Parallelization of bio-inspired algorithms in functional environments

Paralelización de algoritmos bioinspirados en entornos funcionales

Bachelor's thesis in Computer Science

Sergio Domínguez Cabrera

Supervised by

Alberto de la Encina

Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

Madrid, 2023

"Parallelization of bio-inspired algorithms in functional environments"
© 2023 Sergio Domínguez Cabrera

This work is licensed under the ***Creative Commons Attribution 4.0 International (CC BY 4.0)***. To view a copy of this license, visit `http://creativecommons.org/licenses/by/4.0/`.

Acknowledgment

I would like to express my sincere gratitude to the following individuals that have played a significant role in the completion of this thesis:

- My supervisor, Alberto de la Encina, for his guidance and support throughout this project. His knowledge and experience have been invaluable to me.
- My family and friends, for always being there for me and cheering me on. Their belief in me and constant encouragement have been a huge driving force in overcoming challenges and achieving my goals. In particular, I would like to extend a special mention to Javier Lobillo, whose sharp eye and assistance helped find a bug in one of the libraries used in this project. His dedication and willingness to help were priceless, and I am truly grateful for his contribution.
- Paula, for her boundless love, unwavering support, and endless inspiration. She has been a constant source of strength and motivation throughout this journey. I want to express my gratitude for her patience and understanding, and for always being there for me. This project would not have been possible without her love and support. Furthermore, it's not just this project that I owe to her; she has been by my side every step of the way during my studies, and without her, I wouldn't be here writing this.

Abstract

This bachelor's thesis investigates the parallelization of bio-inspired algorithms in functional environments, specifically using Haskell and the Accelerate library. All implementations of the investigated algorithms, namely Ant Colony Optimization, Particle Swarm Optimization, and Differential Evolution is collected in a new library called Shizen. The thesis focuses on implementing these bio-inspired algorithms and then using the parallelization capabilities of Haskell and the Accelerate library to enhance their performance. The implementation and parallelization techniques employed in this thesis are analyzed and compared against traditional imperative approaches to parallel computing, as well as sequential functional implementations of the same algorithms. The findings of this study suggest that while functional programming and the Accelerate library offer parallel implementations of algorithms, their performance falls short compared to traditional imperative approaches and sequential functional implementations of the same algorithms.

Keywords: Parallelization, Bio-inspired algorithms, Haskell, Accelerate library

Resumen

Este Trabajo de Fin de Grado (TFG) investiga la paralelización de algoritmos bioinspirados en entornos funcionales, específicamente utilizando Haskell y la biblioteca Accelerate. Todas las implementaciones de los algoritmos investigados, a saber, Optimización de Colonias de Hormigas, Optimización por Enjambre de Partículas y Evolución Diferencial, se recopilan en una nueva biblioteca llamada Shizen. El trabajo se centra en implementar estos algoritmos bioinspirados y luego utilizar las capacidades de paralelización de Haskell y la biblioteca Accelerate para mejorar su rendimiento. Se analizan y comparan las técnicas de implementación y paralelización empleadas en este TFG con enfoques tradicionales imperativos para la computación paralela, así como con implementaciones funcionales secuenciales de los mismos algoritmos. Los resultados de este estudio sugieren que si bien la programación funcional y la biblioteca Accelerate ofrecen implementaciones paralelas de algoritmos, su rendimiento es inferior en comparación con enfoques imperativos tradicionales y con implementaciones funcionales secuenciales de los mismos algoritmos.

Palabras clave: Paralelización, Algoritmos bioinspirados, Haskell, Librería Accelerate.

To my brother Alberto, who bet
that I would not finish this degree.

Contents

Introduction **1**

1 Basics **5**

1.1 Parallel Computing 5

1.2 Functional Programming 10

1.3 The *Accelerate* Array Language 12

2 Bio-inspired Metaheuristics **17**

2.1 Metaheuristics and Optimization Algorithms 17

2.2 Ant Colony Optimization (ACO) 18

2.3 Particle Swarm Optimization (PSO) 22

2.4 Differential Evolution (DE) 25

2.5 Comparison of Metaheuristics 27

3 Shizen **31**

3.1 Architecture and design 31

3.2 Dependencies 33

3.3 Random number generation 33

3.4 Basic types, data structures, and functions 35

3.5 Algorithms 37

4 Results and Analysis **43**

4.1 Encountered problems 43

4.2 Performance Evaluation 47

5 Conclusions and Future Work 61

5.1 Conclusions 61

5.2 Future work 63

Appendices 65

A Avoiding nested data-parallelism 67

A.1 Understanding nested data-parallelism 67

A.2 Avoiding nested data-parallelism in Accelerate 68

A.3 Efficient parallel implementation of the Roulette Wheel Selection . 70

A.4 Efficient parallel implementation of the Gaussian sampling 71

Bibliography 76

List of Figures

1.1 Concurrency vs Parallelism 6
1.2 Data parallelism 7
1.3 CPU vs GPU 9

2.1 Double bridge experiment 19
2.2 Comparison of functions 29

4.1 Unimodal functions 49
4.2 Multimodal function 50

List of Tables

2.1 Pheromone representation 20

2.2 PSO particle 24

4.1 Benchmark functions 48

4.2 PSO Quality of solutions 52

4.3 PSO Execution time 53

4.4 DE Quality of the solutions 55

4.5 DE Execution time 56

4.6 ACO Quality of the solutions 57

4.7 ACO Execution time 58

List of Listings

1.1 Cuda kernel example 10
1.2 HOAS generated by the frontend for the dot product function. . . . 13
2.1 Continuous Ant Colony Optimization. 22
2.2 Particle Swarm Optimization. 25
2.3 Differential Evolution Optimization. 28
3.1 Scanl example 38
4.1 CUDA Exception I 46
4.2 CUDA Exception II 47
A.1 Fold1 example 69

Introduction

Motivation

In a world where data is becoming increasingly abundant, the need for efficient algorithms to process and analyze it is more important than ever. Optimization problems permeate fields as diverse as engineering, finance, logistics, and machine learning, among others. These problems often involve decision-making processes that require finding the best possible solution from a vast search space. The speed at which optimal solutions can be obtained has emerged as a critical factor in achieving success and maintaining a competitive edge in today's fast-paced world.

Traditional approaches to solving optimization problems often rely on mathematical models that are based on assumptions about the problem. However, these assumptions are not always valid in real-world scenarios, which can lead to suboptimal solutions. In this context, metaheuristics have emerged as a promising alternative to traditional approaches, as they do not require any assumptions about the problem.

In this thesis, we focus on bio-inspired algorithms, which are a class of metaheuristics that are modeled on biological systems and processes, such as swarm intelligence, genetic algorithms, and evolutionary computing. These algorithms have gained significant attention in recent years due to their effectiveness in solving complex optimization problems in various fields. However, the efficiency and scalability of these algorithms remain a challenge, particularly when dealing with large datasets and complex optimization problems.

To address this challenge, parallel computing has emerged as a promising solution to improve the performance of bio-inspired algorithms. However, the complexity of parallel programming has traditionally been a barrier to its adoption. The specific knowledge required to implement efficient parallel algorithms has led to the development of high-level approaches to parallel computing.

This is the reason why functional programming has gained growing popularity in parallel computing, thanks to its inherent advantages compared to traditional imperative methods. Functional programming relies on mathematical functions and immutable data structures, thereby eliminating the requirement for a shared

mutable state and facilitating smoother parallelization. Nonetheless, it is important to note that functional programming is not a panacea, as it presents its own set of challenges, including the initial learning curve.

One of the most popular functional programming languages is Haskell, which provides powerful abstractions for parallel computing. Additionally, the Accelerate library provides a high-level approach to GPU programming in Haskell, enabling the efficient parallelization of algorithms.

