GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

PyGMA: Pythonic Genetic MPI parallelized Algorithm

Winfried Gero Oed

MatrNr: 21674445

Supervisor: Aasish Kumar Sharma

Georg-August-Universität Göttingen
Institute of Computer Science

September 30, 2023

Abstract

Genetic Algorithms (GA) are able to evolve electrical circuits, an approach known as "evolvable hardware". Evolvable hardware usually utilize re-programmable compute devices like Field Programmable Gate Arrays (FPGA's) to test the evolved, different circuit configurations in silicon. This has the advantage that the GA can exploit physical hardware properties. This work focuses on the usage of a GA to evolve logic circuits. As logic circuits only follow their logical operations, e.g. AND, NAND, OR, it is possible to simulate the circuits in a virtual environment. The advantage of simulation is that it allows to utilize modern High Performance Computing (HPC) centers for massive parallel simulations. In particular a binary N bit multiplier based on logic gates is evolved. To evolve the circuit a parallelized GA framework is developed, named PyGMA. PyGMA uses the Message Passing Interface (MPI) to benefit from the computational power of HPC clusters. Performance analysis shows that PyGMA benefits extremely from MPI support if the simulation of the evolved logic circuits - the evaluation of the fitness function - is compute intensive. Additionally the modular program structure of PyGMA allows for easy adoption of the framework to all sorts of evolutionary tasks, not only logic circuits. Unfortunately the goal of evolving a binary multiplier is not quite reached. The GA is found to be stuck in a local optima and evolution does not continue. This is due to several reasons that are addressed and can be potentially resolved in future research. The implementation is available as open source [Oed23].

Statement on the usage of ChatGPT and similar tools in the context of examinations

In this work I have used ChatGPT or a similar AI-system as follows:

- ☑ Not at all, I used my biological Brain
- ☐ In brainstorming
- ☐ In the creation of the outline
- ☐ To create individual passages, altogether to the extent of 0% of the whole text
- ☐ For proofreading
- ☐ Other, namely: -

I assure that I have stated all uses in full.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables iv

List of Figures iv

List of Listings v

List of Abbreviations vi

1 Introduction 1

1.1 Genetic Algorithms . . . 1

2 PyGMA Framework 3

2.1 Implemented Parallelization Approaches . . . 5

3 Logic Circuit Evolving Experiment 7

3.1 Genetic Circuit Coding . . . 8

3.2 Circuit construction . . . 9

3.3 Logical Multiplier Experiment . . . 9

4 Parallelization Performance Analysis 11

4.1 Strong Scaling . . . 13

4.2 Weak Scaling . . . 13

5 Future Work 14

6 Discussion 15

References 16

A Appendix Figures A1

B Appendix Tables A2

C Code samples A3

List of Tables

1 Speedup times for the Strong scaling test as defined by Amdahl's law. The amount of logical gates and as such the computational complexity of the task is not changed. By adding more processing resources (cores) the evolutionary epochs can be computed faster and as such the global runtime will be reduced. The theoretical speedup column shows the speedup calculated by Amdahl's law. The real world speedup column shows the in reality measured speedup when using the own implemented MPI parallelization approach by comparing it to the single core serial execution time. A2

2 Speedup times for the weak scaling test as proposed by Gustafson. The task difficulty is scaled to the number of cores by adding the respecting amount of logic gates. The theoretical speedup should be linearly because the added difficulty will be compensated by adding more compute resources. Real world data shows that for the hardware evolving experiment the speedup is not linear, for several reasons discussed in section 4.2. . . . . A3

List of Figures

1 **A** showing the general procedure that happens in every Genetic Algorithm (GA). Fitness is evaluated for every individual independently. **B** showing the schematic of a 2 point crossover Genetic Operator (GO). From the selected parents, gene strings defined by 2 points will be copied to generate new offspring's as shown in blue and green. **C** shows three island populations. Each population has its own individuals and evolve on its own. At certain points in the evolutionary process genetic exchange between the islands can happen. This enables all island populations to benefit from the gene diversity of other island populations. 3

2 Illustration of the different modules that together form the Pythonic Genetic MPI parallelized Genetic Algorithm (PyGMA) program. The green objects represent dynamically defined components by the user. All user defined components will be collected inside the configuration file. This file will be read by the main program component and define how the program will behave. Generally the core components will not be touched by the user but will perform all the necessary evolutionary algorithmic steps shown in the blue text box. 4

3 Described is the binary coding of a logic circuit. The simple circuit shown does not make any sense despite being very simple to understand and checked in function. Next to the circuit the coding for the logical gates are shown. A NOT gate is coded via the bits 001. The logical circuit can be represented in binary gene strings which are shown below the circuit. Parts in these strings belonging together are separated from other parts using a comma or slash. The coding is dissected into the inputs, the logic gates and outputs. Next to the binary gene strings their bit meaning is described. Finally in the bottom the final gene string is shown, which is the concatenation of all bits into one string. 10

4 **A** shows the results for the strong scaling test using Amdahl's law. The theoretically calculated speedup by Amdahl's law is shown in orange and measured real world MPI parallelization performance in green. Note that the experiment used for performance testing utilize 282 individuals and as such using more then 282 compute cores will not bring any benefit, rather overhead, which is why the speedup is worse for 300 cores. **B** shows the results for the weak scaling test using Gustafson's law. The theoretically calculated speedup by Gustafson's law is shown in orange and measured real world MPI parallelization performance in green. 11

5 Mean generation/epoch runtime in seconds for the Strong scaling test conducted with either MPI or local processes parallelization. It can be clearly seen that spawning local processes yields to much overhead and will slow down computation time compared to a complete serial computation. However using the own developed MPI parallelization, which involves spawning worker processes in the beginning and then pass work to them yields a great benefit (green). Note that for this test 282 individuals (94 in each of the three populations) where simulated. Using more then 282 processing cores should therefore not reduce computation time but rather slightly increase it due to more overhead. A1

List of Listings

1 This code listing shows the implementation of the Message Passing Interface (MPI) master process and how genes are distributed to the workers and their fitness evaluation result stored for each individual. The MPI tags are Integer Enums that facilitate the maintaining and readability of the code. A4

2 The listing shows the implementation of the MPI worker process. The worker will be waiting in the blocking recv statement in line 13 until he receives the go to evaluate the fitness of another gene. Genetic data is extracted an passed into the Experiment which will return a fitness value of the specific gene, which is then send to the master process along with other needed information's in line 38. . . . . A5

List of Abbreviations

GWDG Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

HPC High-Performance Computing

CPU Central Processing Unit

MPI Message Passing Interface

GA Genetic Algorithm

EA Evolutionary Algorithm

GO Genetic Operator

ANN Artificial Neuronal Network

PyGMA Pythonic Genetic MPI parallelized Genetic Algorithm

GIL Global Interpreter Lock

GUI Graphical User Interface

ASIC Application Specific Integrated Circuit

FPGA Fild Programmable Gate Array

1 Introduction

Bio Inspired Design is the process of engineering nature inspired technology. Biological systems have long been used to gain new engineering ideas. For example massive parallel systems are needed in many applications today. Carver Mead, who invented the field of Neuromorphic Hardware - which is specialized, massively parallel hardware to mimic the behaviour of the brain in a computational manner -, said "I was thinking about how you would make massively parallel systems, and the only examples we had were in the brains of animals," [1]. Deep Learning and AI, especially the Transformer architecture, has gained a lot of maturity lately due to the fact that bigger training data sets can be processed in big High-Performance Computing (HPC) data centers featuring massive compute performance. These systems are as well inspired by biology. The basic idea was to encode information, like seen in biology, in a firing rate. These Artificial Neuronal Network (ANN) are therefore rate coded networks rooted in a biological inspiration. But not only the processes in the brain also social processes have found their way into computer algorithms. Bee Colony Optimization and Ant colony Optimization as well as Particle Swarm Optimization are well working Optimization methods. There are yet another whole category of algorithms called Evolutionary Algorithm (EA). Algorithms in this category have in common that they follow some sort of iterative improving approach. There is an initial starting state - which is often randomly defined - from which in each iteration, called generation in this context, a slightly better solution is derived. The process continues until either a satisfied solution is found or another stop criterion is reached. In this work an EA, specifically a Genetic Algorithm (GA) is implemented.

