Article

Distributed Genetic Algorithms for Low-Power, Low-Cost and Small-Sized Memory Devices

Denis R. da S. Medeiros¹ and Marcelo A. C. Fernandes^{1,2,*†}

¹ Laboratory of Machine Learning and Intelligent Instrumentation, Federal University of Rio Grande do Norte, Natal 59078-970, Brazil; dnsricardo@gmail.com

² Department of Computer and Automation Engineering, Federal University of Rio Grande do Norte, Natal 59078-970, Brazil

* Correspondence: mfernandes@dca.ufrn.br

† Current address: John A. Paulson School of Engineering and Applied Sciences, Harvard University, Cambridge, MA 02138, USA.

Received: 30 October 2020; Accepted: 6 November 2020; Published: 11 November 2020

Abstract: This work 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 (μ Cs) and Serial Peripheral Interface (SPI) for data transmission between those devices. 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.

Keywords: distributed system; genetic algorithms; microcontrollers; embedded system

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 [1]. 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 [2]. 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 [3]. 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 [4–7].

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 [8]. 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 μ Cs 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 [9], 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 [10], 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 [11] 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 [12], 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. 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 model presented in [13].

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 [14]. 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 [15–17]. Furthermore, the use of GAs embedded in automobiles to calibrate car engine parameters as presented

in [18,19]. 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 μ Cs 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 basic structure of the standalone genetic algorithm, in Section 2, as well as different approaches to implement the distributed genetic algorithm in Section 3. Inspired on the existing architectures, this work 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 4. Finally, in Section 5 results are presented showing how this new architecture performs in terms of memory, processing time, and power consumption.

2. Genetic Algorithms

For the scope of this work, genetic algorithms can be defined as iterative algorithms that start by randomly generating a population of N individuals and after K iterations, called generations, those individuals will converge to some specific result. Each individual is mapped into M bits and during each generation, k -th iteration of the algorithm, the population passes through operations of evaluation, selection, crossover, and mutation. At the end of the generation, a new population of the same size N is generated and then it will become the starting point of the following generation. After this cyclic process repeats K times, most of the individuals are expected to be concentrated around the same values and the best one can be used as the result.

The Algorithm 1 represents the pseudocode of the GA described above, the same presented in [13], and it is inspired by [3]. 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

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

The GA starts with 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 j_b to be used in the elitism operation later. The fitness value 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 value. 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

$$\mathbf{y}(k) = \begin{bmatrix} y_0[B](k) & \dots & y_{N-1}[B](k) \end{bmatrix}. \quad (2)$$

After the evaluation, the next operation is the selection, where some individuals are selected and the best ones, with better fitness value $y_j[B](k)$, are combined to generate new and possibly better individuals for the next generation. 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, [20]. For this work, the tournament selection is applied since

it is one of the most used and efficient methods according to [21]. The selection function is represented in the pseudocode as SF (Line 10 of Algorithm 1). Finally, the elitism technique can also be applied, so that the best E individuals of the current population are passed directly to the new population without being combined. In this work, $E = 1$ and the best individual is placed on the first position of the new population (Line 16 of Algorithm 1).

Algorithm 1 Genetic Algorithm Pseudocode

```

    ▷ Generation of the initial population
1: Initialize( $X(0)$ )
    ▷ 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:      $y_j[B](k) \leftarrow EF(x_j(k))$ 
5:     if  $y_j[B](k) < y_{jb}[B](k)$  then
6:        $jb \leftarrow j$ 
7:     end if
8:   end for
    ▷ Selection and crossover
9:   for  $i \leftarrow 0$  to  $N - 1$  with step 2 do
10:     $\begin{bmatrix} z_i(k) \\ z_{i+1}(k) \end{bmatrix} \leftarrow CF \left( \begin{bmatrix} SF(y(k), X(k)) \\ SF(y(k), X(k)) \end{bmatrix} \right)$ 
11:   end for
    ▷ Mutation
12:   for  $v \leftarrow 0$  to  $P - 1$  do
13:      $z_v(k) \leftarrow MF(z_v(k))$ 
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
  