Objectives

The main goal of this project is to contribute to ongoing research in the efficient parallelization of bio-inspired algorithms while exploring the effectiveness of functional programming and the Accelerate library within this domain. The specific objectives of this project are as follows:

- **O1**: Investigate the parallelization of bio-inspired algorithms within functional environments, with a particular focus on utilizing Haskell and the Accelerate library.
  - **1.1**: Investigate bio-inspired algorithms and their increasing importance in various fields of research.
  - **1.2**: Investigate the advantages of functional programming for parallel computing.
  - **1.3**: Learn advanced concepts of the Haskell programming language.
  - **1.4**: Investigate the Accelerate library and its features for parallel computing.
- **O2**: Implement and parallelize bio-inspired algorithms using Haskell and the Accelerate library.
- **O3**: Analyze and compare the performance of the parallelized algorithms in contrast to traditional imperative approaches to parallel computing.
  - **3.1**: Investigate how to effectively measure the performance of algorithms.
  - **3.2**: Measure the execution time and quality of the results.
  - **3.3**: Analyze the results and compare them with sequential functional implementations.
  - **3.4**: Compare the performance of the parallelized algorithms with traditional imperative approaches to parallel computing.

By accomplishing these objectives, this thesis aims to contribute valuable insights to the field of efficient parallelization of bio-inspired algorithms, particularly in the context of functional programming and the utilization of the Accelerate library.

Work plan

The work plan for this project is divided into three main phases, which are described below:

P1: *Background research.* This phase involves conducting a thorough review of the literature on bio-inspired algorithms, functional programming, and the Accelerate library. This phase will also include familiarizing myself with advanced concepts of the Haskell programming language and the Accelerate library.

P2: *Implementation.* This phase involves implementing the algorithms in Haskell using the Accelerate library. The implementation will be carried out in a modular fashion, with each algorithm being implemented as a separate module. This will enable the comparison of the performance of the algorithms in a controlled manner.

P3: *Analysis and comparison.* This phase involves analyzing and comparing the performance of the parallelized algorithms in contrast to traditional imperative approaches to parallel computing, as well as sequential functional implementations of the algorithms. This will be done by measuring the execution time of the algorithms and comparing the results.

Document structure

This document follows a linear structure, with each chapter building on the previous one. The first two chapters provide the necessary background for the project, while the remaining chapters describe the implementation and analysis of the algorithms. The document is structured as follows:

- *Basics*: This chapter provides an introduction to parallel computing, typical approaches to parallelization, and the advantages of functional programming. We also provide an overview of the Accelerate library and its features for parallel computing.
- *Bio-inspired Metaheuristics*: This chapter provides an overview of bio-inspired algorithms and their applications. We also provide a brief introduction to the algorithms that will be implemented in this project.
- *Shizen*: This chapter discusses the implementation of the algorithms in Haskell and the parallelization using the Accelerate library.
- *Results and Analysis*: This chapter presents the results of the experiments and analyzes the performance of the algorithms in contrast to traditional imperative approaches to parallel computing.
- *Conclusions and Future Work*: This chapter culminates the work with some conclusions and future work.

Associated code to this project

In this work, we present Shizen: a bio-inspired algorithm library for solving optimization problems. Shizen is written in Haskell and uses the Accelerate library to parallelize the computations. The name Shizen comes from the Japanese word for nature, 自然, and is inspired by the natural phenomena that the algorithms in this library are based on. The library is open-source and available on GitHub.

`https://github.com/sergiodguezc/shizen`

Chapter 1

Basics

In this chapter, we will introduce the basic concepts of parallel programming. We will start with an overview of parallel computing, including its types and approaches. Then, we will discuss data parallelism, a popular technique for parallel programming. Later, we will study the benefits of functional programming for parallel programming. Finally, we will explore some specific tools and models for writing parallel programs, including the CUDA programming model and the Accelerate library.

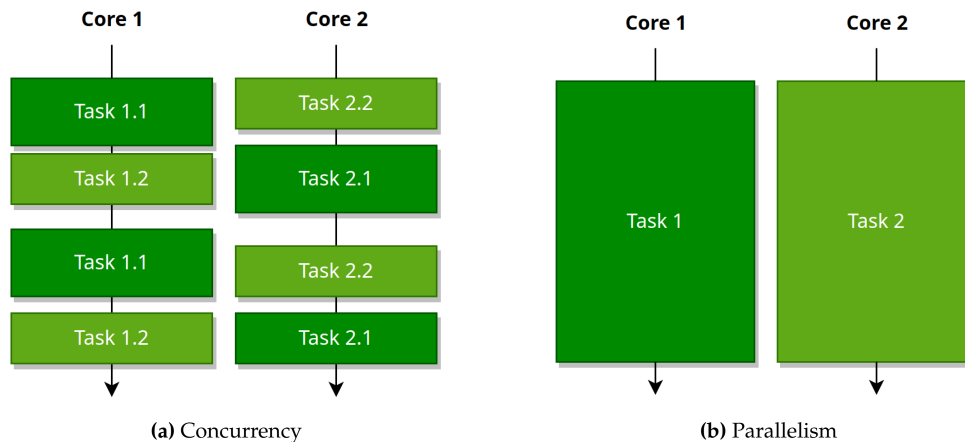
1.1 Parallel Computing

Traditionally, computers have been designed to execute a single instruction at a time. However, as the demand for more computing power has increased, the need for faster computers has also increased.

One way to achieve faster computers is to increase the clock speed of the processor. However, this approach has its limitations. As the clock speed increases, the processor generates more heat, posing a risk of damage. Additionally, the processor has size constraints, and higher clock speeds require smaller processors. This reduction in size limits the processor's capacity for storing data and instructions, ultimately impacting its performance.

Another way to achieve faster computers is to use multiple processors. This is known as *parallel computing*. In this approach, the program is divided into smaller tasks, and each task is executed by a different processor. This concept may seen similar to the concept of concurrency, but there is a difference. Concurrency is the ability of a program to be executed out of order or in partial order. In other words, it is about *dealing* with multiple tasks at the same time. In contrast, in parallel computing we are *doing* multiple tasks at the same time. This is illustrated in Figure 1.1.

It is important to note that parallel computing is not a silver bullet. There exist



(a) Concurrency **(b)** Parallelism

Figure 1.1: Block diagrams of concurrency and parallelism. In concurrency, the tasks are interleaved, while in parallelism, the tasks are executed at the same time.

certain limitations to the speedup that can be achieved by parallelizing a program. This is known as *Amdahl's law* [2] and it states that the speedup of a program is limited by the fraction of the program that cannot be parallelized. Let's consider a program that needs 10 hours to execute using a single thread. If there is a part of the program that cannot be parallelized and it takes 1 hour to execute, then regardless of how many processors we use, the program will take at least 1 hour to execute.

Nowadays, parallelism at multiple levels is the driving force of computer design. Every component of a computer, from the processor to the memory, is designed to implement parallelism. There are two main types of parallelism in applications:

- *Data parallelism* arises because many data items can be operated on independently.
- *Task parallelism* arises because many tasks of work can be executed independently on the same or different data.

In this thesis, we will focus on data parallelism, which is the most appropriate type of parallelism for the applications we are interested in. In the next section, we will discuss it in more detail.