1.1 Genetic Algorithms

GA's are a form of EA and the following will give a fundamental introduction into the concept. GA's can be used for optimisation problems, where the objective is to find parameters to a function that minimizes or maximizes it. GA's are good suited for the task because they have an innate exploratory behaviour of the parameter space. Thus, in the case of function optimisation the GA can be seen as a search algorithm inside the parameter state space of the function that should be optimized. But GA's can also be used in engeneering tasks like evolving an optimal antenna for a NASA space craft [LHL05]. The working mechanism of a GA can be described by elaborate on the several parts that together form the algorithm.

The fundamental part in the GA are the individuals. Individuals are grouped into a population for easy handling. Usually the population size is a tunable parameter. Each individual inside a population is characterized by its genome, the gene. The gene is distinct to each individual.

Genes represent a solution to the problem that the GA is about to solve. Generally genes code arbitrary structures that are mapped from the genotype - the gene itself - to the phenotype - the structure the gene is encoding -. In a function optimisation problem the gene would be a parameter configuration for the function that should be optimized.

[1] Carver Mead
`https://www.hpcwire.com/2013/11/25/carver-mead-quantum-computing-neuromorphic-design/`

Genes change from individual to individual and as such different solutions are encoded by each individual. Additionally new genes and modified genes are produced via Genetic Operators (GO).

GO's are predefined rules which create new genes - new solutions to a problem -. A very basic and often used GO is the mutation operator. It takes a gene and randomly changes some of its values. By doing so a random new solution to a problem is generated which brings in some diversity into the gene pool of the population. Of course random change might not lead to a very good solution. That is why another very common GO is the crossover operator. In its simplest form the crossover operator takes genes from two different individuals, the parents, and produces two new individuals, the offspring's, by combining the parentental genes. A single point crossover operator will generate an offspring by taking the first half of the genome from the first parent and the second half of the genome from the second parent - and vice versa for the second offspring -. Ideally the ofsprings inherit the best parts of the solution from both parents and thus represent a better solution to the problem. A two point crossover operator is visualized in figure 1 B. If new individuals actually represent a better solution is tested during a fitness evaluation which every individual has to pass through.

Fitness evaluation will evaluate how good the solution, represented by a certain individual, is. As such it will assign a usually positive fitness value to each individual. This value can then be further used to sort out the best performing individuals. Fitness evaluation will involve to map the genotype - the gene - to the phenotype - the solution the gene is coding - and then test the phenotype in a certain environment. For example if the goal is to evolve virtual creatures that can swim, the gene of an individual - the genotype - will be mapped into a 3 dimensional creature - the phenotype - which will be put into water - simulated environment - and its swimming ability's, its swimming speed is measured [Sim94]. Based on how good the creature - phenotype - performs the gene - genotype - will get a fitness value assigned. This fitness value will be used to sort the individuals and make offsprings only from the best performing ones.

All the above parts fit together in the main loop of the GA. The general concept of the algorithm can be seen in figure 1 A. Generally there are three phases. An initialisation phase in which the individuals are created and their genes initialized. Next comes the evolutionary phase in which the fitness of the individuals is evaluated and new offsprings are produced by applying GO to create new offsprings. Once this phase has reached a desirable solution or reached another stop criterion, like a certain amount of evolutionary epochs, the end phase begins. In the last phase the best performing individuals are saved such that the found solution can be used in practice.

Beside the above basic concepts there are more advanced ones. One advancement that helps the populations to develop new genetic diversity is that of island models [Del+19]. In a normal GA there is only one population of individuals. GA's implementing the island model will have more than one population and all populations will evolve independently for a given period of time. After this period they're individuals will be combined. This helps the single island populations to benefit from the genetic diversity of the other ones.

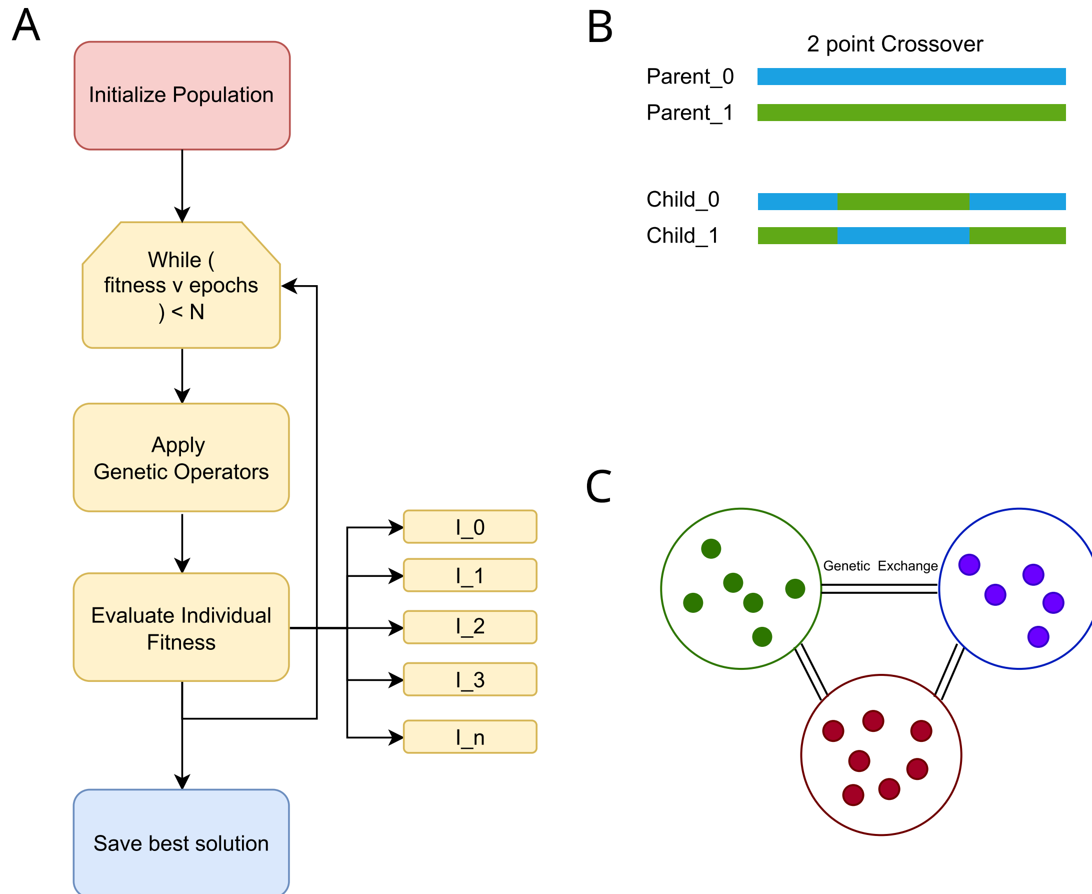

Figure 1: **A** showing the general procedure that happens in every GA. Fitness is evaluated for every individual independently. **B** showing the schematic of a 2 point crossover GO. From the selected parents, gene strings defined by 2 points will be copied to generate new offspring's as shown in blue and green. **C** shows three island populations. Each population has its own individuals and evolve on its own. At certain points in the evolutionary process genetic exchange between the islands can happen. This enables all island populations to benefit from the gene diversity of other island populations.

