **do**
▷ Generate 4 random indices and collect their fitness values.
5: **for** $l \leftarrow 0$ **to** 3 **with step** 1 **do**
6: $\text{indices}_l \leftarrow \text{RNG}() \wedge (N-1)$
7: $\text{scFitness}_l \leftarrow \text{CEVF}(\text{indices}_l)$
8: **end for**
▷ Run the tournament selection method using pairs of fitness values.
9: $\text{indexWinner}_0 \leftarrow \text{TF}(\text{scFitness}_0, \text{scFitness}_1)$
10: $\text{indexWinner}_1 \leftarrow \text{TF}(\text{scFitness}_2, \text{scFitness}_3)$
▷ Collect the 2 individuals that won the tournament from the nodes.
11: $\textbf{scIndividual}_0 \leftarrow \text{CIF}(\text{indexWinner}_0)$
12: $\textbf{scIndividual}_1 \leftarrow \text{CIF}(\text{indexWinner}_1)$
▷ Cross the 2 individuals collected from the nodes and generates 2 new individuals.
13: $\textbf{scNewIndividuals} \leftarrow \text{CF}(\textbf{scIndividual}_0, \textbf{scIndividual}_1)$
▷ Send the 2 new individuals to one node.
14: $\text{SIF}(\text{scNewIndividuals}_0)$
15: $\text{SIF}(\text{scNewIndividuals}_1)$
16: **end for**
▷ Inform all nodes to continue the remaining operations.
17: **for** $q \leftarrow 0$ **to** $Q-1$ **do**
18: $\text{COF}(q)$
19: **end for**
▷ Mutation
20: Same block in Algorithm 1.
▷ Elitism
21: Same block in Algorithm 1.
▷ Updates the population
22: Same block in Algorithm 1.
▷ Collect the best individual of all nodes.
23: **for** $q \leftarrow 0$ **to** $Q-1$ **do**
24: $\textbf{bestIndividuals}_q \leftarrow \mathit{CBIF}(q)$
25: **end for**

is relevant because it is heavily used during the selection and crossover. Since there is an overhead for each data transfer between the nodes, then a big population makes the

selection and crossover slower.

Algorithm 16 Distributed Genetic Algorithm Pseudocode-Slave

```
    ▷ Generation of the initial population
 1: Same block in Algorithm 1.
    ▷ Starts to process the generations
 2: Same block in Algorithm 1.
    ▷ Calculates the fitnesses and evaluates the individuals (or chromosomes)
 3: Same block in Algorithm 1.
    ▷ Selection and crossover
 4: while true do
    ▷ Wait for a command requested by the master node and take an action.
 5:     command ← CPF()
 6:     if command = "Collect Fitness Value" then
 7:         Send Fitness Value to Master Node
 8:     else if command = "Collect Individual" then
 9:         Send Individual to Master Node
10:     else if command = "Send Individual" then
11:         Receive Individual from Master Node
12:     else if command = "Continue Operations" then
13:         break
14:     end if
15: end while
    ▷ Mutation
16: Same block in Algorithm 1.
    ▷ Elitism
17: Same block in Algorithm 1.
    ▷ Updates the population
18: Same block in Algorithm 1.
    ▷ Wait for master request best individual
19: SBIF(X)
```

Another point to be discussed in this new algorithm is the mutation. Since the original function was kept as it is, thus all nodes will process the mutation of P individuals as described in Equation (2.4). That means the mutation rate will be higher and depend directly on the number of nodes, Q. Thus, the new mutation rate R'_M can be defined as

$$R'_M = \frac{P \times Q}{N}. \tag{3.5}$$

Thus, if the project uses several nodes (big value for Q), it is important to use a reasonable population size, otherwise, the mutation rate would increase drastically. For example, by keeping $P = 1$ (lowest value possible), if there are 8 nodes and the global population N

is only 32, then one individual in a sub-population of 4 would mutate, and this represents a mutation rate of 25%, which is considered high.

3.3.2 Communication between Microcontrollers

To implement the distributed genetic algorithm architecture proposed in Section 3.3.1, data transmission between the targeted devices is necessary. Most manufacturers usually implement in these devices at least the following serial interfaces: Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), and Universal Asynchronous Receivers/Transmitter (UART) (Mikhaylov & Tervonen 2012). Thus, developing a distributed system suitable to run over one of these interfaces allows it to be used in a wide range of devices.

A challenge of implementing any distributed system in these limited devices is that those common interfaces are simple and each one has different particularities, which affect transmission speed, the maximum number of connected devices, and energy consumption. It is possible to add additional hardware to the microcontroller to provide other interfaces and protocols, however, this could increase the overall price of the embedded system and increase energy consumption. Therefore, to keep this implementation efficient and with no need for extra hardware to prevent the increase the costs, the interface SPI was chosen as the communication mechanism between the devices that are part of the distributed system.

SPI is a simple synchronous serial bus standard that operates in full-duplex mode and widely supported by different types of low-capacity devices (Ibrahim 2011). It uses a master-slave architecture where the master node provides the clock to all the slaves and controls when the data transfer starts. When the master sends data, it also receives data from a selected slave at the same clock cycle and that explains why it is full-duplex. Another characteristic of the SPI is that it requires at least a four-wire bus for the simplest case with only one slave and for each extra slave a new write is necessary. The SPI wiring structure is shown in Figure 3.7 and the SPI bus is explained as follows:

- SCLK (Serial Clock)—Wire where master node sends the clock signal to the slaves.
- MOSI (Master Output, Slave Input)—Wire used by the master to send data and used by the slave to receive data through.
- MISO (Master Input, Slave Output)—Wire used by the master to receive data and used by the slave to send data through.
- SS (Slave Select)—Wire used to select which slave will be enabled to communicate with the master node.

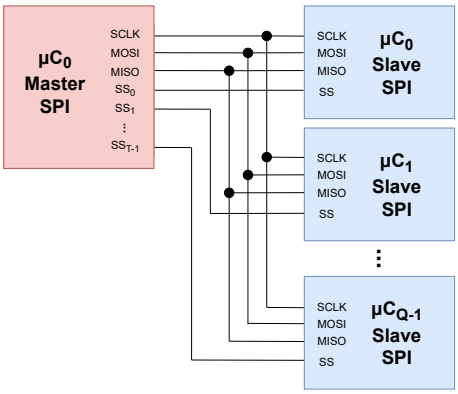

Figure 3.7: Example of the Serial Peripheral Interface (SPI) wiring structure.

While this distributed genetic algorithm implementation may be implemented in any of the communication interfaces, the reason to choose SPI over I2C or UART is that it is simpler to implement, faster, and has lower power consumption for not needing pull-up registers like the I2C (Mishra et al. 2015). Furthermore, the other interfaces have other limitations that would compromise the DGA architecture proposed in this work and its performance as well. UART works point-to-point way and because most devices such as microcontrollers have a limited number of UART interfaces, sometimes only one, it would be impracticable to connect multiple slave nodes to the master node. In respect of I2C, despite the fact that it can support multiple devices, in order to send or receive data it also needs to send the device address before transmitting useful data. This would cause a huge overhead for this DGA proposal because in each generation several data transmissions are done as for individuals as for fitness value.

The SPI interface, which is used in this work, can transmit (send and receive) one byte (8 bits) per time, where 8 clock cycles are necessary for each submission. As described in Section 3.3.1, the distributed genetic algorithm needs to transmit individuals, which can be mapped M bits (either 8, 16 or 32 bits) and have D dimensions, and fitness values mapped in B bits, which usually are float-point numbers (usually 4 bytes for IEEE 754 format). Hence, the clock cycles necessary to transmit these values are

$$c_{\text{CLK}}^{\text{ind}} = M \times D + c_{\text{CLK}}^{\text{trans}} \tag{3.6}$$

$$c_{\text{CLK}}^{\text{fit}} = B + c_{\text{CLK}}^{\text{trans}} \tag{3.7}$$

where $c_{\text{CLK}}^{\text{ind}}$ represents the number of clock cycles to transmit one individual, $c_{\text{CLK}}^{\text{fit}}$ to transmit a fitness value, and $c_{\text{CLK}}^{\text{trans}}$ the clocks cycles necessary as a overhead to start the

transmission.

To abstract the transmission of different data types in the DGA implementation proposed by this work, it was developed a simple 2-step protocol based on commands and acknowledge messages to allow the master inform the selected slave which kind of transfer it is about to make (if the master will send an individual, receive an individual, receive a fitness value, etc). Once the slave receives the command, in the next transmission it will send an acknowledge message as a response, and then, since there is a guarantee they are safely synchronized, the transmission of useful data can begin. This idea is represented in Figure 3.8 and a list of the commands and acknowledge messages is shown in Table 3.1. Therefore, for each transmission of GA content (individual or fitness value), there is an overhead of 16 clock cycles because of the transmission of 2 bytes for command and acknowledge messages. Thus, in Equations (3.6) and (3.7), $c_{\text{CLK}}^{\text{trans}}$ is 16 then.

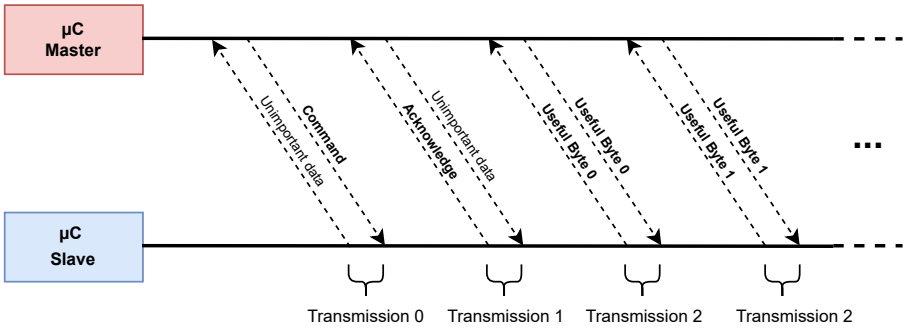

Figure 3.8: Example of how the the protocol based on commands and aknowledge messages works.

3.3.3 Scalability and Overhead

With the Algorithms 15 and 16 and the communication protocol described in Section 3.3.2, it is possible to calculate the number of transferred bytes that are used by the communication protocol. All commands and acknowledge messages are counted together because they work in pairs and individuals and fitness values sent back and forth from the master depends on the Distributed Genetic Algorithm parameters, including the number of individuals (N), number of bits to represent an individual (M), number of dimensions (D), and number of generations (K). With the number of bytes for communication and knowing the transfer speed, in bytes per second, it's possible to calculate the overhead time.

The communication protocol is used in two moments: during the selection and crossover

Table 3.1: List of commands and aknowledge messages used by the proposed protocol.