1.1.1 Data Parallelism

The idea of data parallelism can be traced back to the 1960s with the development of the Solomon machine project [33], which aimed to speed up mathematical computations by utilizing multiple math co-processors. These co-processors were capable of performing arithmetic operations on individual data items, all under the control of a single central processing unit (CPU). This concept was further

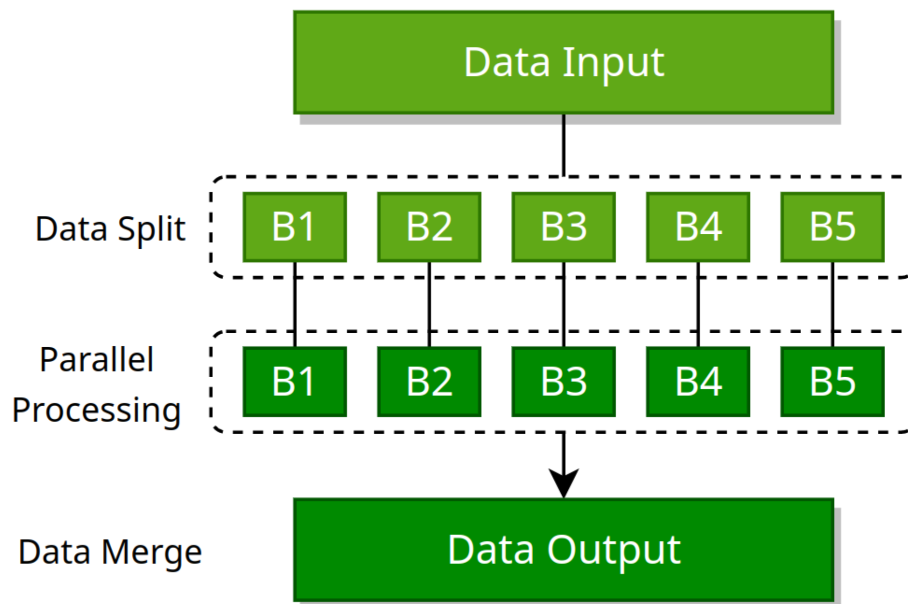


Figure 1.2: Block diagram of how data parallelism works. The data is divided into chunks and each chunk is processed by a different processor. Then the results are combined to produce the final result.

applied in the design of vector processors, which are CPUs that are optimized for efficiently operating on large one-dimensional arrays of data known as vectors.

What are the benefits of data parallelism? From a programmer's perspective, data parallelism presents a single logical thread of execution that is relatively easy to understand and reason about. In addition to its simplicity, there are several advantages to using a data-parallel approach in parallel computing:

- Scalability: The model is independent of the number of processors, allowing it to scale to any number of processors by decomposing data into the appropriate number of chunks.
- Implicit synchronization: All synchronization is handled implicitly, reducing the risk of race conditions and eliminating a major source of errors in parallel programs.
- Memory bandwidth optimization: Improving how data is organized and processed can make it flow faster and enable more tasks to be done at the same time. This can be particularly useful in today's computers, where the amount of data that can be transferred between the memory and the processor at a given time can sometimes be a constraint.

How can we leverage data parallelism in modern computing? There are several ways to leverage data parallelism in modern computing architectures. Some of the most common approaches are:

- Multi-core processors: Modern CPUs come with multiple cores that can execute tasks in parallel.
- Graphics Processing Units (GPUs): GPUs are designed for parallel processing and have thousands of small cores that can perform computations simultaneously.
- Distributed computing: Large datasets can be distributed across multiple machines and processing can be performed on each node in parallel.
- Cloud computing: Cloud platforms like AWS, Azure, and Google Cloud provide scalable computing resources that can be used for data parallelism.

In this thesis, we will focus on leveraging data parallelism through the use of GPUs. However, the work presented in this thesis can be applied to other architectures as we will discuss in Section 1.3.

1.1.2 General Purpose Computing on Graphics Processing Units

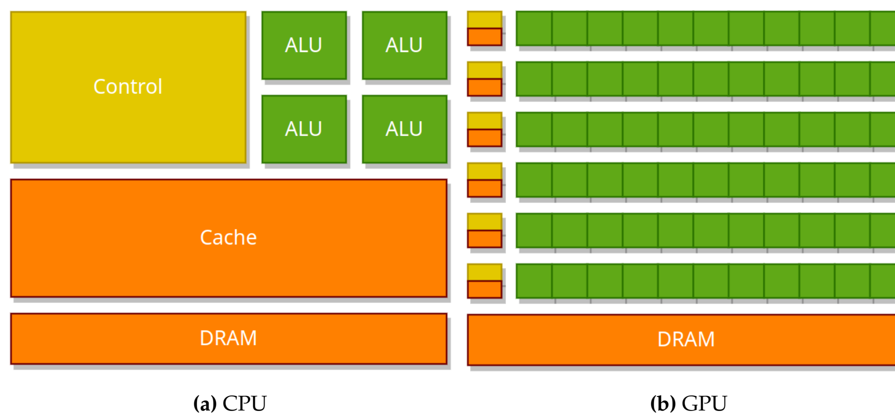
As we discussed in the previous section, data parallelism can be leveraged through the use of a GPU. This is known as *General Purpose Computing on Graphics Processing Units* (GPGPU).

GPUs, originally designed for graphics processing, can perform computations typically handled by CPUs, resulting in improved computational throughput for parallel workloads. In contrast to CPUs in which each core is optimized for single-threaded execution, GPUs are highly parallel and can execute many tasks simultaneously. This is since a significant portion of the CPU's resources, such as die area and power, are utilized for non-computational tasks like caching and branch prediction. However, GPUs can allocate these resources for data-parallel processing, which has made them crucial in accelerating computations for various applications, including machine learning, scientific simulations, and image and video processing. Their capability to handle vast amounts of data and perform calculations on multiple data elements concurrently has made GPUs the preferred choice for data-intensive applications.

The effectiveness of GPUs for data-parallel processing can be observed through the difference in the number of transistors dedicated to data processing on a CPU and a GPU, as illustrated in Figure 1.3.

1.1.3 CUDA

CUDA (or *Compute Unified Device Architecture*) is a parallel computing architecture developed by NVIDIA that enables software developers to access the computing power of NVIDIA GPUs through industry-standard programming languages. It is designed to provide a low learning curve for programmers who



(a) CPU **(b)** GPU

Figure 1.3: Comparison of the number of transistors on a CPU and a GPU. The GPU has more transistors dedicated to data processing than the CPU.

are familiar with standard programming languages such as C.

The CUDA compiler uses programming abstractions to enable parallelism inherent in the CUDA programming model, thus reducing the burden of programming. This programming model is composed of three key language extensions:

- *CUDA blocks*: A group of threads.
- *Shared memory*: Memory shared among all threads within a block.
- *Synchronization barriers*: Enable multiple threads to wait until all threads have reached a particular point of execution before any thread continues.

By using these abstractions, programmers can partition a problem into coarse sub-problems that can be solved independently in parallel, and then into finer pieces that can be solved cooperatively in parallel.

The parallel programming model in CUDA allows threads to cooperate when solving each sub-problem while preserving language expressivity. It also enables transparent scalability, since each sub-problem can be scheduled to be solved on any available processor cores. As a result, a compiled CUDA program can execute on any number of processor cores, and only the runtime system needs to know the physical processor count.

When the Code snippet 1.1 runs on the GPU, it runs in a massively parallel fashion. This is because each element of the vector is executed by a thread in a CUDA block, and all threads run in parallel and independently. This simplifies the parallel programming overhead.

There are three main steps to executing a CUDA program successfully. The first step involves transferring input data from host memory to device memory through host-to-device transfer. Next, the program is loaded onto the GPU, and its execution is performed, with data caching on-chip to improve performance.

```
/** CUDA kernel device code - CUDA Sample Codes Computes the vector
 ↪   addition of A and B
 *  into C. The three vectors have the same number of elements as numElements.
*/
__global__ void vectorAdd( float *A, float *B, float *C, int numElements) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  if (i < numElements) {
    C[i] = A[i] + B[i];
  }
}
```

Listing 1.1: CUDA kernel that performs vector addition of two arrays, A and B, resulting in another array, C. The kernel code is designed to run on the GPU and operates in a scalar fashion, resembling the addition of two individual scalar values, as it adds the two input vectors.

Finally, the results are retrieved by transferring them from device memory to host memory through device-to-host transfer. These steps are necessary to ensure that the program can run correctly on the GPU and that the results can be utilized on the host as required.

What are the main disadvantages of CUDA?