2 PyGMA Framework

PyGMA is a modular Genetic Algorithm written in Python that utilize the Message Passing Interface (MPI) for parallelization of the fitness evaluation function for individuals in multiple island populations [Oed23]. The modular approach makes it easy to extend PyGMA's core functionality to all kinds of needs. A overview of the core and modular components is given in figure 2. The following text will describe the modular parts first, as they are the most interesting ones for the user. Thereafter the core parts are described.

The modular parts in PyGMA represent the user interface. There are no command line parameters or a Graphical User Interface (GUI). All program configurations are done within the configuration file. This file contains a Python dictionary which will have sections for defining all needed parameters. To help the user it will contain a short description of the parameters as comments. Inside the configuration file the user will tell PyGMA how many populations, how many individuals in each population or if MPI

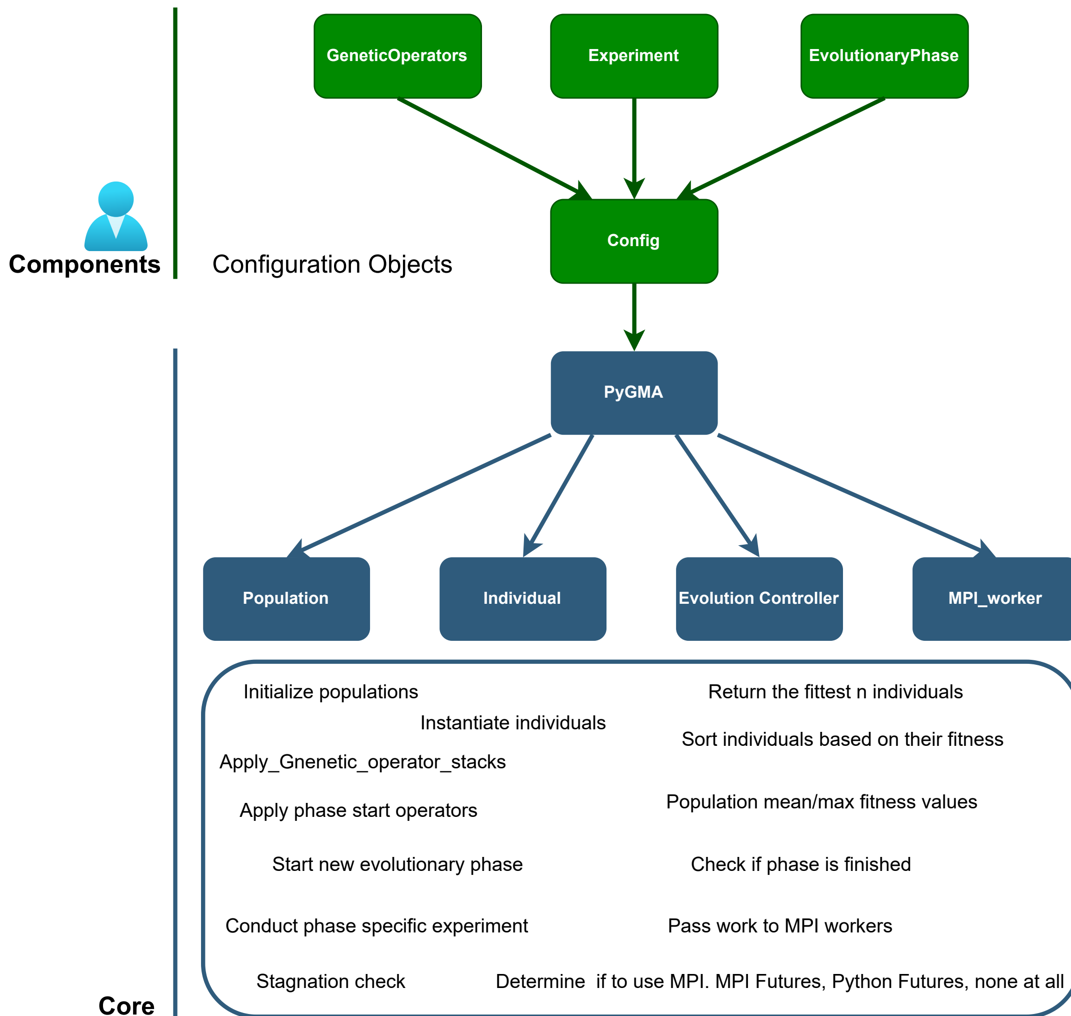

Figure 2: Illustration of the different modules that together form the PyGMA program. The green objects represent dynamically defined components by the user. All user defined components will be collected inside the configuration file. This file will be read by the main program component and define how the program will behave. Generally the core components will not be touched by the user but will perform all the necessary evolutionary algorithmic steps shown in the blue text box.

should be used for parallelization or not. Additionally the user will define the instances of the modular parts used in the evolutionary process. These parts are: the evolutionary phase with its starting operators, the genetic operators that should be applied to the genes in each generation/epoch, the experiment that should be carried out on the individuals of the populations to evaluate their fitness. These three main parts will now be shortly described.

An evolutionary phase can be thought of as a certain time frame in the evolutionary process. It can be used to define intermediate steps in the process. For example it might be to complex to evolve randomly generated genes directly into the desired complex shape. It might be helpful to have intermediate steps. These steps can be defined in different evolutionary phases that will be carried out subsequently. Basically you will first evolve a simple solution and then move on to a more complex one. The simple solution might need another fitness evaluation approach as the complex one, which is why every evolutionary

phase has its own experiment - the individuals are tested in an experiment to evaluate their fitness -. Each evolutionary phase is a Python object which is used to define the stop condition - run for N epochs or run until $fitness > X$ - and used experiment for that phase. It will be user defined, by inherit from the base evolutionary phase class and implement the proper methods that are called by PyGMA.

The experiment describes an algorithm that is used to evaluate the fitness of an individual. It will get a gene G and has to evaluate the fitness of that gene $f(G)$. The fitness evaluation can be very different depending on the problem the user want to solve. It can be a function optimisation task, in which case the gene will be translated into parameters for the function and then these parameters will be put into the function to see the output, but it could be a more complex task. In the presented work a gene will encode a logic circuit that should do some kind of logical operation. Hence the evaluation of the fitness function $f(G)$ will involve to translate the gene G into a logic circuit and then simulate this circuit to determine if its outputs are correct. Depending on a logically correct circuit and if, how many correct outputs are produced by it the gene G can be rated with a fitness value. Since fitness evaluation often involves computational heavy simulations this part of the algorithm is parallelized using local processes or MPI.

Another user definable part are the GO. GO's define rules on how genes are altered in each generation / epoch of the GA. Simple mutation operators will randomly mutate gene parts with a certain probability while other operators like crossover operators produce new offspring's by combining the genes of two parents. The user can define arbitrary GO. These GO can be stacked on top of each other to built a GO stack that will be applied sequentially to the genes. Each population inside PyGMA can have its own, distinct operator stack. The user will define the GO classes and then instantiate the class objects and load them into the stacks inside the configuration file. From here PyGMA will read the stacks and apply them to the genes during the evolutionary process. These conclude the basic description of the user definable components. Next there will be a very brief description of what happens with the core components, which the user does not need to touch. In principle there are several components which in working together built the core functionality of PyGMA. These are the main PyGMA program wrapper, the controller, the populations and individuals of the populations.