Name	Value	Description
CMD_SEND_BYTE	0xC0	Master sends a byte.
ACK_SEND_BYTE	0xA0	
CMD_RECEIVE_BYTE	0xC1	Master receives a byte.
ACK_RECEIVE_BYTE	0xA1	
CMD_SEND_FLOAT	0xC2	Master sends a float.
ACK_SEND_FLOAT	0xA2	
CMD_RECEIVE_FLOAT	0xC3	Master receives a float.
ACK_RECEIVE_FLOAT	0xA3	
CMD_COLLECT_EV	0xC4	Master receives evaluation value.
ACK_COLLECT_EV	0xA4	
CMD_COLLECT_IND	0xC5	Master receives an individual.
ACK_COLLECT_IND	0xA5	
CMD_SEND_IND	0xC6	Master sends an individual.
ACK_SEND_IND	0xA6	
CMD_CONTINUE_OP	0xC7	Master tells slaves to continue.
ACK_CONTINUE_OP	0xA7	
CMD_COLLECT_BEST_IND	0xC8	Master receives the best individual.
ACK_COLLECT_BEST_IND	0xA8	

and end to the collection of the best individual from all nodes. The collection of best individuals is straightforward and deterministic because depends only on the individuals and number of nodes, Q. During the selection and crossover, however, there is some randomness during the selection and only selected individuals from slave nodes require an SPI transfer. For example, in the best-case scenario, if all selected individuals are collected from the master node, thus the only transfer would be to send the new individuals to the slaves. In the worst-case scenario, all selected individuals would be collected from the slaves, therefore more transfers would be necessary. At the end of selection and crossover, the master finally needs to synchronize all the nodes again. The collection of best individuals, in turn, is deterministic and depends only on the individuals and the number of slaves.

The expressions for the number of bytes transferred via SPI by the selection and crossover, and during the collection of the best individuals for the worst-case scenario are

$$\mathrm{H}_{\text{sel-cross}} = \left[6N + \left(2 + \frac{MD}{8}\right) \times N + \left(2 + \frac{MD}{8}\right) \times \left(N - \frac{N}{Q}\right)\right] \times K + 2 \times (Q-1), \tag{3.8}$$

and

$$\mathrm{H_{col}} = \left(2 + \frac{MD}{8}\right) \times (Q-1), \tag{3.9}$$

where $\mathrm{H_{sel\text{-}cross}}$ represents the number of bytes transferred during the selection and crossover, including commands and acknowledge messages to collect fitness values, and commands and acknowledge messages to collect and send individuals; and $\mathrm{H_{col}}$ represents the number of bytes transferred during the collection of the best individuals, which includes the commands and acknowledge messages to collect one individual from each node.

The total number of bytes transferred is the sum of Equations (3.8) and (3.9). Using this result, the equation to calculate the total overhead in seconds is

$$t_{overhead} \leq \frac{8}{c_{\mathrm{CLK}}^{\mathrm{SPI}}} \times (\mathrm{H_{sel\text{-}cross}} + \mathrm{H_{col}}) + \Delta, \tag{3.10}$$

where $t_{overhead}$ is the time spent with the transfers, in seconds; $c_{\mathrm{CLK}}^{\mathrm{SPI}}$ the clock speed in which the SPI is running, in Hz; and Δ is a non-deterministic a value, that may represent delays that are a consequence of limitations in the practical implementation, for instance. Finally, this expression considers that there are no transmissions errors and eventual retransmissions.

The $t_{overhead}$ is an estimation of the maximum time (worst-case scenario) spent only with the overhead, that is, the transmission of fitness values and individuals from the master to slaves. This amount of time, however, is just part of the total execution time, which also depends on the other genetic algorithm operations, including the evaluation function. An important result from the Equation (3.10) though is to notice that the number of nodes, Q, barely affects the total overhead because other variables, such as the number of individuals, N, and the number of generations, K are much greater than Q. For example, a real application could be using a population of $N = 64$ or and $K = 64$ generations with only 2 or 4 nodes ($Q = 2$ or $Q = 4$, respectively). Thus, it is expected that the overhead will not affect significantly the scalability based on the number of nodes.

Finally, using the results presented in (Medeiros et al. 2020*b*) for the processing time for all sections of the GA, it is possible to estimate the total execution time of the DGA. The processing time for the standalone GA can be simplified as

$$t_{\mathrm{GA}} = t_{\mathrm{IFM}} + K \times (t_{\mathrm{FFM}} + t_{\mathrm{NPFM}}), \tag{3.11}$$

where t_{GA} is the processing time for the standalone GA; t_{IFM} the processing time to run the initialization; t_{FFM} the processing time to run the fitness function; and t_{NPFM} to run

the new population function module.

By expanding and simplifying Equation (3.11), the equation can be rewritten as

$$\begin{aligned} t_{\mathrm{GA}} = N\phi_1 + K \times (N\phi_2 + N\phi_3 + N\phi_4 + \phi_5) = \\ N \times [\phi_1 + K \times (\phi_2 + \phi_3 + \phi_4)] + K\phi_5, \end{aligned} \tag{3.12}$$

where ϕ_1 is the internal time to run the initialization operation; ϕ_2 is the internal time to run the fitness operation (evaluation and normalization); ϕ_3 is the internal time to run the selection and crossover operations; ϕ_4 is the internal time to run the population update operation; and ϕ_5 is the sum of other internal times that do not depend on the population size, N. All these values of ϕ changes depending on other parameters, such as number of dimensions, D, or number of bits to represent the individual, M.

Since the distributed genetic algorithm implementation is built on top of the same implementation proposed in (Medeiros et al. 2020*b*), therefore, if the devices are running at the same clock speed, the total time for the DGA is the same expression of Equation (3.12) but with the population divided between Q nodes plus the $t_{overhead}$, that is,

$$t_{\mathrm{DGA}} = \frac{N}{Q} \times [\phi_1 + K \times (\phi_2 + \phi_3 + \phi_4)] + K\phi_5 + t_{overhead}. \tag{3.13}$$

Finally, by putting t_{DGA} in function of t_{GA}, the final expression for t_{DGA} is

$$t_{\mathrm{DGA}} = \frac{1}{Q}(t_{\mathrm{GA}} + K\phi_5) + t_{overhead}. \tag{3.14}$$

The result of Equation (3.14) is important because allows estimating how the processing time for the DGA will be based on how the standalone GA performs. Also, since $t_{\mathrm{GA}} \gg \frac{K\phi_5}{Q}$ and $t_{overhead} \gg \frac{K\phi_5}{Q}$, thus the processing time of the DGA is approximately the processing time of the standalone divided by the number of nodes plus the overhead. Hence, the expression to test if the DGA will be faster than the standalone GA for the same parameters is

$$t_{overhead} \times \frac{Q}{Q-1} \leq t_{\mathrm{GA}}, \text{ where } Q \in \mathbb{N} \text{ and } Q > 1. \tag{3.15}$$

3.4 Results

To validate the implementation proposed in this work as well as analyze its performance and correct operation, an embedded system was developed using the same tech-

nologies employed in (Medeiros et al. 2020*b*). The source code developed on Atmel Studio 7 in language C was used as the base for this project and modified to accommodate both versions of the distributed algorithm (Algorithms 15 and 16). The distributed embedded system targeted Atmel microcontrollers, particularly the same microcontroller ATmega328P that runs on Arduino Uno and was used in the previous work. This µC has an 8-bit processor based on the AVR architecture, which runs by default at 16 MHz, and has 32 KB of program memory and 2 KB of data memory (Mic 2018*a*). The reason to choose an 8-bit microcontroller is that it is one of the simplest and limited devices available with lots of restrictions, thus if the implementation works for it, it will also work for more robust devices.

The construction of the DGA embedded system was done using 2 Arduino Uno boards, which is the minimum number of nodes required to run this project, but it can be used in multiple devices as long as they respect Equation (3.3). Both 8-bit microcontrollers present in these boards were connected between themselves via SPI, configured with a clock frequency of 125 kHz (µC base clock of 16 MHz divided by 128), and it was necessary 4 wires as described in Section 3.3.2. It is important to mention that for each byte transfer via SPI, a delay of 1 ms was added on purpose for each byte transfer to reduce the transmission errors that were happening compared to when SPI was running at full speed. Thus, the value of Δ will be approximately

$$\Delta = \frac{1}{1000} \times (\mathrm{H}_{\text{sel-cross}} + \mathrm{H}_{\text{col}}). \tag{3.16}$$

Moreover, a third Arduino Uno board was connected to the master node using a regular GPIO pin to help with the measurement of processing time. The idea is simple: when some routine needs to be measured in the master node before it starts that pin receives value high and when it finishes that pin receives value low. The third microcontroller will start a timer then it gets value high and then stops it when receives value low, and finally will show the measured time. The wiring of the three µCs is illustrated in Figure 3.9.

The following sections about resource consumption, specifically memory, processing time, and the correction operation using Hardware-In-The-Loop followed the same strategies used in (Medeiros et al. 2020*b*). Also, some experiments had to be done for both master and slave implementations since they have different contents. In the last subsection, there are more results about how this implementation of DGA compares to the standalone GA in terms of performance and energy consumption.

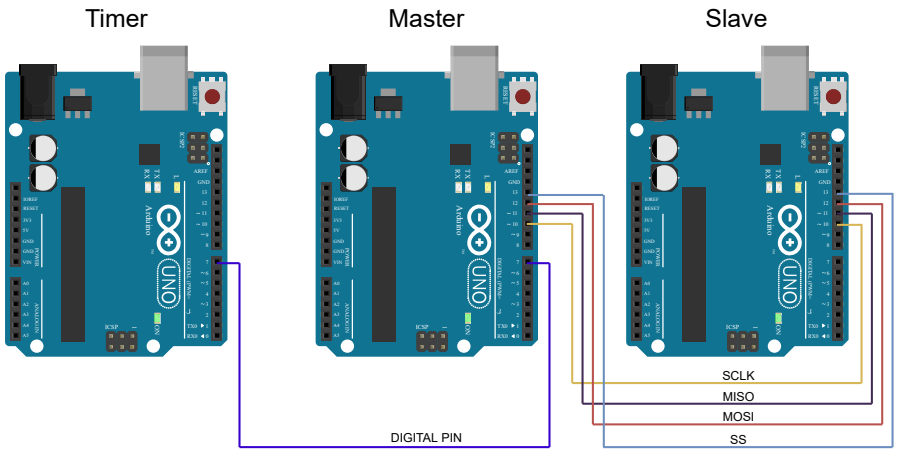

Figure 3.9: Arrangement of how the Arduino Uno boards were connected.

3.4.1 Memory Consumption

The first results collected were the program and data memory consumption. The program memory is non-volatile and is used to store the instructions to be executed by the processor, that is, the compiled program. The data memory is volatile and used to store variables during the run of the program. Also, the data memory can be divided into two segments:

- Static memory: the memory consumed by global and static variables and is kept allocated during the whole program execution. That means this section of the memory cannot be freed and used by other variables.
- Stack memory: the memory used by local variables and that can be allocated and freed according to their lifetime (for example, a local variable defined inside of a function will be freed when the function is finished).

The measurement of static memory is straightforward because the compiler can calculate it. The stack memory, in turn, needs to be calculated empirically. Therefore, both results are shown separately as for the master as for the slave node. To simplify the measurements, all experiments were performed with a fixed number of generations $K = 64$, since this affects only the processing time. Also, the evaluation function used was $f_1(\bar{\mathbf{x}}) = \bar{x}_0^2 - 6\bar{x}_0 + 8$, with dimension $D = 1$, to avoid the use of external libraries. Finally, the crossover was configured as one-point and the number of mutated individuals was $P = 1$.

After running some experiments with the parameters above, the program memory consumption for the master and slave implementations is shown in Tables 3.2 and 3.3

and Figures 3.10 and 3.11, respectively. The compiled program consumes only a small portion of the 32 KB available and practically does not scale, using only about 11% of program memory in the master node and 7% in the slave for almost all scenarios. This result is important because it allows this distributed genetic algorithm implementation to be deployed as part of other projects.

Table 3.2: Program memory consumption (in bytes) in the master node, of a total of 32 KB.