- *Limited portability*: CUDA code is not easily portable to other platforms or architectures, which can be a limitation for projects that need to run on different systems.
- *Steep learning curve*: CUDA requires a significant amount of programming expertise and a deep understanding of GPU architectures to utilize effectively. This can make it challenging for beginners to get started with.
- *Debugging*: Debugging CUDA applications can be challenging and time-consuming due to the complexity of parallel programming and the lack of effective debugging tools.

In this thesis, we will focus on the Accelerate library, which is a high-level approach to parallel programming that addresses the aforementioned limitations of CUDA. This will be discussed in Section 1.3.

1.2 Functional Programming

The programming paradigm of functional programming emphasizes the use of functions to perform computations, and it originated from the lambda calculus, a formal system developed in the 1930s to explore computability solely based on functions.

Programs in functional programming are written declaratively, using expressions or declarations, rather than statements. In contrast to imperative programming, where programs are composed of statements that alter program state, functional programs consist of expressions that do not modify state.

This section will demonstrate that functional programming is a suitable paradigm for parallel programming, as programs that do not modify state are easier to reason about. Haskell is the language of choice for this thesis, and its benefits will be discussed in Section 1.3. Before we discuss how Haskell can be used for parallel programming, let's first discuss some of the key concepts of functional programming and how they can be leveraged to reason about parallel programs.

1.2.1 Why is functional programming well-suited for parallel computing?

As we discussed earlier, programs written in imperative programming involve statements that change the state of the program. In contrast, functional programming uses expressions that don't alter the state. In functional programming, functions are considered *pure* because they don't have any side effects. This makes functional programming ideal for parallel programming because the functions can be executed simultaneously without concerns about race conditions[1] or synchronization problems.

Furthermore, in functional programming, data is typically *immutable*, meaning that once it is created, it cannot be changed. Instead, functions create new data structures based on existing ones, rather than modifying them directly. This also makes it easier to reason about parallelism, since the absence of shared resources makes it impossible for race conditions to occur. Thus, parallel operations can be performed on copies of the data, and the results can be combined later. This can make it easier to write correct parallel code without worrying about low-level synchronization issues.

In functional programming, functions are treated as *first-class citizens*, meaning they can be passed as arguments to other functions, returned as values from functions, and assigned to variables. This can make it easier to create *higher-order functions*[2] that can operate on collections of data in parallel. For example, the map function can be used to apply a function to each element of a list in parallel, which can be more efficient than performing the same operation sequentially.

Recursion is a technique that is heavily used in functional programming, al-

[1]Race conditions refer to the situation where two or more threads or processes read and write a shared resource, and the final result depends on the order in which the threads are scheduled.

[2]Higher-order functions refer to functions that can either receive other functions as arguments or return them as output. In calculus, an example of a higher-order function is the differential operator $\frac{d}{dx}$, which provides the derivative of a function f.

though it is not exclusive to it. This powerful tool allows us to divide a problem into smaller subproblems that can be solved in parallel. For example, a parallel merge sort algorithm can be implemented using recursive calls to sort smaller partitions of the input data in parallel and then merge the results.

Overall, functional programming concepts can simplify the development of correct, efficient, and scalable parallel programs that take full advantage of modern computing architectures. Additionally, programs created with these concepts are often easier to comprehend and reason about than those developed with imperative languages. As a result, functional languages offer a wealth of tools for creating concurrent and parallel programs, and Haskell is one of the most widely used among them. Haskell provides a broad range of tools, including Eden[3], Repa[4], Accelerate, and many others. For our work, we have chosen to use Accelerate, which will be discussed in the following section.

1.3 The *Accelerate* Array Language

As we discussed in Section 1.1.3, CUDA is a low-level approach to parallel programming that requires a significant amount of programming expertise and a deep understanding of GPU architectures to utilize effectively. Instead, we can use a high-level approach to parallel programming that abstracts away the low-level details of parallel programming and provides a simpler interface for writing parallel programs. This is where the Accelerate library comes in.

Accelerate is an embedded domain-specific language (EDSL) for high-performance array computations in Haskell. There are a lot of questions to unpack in this statement, so let's start with the basics. A DSL is a computer language that is designed to solve a specific problem domain. Just as SQL is used for querying databases and HTML is used for creating web pages, Accelerate is used for array computations.

Being embedded in Haskell means that Accelerate is implemented as a Haskell library that provides a range of functions and types for writing array computations. These functions are modeled after Haskell's list library, such as fold, zipWith, and map, but also include array-specific functions such as permute and stencil convolution.

To illustrate how Accelerate works, consider the following Haskell function that performs the dot product of two vectors:

[3] Eden is a language extension for Haskell that simplifies the implementation of parallel algorithms by introducing automatic communication, synchronization, and process handling. For more information, see [20].

[4] Repa is a Haskell library that provides high-performance, shape-polymorphic parallel arrays, automatically parallelized using combinators and stored unboxed. Repa stands for "turnip" in Russian.

```
Fold add (Const 0) (ZipWith mul xs' ys')
  where
    add = Lam x y -> PrimAdd (helided type infoi)
            'PrimApp'
            Tuple (NilTup 'SnocTup' x 'SnocTup' y)
    mul = Lam x y -> PrimMul (helided type infoi)
            'PrimApp'
            Tuple (NilTup 'SnocTup' x 'SnocTup' y)
```