The program wrapper is the main entry point in the execution tree of PyGMA. It will be called in serial and parallelized execution and by reading the configuration file determines if it should spawn MPI workers or local processes or none at all. It will initiate the controller object which will handle all other program parts like applying GO's or distributing the fitness evaluation of the single individuals to the worker processes - the worker processes will also be spawned by the main program wrapper -. Populations and individuals will be handled by the controller class and hold the individuals which hold their specific gene and fitness.

2.1 Implemented Parallelization Approaches

In total there are four parallelization approaches implemented in PyGMA. These are none at all, local processes via Pythons inbuilt Processpool executor, dynamic MPI worker processes via MPI4Py futures (MPIPoolExecutor) or static MPI worker processes implemented in an own manner. The last three approaches are now described where the focus

will be on the final approach as this is the one developed in this course.

Python is an interpreter language and features a Global Interpreter Lock (GIL). Only one thread can control the Python interpreter at a time. This makes threaded parallelization a not always successful endeavour. It only makes sense for tasks that are waiting most of their time - for example downloading data - but not for computationally heavy tasks, as they will not benefit at all since only one thread can work at a time. To overcome this many processes can be used where each process will have its own context and with it its own Python interpreter. While overcoming the GIL, processes need to be forked and spawned which, unlike for spawning threads, involves copying the whole program memory to create a new and independent context for the process. This is a slow operation and only brings a computational benefit if the work done in the single processes is very compute intensive, meaning computation time is definitely larger as process spawning time. With the inbuilt Python Procespool executor it is only possible to dynamically spawn local processes. Hence the MPIPoolExecutor provided via the MPI4Py library is used to spawn worker processes dynamically on remote worker nodes. Both of these approaches are implemented in PyGMA. For the speedup testing carried out in this work it was not the case that computation time is definitely larger then process spawning time. As such it can be seen in figure 5 that using more processes actually slows down the overall execution time of the program.

Despite dynamically spawned worker processes MPI makes it possible to spawn ever living worker processes on remote nodes in the beginning of the execution of a program. Every worker has its own program context all the time and will only exchange the definitely needed data fragments with the main process. This makes the approach very fast but also involves more manual handling in the data exchange process between the master and worker processes. The MPI implementation in PyGMA will spawn worker processes in the beginning of the program, then start them and let them evaluate the fitness of each individual in every evolutionary generation/epoch. The main program flow implementation for the master process can be seen in the appendix in code listing 1 and the implementation of the worker process is listed in the appendix in listing 2. Generally the main idea is the following. The master process has a list which contains the MPI ranks of each worker. As long as there are genes which fitness has to be evaluated a loop over all worker ranks is conducted. For each worker it will be checked if the worker is idling, then a gene is passed to this worker, or if there is a result which can be retrieved. If a result is retrieved it is marked that a gene was successfully evaluated, reducing the amount of genes which fitness has to be evaluated by one and as such the most outer loop will end at some point. In principle there can be no deadlock for the workers in this implementation. Each worker will never touch or wait for a resource another worker has to provide and as such workers are only depended on the master process. However the master process can get stuck if a worker is not finish processing or the sending process fails. In this case the master process would wait indefinitely for the result of the worker process and evolution would not continue. To overcome this a check could be implemented which marks the genes that are in processing with a time stamp and if a certain processing time limit is exceeded sends the gene to another worker process. Additionally it should be checked if the worker who failed is still alive and functioning, if not this rank should be removed from the worker list to not distribute data to it anymore.

Code snippets and tests for other parallelization approaches where made as well. Inde-

pendent island evolution's could be used as parallelization approach. In this case the single island populations would be computed by they're own process. This would allow for many island populations and brings the benefit that all GO would be applied in a parallelized environment alongside their populations. Since having problems and genetic evolution strategies - mainly the way the GO's and fitness function are designed - that benefit from this approach are very complex it was not implemented in this project.

Another parallelization approach would be to send a whole pack of genes to the workers at once, instead of always sending only a single gene for fitness evaluation. This would have as drawback that some gene packs might need longer to be evaluated and as such the longest pack would determine the computational time. This is mitigated when using single genes as a worker, in the worst case, gets only one complex gene, not a whole pack.

Another maybe faster approach is to use mpi4py's inbuilt functionality to send numpy arrays. If not explicitly defined mpi4py will send Python objects by pickling them. This is slow compared to directly sending only the raw data. However to send raw data the buffer to receive this data has to be defined. This usually happens in the beginning of the program and the buffer has a fixed size. As such the send data should also not change in size. Since genes can change in their size due to the different evolutionary phases the slower pickle approach is used.

3 Logic Circuit Evolving Experiment

In this project a mechanism that allows a genetic algorithm to synthesize logic circuits is developed. Synthesize logical circuits is in line with synthesize electrical circuits known as "evolvable hardware" [Tho98]. Evolvable hardware brings together evolutionary algorithms and reconfigurable electronic devices. Normal Application Specific Integrated Circuit (ASIC) compute chips can not change their internal wiring's. Reconfigurable compute devices like Fild Programmable Gate Array (FPGA) can change their internal wiring and function, they are re-programmable. This enables evolutionary inspired algorithms to evolve a FPGA configuration that will solve a given problem. It can also be used to evolve fault tolerant systems [Gar05]. Evolution of such circuits can take place in an "intrinsic" or "extrinsic" manner. Intrinsic means that configurations produced by the GA will be downloaded on a physical device for evaluation. Extrinsic means that each configuration will be evaluated using a hardware simulator. Both approaches have their advantages and drawbacks. Simulation can be faster because it can be easily parallelized - in a HPC environment - and it usually comes with a much lower initialisation time. On real physical substrate the evolved configuration has to be loaded onto the FPGA and its logical tiles configured accordingly. This can take up to 9 seconds [HFB22]. If one think of the fact that an GA can take several hundreds of evolutionary generations and in each generation several individual configurations - the genes of the individuals inside the populations - have to be evaluated, reconfiguration time of the physical substrate becomes a crucial factor. However evolving directly on the physical substrate benefits from hardware effects that are and can not be part of the simulation, like material nuances or electromagnetic fields [Tho98].

This work, PyGMA is specialized on extrinsic evolution of logical - not electrical - circuits.

To simulate the logical circuits the Python library *Circuit* [2] [3] is used. Circuit allows for construction of logic circuits and their evaluation on specified inputs. The general idea is to have a binary gene encode the circuit, meaning it will encode how many logic gates, the function of the gates, connection of the gates and inputs and outputs it has. Each of such a logic circuit configuration is simulated to obtain a fitness value for the gene. This involves supplying certain inputs into the circuit and obtain the outputs, then compare them to the desired output. The next subsection will describe how the mapping from a gene to a logic circuit takes place.

3.1 Genetic Circuit Coding

Binary gene strings are used to encode the logic circuit. The description that follows now is based on figure 3 and will describe the coding by walking through the figure as this will give a more intuitive understanding of the coding to the reader. A logical circuit has inputs - in figure 3 named I1, I2, I3, I4 -, logical gates - in figure 3 G4, G5, G6, G7 - and outputs - in figure 3 O8, O9 -. All these parts have to be coded by the gene string.

Inputs are coded by a fixed number of bits. For each input it is coded to which logical gates it is connected. This is done by having two bits for each logical gate, where the first bit codes if the input is connected to input one of this gate and the second if the input is connected to input two. Hence there are $2 \cdot N$ genetic bits for each input, where N is the number of logic gates in the circuit.