```

The operation following selection is called crossover, where two or more selected individuals from the current population, $X(k)$, are combined to generate new ones that will be inserted into the new population, $X(k + 1)$, 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 [22]. In this work, either of these three options can be used. The crossover function is defined as CF (Line 10 of Algorithm 1) and the offspring is stored into the matrix $Z(k)$, which is defined as

$$Z(k) = \begin{bmatrix} z_0(k) & \dots & z_{N-1}(k) \end{bmatrix} = \begin{bmatrix} z_{0,0}[M](k) & \dots & z_{0,D-1}[M](k) \\ \vdots & \ddots & \vdots \\ z_{N-1,0}[M](k) & \dots & z_{N-1,D-1}[M](k) \end{bmatrix}. \quad (3)$$

After the new individuals are inserted in the $Z(k)$, they are processed through the operation called 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). The mutation rate, called R_M , defines the proportion of individuals that suffer mutation, hence P can be specified as

$$P = \lceil R_M \times N \rceil. \quad (4)$$

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 substituted [23]. 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).

3. 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 [24] 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 4.

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 1.

Figure 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

$$V = \frac{N}{Q}. \quad (5)$$

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

$$\begin{aligned} \mathbf{X}'_{\mathbf{q}}(k) &= \begin{bmatrix} \mathbf{x}_{q \times V}(k) & \dots & \mathbf{x}_{(q+1) \times V-1}(k) \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{x}_{q \times V,0}[M](k) & \dots & \mathbf{x}_{q \times V,D-1}[M](k) \\ \vdots & \ddots & \vdots \\ \mathbf{x}_{(q+1) \times V-1,0}[M](k) & \dots & \mathbf{x}_{(q+1) \times V-1,D-1}[M](k) \end{bmatrix}. \end{aligned} \quad (6)$$

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 2 and 3, respectively.

Figure 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. 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 4.

Figure 4. Pool model for distributed genetic algorithms.

4. Implementation

4.1. 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, 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 chance 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 5.

Once these operations are done, all μ Cs 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 6.

Figure 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 [13] 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

$$Q = 2^{A_1}, \text{ where } A_1 \in \mathbb{N}^*. \quad (7)$$

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

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

Ultimately, another consequence of this new N' is that it can be bigger than 256. Therefore, the type `popsizet`, 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.

Figure 6. Master-slave distributed GA model proposed by this work for distributed genetic algorithms showing asynchronous and synchronous operations.

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 2 and 3. In both Algorithms 2 and 3, 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.
 - Send Best Individual Function (SBIF)—Slave sends its best individual to master.