Individual Size (M)	**Population Size (N)**			
	32	**64**	**128**	**256**
8-bit	3206	3206	3214	3320
16-bit	3328	3378	3382	3412
32-bit	3602	3614	3636	3734

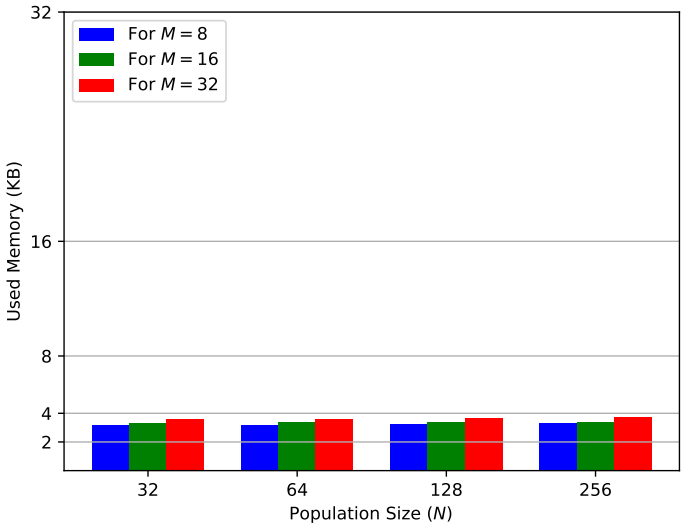

Figure 3.10: Comparison of program memory consumption (in bytes) in the master node, of a total of 32 KB, for different values of N and M.

The results regarding data memory are divided into static and stack memory. For all the scenarios tested above, the static memory was always 8 bytes. This was expected because this project does not use global or static variables so that almost all data memory can be used dynamically as stack memory. The results of stack memory, in turn, are shown in Tables 3.4 and 3.5 and Figures 3.12 and 3.13, sequentially. The numbers ob-

Table 3.3: Program memory consumption (in bytes) in the slave node, of a total of 32 KB.

Individual Size (M)	**Population Size (N)**			
	32	**64**	**128**	**256**
8-bit	2060	2060	2064	2086
16-bit	2140	2144	2144	2158
32-bit	2314	2314	2306	2354

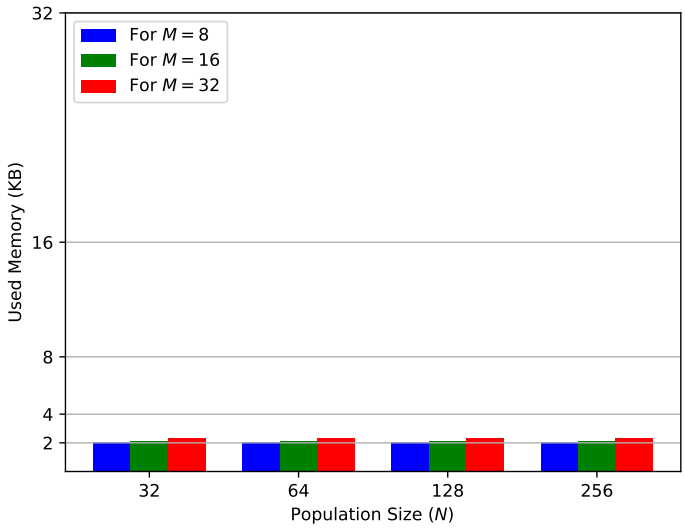

Figure 3.11: Comparison of program memory consumption (in bytes) in the slave node, of a total of 32 KB, for different values of N and M.

tained in this work are similar to those obtained in (Medeiros et al. 2020*b*), because after dividing the global population both microcontrollers got the same population size used in the experiments in that work.

Table 3.4: Stack memory consumption (in bytes) in the master node, of a total of 2 KB.

Individual Size (M)	**Population Size (N)**			
	32	**64**	**128**	**256**
8-bit	211	307	499	888
16-bit	249	377	633	1152
32-bit	312	508	901	1674

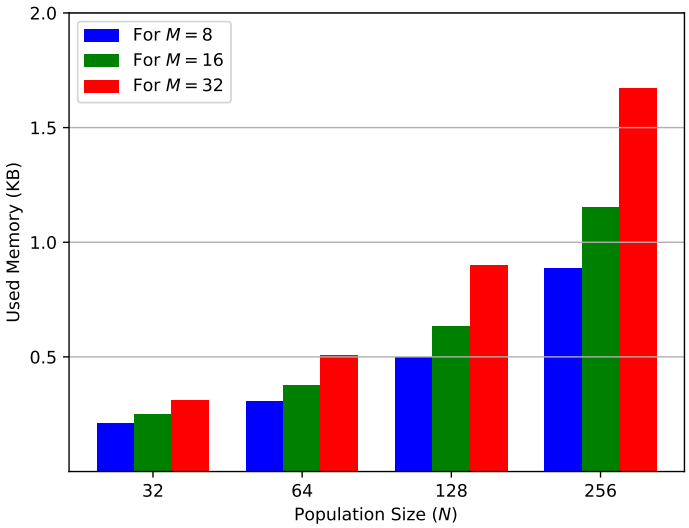

Figure 3.12: Comparison of stack memory consumption (in bytes) in the master node, of a total of 2 KB, for different values of N and M.

Table 3.5: Stack memory consumption (in bytes) in the slave node, of a total of 2 KB.

Individual Size (M)	**Population Size (N)**			
	32	**64**	**128**	**256**
8-bit	184	264	453	839
16-bit	200	327	583	1097
32-bit	275	467	851	1611

The numbers were also plotted in a chart in Figures 3.14 and 3.15, and the best approximation of a linear function was done for all the cases. The stack memory consumption seems to increase linearly with the population size N and at a slower rate with the increase of the individual size M. While not presented, the same linear increase is expected for the number of dimensions D in the individuals since another dimension is equivalent to add another individual. For a typical situation using 2 microcontrollers with a population size of 128, with individuals mapped into 16 bits, the total memory consumption is around 31% for the master and 28% for the slave. This low usage is important because it leaves about 70% of the memory available and allows this DGA implementation to reside together with other projects in the microcontrollers.

Therefore, it is important to consider the peculiarities of each application of this implementation. For this scenario with 2 microcontrollers, a global population of 512 indi-

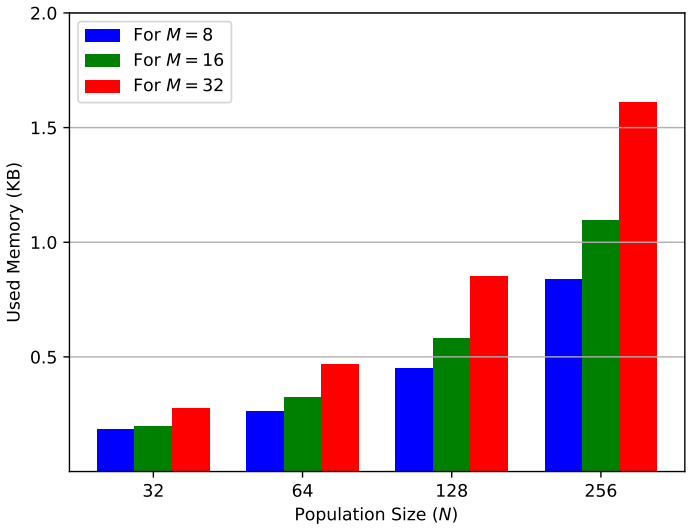

Figure 3.13: Comparison of stack memory consumption (in bytes) in the slave node, of a total of 2 KB, for different values of N and M.

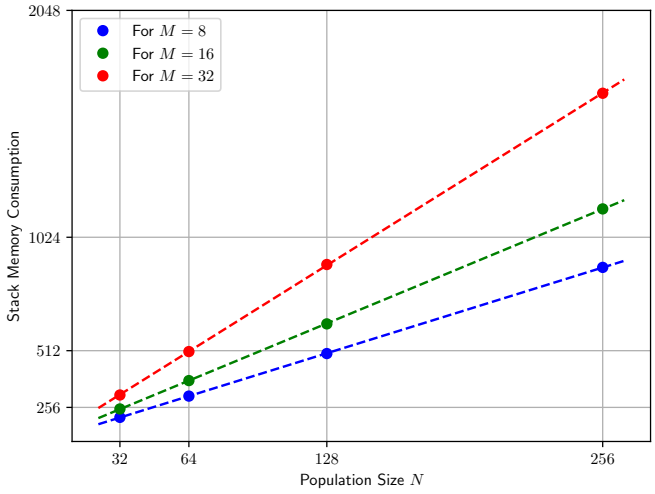

Figure 3.14: Stack memory consumption in master node.

viduals mapped into 32-bit would not be viable because the data memory would not be enough (by following the trend, it would be necessary more than 3.2 KB at least in each µC). As possible solutions, the population N or the precision M could be reduced or more microcontrollers could be added to provide more resources. The only problem with this last approach is that it would double the costs with hardware since the number of nodes Q must be a power of 2 as explained in Equation (3.3).

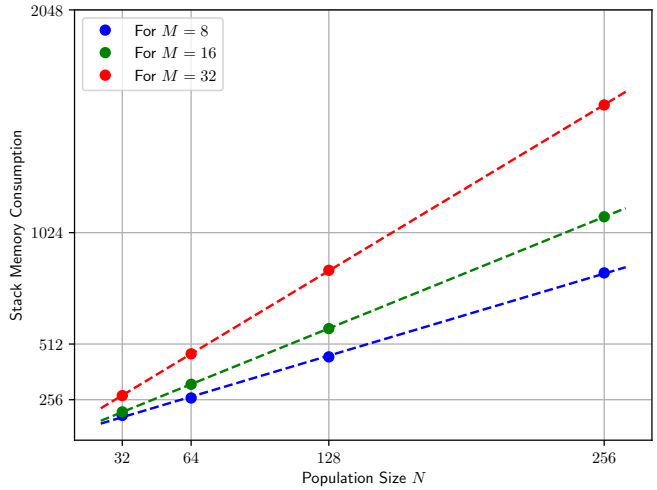

Figure 3.15: Stack memory consumption in slave node.

3.4.2 Processing Time

The second results collected from this distributed genetic algorithm implementation was the processing time. The methodology used in (Medeiros et al. 2020*b*), which was mostly based on measuring the number of clock cycles using the Atmel Studio 7 debugger, is not interesting for this work because the communication between multiple microcontrollers may not keep the algorithm fully deterministic. As shown in Figure 3.6, some part of this implementation is not synchronized and some nodes may finish the run before others. Another issue that can happen is when the master sends a byte with a command and because of some error the slave didn't receive it properly, then the slave will not send the acknowledge message and will wait for a resending of the command again. Thus, the following results present the real run time, experimentally measured with an external timer.

To evaluate the processing time, the following evaluation functions were used:

- $f_1(\bar{\mathbf{x}}) = \bar{x}_0^2 - 6\bar{x}_0 + 8$
- $f_2(\bar{\mathbf{x}}) = \bar{x}_0^2 + \bar{x}_1^2$
- $f_3(\bar{\mathbf{x}}) = 2\bar{x}_0 + 3\bar{x}_1 + 5$
- $f_4(\bar{\mathbf{x}}) = 21{,}5 + \bar{x}_0\left(\sin\left(40\pi\bar{x}_0\right) + \cos\left(20\pi\bar{x}_0\right)\right)$
- $f_5(\bar{\mathbf{x}}) = 10 + \bar{x}_0^2 - 10\cos\left(2\pi\bar{x}_0\right) + \bar{x}_1^2 - 10\cos\left(2\pi\bar{x}_1\right)$

For all these functions, the following GA parameters were fixed: population size $N = 32$, individuals mapped into $M = 16$ bits, number of generations $K = 64$, and number of mutated individuals $P = 2$. The results are presented in Table 3.6. The processing time

seems to not change so much with the type of evaluation function and this can be noticed when comparing functions $f_2(\bar{\mathbf{x}})$, $f_3(\bar{\mathbf{x}})$ and $f_5(\bar{\mathbf{x}})$, which have different mathematical operations but same number of dimensions and similar run times. The same happened for $f_1(\bar{\mathbf{x}})$ and $f_4(\bar{\mathbf{x}})$, which have one dimension as common characteristic. Thus, this suggests the time spent with the communication is being predominantly the part that is consuming more time.