Gates are coded by first defining their function. This can be any of the 8 logical functions and as such the first 3 binary bits are devoted for this. Each logic gate has at least one input. This input is the output of one of the components in the logical circuit. As such it is determined by choosing one of the components of the circuit. This is done by coding three variables, the addressing mode, the direction and the length. The variables essentially tell how to choose the input component by walking on the gene string in a certain direction. First the addressing mode bit tells if walking should start from the beginning of the gene string or from the current position of the current gate. The direction bit tells if walking should be done to the left or the right on the gene string. The length bit's defines how many gates should be walked. As example take the coding $0, 1, 0011$ from the input one of gate G4. The first bit, the addressing mode, is 0 meaning we start walking from the current gate position in the gene string. The second bit, the direction bit, is 1 meaning we walk right - we increase the counter -. The next four bits, 0011, tell how long we shall walk, in this case three gates. We will therefore determine gate G7 as the first input to gate G4. For each gate above coding will need $3 + (1 + 1 + bin(N)) \cdot 2$ coding bits, where N is the amount of gates in the circuit, $bin(N)$ the number of bits needed to code N in the base two numerical system.

Outputs have only one input and can only connect to logical gates, not inputs or other outputs. They're first coding bit is a direction bit which will determine the walking direction. Following bits code the walking length, starting from the beginning of the gates. Take as example the output O9 which is coded by $0, 001$. The first direction bit is 0 hence we should walk from beginning by increasing our walking counter. Next we

[2] `https://github.com/reity/circuit`

[3] `https://circuit.readthedocs.io/en/2.0.1/`

should walk 001 = 1 gate. Since gate zero is G4 we will end up with G5.

Above coding schema, as such, allows for arbitrary circuit coding.

3.2 Circuit construction

Having defined a gene coding it is possible to map a binary string into a logical circuit. In practice this leads to some issues that has to be addressed.

The coding allows for arbitrary configurations of the logic gates. However some configurations are not valid. Mapping the output of a logic gate to the exact same gate is inappropriate. This intrinsically includes all circles where an output of gate A is fed back to gate A, potentially by feeding it to B, then C and then from C back to A. Such circles have to be detected.

Constructing the logic gates for simulation in the used Circuit library also puts some constraints on the initialisation process. Each logic gate inside the Circuit library is a Python object. To instantiate the object the input and output gates, which are Python objects as well , have to be provided to the constructor. Unfortunately the binary gene string does not sort the logic gates in any way. Meaning that when starting to process the gene string showed in figure 3 and trying to construct gate G4, first G6 and G7 needs to be constructed. To overcome this problem a gate construction algorithm is developed. This algorithm loops through all gates and try's to construct them. If it finds that it can not construct the gate due to the fact that it's inputs do rely on a gate not constructed already, it will create a stack and puts the gate on this stack. Then continue with the gate that was needed as input. If this gate can not be constructed because it has input from another gate not yet constructed it will be put on the stack as well and processing is continued with the input gate in need. The stack will grow until finally all inputs are constructible, which is at least always possible for input gates as an input does not rely on any other gate. Finally it will be possible to construct all gates on the stack. There are lists and structures involved to remember which gates have already been constructed. The full implementation could be viewed in the repository if necessary but will be not further elaborated here.

3.3 Logical Multiplier Experiment

The goal in this evolutionary experiment is to evolve a binary N bit multiplier. A binary N bit multiplier will get two N bit numbers and multiply them to retrieve a maximal $N + N$ long bit number - e.g. $11 * 11 = 1001$ which is $3 * 3 = 9$ -. To evolve such a circuit the above described gene coding is used since it allows for arbitrary coding of logical circuits.

Fitness evaluation is implemented in the following way. Each circuit - represented as a binary gene string from an individual inside the GA - is constructed, if possible. If it is not possible to construct the circuits, e.g. because it contains loops, the fitness is 0. If it is possible then all possible combinations of two N bit numbers are given into the circuit and its outputs are compared to the correct output defined by multiplication - meaning that for $N = 3$ and the inputs $I_0 = 010$ $I_1 = 100$ the output should be

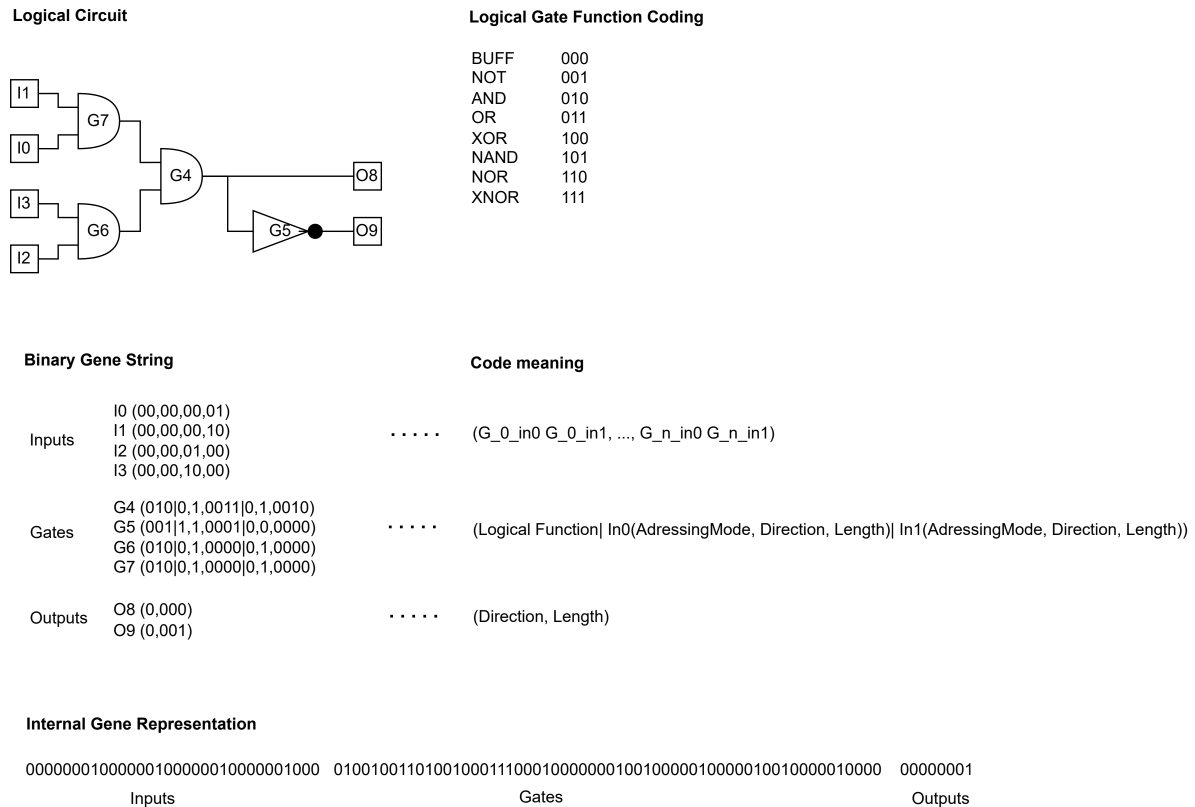

Figure 3: Described is the binary coding of a logic circuit. The simple circuit shown does not make any sense despite being very simple to understand and checked in function. Next to the circuit the coding for the logical gates are shown. A NOT gate is coded via the bits 001. The logical circuit can be represented in binary gene strings which are shown below the circuit. Parts in these strings belonging together are separated from other parts using a comma or slash. The coding is dissected into the inputs, the logic gates and outputs. Next to the binary gene strings their bit meaning is described. Finally in the bottom the final gene string is shown, which is the concatenation of all bits into one string.

$O = 001000$ -. Based on how many correct bits the circuit can produced it is rated by $fitness = 1/(error + 0.00001)$. This ensures that if the error is very small the fitness will be very high.