Algorithm 2 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:      $indices_l \leftarrow RNG() \wedge (N - 1)$ 
7:      $scFitness_l \leftarrow CEVF(indices_l)$ 
8:   end for
  ▷ Run the tournament selection method using pairs of fitness values.
9:    $indexWinner_0 \leftarrow TF(scFitness_0, scFitness_1)$ 
10:   $indexWinner_1 \leftarrow TF(scFitness_2, scFitness_3)$ 
  ▷ Collect the 2 individuals that won the tournament from the nodes.
11:   $scIndividual_0 \leftarrow CIF(indexWinner_0)$ 
12:   $scIndividual_1 \leftarrow CIF(indexWinner_1)$ 
  ▷ Cross the 2 individuals collected from the nodes and generates 2 new individuals.
13:   $scNewIndividuals \leftarrow CF(scIndividual_0, scIndividual_1)$ 
  ▷ Send the 2 new individuals to one node.
14:   $SIF(scNewIndividuals_0)$ 
15:   $SIF(scNewIndividuals_1)$ 
16: end for
  ▷ Inform all nodes to continue the remaining operations.
17: for  $q \leftarrow 0$  to  $Q - 1$  do
18:    $COF(q)$ 
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:    $bestIndividuals_q \leftarrow CBIF(q)$ 
25: end for

```

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 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 3 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 (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

$$R'_M = \frac{P \times Q}{N}. \quad (9)$$

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.

4.2. Communication between Microcontrollers

To implement the distributed genetic algorithm architecture proposed in Section 4.1, 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) [25]. 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 [26]. 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 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 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 [27]. 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 4.1, 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

$$c_{\text{CLK}}^{\text{ind}} = M \times D + c_{\text{CLK}}^{\text{trans}} \quad (10)$$

$$c_{\text{CLK}}^{\text{fit}} = B + c_{\text{CLK}}^{\text{trans}} \quad (11)$$

where c_{CLK}^{ind} represents the number of clock cycles to transmit one individual, c_{CLK}^{fit} to transmit a fitness value, and c_{CLK}^{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 8 and a list of the commands and acknowledge messages is shown in Table 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 (10) and (11), c_{CLK}^{trans} is 16 then.

Figure 8. Example of how the the protocol based on commands and acknowledge messages works.

Table 1. List of commands and acknowledge messages used by the proposed protocol.

Name	Value	Description
CMD_SEND_BYTE	$0 \times C0$	Master sends a byte.
ACK_SEND_BYTE	$0 \times A0$	
CMD_RECEIVE_BYTE	$0 \times C1$	Master receives a byte.
ACK_RECEIVE_BYTE	$0 \times A1$	
CMD_SEND_FLOAT	$0 \times C2$	Master sends a float.
ACK_SEND_FLOAT	$0 \times A2$	
CMD_RECEIVE_FLOAT	$0 \times C3$	Master receives a float.
ACK_RECEIVE_FLOAT	$0 \times A3$	
CMD_COLLECT_EV	$0 \times C4$	Master receives evaluation value.
ACK_COLLECT_EV	$0 \times A4$	
CMD_COLLECT_IND	$0 \times C5$	Master receives an individual.
ACK_COLLECT_IND	$0 \times A5$	
CMD_SEND_IND	$0 \times C6$	Master sends an individual.
ACK_SEND_IND	$0 \times A6$	
CMD_CONTINUE_OP	$0 \times C7$	Master tells slaves to continue.
ACK_CONTINUE_OP	$0 \times A7$	
CMD_COLLECT_BEST_IND	$0 \times C8$	Master receives the best individual.
ACK_COLLECT_BEST_IND	$0 \times A8$	

4.3. Scalability and Overhead

With the Algorithms 2 and 3 and the communication protocol described in Section 4.2, 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 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

$$H_{\text{sel-cross}} = \left[6N + \left(2 + \frac{MD}{8} \right) \times N + \left(2 + \frac{MD}{8} \right) \times \left(N - \frac{N}{Q} \right) \right] \times K + 2 \times (Q - 1), \quad (12)$$

and

$$H_{\text{col}} = \left(2 + \frac{MD}{8} \right) \times (Q - 1), \quad (13)$$

where $H_{\text{sel-cross}}$ 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 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 (12) and (13). Using this result, the equation to calculate the total overhead in seconds is

$$t_{\text{overhead}} \leq \frac{8}{c_{\text{SPI}}^{\text{CLK}}} \times (H_{\text{sel-cross}} + H_{\text{col}}) + \Delta, \quad (14)$$

where t_{overhead} is the time spent with the transfers, in seconds; $c_{\text{CLK}}^{\text{SPI}}$ 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 (14) 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 [13] 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

$$t_{\text{GA}} = t_{\text{IFM}} + K \times (t_{\text{FFM}} + t_{\text{NPFM}}), \quad (15)$$

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 (15), the equation can be rewritten as

$$t_{GA} = N\phi_1 + K \times (N\phi_2 + N\phi_3 + N\phi_4 + \phi_5) = N \times [\phi_1 + K \times (\phi_2 + \phi_3 + \phi_4)] + K\phi_5, \quad (16)$$

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 [13], therefore, if the devices are running at the same clock speed, the total time for the DGA is the same expression of Equation (16) but with the population divided between Q nodes plus the $t_{overhead}$, that is,

$$t_{DGA} = \frac{N}{Q} \times [\phi_1 + K \times (\phi_2 + \phi_3 + \phi_4)] + K\phi_5 + t_{overhead}. \quad (17)$$

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

$$t_{DGA} = \frac{1}{Q} (t_{GA} + K\phi_5) + t_{overhead}. \quad (18)$$

The result of Equation (18) is important because allows estimating how the processing time for the DGA will be based on how the standalone GA performs. Also, since $t_{GA} \gg \frac{K\phi_5}{Q}$ 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

$$t_{overhead} \times \frac{Q}{Q-1} \leq t_{GA}, \text{ where } Q \in \mathbb{N} \text{ and } Q > 1. \quad (19)$$