Table 3.6: Processing time for different evaluation functions.

Evaluation Function	**Average Time (s)**	**Standard Deviation (s)**
$f_1(\bar{\mathbf{x}})$	14.759885	0.0675244
$f_2(\bar{\mathbf{x}})$	17.248525	0.0985764
$f_3(\bar{\mathbf{x}})$	17.221671	0.1402979
$f_4(\bar{\mathbf{x}})$	15.003993	0.0635690
$f_5(\bar{\mathbf{x}})$	17.513101	0.0940247

To understand how the SPI communication is affecting the global time, several experiments were done to evaluate how the processing time changes according to the size of population N, number of generations K, and the number of dimensions D-all related with the number of bits per individual M. For the experiments to analyze N, K,and M, the function $f_4(\bar{\mathbf{x}})$ was used. For evaluate D, it was used the evaluation function $f_2(\bar{\mathbf{x}})$, by adding or removing more terms $\bar{x}_D^2$ when the dimension was greater than 2. For example, to evaluate the version with 4 dimensions, the terms $\bar{x}_2^2$ and $\bar{x}_3^2$ are added to the function and so on.

The results for processing time for N, K, and D are presented in Tables 3.7–3.9, respectively. For the results of N and K, by observing the lines from the top to the bottom, the value of M does not affect so much the processing time and the difference in time when using 16-bit and 32-bit individuals is small. However, by analyzing the columns from left to right, it was noticed a sort of linear increase of processing time proportional as to N as to K in both Tables 3.7 and 3.8. This impression can be proved in Figures 3.16 and 3.17, wherein both cases the points seem to represent a first-degree polynomial function. Finally, the results for the number of dimensions D are presented in Table 3.9. The value of M seems to affect more the time than the other 2 previous cases (N and K). On the other hand, even though the increase in the number of dimensions D affects the consumption of data memory, it produces only a slight increase in the processing time. A first-degree polynomial function is plotted in Figure 3.18 and shows how the increase is expected for different values of D.

Table 3.7: Average processing time (in seconds) for different values of N, using $f_4(\bar{\mathbf{x}})$ with $K = 8$ and $D = 1$.

Individual Size (M)	Population Size (N)			
	32	**64**	**128**	**256**
8-bit	1.826240	3.426816	6.537408	13.276224
16-bit	2.034240	3.885440	7.593472	15.342976
32-bit	2.175488	3.947008	7.800448	15.710848

Table 3.8: Average processing time (in seconds) for different values of K, using $f_4(\bar{\mathbf{x}})$ with $N = 8$ and $D = 1$.

Individual Size (M)	Number of Generations (K)			
	32	**64**	**128**	**256**
8-bit	1.784064	3.451264	6.886272	13.689343
16-bit	2.047936	4.092608	8.073536	16.024448
32-bit	2.075712	4.126912	8.097472	16.096191

Table 3.9: Average processing time (in seconds) for different values of D, using $f_2(\bar{\mathbf{x}})$ with $N = 32$ and $K = 8$.

Individual Size (M)	Number of Dimensions (D)			
	1	**2**	**4**	**8**
8-bit	1.763648	1.654784	1.914048	1.887424
16-bit	1.919808	2.217280	2.843584	3.648960
32-bit	2.065408	2.295168	2.912192	4.293888

Therefore, the results of processing time are importing to show how it increases based on important parameters of the distributed genetic algorithm. All the main four variables analyzed above (N, K, D, and M) influence directly on the time spent with communication between the nodes, which is the main overhead in this case, because an 8-bit microcontroller can transfer only one byte (8 bits) at once via SPI. The variables D and M define how large is each individual in terms of bytes and N and K how many transfers need to be done during a run of the distributed GA. For that reason, it is crucial to select the proper GA parameters to have control over the processing time.

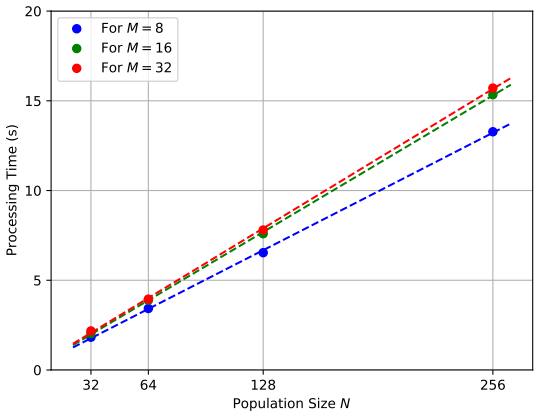

Figure 3.16: Best polynomial function that fits processing time for different values of population size, N.

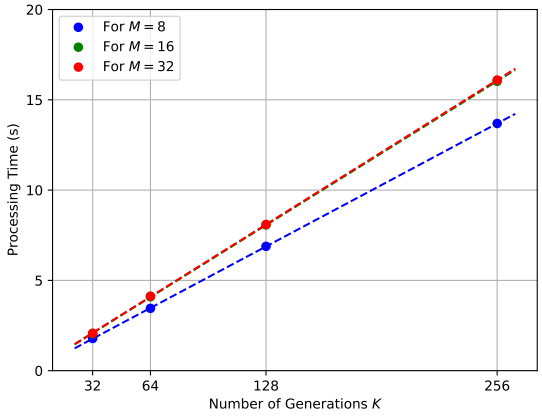

Figure 3.17: Best polynomial function that fits processing time for different values of number of generations, K.

3.4.3 Validation with Hardware-In-The-Loop

Another important experiment was the verification of the proper functioning of this implementation. To collect the data, it was used the Hardware-In-The-Loop (HIL) technique, where the microcontrollers are connected to a computer via some interface and then they can exchange messages during the run, such as parameters and results. In this project, both master and slave nodes were connected to the computer by using the USART interface and during each generation, they were set to send the current best individual. In the computer, there is a Python program running and collecting the data and after all generations, it plotted a chart showing the convergence of the DGA. The functions employed in this section are $f_2(\bar{\mathbf{x}})$ and $f_4(\bar{\mathbf{x}})$, which are shown in Figures 3.19 and 3.20, respectively.

The first experiment was using the evaluation function $f_2(\bar{\mathbf{x}})$, where the goal is to find the global minimum. The search space for all dimensions was defined between -5 and

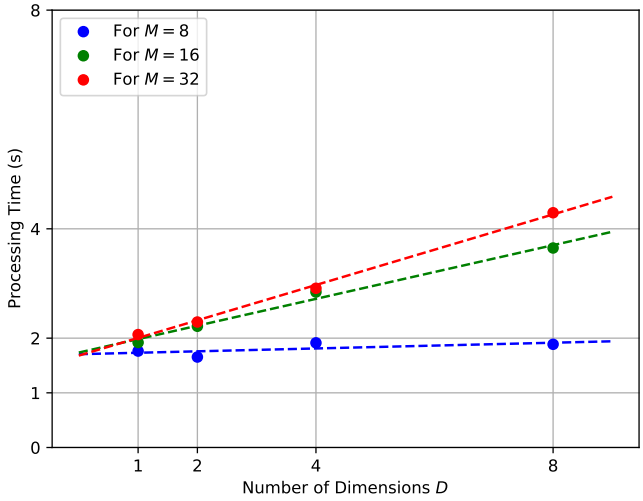

Figure 3.18: Best polynomial function that fits processing time for different values of number of dimensions, D.

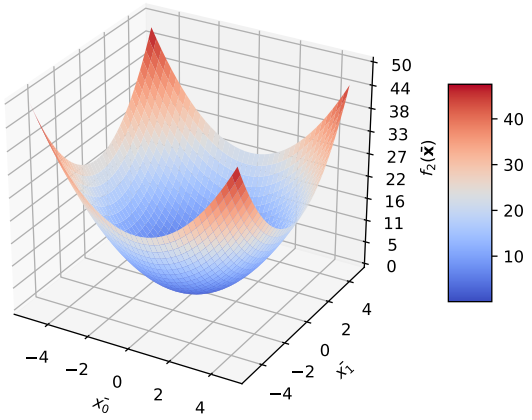

Figure 3.19: Chart showing values of $f_2(\bar{\mathbf{x}})$, with x_0 and x_1 from -5 to 5.

5 and the DGA was set up with the following parameters: population size $N = 16$ individuals mapped into $M = 16$ bits, dimensions $D = 2$, number of generations $K = 64$, and number of individuals mutated $P = 1$. After running the distributed genetic algorithm, the local population in both nodes converged to close to the right result, which is $(0,0)$. This is shown separately for the master in Figure 3.21 and for the slave in Figure 3.22, where each dimension is independent and converge in different moments. For this particular run, after finishing all the generations and comparing the best individual of all nodes, the one from the slave was the selected to be the final result, which was the value $(0.000076, 0.000687)$.

The second function used for the HIL validation was $f_4(\bar{\mathbf{x}})$. The intention was to find

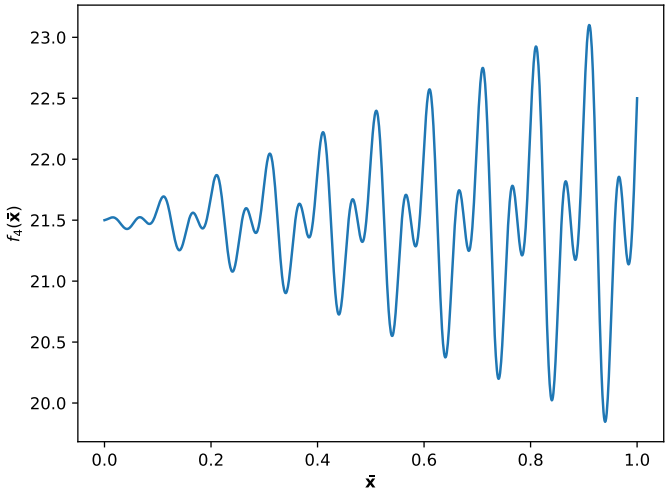

Figure 3.20: Chart showing values of $f_4(\bar{\mathbf{x}})$, with x from 0.8 to 1.0.

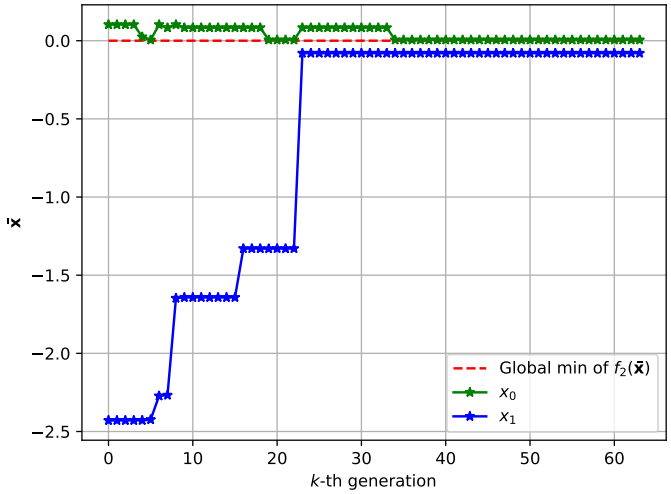

Figure 3.21: Convergence of the dimensions x_0 and x_1 of $f_2(\bar{\mathbf{x}})$ in master node.

the local maximum for the search space between 0 and 1. The distributed genetic algorithm was configured with population size $N = 32$ individuals mapped into $M = 32$ bits, dimensions $D = 1$, number of generations $K = 64$, and number of individuals mutated $P = 4$. As in the master as in the slave node, both populations converged to the expected maximum local maximum, which is located around $x = 0.91$. At the end of the algorithm, the populations in both nodes were homogenous and the best individual had the same value $x = 0.910204$, thus the best individual from the master was used as the final result. The results for the master and the slave are presented in Figures 3.23 and 3.24, respectively.