Listing 1.2: HOAS generated by the frontend for the dot product function.

```
dotp :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

The key difference between this function and a regular Haskell function is the use of the Acc type constructor, which represents embedded data-parallel array computations. The Vector a and Scalar a types represent one-dimensional arrays and zero-dimensional arrays, respectively.

1.3.1 Embedding array computations

According to [7], the Accelerate library was designed not only to support NVIDIA GPUs but also to be adaptable to other architectures. Over time, the library has been consistently updated to facilitate the implementation of backends for different hardware, such as the multicore CPU backend, which is now a reliable option. This knowledge is fundamental to understanding the library's design.

The Accelerate library was designed with a modular architecture to support the implementation of new backends. This architecture comprises a single frontend and multiple backends. The embedded array operations do not perform any computation; rather, they create a Higher Order Abstract Syntax (HOAS) using de Bruijn indices. This can be illustrated in Listing 1.2. Additionally, the frontend handles all backend-independent analyses, such as fusion and simplification. Later, the HOAS is then passed on to the backend, which generates the code that executes on the target architecture.

The frontend design of Accelerate enforces the utilization of algorithmic skeletons in the backends [11]. These are higher-order functions that implement parallel computation patterns such as map, fold, and zipWith. This approach differs from that used in Eden, where skeletons were not provided as language constructs. Instead, programmers were required to define their own skeletons on top of the process abstraction.

The Accelerate CUDA backend was implemented by employing the aforementioned approach, where each skeleton was transformed into a CUDA kernel template, which took array types and worker functions, such as the mapped function, as parameters. The implementation of the CUDA backend relied on C++ templates and C preprocessing macros, which made the code difficult to maintain, extend, and scale. Additionally, this approach was incapable of supporting producer-consumer skeleton fusion, which is a crucial optimization, even for relatively simple computations such as the dot product.

To address these issues, the backend was redesigned utilizing Template Haskell, which is a Haskell library that enables generating Haskell code at compile time. This implies that we can write programs that create other programs, a technique referred to as *metaprogramming*. Specifically, the Accelerate team employed the quasiquotes [21] extension of Template Haskell, which permits embedding a language in Haskell.

At the time, there were only two stable backends: CUDA and the interpreter backend, the latter of which served as a reference implementation for the language's semantics. This was due to the difficulty of writing high-performance code for various architectures. To address this limitation, the LLVM compiler infrastructure was employed. LLVM is a set of technologies that can be used to develop a frontend for any programming language and a backend for any instruction set architecture.

LLVM's power stems from its intermediate representation (IR), which is a low-level programming language that serves as a common representation for compilers. The IR is designed to represent programs in a form suitable for analysis and transformation by compiler optimization techniques. Furthermore, the intermediate representation is designed to be independent of any specific programming language, compiler, or machine architecture. This implies that it can be utilized as a portable intermediate representation to implement the frontend of a compiler for any programming language and the backend for any instruction set architecture.

Some of the advantages of utilizing LLVM include:

- *Portability*: LLVM IR is portable across various architectures, including x86, ARM, and PowerPC. Moreover, it has support for high-throughput instruction sets, such as PTX [26].
- *Optimization*: LLVM includes a broad range of compiler optimizations, including those that necessitate machine-specific knowledge.
- *Online compilation*: LLVM can compile code at runtime, allowing for dynamic compilation and optimization.

By utilizing LLVM, the Accelerate library can generate optimized code at runtime for any architecture with an LLVM backend, including multicore CPUs

and GPUs. This was a major improvement over the previous approach, which required implementing a separate backend for each architecture.

To sum up, Accelerate is an embedded domain-specific language (EDSL) for high-performance array computations in Haskell. This means that programs are written using Haskell, but the method of execution is different. Instead of executing the program directly, the fragment of code that use Accelerate works in two phases:

- *Frontend*: The Haskell code generates a data structure that represents the computation.
- *Backend*: The data structure is transformed into a program that can be executed on the target architecture at runtime. Once the program is generated, it is executed on the target architecture.

Chapter 2

Bio-inspired Metaheuristics

This chapter provides an overview of bio-inspired metaheuristics as an efficient problem-solving technique for optimization problems. It highlights the advantages of these methods over traditional optimization algorithms and provides detailed explanations of popular algorithms such as Ant Colony Optimization, Differential Evolution, and Particle Swarm Optimization. By the end of this chapter, readers should have a clear understanding of how bio-inspired metaheuristics work and their potential applications in various fields.

2.1 Metaheuristics and Optimization Algorithms

Optimization problems are ubiquitous in many fields, from engineering and computer science to economics and finance. These problems involve finding the optimal solution to a given problem, subject to certain constraints and objectives. Traditional optimization algorithms, such as linear programming and gradient descent, have been widely used to solve these problems, but they can often become inefficient or even fail to converge to a satisfactory solution.

Metaheuristics offer a powerful alternative to traditional optimization algorithms, by providing a framework for exploring and exploiting the search space of solutions to find near-optimal or satisfactory solutions. Metaheuristics are general-purpose optimization algorithms that can be applied to a wide range of problems and do not rely on the problem structure or specific problem information. Instead, they use iterative search procedures that mimic natural phenomena such as evolution, swarm behavior, and colony behavior, among others. These algorithms are usually classified as bio-inspired metaheuristics. They provide robust, efficient, and scalable solutions to complex optimization problems.

In this thesis, we focus on solving optimization problems using this class of metaheuristics. In particular, we will solve problems that can be expressed as a

function

$$f : S \subset \mathbb{R}^n \to \mathbb{R}, \qquad (x_1, x_2, \ldots, x_n) \mapsto f(x_1, x_2, \ldots, x_n)$$

where S is the search space and f is the objective function. The goal is to find the global minimum of f within S. Not all optimization problems can be expressed in this form, but many can be transformed into this form by introducing additional constraints. Thus, the algorithms presented in this thesis can be applied to a wide range of problems. However, transforming problems is beyond the scope of this thesis, and we will focus on problems that can be expressed in this form.

The algorithms presented in this thesis have different parameters that can be tuned to improve their performance. This process is known as *meta-optimization*, and consists of using an optimization algorithm to find the optimal parameters for another optimization algorithm. For more details on meta-optimization, see [12] [3]. In this thesis, we will point out the parameters that can be tuned for each algorithm, and provide guidelines on how to tune them.

Remark 2.1.1. Note that we will not use the gradient of the objective function f in any of the algorithms presented in this thesis. This is because the gradient is not always available, and it can be computationally expensive to compute. This means that the objective function does not need to be differentiable as is required in classical optimization algorithms such as gradient descent and Newton's method.

2.2 Ant Colony Optimization (ACO)

The first bio-inspired metaheuristic we will discuss is Ant Colony Optimization (ACO). ACO is a population-based metaheuristic that is inspired by the foraging behavior of ants. This metaheuristic was first introduced by Marco Dorigo in 1992, in his Ph.D. thesis [14]. This algorithm was initially developed to solve the traveling salesman problem, which is a well-known NP-hard problem. Since then, it has been applied to a wide range of problems, including those in continuous domains [34] [35]. In this section, we will provide a detailed explanation of how ACO works, and how it can be applied to solve optimization problems.

2.2.1 Description

Ants are social insects that work together in colonies to find food sources, despite having limited cognitive abilities and no centralized control system or leader. Instead, they rely on the use of *pheromones* to communicate with each other and locate food. When an ant discovers a food source, it leaves a pheromone trail on the ground, which other ants can then follow to the food source. As more ants use a particular path, the pheromone trail becomes stronger and more attractive,

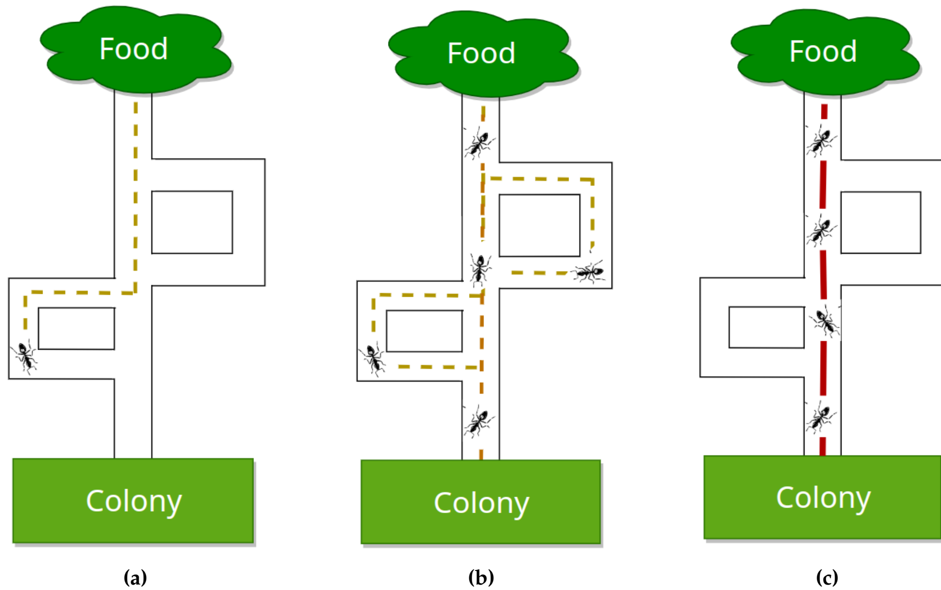


Figure 2.1: Double bridge experiment. **(a)** The first ant to find the food source, and comes back to the nest, leaving a pheromone trail. **(b)** The ants follows one of the 4 possible paths to the food source. **(c)** The shortest path is the one with the most pheromone.

leading to an increased likelihood that other ants will follow the same path. This process is called stigmergy, and it forms the basis of ACO.

In a well-known experiment by Deneubourg *et al.* [13], the researchers demonstrated that ants prefer to use the shortest available path to reach a food source. This can be seen in Figure 2.1, where the ants were presented with two bridges of different lengths to reach the food source. Over time, the ants consistently used the shorter path, as ants that used this path returned to the nest faster and left more pheromones on the ground, which attracted more ants to follow that path. In contrast, ants that used the longer path took longer to return to the nest and left less pheromone, which resulted in fewer ants following that path. Consequently, the shortest path had the strongest pheromone trail, making it the most attractive path for the ants. This is the basic principle behind ACO.

ACO algorithm effectiveness is reliant on the utilization of pheromones to steer the search process, which can be accomplished through a well-structured pheromone trail representation. By [35], we will adopt the solution archive as our pheromone trail representation. The solution archive is a compilation of previously found solutions, updated with every iteration of the algorithm. These solutions are ranked in order of decreasing fitness, with the best solution, so far, occupying the topmost position in the archive. An illustration of this concept can be found in Table 2.1.

The algorithm begins by initializing a population of k artificial ants, each of which represents a potential solution to the problem. Each ant is assigned a

a_1	a_{11}	a_{12}	$\cdots$	a_{1n}		$f(a_1)$
a_2	a_{21}	a_{22}	$\cdots$	a_{2n}		$f(a_2)$
	$\vdots$	$\vdots$	$\ddots$	$\vdots$		$\vdots$
a_j	a_{j1}	a_{j2}	$\cdots$	a_{jn}		$f(a_j)$
	$\vdots$	$\vdots$	$\ddots$	$\vdots$		$\vdots$
a_k	a_{k1}	a_{k2}	$\cdots$	a_{kn}		$f(a_k)$

Table 2.1: A typical solution archive / Pheromone Table. The solutions are organized in descending order of fitness, placing the best solution at the top for the ease of generating PDFs.

random initial position in the search space. Later on, the algorithm proceeds to the main loop, which consists of two phases: the construction phase and the pheromone update phase.

In the construction phase, m ants are created based on the solutions in the archive. The first step is to generate a probability density function (PDF) from the solutions in the archive. Each ant of the archive is assigned a fitness value, which is used to generate the PDF. The fitness value of the i-th ant determined by:

$$\omega_i = \begin{cases} 1 + |f(x_i)| & f(x_i) \leq 0 \\ \frac{1}{1+f(x_i)} & f(x_i) > 0 \end{cases}$$

This function is used to ensure that the fitness values are always positive, and that the best solution has the highest fitness value. The PDF is then generated using the following formula:

$$p_i = \frac{w_i}{\sum_{j=1}^{k} \omega_j}$$

This technique is known as the fitness-proportionate selection, more commonly as roulette wheel selection. Other selection strategies can be used to generate the PDF, such as rank-based selection and tournament selection, but, following the results of [27], the fitness-proportionate selection is the most effective for this algorithm.

The PDF is then generated using these probabilities, and the ants are selected using it. Once the ants have been selected, they are used to generate new solutions by performing a Gaussian sampling around them. The Gaussian density function is defined as follows:

$$\varphi_{\mu,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where μ is the mean and σ is the standard deviation. Let's illustrate how the construction phase works with an example.

Suppose that the selected solution is $a_s = (a_{s1}, a_{s2}, \ldots, a_{sn})$, then the new solution is generated as follows: for each dimension i of the search space, a

random number r is generated from a normal distribution with mean $\mu = a_{si}$ and standard deviation $\sigma_i = \xi * D_i$, where ξ is a parameter of the algorithm called the *evaporation rate*, and D_i is the average distance between the i-th component of the solutions in the archive and the selected solution a_s at the same component. For example, if we use the Manhattan distance,

$$D_i = \frac{1}{k-1} \sum_{j=1}^{k} |a_{ji} - a_{si}|$$

To sum up, the new solution is generated as follows:

$$a = (a_1, \ldots, a_n), \text{ where } a_i \sim N(\mu_i, \sigma_i) \text{ for } i = 1, \ldots, n$$

This process is repeated until m ants have been created. The new solutions are then added to the archive, and the archive is truncated to its original size, keeping only the best solutions. Finally, the pheromone values are updated based on the new solutions. This process is repeated for a fixed number of iterations. The pseudocode for this algorithm can be found in Listing 2.1.

2.2.2 Parameters and Tuning

The ACO algorithm has several parameters that can be tuned to improve its performance. These parameters are as follows:

- The number of ants created per iteration (m). A larger value of m will result in a larger number of solutions being generated, which can lead to a better exploration of the search space. However, this comes at the cost of increased computational time.
- The pheromone evaporation rate (ξ). A larger value of ξ will result in faster evaporation of the pheromone, which can lead to faster convergence to a solution. However, this comes at the cost of a reduced exploration of the search space.
- The number of iterations (N). A larger value of N will result in a longer runtime, but it can improve the quality of the solution.

Furthermore, additional design choices can be implemented to enhance the algorithm's performance. For instance, modifying the selection strategy employed to generate the probability density function (PDF) or alter the distance metric used to calculate the average solution distance can improve the exploration of the search space. Under the analysis conducted by Ojha et al. [27], we will adopt the Roulette Wheel Selection (RWS) as the selection strategy and utilize the Manhattan distance as the distance metric, as they have proven to be highly effective for ACO (as per their experiments).