5. 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 technologies employed in [13]. 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 2 and 3). 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 [28]. 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 (7). 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 4.2. 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

$$\Delta = \frac{1}{1000} \times (H_{\text{sel-cross}} + H_{\text{col}}). \quad (20)$$

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 9.

Figure 9. Arrangement of how the Arduino Uno boards were connected.

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 [13]. 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.

5.1. 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 $f_1(\bar{x}) = \bar{x}_0^2 - 6\bar{x}_0 + 8$, 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 2 and 3, 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 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

Table 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

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 4 and 5, sequentially. The numbers obtained in this work are similar to those obtained in [13], because after dividing the global population both microcontrollers got the same population size used in the experiments in that work.

Table 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

Table 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 10 and 11, 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.

Figure 10. Stack memory consumption in master node.

Figure 11. Stack memory consumption in slave node.

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 individuals 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 (7).

5.2. Processing Time

The second results collected from this distributed genetic algorithm implementation was the processing time. The methodology used in [13], 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 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:

- $f_1(\bar{x}) = \bar{x}_0^2 - 6\bar{x}_0 + 8$
- $f_2(\bar{x}) = \bar{x}_0^2 + \bar{x}_1^2$
- $f_3(\bar{x}) = 2\bar{x}_0 + 3\bar{x}_1 + 5$
- $f_4(\bar{x}) = 21.5 + \bar{x}_0 (\sin(40\pi\bar{x}_0) + \cos(20\pi\bar{x}_0))$
- $f_5(\bar{x}) = 10 + \bar{x}_0^2 - 10\cos(2\pi\bar{x}_0) + \bar{x}_1^2 - 10\cos(2\pi\bar{x}_1)$

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 6. The processing time seems to not change so much with the type of evaluation function and this can be noticed when comparing functions $f_2(\bar{x})$, $f_3(\bar{x})$ and $f_5(\bar{x})$, which have different mathematical operations but same number of dimensions and similar run times. The same happened for $f_1(\bar{x})$ and $f_4(\bar{x})$, 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 6. Processing time for different evaluation functions.

Evaluation Function	Average Time (s)	Standard Deviation (s)
$f_1(\bar{x})$	14.759885	0.0675244
$f_2(\bar{x})$	17.248525	0.0985764
$f_3(\bar{x})$	17.221671	0.1402979
$f_4(\bar{x})$	15.003993	0.0635690
$f_5(\bar{x})$	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 $f_4(\bar{x})$ was used. For evaluate D , it was used the evaluation function $f_2(\bar{x})$, 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 7–9, 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 7 and 8. This impression can be proved in Figures 12 and 13, wherein both cases the points seem to represent a first-degree polynomial function.

Table 7. Average processing time (in seconds) for different values of N , using $f_4(\bar{x})$ 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 8. Average processing time (in seconds) for different values of K , using $f_4(\bar{x})$ 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 9. Average processing time (in seconds) for different values of D , using $f_2(\bar{x})$ 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

Figure 12. Best polynomial function that fits processing time for different values of population size, N .

Finally, the results for the number of dimensions D are presented in Table 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 14 and shows how the increase is expected for different values of D .

Figure 13. Best polynomial function that fits processing time for different values of number of generations, K .

Figure 14. Best polynomial function that fits processing time for different values of number of dimensions, D .

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.

5.3. 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 $f_2(\bar{x})$ and $f_4(\bar{x})$, which are shown in Figures 15 and 16, respectively.

Figure 15. Chart showing values of $f_2(\bar{x})$, with x_0 and x_1 from -5 to 5 .

Figure 16. Chart showing values of $f_4(\bar{x})$, with x from 0.8 to 1.0 .

The first experiment was using the evaluation function $f_2(\bar{x})$, where the goal is to find the global minimum. The search space for all dimensions was defined between -5 and 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 17 and for the slave in Figure 18, 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 $f_4(\bar{x})$. The intention was to find 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 19 and 20, respectively.

Figure 17. Convergence of the dimensions x_0 and x_1 of $f_2(\bar{x})$ in master node.

Figure 18. Convergence of the dimensions x_0 and x_1 of $f_2(\bar{x})$ in slave node.

Figure 19. Convergence of the dimension x of $f_4(\bar{x})$ in master node.

Figure 20. Convergence of the dimension x of $f_4(\bar{x})$ in slave node.

5.4. 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 [13]. 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 to the standalone version.

By analyzing the results presented in Section 5.2, 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 4.

Algorithm 4 Redefinition of Evaluation Function to Become Slower