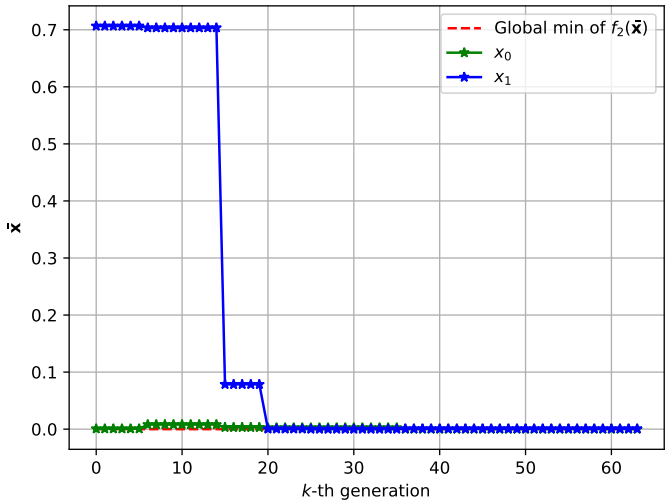

Figure 3.22: Convergence of the dimensions x_0 and x_1 of $f_2(\bar{\mathbf{x}})$ in slave node.

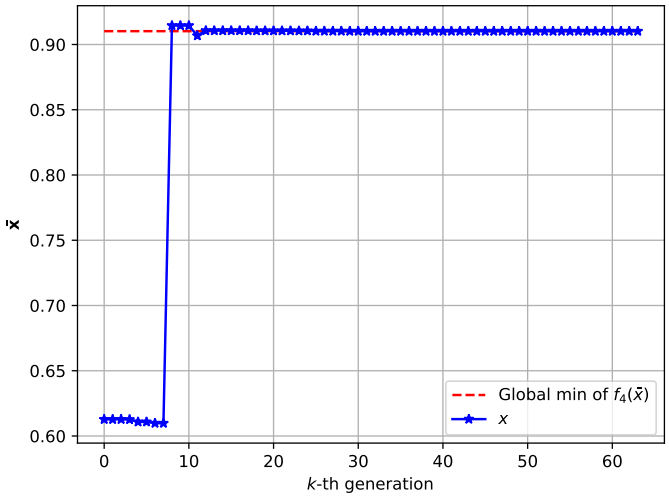

Figure 3.23: Convergence of the dimension x of $f_4(\bar{\mathbf{x}})$ in master node.

3.4.4 Comparison with Standalone Version

A final experiment was to investigate how the distributed genetic algorithm proposed in this work is compared to the standalone version, that is, the genetic algorithm that runs in one single 8-bit microcontroller, which is presented in (Medeiros et al. 2020*b*). There are two motivations for this result:

- Verify if it is possible to accelerate the genetic algorithm for a certain application by adding more microcontrollers;
- Evaluate if, by using multiple microcontrollers configured with lower voltage and lower clock frequency, it is possible to save energy and have a similar performance

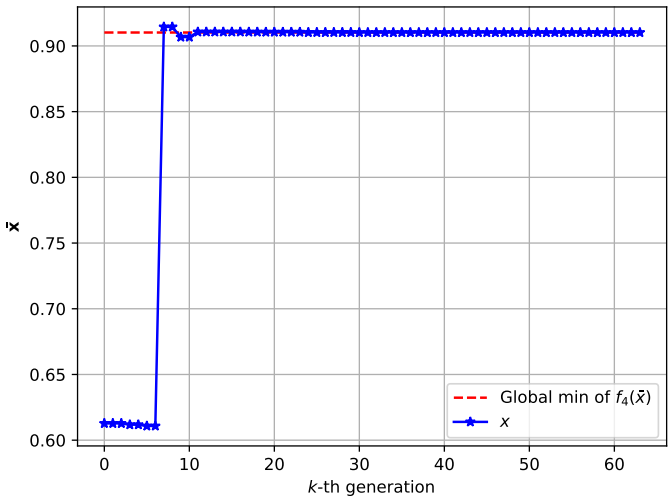

Figure 3.24: Convergence of the dimension x of $f_4(\bar{\mathbf{x}})$ in slave node.

to the standalone version.

By analyzing the results presented in Section 3.4.2, there is a large overhead due to the SPI communication between the microcontrollers, which is consuming a lot of processing time even using those simple evaluation functions. Thus, in order to have some advantage with multiple cores, the evaluation function needs to be complex enough so that the processing time spent with it is much higher than the time spent with the data transfer between the nodes. To not change the original evaluation functions, they were changed in such a way to consume more clock cycles but generating the same result. This idea is expressed in the Algorithm 17.

Algorithm 17 Redefinition of Evaluation Function to Become Slower

▷ Define how many times the evaluation function will repeat (2000 times, for example).
1: REPEAT $\leftarrow$ 2000
▷ The original evaluation function will run REPEAT times.
2: **function** EFM($\bar{\mathbf{x}}(k)$)
3: result $\leftarrow$ 0
4: **for** $r \leftarrow 0$ **to** REPEAT $-$ 1 **do**
5: result $\leftarrow$ result $+ f(\bar{x}_0[M](k), \ldots, \bar{x}_i[M](k), \ldots, \bar{x}_{D-1}[M](k))$
6: **end for**
7: **return** (result/REPEAT)
8: **end function**

For the following experiments, the GA was set up with the following parameters: population size $N = 32$, number of generations $K = 64$, individual size $M = 16$, number of

mutated individual $P = 1$, and evaluation function $f_4(\bar{\mathbf{x}})$, which was set up to repeat 1000, 2000, 4000 and 8000 times using the strategy proposed in Algorithm 17. By measuring the number of clock cycles that this function needs to run for each case, the processing time of the modified evaluation function, $t_{\text{EFM-slow}}$, can be calculated as

$$t_{\text{EFM-slow}} = c_{\text{CLK}}^{f_{slow}(\times)} \times \frac{1}{\text{CLK}}, \tag{3.17}$$

where $c_{\text{CLK}}^{f_{slow}(\times)}$ is the number of clock cycles to run the modified evaluation function and CLK the clock frequency of the microcontroller. This processing time is used below for the different scenarios. The values of $c_{\text{CLK}}^{f_{slow}(\times)}$ were collecting via experiments in Atmel Studio 7 and are shown in Table 3.10.

Table 3.10: Clock cycles of $f_4(\bar{\mathbf{x}})$ after being slowed down.

Number of Repetitions			
1000	**2000**	**4000**	**8000**
122,055	246,193	505,304	1,055,503

The first measures were done using both standalone and distributed version of the GA running at the same clock speed and voltage. As shown in Table 3.11 and Figure 3.25, when the evaluation function is not complex enough, the overhead due SPI communication makes the distributed GA slower than the standalone GA. However, as the evaluation function becomes more complex, the distributed GA becomes faster. In fact, this can also be noticed by analyzing both polynomial functions that fit those points shown in Figure 3.25, which is in the format $t = a \times c_{\text{CLK}} + b$, and is defined as

$$t_{\text{std}} = 0.0001299 \times c_{\text{CLK}} + 0.06282 \tag{3.18}$$

for the standalone version, and

$$t_{\text{dist}} = 0.00006501 \times c_{\text{CLK}} + 14.97 \tag{3.19}$$

for the distributed version, where t represents the processing time in seconds and c_{CLK} represents the evaluation function clock cycles. When the number of clock cycles c_{CLK} is large enough, the distributed version will run approximately 2 times faster than standalone

as demonstrated as follows

$$\lim_{c_{\text{CLK}} \to \infty} \frac{t_{\text{std}}}{t_{\text{dist}}} = \frac{0.0001299}{0.00006501} = 1.998154. \tag{3.20}$$

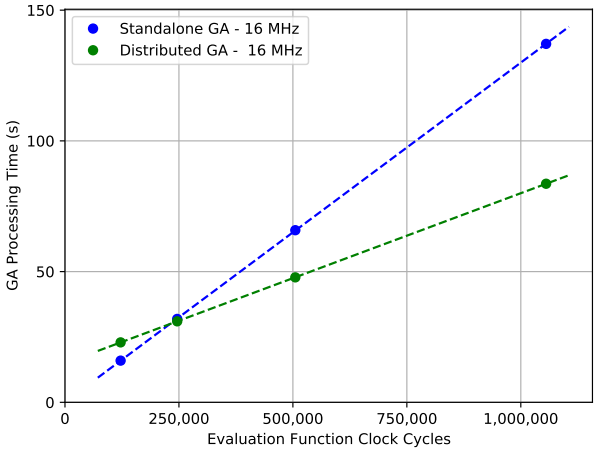

Figure 3.25: Comparison of standalone GA (1 node) and distributed GA (2 nodes) processing time with same clock frequencies.

Table 3.11: Processing time (in seconds) for standalone (1 node) and distributed GA (2 nodes), both running at 16 MHz.

Version	**Evaluation Function Clock Cycles**			
	122,055	**246,193**	**505,304**	**1,055,503**
Standalone GA	15.948736	31.897600	65.834305	137.102340
Distributed GA	22.925825	30.978945	47.796932	83.606339

Another important analysis from Equation (3.19) is the high overhead. By applying the Genetic Algorithm parameters and the SPI clock frequency, defined in 125 kHz, to Equations (3.10) and (3.16), it is possible to see how the theoretical model compares to the experiments. The $t_{overhead}$ is calculated as follows

$$t_{overhead} \leq \frac{8}{125000} \times (24578+4) + \frac{1}{1000} \times (24578+4) = 26.155, \tag{3.21}$$

where the value of 26.155 would be the maximum overhead in seconds for the worst-case scenario, that is, if all individuals were selected from the slave. However, since this is unlikely to happen, then the 14.97 seconds in Equation (3.19) is reasonable and under the theoretical limit.

Finally, to validate the theoretical model presented in Equation (3.14), by applying the results from the experiment shown in Equation (3.18) and from Equation (3.21), the expected equation for the distributed versions would be

$$t'_{\text{dist}} \leq \frac{t_{\text{std}}}{2} + t_{overhead} = 0.00006495 \times c_{\text{CLK}} + 26.186, \tag{3.22}$$

where t'_{dist} is the estimated processing time for the distributed GA with the same configuration. The result of Equation (3.22) is similar to the one obtained experimentally in Equation (3.19). It is important to emphasize again that the $t_{overhead}$ is calculated for the worst-case scenario (all individuals selected from the slave) and that is why the second term 26.186 is greater than 14.97. Figure 3.26 illustrates how the theoretical model is reasonable when compared to the experiments, by showing that the theoretical model (blue line) has approximately the same inclination of the experimental result (cyan line). What makes the theoretical model to be higher is because it represents the time for the DGA when the overhead is maximum (the worst-case scenario). For most practical applications, the overhead will be lower than this and the line will be shifted vertically to a lower position.