```
1: function ACO(k, n, b, f, m, ξ, N)    ▷ k: archive size, n: number of
   dimensions, b: boundaries of the search space, f: objective function,
   m: number of ants created per iteration, ξ: pheromone evaporation
   rate, N: number of iterations.
2:     A ← ∅                                        ▷ Initialize archive
3:     for i = 1 to k do
4:         x_i ← random position in b
5:         f_i ← f(x_i)
6:         A ← A ∪ {(x_i, f_i)}
7:     end for
8:     A ← sort(A)                              ▷ Sort archive by fitness
9:     for it = 1 to N do                                   ▷ Main loop
10:        P ← generate PDF using A
11:        S ← ∅                                ▷ Initialize solution set
12:        for j = 1 to m do                       ▷ Construct solutions
13:            (x_s, _) ← select(P, A)
14:            x_j ← N(μ, σ)                 ▷ μ = x_s, σ as explained above
15:            f_j ← f(x_j)
16:            S ← S ∪ {(x_j, f_j)}
17:        end for
18:        A ← A ∪ S                           ▷ Add solutions to archive
19:        A ← sort(A)                          ▷ Sort archive by fitness
20:        A ← truncate(A, k)                         ▷ Truncate archive
21:    end for
        return head(A)                            ▷ Return best solution
22: end function
```

Listing 2.1: Continuous Ant Colony Optimization.

Additionally, their experimental findings demonstrate that the performance of ACO is greatly influenced by the value of the evaporation rate, denoted as ξ. A low ξ value may result in premature convergence, while a high ξ value can lead to slow convergence. Therefore, it is crucial to determine an appropriate ξ value that ensures the algorithm converges to a favorable solution within a reasonable timeframe. Since this parameter's suitability varies depending on the specific problem being addressed, it is essential to fine-tune it for each problem. Nevertheless, Ojha et al. discovered that $\xi = 0.5$ is an effective value for most problems, and thus, we will utilize this value in our tests.

2.3 Particle Swarm Optimization (PSO)

This section provides a detailed explanation of Particle Swarm Optimization (PSO), a population-based metaheuristic that is inspired by the social behavior of bird flocks and fish schools. PSO was first introduced by Kennedy, Eberhart,

and Shi [19] [31] in 1995. This algorithm was initially developed to perform simulations of social behavior, however, it was later found that it could be used to solve optimization problems as well. Since then, it has been applied to a wide range of problems, pointing out the continuous domain as its most common application.

2.3.1 Description

PSO is a population-based metaheuristic that is inspired by the social behavior of bird flocks and fish schools. In these groups, the individuals can find food sources and avoid predators by using simple rules that allow them to coordinate their movements. These rules are as follows:

- Each individual moves in the direction of the best solution it has found so far.
- Each individual moves in the direction of the best solution found by its neighbors.

These rules allow individuals to coordinate their movements and find food sources efficiently. This is the basic principle behind PSO, it combines the erratic movement of individuals with the social behavior of the group to find the optimal solution to a given problem.

The process begins by setting up a population of k particles, where each particle represents a possible solution to the given problem. Each particle is assigned a random position within the search space and a random velocity. The boundaries within which the vector is initialized are determined by the following equation:

$$v_{ij} \sim U(-|b_j^v|, |b_j^v|) \text{ where } b_{vj} = b_j^{max} - b_j^{min}$$

where v_{ij} is the j-th dimension of the i-th particle's velocity, b_j^v is the j-th dimension of the velocity boundaries, and b_j^{max} and b_j^{min} are the upper and lower boundaries of the search space in dimension j, respectively.

Additionally, every particle keeps track of its best solution found so far, known as its *personal best*. In the first iteration, the personal best is set as the particle's initial position. The algorithm also keeps a record of the best solution discovered by the entire population, called the *global best*. Initially, the global best is set to the best solution found among the particles in the initial population.

Later on, the algorithm proceeds to the main loop, which consists of two phases: the movement phase and the update phase. In the movement phase, each particle updates its velocity and position based on the following equation:

$$v_{ij} = \omega v_{ij} + \varphi_p r_1 (p_{ij} - x_{ij}) + \varphi_g r_2 (g_j - x_{ij}) \text{ where } r_1, r_2 \sim U(0, 1),$$

v_{ij} is the j-th dimension of the i-th particle's velocity, x_{ij} is the j-th dimension of the i-th particle's position, p_{ij} is the j-th dimension of the i-th particle's personal best,

x	x_1	x_2	$\cdots$	x_n	$f(x)$
v	v_1	v_2	$\cdots$	v_n	
p	p_1	p_2	$\cdots$	p_n	

Table 2.2: A typical representation of a particle. The vector x represents the particle's position, v represents the particle's velocity, and p represents the particle's personal best.

g_j is the j-th dimension of the global best, and ω, φ_p, and φ_g are parameters of the algorithm called the *inertia weight*, *cognitive weight*, and *social weight*, respectively.

The last step of the movement phase is to update the particle's position using the following simple equation:

$$x_{ij} = x_{ij} + v_{ij}$$

Finally, in the update phase, the particle updates its personal best and the global best. If the particle's new position is better than its personal best, then the personal best is updated to the new position. Similarly, if the particle's new position is better than the global best, then the global best is updated to the new position. This process is repeated for a fixed number of iterations. The pseudocode for this algorithm can be found in Listing 2.2.

2.3.2 Parameters and Tuning

The choices of the parameters for PSO are crucial to the algorithm's effectiveness and have been the subject of much research [40] [32] [15]. The parameters that can be tuned are as follows:

- The inertia weight (ω). A larger value of ω will result in a larger velocity, which can lead to a better exploration of the search space. However, this comes at the cost of the worse convergence to a solution.
- The cognitive weight (φ_p) determines the impact of a particle's personal best on its velocity. A higher value of φ_p increases the influence of the personal best, potentially leading to convergence towards local minima.
- The social weight (φ_g) dictates the effect of the global best on a particle's velocity. A higher value of φ_g amplifies the influence of the global best, potentially accelerating convergence towards local minima. This can be advantageous when seeking a prompt solution.

Now, following the analysis conducted by Pedersen [28], in his Ph.D. thesis, in our experiments we will adopt the following values for the parameters:

$$\omega = -0.1618, \qquad \varphi_p = 1.8903 \qquad \varphi_g = 2.1225$$

Our experiments will incorporate these values as they have demonstrated effectiveness across a broad spectrum of problems.