```

▷ Define how many times the evaluation function will repeat (2000 times, for example).
1: REPEAT ← 2000
▷ The original evaluation function will run REPEAT times.
2: function EFM( $\bar{x}(k)$ )
3:   result ← 0
4:   for  $r \leftarrow 0$  to REPEAT − 1 do
5:     result ← result +  $f(\bar{x}_0[M](k), \dots, \bar{x}_i[M](k), \dots, \bar{x}_{D-1}[M](k))$ 
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 $f_4(\bar{x})$, which was set up to repeat 1000, 2000, 4000 and 8000 times using the strategy proposed in Algorithm 4. 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

$$t_{\text{EFM-slow}} = c_{\text{CLK}}^{f_{\text{slow}}(\times)} \times \frac{1}{\text{CLK}}, \quad (21)$$

where $c_{CLK}^{f_{slow}(\times)}$ 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 $c_{CLK}^{f_{slow}(\times)}$ were collecting via experiments in Atmel Studio 7 and are shown in Table 10.

Table 10. Clock cycles of $f_4(\bar{x})$ 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 11 and Figure 21, 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 21, which is in the format $t = a \times c_{CLK} + b$, and is defined as

$$t_{std} = 0.0001299 \times c_{CLK} + 0.06282 \quad (22)$$

for the standalone version, and

$$t_{dist} = 0.00006501 \times c_{CLK} + 14.97 \quad (23)$$

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

$$\lim_{c_{CLK} \rightarrow \infty} \frac{t_{std}}{t_{dist}} = \frac{0.0001299}{0.00006501} = 1.998154. \quad (24)$$

Figure 21. Comparison of standalone GA (1 node) and distributed GA (2 nodes) processing time with same clock frequencies.

Table 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 (23) is the high overhead. By applying the Genetic Algorithm parameters and the SPI clock frequency, defined in 125 kHz, to Equations (14) and (20), it is possible to see how the theoretical model compares to the experiments. The $t_{overhead}$ is calculated as follows

$$t_{overhead} \leq \frac{8}{125,000} \times (24,578 + 4) + \frac{1}{1000} \times (24,578 + 4) = 26.155, \quad (25)$$

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 (23) is reasonable and under the theoretical limit.

Finally, to validate the theoretical model presented in Equation (18), by applying the results from the experiment shown in Equation (22) and from Equation (25), the expected equation for the distributed versions would be

$$t'_{dist} \leq \frac{t_{std}}{2} + t_{overhead} = 0.00006495 \times c_{CLK} + 26.186, \quad (26)$$

where t'_{dist} is the estimated processing time for the distributed GA with the same configuration. The result of Equation (26) is similar to the one obtained experimentally in Equation (23). 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 22 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 22. 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 [29,30]. By reducing the frequency and voltage, it is possible to reduce the power and consequently 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

$$P = P_{\text{dynamic}} + P_{\text{static}} = C \times f \times V^2 + P_{\text{static}}, \quad (27)$$

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 (27) can also be found out in the datasheet of the microcontroller ATmega328P [28]. The Figure 23 shows what is the current I_{CC} consumed by the μC for different combinations of frequency (from 0 to 20 MHz) and voltage (from 2.7 V to 5.5 V). Since power can be also defined as

$$P = V \times I_{CC}, \quad (28)$$

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 23).

Figure 23. 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 24. The processing time for the same configuration in the previous experiment is shown in Table 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 11.

Table 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 24. Operational levels of voltage and frequency in ATmega328P.

The comparison between these new results with the standalone version is presented in Figure 25. 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

$$t_{\text{red}} = 0.000131 \times c_{\text{CLK}} + 16.43, \quad (29)$$

where t_{red} is the processing time for the DGA running at reduced clock speed. This equation has almost the same inclination of Equation (22) 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.

Figure 25. Comparison of standalone GA (1 node) and distributed GA (2 nodes) processing time with different clock frequencies.