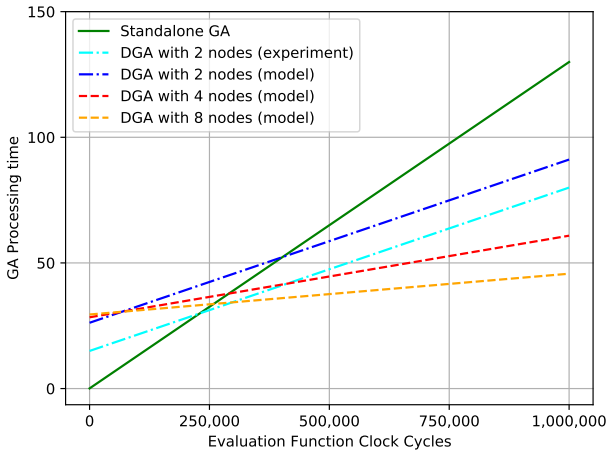

Figure 3.26: Comparison of model and experiments about how processing time of distributed genetic algorithm (DGA) scales with number of nodes, Q.

The second experiment was with the distributed version set up with reduced voltage and lower clock frequency for the same GA configuration used above. The motivation for this configuration is to take advantage of how dynamic power is defined for CMOS systems, which is present in regular microcontrollers (Le Sueur & Heiser 2010, Haririan 2020). By reducing the frequency and voltage, it is possible to reduce the power and con-

sequently energy consumption in a higher rate. This idea can be verified in the equation that defines the power, P, as the sum of the dynamic power, P_{dynamic}, and static power, P_{static}, in a CMOS integrated circuit and is defined as

$$P = P_{\text{dynamic}} + P_{\text{static}} = C \times f \times V^2 + P_{\text{static}}, \tag{3.23}$$

where C is the capacitance of the transistor gates, f the operating frequency, V the power supply voltage, and P_{static} the static power which depends mostly on the number of transistors and how they are organized spatially. Thus, by reducing the voltage V in the system, the reduction in the dynamic power will be in a quadratic level.

The behavior of Equation (3.23) can also be found out in the datasheet of the microcontroller ATmega328P (Mic 2018*a*). The Figure 3.27 shows what is the current I_{CC} consumed by the µC for different combinations of frequency (from 0 to 20 MHz) and voltage (from 2.7V to 5.5V). Since power can be also defined as

$$P = V \times I_{\text{CC}}, \tag{3.24}$$

then the power will be reduced for low values of voltage and frequency as well (power reduces from right to left and from top to bottom in Figure 3.27).

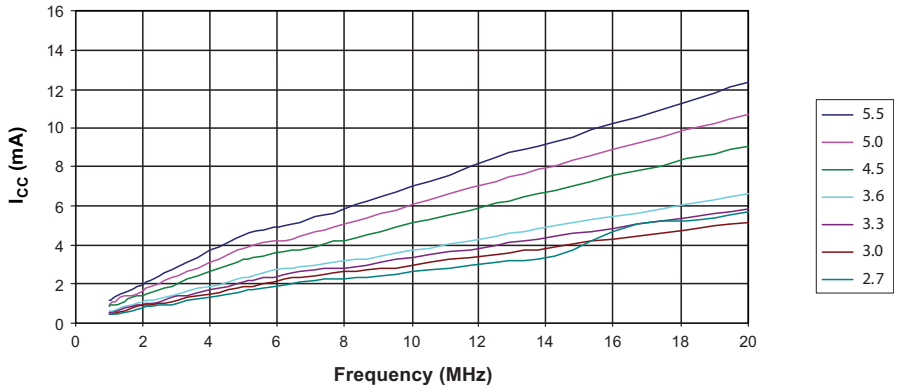

Figure 3.27: Relation of voltage, frequency and current in microcontroller ATmega328p.

To run this last experiment, both microcontrollers in the DGA were arranged to run at 8 MHz at a voltage of 2.7 V. This is the minimum operational voltage for this frequency, as shown in Figure 3.28. The processing time for the same configuration in the previous experiment in shown in Table 3.12. As expected, by running at a slower clock frequency made the processing time increase, and even in situations where the evaluation function is complex, the processing time for the distributed GA is always slower than the standalone

GA running at 16 MHz, as expressed in Table 3.11.

Table 3.12: Processing time (in seconds) distributed GA (2 nodes) running at 8 MHz.

Version	Evaluation Function Clock Cycles			
	122,055	**246,193**	**505,304**	**1,055,503**
Distributed GA	32.548286	48.669441	82.438400	154.786700

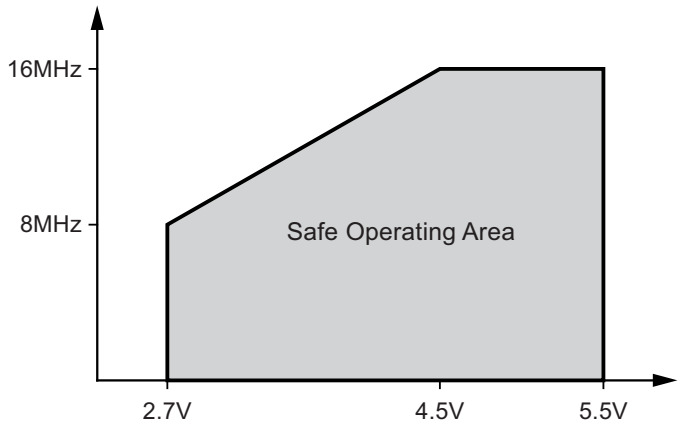

Figure 3.28: Operational levels of voltage and frequency in ATmega328P.

The comparison between these new results with the standalone version is presented in Figure 3.29. Both lines seem to be parallel, which suggests the distributed version with 2 nodes and half of the clock speed will never be faster than the standalone version. In fact, the first-degree polynomial functions that fit these points are calculated as follows

$$t_{\text{red}} = 0.000131 \times c_{\text{CLK}} + 16.43, \tag{3.25}$$

where t_{red} is the processing time for the DGA running at reduced clock speed. This equation has almost the same inclination of Equation (3.18) and the small difference may be a consequence of error/lack of precision of the measurements. Thus, this result suggests that for this GA configuration the DGA will always be about 16.43 s slower than the standalone GA, no matter how complex is the evaluation function. However, for long runs, the time difference will decrease relatively. For example, if the standalone GA takes 5 min, the DGA will take 5 min plus 16.43 s, which is only about 5% slower.

Even though the distributed genetic algorithm with 2 microcontrollers running at a half frequency of the standalone version is always slower, the main advantage of this

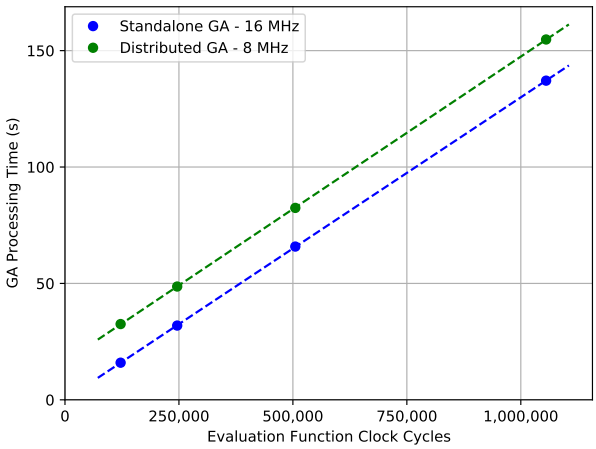

Figure 3.29: Comparison of standalone GA (1 node) and distributed GA (2 nodes) processing time with different clock frequencies.

structure is the save of power and consequently energy. This is one of the most common goals in embedded systems because they normally run on batteries and need to be power-efficient. The equation of energy consumption, E, is the product of the power equation by the elapsed time, defined as follows

$$E = P \times \Delta t = V \times I_{CC} \times \Delta t, \tag{3.26}$$

where Δt is the elapsed time. Since the elapsed time for the standalone and distributed versions on lower frequency, represented by t_{std} and t_{dist} respectively, were calculated in Equations (3.18) and (3.25), after applying them to Equation (3.26), the energy consumption equations for both cases are determined as

$$E_{std} = P_{std} \times t_{std} = P_{std} \times (0.0001299 \times c_{CLK} + 0.06282) \tag{3.27}$$

and

$$E_{red} = Q \times P_{red} \times t_{red} = Q \times P_{red} \times (0.000131 \times c_{CLK} + 16.43), \tag{3.28}$$

where E_{std}, P_{std}, and t_{std} are respectively the energy, power, and time consumption in the standalone system; E_{red}, P_{red}, and t_{red} are respectively the energy, power, and time consumption in the distributed system with reduced clock speed; and Q is the number of nodes in the distributed system ($Q = 2$ in these results).

From Figures 3.27 and 3.28 and considering both GA versions running at the lowest operating voltage for the specified frequency, the standalone version at 16 MHz needs

approximately 4.5 V and consumes a current of 7.5 mA, and the distributed version needs approximately 2.7 V and consumes a current of 2.3 mA. By using these values in Equations (3.27) and (3.28), respectively, the final expressions for energy are defined as

$$\begin{aligned} E_{\text{std}} &= 4.5 \times 7.5 \times (0.0001299 \times c_{\text{CLK}} + 0.06282) \\ &= 0.0043841 \times c_{\text{CLK}} + 2.120175 \text{ (mAh)} \end{aligned} \tag{3.29}$$

and

$$\begin{aligned} E_{\text{red}} &= 2 \times 2.7 \times 2.3 \times (0.000131 \times c_{\text{CLK}} + 16.43) \\ &= 0.0016270 \times c_{\text{CLK}} + 204.0606 \text{ (mAh)}, \end{aligned} \tag{3.30}$$

where the unity mAh means milliampere hour.

By plotting the energy consumption equations in Figure 3.30, the equation of the distributed version grows slower than the standalone version. The energy consumption in the distributed genetic algorithm for this configuration will be lower than the standalone GA when the evaluation function has at least 73,244 clock cycles, as demonstrated in

$$E_{\text{red}} = E_{\text{std}} = 0.0043841 \times c_{\text{CLK}} + 2.120175 = 0.0016270 \times c_{\text{CLK}} + 204.0606, \tag{3.31}$$

where the value of c_{CLK} that solves this equation is 73244.

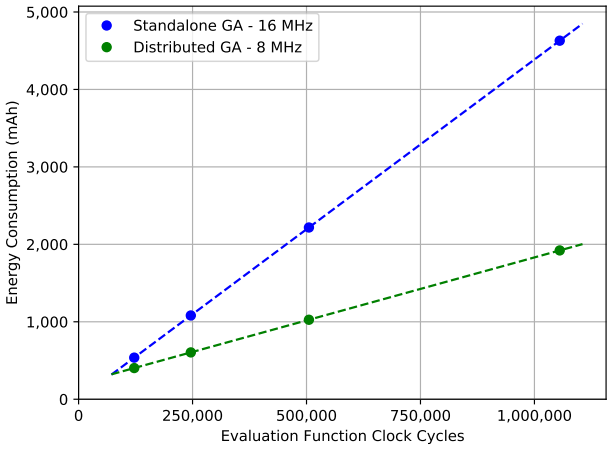

Figure 3.30: Comparison of standalone GA (1 node) and distributed GA (2 nodes) energy consumption with different clock frequencies.