To increase the fitness two standard GO are applied. First a binary mutation operator which will mutate every bit inside the binary genetic string with a chance of x percent. Secondly a single point crossover operator which generates new genes by taking two parent genes (p_0, p_1) and a random point k, then composing a child by taking the first k bits from p_0 and the remaining bits from p_1.

With the above described setup it is possible to evolve logic circuits that have a fitness $0 < fitness$. However the evolutionary process gets stuck in a local optima and is not able to evolve further. Applying different and more genetic operators as well as having a more appropriate fitness function which guides the evolutionary process is necessary. These issues and solution strategies are addressed in section 5.

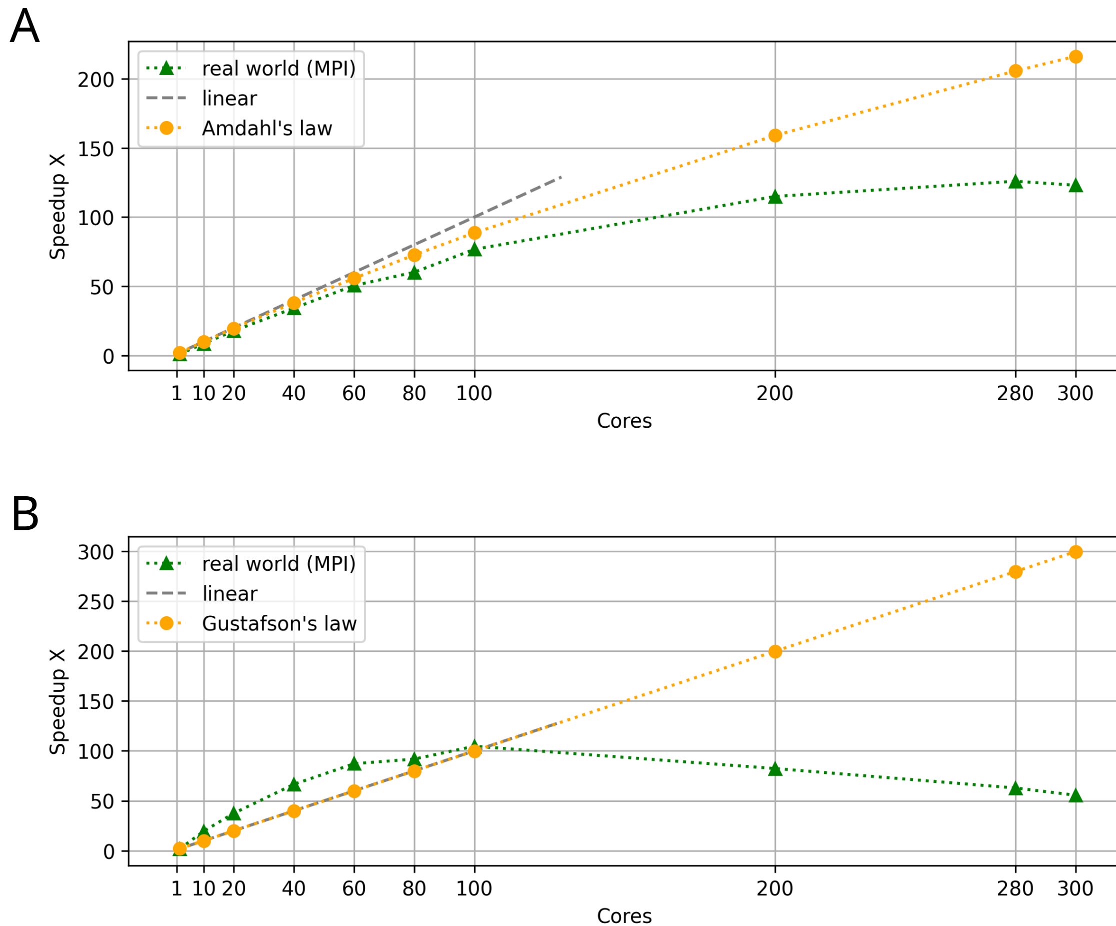

Figure 4: **A** shows the results for the strong scaling test using Amdahl's law. The theoretically calculated speedup by Amdahl's law is shown in orange and measured real world MPI parallelization performance in green. Note that the experiment used for performance testing utilize 282 individuals and as such using more then 282 compute cores will not bring any benefit, rather overhead, which is why the speedup is worse for 300 cores. **B** shows the results for the weak scaling test using Gustafson's law. The theoretically calculated speedup by Gustafson's law is shown in orange and measured real world MPI parallelization performance in green.

4 Parallelization Performance Analysis

Performance analysis is able to tell if an application can benefit from highly parallelized HPC environments. It usually consists of two parts, a theoretical part and practical part. In the theoretical part the speedup gained by parallelization is calculated using Amdahl's and or Gustafson's law. Thereafter the real world speedup is evaluated by measuring the execution time of the application using no, up to strong parallelization enabled.

In PyGMA parallelization is implemented for the fitness evaluation of the single individuals inside the populations. The fitness is evaluated in every evolutionary epoch. Fitness evaluation usually requires a mapping from the genotype to the phenotype followed by a simulation of the phenotype which allows for the fitness measurement. This steps are the most execution heavy task in a GA. Normally fitness evaluation of one individual

does not depend on another. This makes the fitness evaluation a well suited target for parallelization approaches.

However there are serial parts in every evolutionary epoch that, assuming normal conditions, can not be parallelized very well. This includes applying the GO and evolutionary stagnation checks. GO are independently applied to the genes of all individuals and as such could be subject to parallelization. Normally they involve very lightweight operations - e.g. a mutation operator on a binary gene string will flip some bits -. Therefore starting a new process to apply the operator or send the gene to another worker process over the network yields to much overhead. The only reason to apply a parallel approach to GO is when these contain many and heavy operations, which is not the usual case. PyGMA therefore apply's all GO's in every evolutionary epoch in a serial manner.

To measure the speedup gained by parallelization in PyGMA only the time spend for computing the evolutionary epochs is measured. General initialisation time and shutdown time of the worker processes is not considered since they do not contribute reasonable values to the execution time. The initialisation process is parallelized an starting several MPI processes will initiate the same initialisation part on each worker as if PyGMA would run on a single Central Processing Unit (CPU). Shutdown of MPI processes is very fast.

Since the individuals in the populations of a GA are subject to change, the evaluation of the fitness - execution heavy part - can take a different amount of execution time. Therefore the mean execution time for 300 epochs is measured and represented in the practical parts of the test. All tests where carried out on the Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG) compute cluster using the medium partition. This partition features 95 compute nodes each having two Intel(R) Xeon(R) Platinum 9242 CPU's @ 2.30GHz. For benchmarking not more then 20 CPU cores are used on each node. Meaning to allocate 200 compute cores, 10 nodes, each running 20 processes where allocated.

Performance was tested on the logical circuit evolving experiment described in section 3. Parameters for this experiment where chosen in the following way. 3 population's where generated each having 94 individuals. This results in 282 genes which means the fitness evaluation has to be conducted 282 times each epoch. 3 genetic operators where used on each population. The first being the standard removal operator which deletes the 90 worst performing individuals and leaves the fittest 4 - elitism - for further gene generation. Secondly a mutation operator is applied which will generate 45 new individuals by taking a gene from the fittest 4 remaining ones and randomly flips bits with a probability of 5 percent. Third a single point crossover operator is applied which will take two of the 4 fittest genes and combines them into new offspring's until 45 new offspring's are generated. Overall there where 128 logic gates used to find a logical multiplier that can multiply two 3 bit numbers.