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 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

$$E = P \times \Delta t = V \times I_{CC} \times \Delta t, \quad (30)$$

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 (22) and (29), after applying them to Equation (30), the energy consumption equations for both cases are determined as

$$E_{std} = P_{std} \times t_{std} = P_{std} \times (0.0001299 \times c_{CLK} + 0.06282) \quad (31)$$

and

$$E_{red} = Q \times P_{red} \times t_{red} = Q \times P_{red} \times (0.000131 \times c_{CLK} + 16.43), \quad (32)$$

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 23 and 24 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 (31) and (32), respectively, the final expressions for energy are defined as

$$\begin{aligned} E_{std} &= 4.5 \times 7.5 \times (0.0001299 \times c_{CLK} + 0.06282) \\ &= 0.0043841 \times c_{CLK} + 2.120175 \text{ (mAh)} \end{aligned} \quad (33)$$

and

$$\begin{aligned} E_{red} &= 2 \times 2.7 \times 2.3 \times (0.000131 \times c_{CLK} + 16.43) \\ &= 0.0016270 \times c_{CLK} + 204.0606 \text{ (mAh)}, \end{aligned} \quad (34)$$

where the unity mAh means milliamperere hour.

By plotting the energy consumption equations in Figure 26, 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

$$E_{red} = E_{std} = 0.0043841 \times c_{CLK} + 2.120175 = 0.0016270 \times c_{CLK} + 204.0606, \quad (35)$$

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

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

$$\lim_{c_{CLK} \rightarrow \infty} \frac{E_{red}}{E_{std}} = \frac{0.0016270}{0.0043841} = 0.3711138. \quad (36)$$

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.

Figure 26. Comparison of standalone GA (1 node) and distributed GA (2 nodes) energy consumption with different clock frequencies.

6. 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.

Author Contributions: Conceptualization, D.R.d.S.M. and M.A.C.F.; methodology, D.R.d.S.M. and M.A.C.F.; software and validation, D.R.d.S.M. and M.A.C.F.; data curation, D.R.d.S.M. and M.A.C.F.; writing—original draft preparation, D.R.d.S.M.; writing—review and editing, D.R.d.S.M. and M.A.C.F.; supervision, M.A.C.F.; project administration, M.A.C.F. All authors have read and agreed to the published version of the manuscript.