For example, when the evaluation function requires around 1,000,000 clock cycles,

the standalone genetic algorithm needs approximately 130 s and 4400 mAh and the distributed GA approximately 147 s to run but only 1832 mAh, which is less than half of the energy spent by the standalone one. When the number of clock cycles is big enough, the distributed version will consume merely 37.1% regarding the standalone as demonstrated in

$$\lim_{c_{\text{CLK}} \to \infty} \frac{E_{\text{red}}}{E_{\text{std}}} = \frac{0.0016270}{0.0043841} = 0.3711138. \tag{3.32}$$

Therefore, the results presented in this section show some possible scenarios where the distributed genetic algorithm can have some advantages over a regular GA running on a single microcontroller. For situations where the evaluation function is not too complex, the standalone version is still the best option because it runs faster and consumes less energy. However, if it is complex enough, this proposed DGA, even having a large overhead due to the SPI communication, can be used either to accelerate the execution by running the microcontrollers in high frequency or to save power by reducing voltage and frequency. Finally, similar results are expected in case of employing more microcontrollers (4, 8, etc) and with more cores, the global clock could be even more reduced to 4 MHz, 2 MHz, and so on.

3.5 Conclusions

This work proposed a strategy to implement distributed genetic algorithms in 8-bit microcontrollers. Details about the implementation, constraints, and limitations were presented, as well as how this strategy is compared to others in the literature. Several experiments were done and showed that the DGA deployed as an embedded system has a low consumption of memory and works properly. Furthermore, the results regarding processing time exposed that there is a large overhead due to the communication via SPI, which makes this implementation not the best choice for problems where the evaluation function is not very complex. Nevertheless, when it is sufficiently complex, the distributed version can be used either to accelerate the run or to reduce the energy consumption by reducing the voltage and clock speed without losing so much performance compared to the regular GA.

Therefore, we concluded this implementation has demonstrated that it is feasible to be applied in embedded systems using 8-bit microcontrollers and can be a good alternative to a regular GA when the processing time of the valuation function is high. In this sense, it can be applied in numerous situations where this time limitation due to the SPI communication overhead is not a problem and may be useful for some non-real-time applications

in IoT, for instance. Finally, as future works, more results can be obtained by analyzing the performance scale with different clock frequencies for the SPI, with different communication protocols, with different distributed GA architectures, and with the addition of more microcontrollers as slaves.

Chapter 4

Conclusion

This work proposed and investigated the implementations of two versions of embedded genetic algorithms for low-power, low-cost and low-size-memory devices: a standalone version and a distributed version. The detailed implementations for both versions were presented, including the source code so that other researchers can compare their work with this one. For both cases, the results showed that it was possible to optimize the implementation to reduce memory, processing time, and energy consumption.

The standalone version was presented in Chapter 2 and was the basis of this work. The genetic algorithm was dismembered in several parts so that it was possible to identify areas that could be optimized and others that needed to have some restrictions to make it viable for limited hardware. The results for this version were satisfactory and identified different contexts where it may be applicable or not. Lastly, experiments showed that this version performed extremely well in all aspects when compared to other work found in the literature.

The distributed version was presented in Chapter 3 and took advantage of the results already obtained in the standalone version. Inspired in some existing strategies of distributed genetic algorithms, it was proposed a new architecture to allow the GA to run in multiple devices by transmitting data between themselves via the SPI interface. The results showed that there is a high overhead caused by the communication between the microcontrollers and that it is suitable for situations where this extra time is tolerable. However, the most interesting result was the possibility of running the GA with multiple μC but at a reduced clock speed and voltage with similar performance to the standalone version for certain scenarios where the evaluation function is costly enough. Hence, this version can be used to reduce power consumption without losing so much performance in some cases.

Therefore, the results obtained in this work are important because it is another effort of allowing an important artificial intelligence technique to optimized and viable to be used on such low-power, low-cost, and low-size-memory devices. These devices currently play

an essential role in the industry and household appliances and will be even more important with the growth of areas such as IoT, M2M, Smart Grid, etc. Thus, the implementations proposed in this work have can potentially be an important tool for existing and new embedded applications that need to solve optimization and search problems, and by being energetically efficient and sustainable.

4.1 Future works

The investigation made in this work generated important and satisfactory results but it also opens opportunities for further investigations in this area. Currently, several studies are aiming to implement artificial intelligence techniques directly in hardware, such as using reconfigurable computing like Field Programmable Gate Arrays (FPGAs), instead of aiming general-purpose processors like the ones found in microcontrollers. The hardware implementation can be much faster and more efficient and could be produced on a large scale at a much lower cost. Thus, the implementations proposed in this work could be ported to these platforms.

One possible way to make a transition from these implementations to FPGA is using the High-level Synthesis (HLS), which consists of an automated design process that converts an algorithm implemented in a high-level programming language to a hardware description language (HDL). By using this approach, both implementations could be adapted to run even faster and consuming less power. Moreover, the distributed version could be configured so that master and slave nodes would be blocks under the same chip and the data transfer would be much faster than SPI, which would reduce the global overhead and make it viable for more situations.

Another possible work is to analyze the distributed GA implementation in scenarios with more microcontrollers, such as with 4, 8, or 16 devices. In this context, a new analysis could be done in terms of parallelism, such as to calculate the scalability and speedup, for example. With these results, it would be possible to have a good understanding of what would be the ideal number of nodes to not increase more the price without gaining more performance.

Regarding the standalone GA implementation, a possible modification to be investigated is to update the population in-place and run some shuffling operation to not reduce the entropy for the search space. This might increase the processing time but reduce memory consumption for not needing a second array to store the new individuals. For the distributed GA, different types of topology could be evaluated for the communication between the devices to find out what is the one with less overhead. Finally, more exper-

iments could be done to calculate experimentally the exact energy consumption for both implementations using a oscilloscope or a auxiliary battery, for example.

Bibliography

Abdelhafez, Amr, Enrique Alba & Gabriel Luque (2019), 'Performance analysis of synchronous and asynchronous distributed genetic algorithms on multiprocessors', *Swarm and Evolutionary Computation* **49**, 147–157.

Abdulzaher, Mohamed (2019), *Artificial Intelligence Journalism - The 4IR and Media Restructuring*, 1ª edição, Badael Publishing House.

Al-Kofahi, Majd M., Mohammad Y. Al-Shorman & Osameh M. Al-Kofahi (2019), 'Toward energy efficient microcontrollers and internet-of-things systems', *Computers & Electrical Engineering* **79**, 106457.
URL: *http://www.sciencedirect.com/science/article/pii/S0045790618304129*

Alves, Nuno (2010), 'Implementing genetic algorithms on arduino micro-controllers', *CoRR* **abs/1002.2012**.
URL: *http://arxiv.org/abs/1002.2012*

Ard (2018), *Arduino UNO Rev3*.
URL: *https://store.arduino.cc/usa/arduino-uno-rev3*

Burkert, Andreas (2019), 'Artificial intelligence conquers the microcontroller', *ATZelectronics worldwide* **14**(4), 56–59.

Chen, Xuanang & Peijun Gao (2019), 'Path planning and control of soccer robot based on genetic algorithm', *Journal of Ambient Intelligence and Humanized Computing* pp. 1–10.

Cislak, Aleksander & Szymon Grabowski (2017), 'Lightweight fingerprints for fast approximate keyword matching using bitwise operations', *CoRR* **abs/1711.08475**.
URL: *http://arxiv.org/abs/1711.08475*

Coulouris, George, Jean Dollimore, Tim Kindberg & Gordon Blair (2011), *Distributed Systems: Concepts and Design*, 5thª edição, Addison-Wesley Publishing Company, USA.

de Oliveira, Atila V. F. M. & Marcelo A. C. Fernandes (2016), 'Dynamic planning navigation strategy for mobile terrestrial robots', *Robotica* **34**(3), 568–583.

Dendaluce, Martin, Juan José Valera, Vicente Gómez-Garay, Eloy Irigoyen & Ekaitz Larzabal (2014), Microcontroller implementation of a multi objective

genetic algorithm for real-time intelligent control, *em* Á.Herrero, B.Baruque, F.Klett, A.Abraham, V.Snášel, A. C.de Carvalho, P. G.Bringas, I.Zelinka, H.Quintián & E.Corchado, eds., 'International Joint Conference SOCO'13-CISIS'13-ICEUTE'13', Springer International Publishing, Cham, pp. 71–80.

Deshmukh, Ajay V (2005), *Microcontrollers: theory and applications*, Tata McGraw-Hill Education.

DiCarlo, David F (2012), 'Random number generation: Types and techniques'.
URL: *http://digitalcommons.liberty.edu/honors/308/*

Dong, Hongzhou, Yong Liu, Chunping Yang & Mingwu Ao (2017), 'Coherent accumulation by using sequence shifting and genetic algorithm', *Optik* **138**, 137 – 144.
URL: *http://www.sciencedirect.com/science/article/pii/S0030402617303479*

dos Santos, Willian GV, Wesley S Costa, Menno J Faber, Jair AL Silva, Helder RO Rocha & Marcelo EV Segatto (2019), Sensor allocation in a hybrid star-mesh iot network using genetic algorithm and k-medoids, *em* '2019 IEEE Latin-American Conference on Communications (LATINCOM)', IEEE, pp. 1–6.

Eiben, Agoston E & James E Smith (2015), *Introduction to evolutionary computing*, 3ª edição, Springer.

El-Saadawi, Magdi M, Eid Abdelbaqi Gouda, Mostafa A Elhosseini & Mohamed Said Essa (2020), 'Identification and speed control of dc motor using fractional order pid: Microcontroller', *European Journal of Electrical Engineering and Computer Science* **4**(1).

Elazhary, Hanan (2019), 'Internet of things (iot), mobile cloud, cloudlet, mobile iot, iot cloud, fog, mobile edge, and edge emerging computing paradigms: Disambiguation and research directions', *Journal of Network and Computer Applications* **128**, 105–140.

Eldahab, Yasser E. Abu, Naggar H. Saad & Abdalhalim Zekry (2017), 'Enhancing the tracking techniques for the global maximum power point under partial shading conditions', *Renewable and Sustainable Energy Reviews* **73**, 1173 – 1183.
URL: *http://www.sciencedirect.com/science/article/pii/S1364032117302320*

Fre (2018*a*), *The C Preprocessor*.
URL: *https://gcc.gnu.org/onlinedocs/cpp/index.html*

Fre (2018*b*), *The GNU C Library*.
URL: *https://www.gnu.org/software/libc/manual/*

Gaikwad, P. P., J. P. Gabhane & S. S. Golait (2015), A survey based on smart homes system using internet-of-things, *em* '2015 International Conference on Computation of Power, Energy, Information and Communication (ICCPEIC)', pp. 0330–0335.

Garzia, F. & G. M. Veca (2002), Evolutionary computation and genetic algorithms for energy management and conservation, *em* '24th Annual International Telecommunications Energy Conference', pp. 386–393.

Gendreau, Michel & Jean-Yves Potvin (2010), *Handbook of metaheuristics*, Vol. 2, 2ª edição, Springer.

Girolami, Alberto, Davide Brunelli & Luca Benini (2017), Low-cost and distributed health monitoring system for critical buildings, *em* '2017 IEEE Workshop on Environmental, Energy, and Structural Monitoring Systems (EESMS)', IEEE, pp. 1–6.

Gong, Yue-Jiao, Wei-Neng Chen, Zhi-Hui Zhan, Jun Zhang, Yun Li, Qingfu Zhang & Jing-Jing Li (2015), 'Distributed evolutionary algorithms and their models: A survey of the state-of-the-art', *Applied Soft Computing* **34**, 286–300.