```
1: function PSO(k, n, b, f, φ_p, φ_g, ω, N)                                   ▷ k:
   archive size, n: number of dimensions, b: boundaries of the search
   space, f: objective function, φ_p: cognitive weight, φ_g: social weight,
   ω: inertia weight, N: number of iterations.
2:     A ← ∅                                                   ▷ Initialize archive
3:     for i = 1 to k do
4:         x_i ← random position in b
5:         v_i ← random velocity in b_v                     ▷ b_v as explained above
6:         f_i ← f(x_i)
7:         p_i ← x_i                                        ▷ Initialize personal best
8:         A ← A ∪ {(x_i, f_i, v_i, p_i)}
9:     end for
10:    g ← best solution in A                               ▷ Initialize global best
11:    for it = 1 to N do                                              ▷ Main loop
12:        for i = 1 to k do                                      ▷ Movement phase
13:            for j = 1 to n do
14:                r_1, r_2 ← random numbers in U(0,1)
15:                v_ij ← ωv_ij + φ_p r_1(p_ij − x_ij) + φ_g r_2(g_j − x_ij)
16:                x_ij ← x_ij + v_ij
17:            end for
18:            f_i ← f(x_i)                                         ▷ Update phase
19:            if f(x_i) < f(p_i) then
20:                p_i ← x_i
21:            end if
22:        end for
23:        g ← best solution in A
24:    end for
        return g                                           ▷ Return best solution
25: end function
```

Listing 2.2: Particle Swarm Optimization.

2.4 Differential Evolution (DE)

The third bio-inspired metaheuristic we will discuss is Differential Evolution (DE), a population-based metaheuristic that was first introduced by Storn and Price in 1997 [38]. This algorithm belongs to the family of evolutionary algorithms, which are inspired by the process of natural selection. In this section, we will provide a detailed explanation of how DE works, and how it can be applied to solve optimization problems.

2.4.1 Description

DE is inspired by the process of natural selection. In nature, individuals with favorable traits are more likely to survive and reproduce, while those with unfavorable traits are less likely to survive and reproduce. This process is known as *survival of the fittest*[1], and it is the basis of DE.

As with other evolutionary algorithms, the process begins by setting up a population of k individuals, where each individual represents a possible solution to the given problem. Each individual is assigned a random position within the search space. The solution archive is identical to the one used in ACO (Table 2.1).

Afterward, the algorithm proceeds to the main loop, which consists of three phases: the mutation phase, the crossover phase, and the selection phase. In the mutation phase, new individuals are generated by mutating the current ones. This process is carried out as follows:

$$m_i = x_{r_1} + F(x_{r_2} - x_{r_3}), \qquad i = 1, \ldots, k$$

where m_i is the mutated solution, x_{r_1}, x_{r_2}, and x_{r_3} are three randomly selected individuals and F is a parameter of the algorithm called the *mutation factor*. The individuals are selected from the archive and initially were intended to be different from each other. However, it was later found that the algorithm performs better when the individuals are selected randomly, without any restrictions, provided that the population size is large enough to ensure diversity. Also, note that in the above equation, the operations are performed over vectors, not scalars. This means that the mutation is performed in each dimension of the search space.

In the crossover phase, the new mutated solution is combined with the current solution to generate a new solution. The new solution is generated by the following equation:

$$x_i = \begin{cases} m_i & \text{if } r < CR \\ x_i & \text{otherwise} \end{cases}, \qquad r \sim U(0,1)$$

where x_i is the current solution, m_i is the mutated solution, and CR is a parameter of the algorithm called the *crossover rate*.

Finally, in the selection phase, the new solution is compared with the current solution. If the new solution is better than the current solution, then the new solution substitutes the current solution. This process is repeated for a fixed number of iterations. The pseudocode for this algorithm can be found in Listing 2.3.

[1] The phrase "survival of the fittest" was first used by Herbert Spencer in his 1864 book *Principles of Biology* [36]. He attempted to explain the Darwinian principle of natural selection.

2.4.2 Parameters and Tuning

The DE algorithm has three parameters that can be tuned to improve its performance. These parameters are as follows:

- Number of individuals in the population (k): This parameter determines the number of solutions generated in each iteration.
- Mutation factor (F): The magnitude of mutation is determined by this parameter. A larger value of F results in a larger mutation, allowing for better exploration of the search space. However, it may lead to worse convergence to a solution.
- Crossover rate (CR): This parameter determines the probability of performing the crossover operation. A higher value of CR leads to a greater number of crossover operations, enabling better exploration of the search space. However, it may also result in worse convergence to a solution.

The number of individuals must be at least 4 to ensure that the algorithm has enough individuals to perform the mutation. The mutation factor determines the magnitude of the mutation, and it is usually set to a value between 0 and 2, and the crossover rate is a value between 0 and 1 (it is a probability). The number of individuals in the population is usually set to a value between 5 and 10 times the number of dimensions of the search space. The values of these parameters can be tuned to improve the performance of the algorithm. However, it is important to note that the suitability of these parameters varies depending on the specific problem being addressed. Therefore, it is essential to fine-tune them for each problem.

In their paper, Storn and Price [38] recommend the following values for a starting point:

$$k = 10n, \qquad F = 0.5, \qquad CR = 0.9$$

and so we will adopt these values in our experiments.

2.5 Comparison of Metaheuristics

In this section, we will compare the algorithms described above based on their approach to the problem: exploration vs. exploitation. Exploration is the process of searching the search space for new solutions, while exploitation is the process of refining the current solution.

All of the algorithms described above mix exploration and exploitation to different degrees, however, it is essential to understand what their main approach is. This is especially useful when deciding which algorithm to use (or how to fine-tune it) for a specific problem. For instance, when a problem is recognized to possess numerous local minima, as depicted in Figure 2.2b, it is preferable