Funding: This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES)—Finance Code 001.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Coulouris, G.; Dollimore, J.; Kindberg, T.; Blair, G. *Distributed Systems: Concepts and Design*, 5th ed.; Addison-Wesley Publishing Company: Boston, MA, USA, 2011.
2. van Steen, M.; Tanenbaum, A.S. A brief introduction to distributed systems. *Computing* **2016**, *98*, 967–1009. [\[CrossRef\]](#)
3. Eiben, A.E.; Smith, J.E. *Introduction to Evolutionary Computing*, 3rd ed.; Springer: Berlin, Germany, 2015; pp. 99–100.
4. Tangen, H. Wind Farm Layout Optimization Using Population Distributed Genetic Algorithms. Master's Thesis, NTNU, Trondheim, Norway, 2016.
5. Guo, L.; Funie, A.I.; Thomas, D.B.; Fu, H.; Luk, W. Parallel genetic algorithms on multiple FPGAs. *ACM SIGARCH Comput. Archit. News* **2016**, *43*, 86–93. [\[CrossRef\]](#)
6. Abdelhafez, A.; Alba, E.; Luque, G. Performance analysis of synchronous and asynchronous distributed genetic algorithms on multiprocessors. *Swarm Evol. Comput.* **2019**, *49*, 147–157. [\[CrossRef\]](#)
7. Harada, T.; Alba, E. Parallel Genetic Algorithms: A Useful Survey. *ACM Comput. Surv. (CSUR)* **2020**, *53*, 1–39. [\[CrossRef\]](#)
8. Deshmukh, A.V. *Microcontrollers: Theory and Applications*; Tata McGraw-Hill Education: New Delhi, India, 2005.
9. Girolami, A.; Brunelli, D.; Benini, L. Low-cost and distributed health monitoring system for critical buildings. In Proceedings of the 2017 IEEE Workshop on Environmental, Energy, and Structural Monitoring Systems (EESMS), Milan, Italy, 24–25 July 2017; IEEE: New York City, NY, USA, 2017; pp. 1–6.
10. Sumalan, R.L.; Stroia, N.; Moga, D.; Muresan, V.; Lodin, A.; Vintila, T.; Popescu, C.A. A Cost-Effective Embedded Platform for Greenhouse Environment Control and Remote Monitoring. *Agronomy* **2020**, *10*, 936. [\[CrossRef\]](#)
11. Wu, Z.; Qiu, K.; Zhang, J. A Smart Microcontroller Architecture for the Internet of Things. *Sensors* **2020**, *20*, 1821. [\[CrossRef\]](#)
12. Morell, J.; Alba, E. Distributed genetic algorithms on portable devices for smart cities. In *International Conference on Smart Cities*; Springer: Berlin, Germany, 2017; pp. 51–62.
13. Medeiros, D.R.d.S.; Torquato, M.F.; Fernandes, M.A.C. Embedded genetic algorithm for low-power, low-cost, and low-size-memory devices. *Eng. Rep.* **2020**, *8*, 1–10. [\[CrossRef\]](#)
14. Zomaya, A.Y.; Lee, Y.C. *Energy-Efficient Distributed Computing Systems*; John Wiley & Sons: Hoboken, NJ, USA, 2012; Volume 88.
15. de Oliveira, A.V.F.M.; Fernandes, M.A.C. Dynamic planning navigation strategy for mobile terrestrial robots. *Robotica* **2016**, *34*, 568–583. [\[CrossRef\]](#)
16. Silva, C.A.; De Oliveira, Á.V.; Fernandes, M.A. Validation of a dynamic planning navigation strategy applied to mobile terrestrial robots. *Sensors* **2018**, *18*, 4322. [\[CrossRef\]](#) [\[PubMed\]](#)
17. Lamini, C.; Benhlila, S.; Elbekri, A. Genetic algorithm based approach for autonomous mobile robot path planning. *Procedia Comput. Sci.* **2018**, *127*, 180–189. [\[CrossRef\]](#)
18. Millo, F.; Arya, P.; Mallamo, F. Optimization of automotive diesel engine calibration using genetic algorithm techniques. *Energy* **2018**, *158*, 807–819. [\[CrossRef\]](#)
19. Kalaivanan, A.P.; Sakthivel, G. *Self Tuning Genetic Algorithm to Achieve Maximum Thermal Efficiency by Monitoring Combustion Characteristics with Vibration/Acoustic Sensors*; Technical Report, SAE Technical Paper; SAE: Warrendale, PA, USA, 2017. [\[CrossRef\]](#)
20. Talbi, E.G. *Metaheuristics: From Design to Implementation*, 1st ed.; John Wiley & Sons: Hoboken, NJ, USA, 2009.
21. Noraini, M.R.; Geraghty, J. Genetic algorithm performance with different selection strategies in solving TSP. In Proceedings of the International Conference of Computational Intelligence and Intelligent Systems (ICCIIS'11), London, UK, 6–8 July 2011.
22. Luke, S. *Essentials of Metaheuristics*, 2nd ed.; Lulu: Morrisville, NC, USA, 2013. Available online: <http://cs.gmu.edu/~sean/book/metaheuristics/> (accessed on 8 November 2020).
23. Gendreau, M.; Potvin, J.Y. *Handbook of Metaheuristics*, 2nd ed.; Springer: Berlin/Heidelberg, Germany, 2010; Volume 2.
24. Gong, Y.J.; Chen, W.N.; Zhan, Z.H.; Zhang, J.; Li, Y.; Zhang, Q.; Li, J.J. Distributed evolutionary algorithms and their models: A survey of the state-of-the-art. *Appl. Soft Comput.* **2015**, *34*, 286–300. [\[CrossRef\]](#)

25. Mikhaylov, K.; Tervonen, J. Evaluation of Power Efficiency for Digital Serial Interfaces of Microcontrollers. In Proceedings of the 5th International Conference on New Technologies, Mobility and Security (NTMS), Istanbul, Turkey, 7–10 May 2012; pp. 1–5.
26. Ibrahim, D. *Advanced PIC Microcontroller Projects in C: From USB to RTOS with the PIC 18F Series*; Newnes: Burlington, MA, USA, 2011.
27. Mishra, S.; Singh, N.K.; Rousseau, V. *System on Chip Interfaces for Low Power Design*; Morgan Kaufmann: Waltham, MA, USA, 2015.
28. Microchip. ATmega328P Datasheet. 2018. Available online: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf (accessed on 11 August 2020).
29. Le Sueur, E.; Heiser, G. Dynamic voltage and frequency scaling: The laws of diminishing returns. In Proceedings of the 2010 International Conference on Power aware Computing and Systems, Vancouver, BC, Canada, 3 October 2010; pp. 1–8.
30. Haririan, P. DVFS and Its Architectural Simulation Models for Improving Energy Efficiency of Complex Embedded Systems in Early Design Phase. *Computers* **2020**, *9*, 2. [[CrossRef](#)]

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).