4.1 Strong Scaling

In this scaling test it is assumed that certain parts of the program can not be parallelized and have to be carried out in serial operations. As such the ultimate speedup gained by using more processors is limited to at least to the time the program is spending in its serial parts. Strong scaling is defined by Amdahl's law [Amd67]. The law can be formulated as

$$s_{speedup} = T/(s + p/N) \tag{1}$$

where s is the time spend in the serial parts of the program and p the time spend in the parallelized sections, $T = s + p$ the overall execution time on one processor core - execution time without parallelization - and N the amount of processor cores used for parallelization.

In each evolutionary epoch PyGMA's serial sections s are the application of the GO to the individual genes. These step takes a mean execution time of 0.00635 seconds in each epoch for above described circuit evolving experiment. The parallel parts are the evaluation of the fitness for each individual/gene. Evaluating the fitness on one processor core for all 282 genes takes a mean execution time of 4.89924 seconds. The resulting strong scaling speedup can be seen in figure 4 A and in the appendix table 1. For the maximal speedup we can assume that we minimize the parallel execution time with infinite compute resources to zero. We then get $s_{speedup} = T/(s + p/N) \leq T/s = s_{max}$, where $s_{max} = 4.90559/0.00635 = 772.533$. Meaning despite how many parallel compute resource we use the speedup limit is 772.533

4.2 Weak Scaling

Weak scaling, as proposed by John Gustafson, takes into account the fact that parallelization is applied when scaling the problem size - this is in contradiction with Amdahl's law where the problem size is fixed - [Gus88]. This means that theoretically the problem size can grow in size by N but the runtime is kept the same due to adding N more compute cores that handle the parallel parts. Gustavson's law can be formalized by

$$s_{scaledspeedup} = s + p \cdot N \tag{2}$$

where s is the serial part execution fraction, p is the parallel part execution fraction, which means that $1 = s + p$, N is the number of processors used. Since the problem size is scaled, but due to adding more compute units the runtime is kept the same, the speedup evolves linear in theory.

In practice some problems do not follow this theoretical speedup because increasing the problem size will additionally increasing the linear parts and not only the parallel ones. In a GA it is most likely that increasing the problem size will also increase the serial execution time of the algorithm. This is exactly the case for the logic circuit evolving experiment that is used in testing here. To increase the problem complexity there are more logic gates added to the circuit that is evolved. Unfortunately adding logic gates means that they have to be represented inside the genome of the individuals. Therefore adding logic gates leads to longer genes. Longer genes mean more work for the GO and therefore will result in longer serial execution time as the GO are applied in a serial fashion. The

serial part of applying the Genetic Operators (OP) went from 0.00635 to 0.49579 seconds for an experiment with 128 and 19200 logic gates respectively. The parallel problem difficulty - the evaluation of the fitness function - also does not scale linearly by adding more logic gates. For example using double the amount of logic gates does not increase the difficulty of the fitness evaluation by a factor of two. It does add more gates that have to be constructed and theoretically be simulated in the evaluation process but due to the fact that many gates might not be used in the task their simulation is not carried out. Additionally construction of many logic gates has huge overhead in the construction algorithm which need to check if there are connection loops for the logic gates. The used Circuit library for simulating the logic gates might as well follow a non linear execution behaviour if the number of gates increase.

The scaling problems can be seen in the first speedup tests ranging from 10 to 100 used processor cores in figure 4 B. The speedup should be linearly but it is much more, because the problem did not scaled correctly and the task was to easy. Whereas for more then 6400 logic gates, which is the case for more then 100 used processor cores, the problem becomes to complex and generates to much overhead, in other words does not scale very well and as such takes much more execution time. Hence the speedup does not follow Gustavson's law.

5 Future Work

Future work has to be carried out to make logical circuit evolving with PyGMA a success. In the current state the evolutionary process will stuck in a local optima and is not able to evolve further. Due to time limitations there are several unaddressed problems.

The fitness evaluation is not optimized for the search space of the GA. It will measure the fitness only based on the correct input output pairs. These pairs do not give a great guide for the evolutionary search process. It is necessary to test the output but for the circuit design it might be more fruitful if other metrics are applied that really guide the search process in how the circuit should be shaped. For example a metric that takes into account the shape of already existing, human engineered circuits might help. This is based on the idea of nest building using GA in Bonobo et al. [BDT99]. Bonobo et. al. used human rated geometrical metrics to evolve agents that build certain structures. Similar ideas might be used for the circuit design evaluation. It might enable the circuits to evolve in a certain already well established structure in human design approaches and from there to further improvements.

PyGMA also features evolutionary stages that enable the process to come up with a simple solution which can be enhanced further. This stages are currently unused. Using these it might be possible to first evolve genes that represent simple logic gate structures like a 2 bit multiplier and the enhance this multiplier further.

The used GO's are very basic operators and are not tuned for hardware evolving. At least a k point - not only single point - mutation operator should be implemented. Additionally it could be helpful to implement a connection modification operator. This operator would not modify all parts of the gene string but only the ones encoding the connections of the gates. By applying it, not the gate function, only the connections are changed. It was

found that such operators work well for designing fault tolerant electronic circuits [Gar05].

The used coding for the inputs is another problem that needs to be adressed. Currently, for each input its connections to all other gates is directly coded. Hence for 128 logical gates one input needs $128 * 2 = 264$ genetic bits. This makes the input coding part the longest on the whole genome. Additionally it leads to the fact that the inputs will connect to many logic gates, which is usually not the case in a logic circuit.

All the above points should be addressed in future. The modular design of PyGMA provides a solid foundation needed for the improvements.

6 Discussion

The developed GA framework, PyGMA was introduced. It allows for user definable, modular evolutionary experiments. Running the framework in a HPC environment with strong parallelization applied showed reasonable speedup values. As such PyGMA is well suited to run on highly parallel compute environments. Evolution of functional logic circuits however was not possible due to the fact that the algorithm get stuck in a local optima. Possible solutions to this problem like advancing the used fitness function and GO have been pointed out and can be solved in future research.

References

[Amd67] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. URL: https://doi.org/10.1145/1465482.1465560.

[BDT99] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, Oct. 1999. ISBN: 9780195131581. DOI: 10.1093/oso/9780195131581.001.0001. URL: https://doi.org/10.1093/oso/9780195131581.001.0001.

[Del+19] Javier Del Ser et al. "Bio-inspired computation: Where we stand and what's next". In: *Swarm and Evolutionary Computation* 48 (2019), pp. 220–250. ISSN: 2210-6502. DOI: https://doi.org/10.1016/j.swevo.2019.04.008. URL: https://www.sciencedirect.com/science/article/pii/S2210650218310277.

[Gar05] Michael Garvie. "Reliable Electronics through Artificial Evolution". PhD thesis. Jan. 2005.

[Gus88] John L. Gustafson. "Reevaluating Amdahl's Law". In: *Commun. ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415. URL: https://doi.org/10.1145/42411.42415.

[HFB22] Jörn Hoffmann, Clemens Fritzsch, and Martin Bogdan. "CoBEA: framework for evolving hardware by direct manipulation of FPGA bitstreams". In: July 2022, pp. 112–115. DOI: 10.1145/3520304.3528821.

[LHL05] Jason D. Lohn, Gregory S. Hornby, and Derek S. Linden. "An Evolved Antenna for Deployment on Nasa's Space Technology 5 Mission". In: *Genetic Programming Theory and Practice II*. Ed. by Una-May O'Reilly et al. Boston, MA: Springer US, 2005, pp. 301–315. ISBN: 978-0-387-23254-6. DOI: 10.1007/0-387-23254-0_18. URL: https://doi.org/10.1007/0-387-23254-0_18.