Guo, Liucheng, Andreea Ingrid Funie, David B Thomas, Haohuan Fu & Wayne Luk (2016), 'Parallel genetic algorithms on multiple fpgas', *ACM SIGARCH Computer Architecture News* **43**(4), 86–93.

Harada, Tomohiro & Enrique Alba (2020), 'Parallel genetic algorithms: A useful survey', *ACM Computing Surveys (CSUR)* **53**(4), 1–39.

Haririan, Parham (2020), 'Dvfs and its architectural simulation models for improving energy efficiency of complex embedded systems in early design phase', *Computers* **9**(1), 2.

Hassija, Vikas, Vinay Chamola, Vikas Saxena, Divyansh Jain, Pranav Goyal & Biplab Sikdar (2019), 'A survey on iot security: application areas, security threats, and solution architectures', *IEEE Access* **7**, 82721–82743.

Hissoiny, Sami, Philippe Després & Benoît Ozell (2011), 'Using graphics processing units to generate random numbers', *arXiv preprint arXiv:1101.1846* .

Holland, John Henry (1992), *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*, MIT press.

Hunter, J. D. (2007), 'Matplotlib: A 2d graphics environment', *Computing In Science & Engineering* **9**(3), 90–95.

Ibrahim, Dogan (2011), *Advanced PIC microcontroller projects in C: from USB to RTOS with the PIC 18F series*, Newnes.

Kalaivanan, Anand Prabu & Gnanasekaran Sakthivel (2017), Self tuning genetic algorithm to achieve maximum thermal efficiency by monitoring combustion characteristics with vibration/acoustic sensors, Relatório técnico, SAE Technical Paper.

Kim, Minje & Paris Smaragdis (2016), 'Bitwise neural networks', *CoRR* **abs/1601.06071**. **URL:** *http://arxiv.org/abs/1601.06071*

Krope, J, A Ghani Olabi, D Goričanec & S Božičnik (2017), Optimization of automotive diesel engine calibration using genetic algorithm techniques, *em* '10TH International Conference on Sustainable Energy and Environmental Protection Technical Developments in Vehicles', p. 27.

Lamini, Chaymaa, Said Benhlima & Ali Elbekri (2018), 'Genetic algorithm based approach for autonomous mobile robot path planning', *Procedia Computer Science* **127**(C), 180–189.

Le Sueur, Etienne & Gernot Heiser (2010), Dynamic voltage and frequency scaling: The laws of diminishing returns, *em* 'Proceedings of the 2010 international conference on Power aware computing and systems', pp. 1–8.

Lu, Yang (2019), 'Artificial intelligence: a survey on evolution, models, applications and future trends', *Journal of Management Analytics* **6**(1), 1–29.

Luke, Sean (2013), *Essentials of Metaheuristics*, second[a] edição, Lulu. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

Mamdoohi, Ghazaleh, Ahmad Fauzi Abas, Khairulmizam Samsudin, Noor Hisham Ibrahim, Ariya Hidayat & Mohd Adzir Mahdi (2012), 'Implementation of genetic algorithm in an embedded microcontroller-based polarization control system', *Engineering Applications of Artificial Intelligence* **25**(4), 869 – 873. Special Section: Dependable System Modelling and Analysis.
URL: *http://www.sciencedirect.com/science/article/pii/S095219761200022X*

Medeiros, Denis R. da S., Matheus F. Torquato & Marcelo A. C. Fernandes (2020*a*), '[dataset] source code of embedded genetic algorithm for low-power, low-cost andlow-size-memory devices', https://doi.org/10.5281/zenodo.3890704.

Medeiros, Denis R. da S., Matheus F. Torquato & Marcelo A. C. Fernandes (2020*b*), 'Embedded genetic algorithm for low-power, low-cost and low-size-memory devices', *Engineering Reports* .

Megantoro, Prisma, Yabes Dwi Nugroho, Fajar Anggara, Aji Pakha & Brahmantya Aji Pramudita (2018), The implementation of genetic algorithm to mppt technique in a dc/dc buck converter under partial shading condition, *em* '2018 3rd International Conference on Information Technology, Information System and Electrical Engineering (ICITISEE)', IEEE, pp. 308–312.

Mic (2018*a*), *ATmega328P Datasheet*.
URL: *http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf*

Mic (2018*b*), *Atmel AVR4027: Tips and Tricks to Optimize Your C Code for 8-bit AVR Microcontrollers*.
URL: *http://ww1.microchip.com/downloads/en/AppNotes/doc8453.pdf*

Mikhaylov, K. & J. Tervonen (2012), Evaluation of power efficiency for digital serial interfaces of microcontrollers, *em* '2012 5th International Conference on New Technologies, Mobility and Security (NTMS)', pp. 1–5.

Millo, Federico, Pranav Arya & Fabio Mallamo (2018), 'Optimization of automotive diesel engine calibration using genetic algorithm techniques', *Energy* **158**, 807 – 819.
URL: *http://www.sciencedirect.com/science/article/pii/S0360544218311095*

Mishra, Sanjeeb, Neeraj Kumar Singh & Vijayakrishnan Rousseau (2015), *System on chip interfaces for low power design*, Morgan Kaufmann.

Morell, JA & Enrique Alba (2017), Distributed genetic algorithms on portable devices for smart cities, *em* 'International Conference on Smart Cities', Springer, pp. 51–62.

Nazarahari, Milad, Esmaeel Khanmirza & Samira Doostie (2019), 'Multi-objective multi-robot path planning in continuous environment using an enhanced genetic algorithm', *Expert Systems with Applications* **115**, 106–120.

Noraini, Mohd Razali & John Geraghty (2011), 'Genetic algorithm performance with different selection strategies in solving tsp', *International Conference of Computational Intelligence and Intelligent Systems (ICCIIS'11)* .

Ortega-Zamorano, F., J. M. Jerez, D. Urda Muñoz, R. M. Luque-Baena & L. Franco (2016), 'Efficient implementation of the backpropagation algorithm in fpgas and microcontrollers', *IEEE Transactions on Neural Networks and Learning Systems* **27**(9), 1840–1850.

Panda, Nihar Ranjan, PNS Sailaja & Rupali Satapathy (2019), 'Applications of microcontrollers', *Electronics and Communications Engineering: Applications and Innovations* pp. 293–303.

Pedemonte, Martín, Enrique Alba & Francisco Luna (2011), Bitwise operations for gpu implementation of genetic algorithms, *em* 'Proceedings of the 13th annual conference companion on Genetic and evolutionary computation', pp. 439–446.

Pohlheim, Hartmut (2007), 'Examples of objective functions', *Retrieved* **4**(10), 2012.

Qian, Kai, David Den Haring & Li Cao (2009), Survey of popular microcontrollers, *em* 'Embedded Software Development with C', Springer, pp. 203–221.

Risset, Tanguy (2011), *SoC (System on Chip)*, Springer US, Boston, MA, pp. 1837–1842.

Sakr, Fouad, Francesco Bellotti, Riccardo Berta & Alessandro De Gloria (2020), 'Machine learning on mainstream microcontrollers', *Sensors* **20**(9), 2638.

Santos, Pedro Henriquede Oliveira, Gustavo Luís Soares, Thiago Melo Machado-Coelho, Bernardo Augusto Godinho de Oliveira, Petr Iakovlevitch Ekel, Flávia Magalhães Freitas Ferreira & Carlos Augusto Paiva da Silva Martins (2018), Multi-objective genetic algorithm implemented on a stm32f microcontroller, *em* '2018 IEEE Congress on Evolutionary Computation (CEC)', IEEE, pp. 1–7.

Sawulski, J. & M. Ławryńczuk (2019), Real-time optimisation of control strategy for a low fuel consumption vehicle engine using stm32 microcontroller: Preliminary results, *em* '2019 18th European Control Conference (ECC)', pp. 4246–4251.

Shelokar, Prakash, Abhijit Kulkarni, Valadi K. Jayaraman & Patrick Siarry (2014), *Metaheuristics in Process Engineering: A Historical Perspective"*, Springer International Publishing, Cham, pp. 1–38.

Silva, Caroline AD, Átila VFM De Oliveira & Marcelo AC Fernandes (2018), 'Validation of a dynamic planning navigation strategy applied to mobile terrestrial robots', *Sensors* **18**(12), 4322.

Sittón-Candanedo, Inés, Ricardo S Alonso, Óscar García, Lilia Muñoz & Sara Rodríguez-González (2019), 'Edge computing, iot and social computing in smart energy scenarios', *Sensors* **19**(15), 3353.

Situmorang, Z. & J. A. Situmorang (2015), Intelligent fuzzy controller for a solar energy wood dry kiln process, *em* '2015 International Conference on Technology, Informatics, Management, Engineering Environment (TIME-E)', pp. 152–157.

Steele Jr, Guy L, Doug Lea & Christine H Flood (2014), Fast splittable pseudorandom number generators, *em* 'ACM SIGPLAN Notices', Vol. 49, ACM, pp. 453–472.

Sumalan, Radu L, Nicoleta Stroia, Daniel Moga, Vlad Muresan, Alexandru Lodin, Teodor Vintila & Cosmin A Popescu (2020), 'A cost-effective embedded platform for greenhouse environment control and remote monitoring', *Agronomy* **10**(7), 936.

Talbi, El-Ghazali (2009), *Metaheuristics: from design to implementation*, 1ª edição, John Wiley & Sons, Hoboken, NJ, USA.

Tangen, Helene (2016), Wind farm layout optimization using population distributed genetic algorithms, Dissertação de mestrado, NTNU.

The 2019 AI Index Report (n.d.).
URL: *https://hai.stanford.edu/research/ai-index-2019*

Torquato, Matheus F. & Marcelo A. C. Fernandes (2019), 'High-performance parallel implementation of genetic algorithm on fpga', *Circuits, Systems, and Signal Processing* **38**(9), 4014–4039.
URL: *https://doi.org/10.1007/s00034-019-01037-w*

van Steen, Maarten & Andrew S Tanenbaum (2016), 'A brief introduction to distributed systems', *Computing* **98**(10), 967–1009.

Wang, Yu Emma, Gu-Yeon Wei & David Brooks (2019), 'Benchmarking tpu, gpu, and cpu platforms for deep learning', *arXiv preprint arXiv:1907.10701* .

Wu, Zhenyu, Kai Qiu & Jianguo Zhang (2020), 'A smart microcontroller architecture for the internet of things', *Sensors* **20**(7), 1821.

Yampolskiy, Roman V (2020), 'On defining differences between intelligence and artificial intelligence', *Journal of Artificial General Intelligence* **11**(2), 68–70.

Yordzhev, Krasimir (2013), 'The bitwise operations related to a fast sorting algorithm', *CoRR* **abs/1312.0138**.
URL: *http://arxiv.org/abs/1312.0138*

Yordzhev, Krasimir Yankov (2012), 'An example for the use of bitwise operations in programming', *CoRR* **abs/1201.1468**.
URL: *http://arxiv.org/abs/1201.1468*

Zakaria, Mohamed, TM Abdel-Moneim, Hesham Abdin, Alaa El-Din Hafez & Samy Darwish (2017), Optimization and mechanical simulation of a pursuit-evader scenario using genetic algorithm and stewart platform, *em* '2017 8th International Conference on Mechanical and Aerospace Engineering (ICMAE)', IEEE, pp. 314–319.

Zomaya, Albert Y & Young Choon Lee (2012), *Energy-efficient distributed computing systems*, Vol. 88, John Wiley & Sons.