[Oed23] Winfried Gero Oed. *Pythonic Genetic MPI parallelized Algorithm*. https://gitlab-ce.gwdg.de/winfired.oed/pygma. 2023.

[Sim94] Karl Sims. "Evolving Virtual Creatures". In: (1994).

[Tho98] Adrian Thompson. "Evolving Electronic Robot Controllers that Exploit Hardware Resources". In: (Dec. 1998).

A Appendix Figures

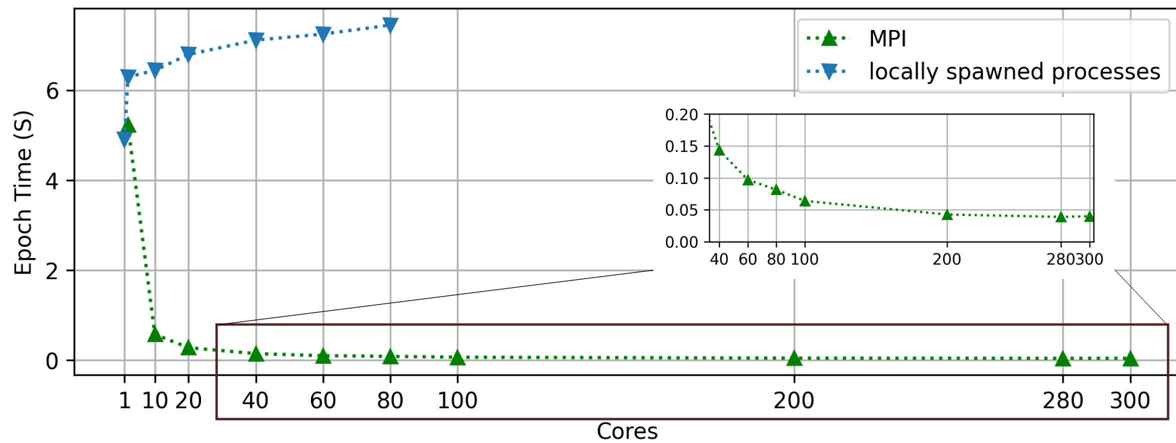

Figure 5: Mean generation/epoch runtime in seconds for the Strong scaling test conducted with either MPI or local processes parallelization. It can be clearly seen that spawning local processes yields to much overhead and will slow down computation time compared to a complete serial computation. However using the own developed MPI parallelization, which involves spawning worker processes in the beginning and then pass work to them yields a great benefit (green). Note that for this test 282 individuals (94 in each of the three populations) where simulated. Using more then 282 processing cores should therefore not reduce computation time but rather slightly increase it due to more overhead.

B Appendix Tables

Logical Gates	Cores	Epoch Time (S)	Theoretical Speedup	Real World Speedup
128	2	5.226	2.00	0.94
128	10	0.570	9.88	8.61
128	20	0.276	19.52	17.76
128	40	0.143	38.08	34.19
128	60	0.097	55.74	50.53
128	80	0.082	72.58	60.13
128	100	0.064	88.64	76.68
128	200	0.043	159.03	114.96
128	280	0.039	205.71	125.95
128	300	0.040	216.29	123.04

Table 1: Speedup times for the Strong scaling test as defined by Amdahl's law. The amount of logical gates and as such the computational complexity of the task is not changed. By adding more processing resources (cores) the evolutionary epochs can be computed faster and as such the global runtime will be reduced. The theoretical speedup column shows the speedup calculated by Amdahl's law. The real world speedup column shows the in reality measured speedup when using the own implemented MPI parallelization approach by comparing it to the single core serial execution time.

Logical Gates	Cores	Epoch Time (S)	Theoretical Speedup	Real World Speedup
128	2	5.226	2.00	0.94
640	10	2.508	9.99	19.56
1280	20	2.615	19.98	37.52
2560	40	2.949	39.95	66.54
3840	60	3.378	59.92	87.13
5120	80	4.276	79.90	91.77
6400	100	4.688	99.87	104.64
12800	200	11.919	199.74	82.32
17920	280	21.868	279.64	62.81
19200	300	26.473	299.61	55.59

Table 2: Speedup times for the weak scaling test as proposed by Gustafson. The task difficulty is scaled to the number of cores by adding the respecting amount of logic gates. The theoretical speedup should be linearly because the added difficulty will be compensated by adding more compute resources. Real world data shows that for the hardware evolving experiment the speedup is not linear, for several reasons discussed in section 4.2.

C Code samples

```python
from mpi4py import MPI
from core.mpi_tags import mpi_tags
comm = MPI.COMM_WORLD
world_size = comm.Get_size()

# while we have data distribute it to free workers
# collect finished data as well
# a gene is computed once the result is written back
genes_to_compute = len(worker_tuples)
while genes_to_compute > 0:
    # check all workers
    for worker_index in range(1, world_size):

        # check if worker can compute new data (Is idle)
        # and we have data to compute
        if (comm.iprobe(source=worker_index, tag=mpi_tags.IDLE) and
            worker_tuples):
            # eat up idle message to clean the pipeline
            comm.recv(source=worker_index, tag=mpi_tags.IDLE)
            # pop the next data to distribute
            data = worker_tuples.pop()
            # signalling the worker that it should continue to work
            comm.send(True, dest=worker_index,
                      tag=mpi_tags.CONTINUE_PROCESSING)
            # send data to worker
            comm.send(data, dest=worker_index,
                      tag=mpi_tags.DATA)

        # check if we can retrieve a result from the worker
        if comm.iprobe(source=worker_index, tag=mpi_tags.DATA_RETURN):
            # retrieve the result
            w_result = comm.recv(
                source=worker_index, tag=mpi_tags.DATA_RETURN)
            # write the fitness result into the appropriate individual
            self.populations[w_result[0]
                ].individuals[w_result[1]].fitness = w_result[2]
            # note that we have computed a gene
            genes_to_compute -= 1

```

Listing 1: This code listing shows the implementation of the MPI master process and how genes are distributed to the workers and their fitness evaluation result stored for each individual. The MPI tags are Integer Enums that facilitate the maintaining and readability of the code.

```python
class MPI_worker():
    def __init__(self, config):
        self.config = config
        self.comm = MPI.COMM_WORLD
        self.rank = self.comm.Get_rank()
    def start(self):
        # work loop until stop is signalled
        while True:
            # signaling that we are waiting for new work
            self.comm.isend(None, dest=0, tag=mpi_tags.IDLE)

            # we receive a bool if we shall to continue to process
            continue_processing = self.comm.recv(
                source=0, tag=mpi_tags.CONTINUE_PROCESSING)

            # if we shall not continue stop the worker
            if not continue_processing:
                break

            # get the working data
            # format will be (pop_index, indiv_index, phase_index, gene)
            data = self.comm.recv(source=0, tag=mpi_tags.DATA)

            # extract data
            pop_index = data[0]
            indiv_index = data[1]
            phase_index = data[2]
            gene = data[3]

            # instantiate the experiment and evaluate the individual
            experiment = self.config['evolutionary_phases'][
                                                    phase_index].experiment

            # conduct the experiment
            fitness = experiment.conduct(gene)

            # return the fitness containing result tuple
            self.comm.send((pop_index, indiv_index, fitness),
                           dest=0, tag=mpi_tags.DATA_RETURN)
```

Listing 2: The listing shows the implementation of the MPI worker process. The worker will be waiting in the blocking recv statement in line 13 until he receives the go to evaluate the fitness of another gene. Genetic data is extracted an passed into the Experiment which will return a fitness value of the specific gene, which is then send to the master process along with other needed information's in line 38.