```
1: function DE(k, n, b, f, F, CR, N)    ▷ k: archive size, n: number of
   dimensions, b: boundaries of the search space, f: objective function,
   F: mutation factor, CR: crossover rate, N: number of iterations.
2:    A ← ∅                                         ▷ Initialize archive
3:    for i = 1 to k do
4:       x_i ← random position in b
5:       f_i ← f(x_i)
6:       A ← A ∪ {(x_i, f_i)}
7:    end for
8:    for it = 1 to N do                            ▷ Main loop
9:       for i = 1 to k do
10:         r_1, r_2, r_3 ← random integers in U(1, k)
11:         for j = 1 to n do                       ▷ Mutation phase
12:            m_ij ← x_{r_1 j} + F(x_{r_2 j} − x_{r_3 j})
13:         end for
14:         for j = 1 to n do                       ▷ Crossover phase
15:            r ← random number in U(0, 1)
16:            if r < CR then
17:               y_ij ← m_ij
18:            else
19:               y_ij ← x_ij
20:            end if
21:         end for
22:         if f(y_i) < f(x_i) then                 ▷ Selection phase
23:            x_i ← y_i
24:            f_i ← f(y_i)
25:         end if
26:      end for
27:   end for
      return g                                      ▷ Return best solution
28: end function
```

Listing 2.3: Differential Evolution Optimization.

to employ an exploration-focused algorithm. Such an algorithm will expedite the discovery of the global minimum. Conversely, if a problem is known to have a solitary global minimum, as illustrated in Figure 2.2a, it is more advantageous to utilize an exploitation-focused algorithm. This choice will facilitate the identification of the global minimum more efficiently.

The balance between exploration and exploitation in Particle Swarm Optimization (PSO) is determined by the inertia weight (ω). When ω is set to a higher value, the particles' velocities increase, enabling a more extensive exploration of the search space. However, it is generally acknowledged that PSO primarily focuses on exploitation. This means that the algorithm tends to move all particles toward the best solution discovered thus far, emphasizing exploitation rather than

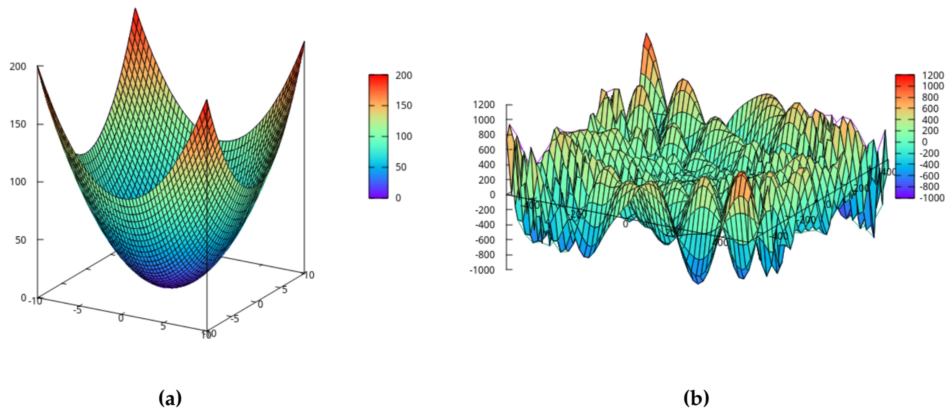


Figure 2.2: (a) The function has a single minimum, so an exploitation-focused algorithm is preferable. **(b)** The function has several local minima, so an exploration-focused algorithm is preferable.

exploration.

In Ant Colony Optimization (ACO), the balance between exploration and exploitation is governed by the evaporation rate (ξ). When ξ is set to a higher value (near 1), the pheromone 'evaporates' at a faster rate, allowing the ants to explore the search space more thoroughly. Conversely, when ξ is set to a lower value (near 0), the pheromone persists, leading to quicker convergence toward a solution. Nonetheless, ACO is generally regarded as an exploration-focused algorithm. It generates new solutions around the best solutions found so far, emphasizing exploration rather than exploitation.

Differential Evolution (DE) in its canonical form is an exploitation-focused algorithm. Even though the mutation factor (F) and the crossover rate (CR) can facilitate the exploration of the search space, the algorithm is mainly biased toward exploitation. This is because DE generates new solutions by combining current solutions and retaining the best ones.

Chapter 3

Shizen

This chapter unveils Shizen, an algorithm library designed to tackle optimization problems, drawing inspiration from the intricate workings of nature. Implemented in Haskell, Shizen leverages the Accelerate library to parallelize computations, intending to improve performance. The name originates from the Japanese word for nature, 自然 (pronounced *shizen*), reflecting the profound influence of natural phenomena on the algorithms encapsulated within this library. Shizen is an open-source project, and its codebase is freely accessible on GitHub.

3.1 Architecture and design

When designing Shizen, the main goal was to create a library that was easy to use and extend. To achieve this, we decided to use a modular design. Modularity serves as a core principle in software engineering, enabling the division of intricate systems into smaller, more manageable components. This division greatly enhances comprehension and maintenance efforts. In the context of Shizen, we employ modules to break down the library into smaller segments, facilitating comprehension and extension. By adopting a modular design, we can also reuse code across various sections of the library. This approach streamlines maintenance tasks, as modifications need only be made in one location. Additionally, it promotes greater confidence in code correctness, as testing can be focused on smaller, isolated parts.

The library is divided into modules under the Shizen namespace. The main modules are:

- Shizen.Types: Contains the basic types and type classes [1] used in the library.

[1] Type classes are a powerful feature of the Haskell programming language that allows for ad hoc polymorphism, which means that different functions can be defined for different types, even if those types are unrelated or defined independently. Type classes provide a way to define generic interfaces that can be implemented by multiple types.

This will be discussed in more detail in Section 3.4.1.

- Shizen.Utils: Contains the implementations of some utility functions used in the library. This will be discussed in more detail in Section 3.4.2.
- There is a module for each algorithm implemented in the library. These modules are named after the algorithm they implement. For example, the module that implements the Ant Colony Optimization algorithm is called Shizen.AntColony. These modules contain the implementation of the algorithm as well as the types and functions needed to use it. These modules will be discussed in more detail in Section 3.5.

The modular design wasn't only used to make the library easy to extend, but also to make it as general as possible. To achieve this, we decided to use type classes to abstract even the most basic types. This allows us to use the same code for different types, as long as they implement the required type classes. For example, we can use the same code to compute the fitness of a solution for a n-dimensional problem, regardless of which n we are using.

Modularity and generality are important, but they are not the only things that make a library easy to use. We also need to make sure that the library is easy to integrate with other code. To achieve this, we decided to use the Accelerate library to parallelize the computations. This allows us to use the same code for both sequential and parallel computations. The only thing we need to do is to change the Accelerate backend we want to use.

As the backend we are targeting is GPU, it is important to minimize unnecessary data transfers between the CPU and GPU. To accomplish this, we adopt a strategy of storing all intermediate computations within the GPU itself, specifically within the Acc type constructor. This design decision plays a crucial role in potentially enhancing the library's performance. Moreover, as not all computing devices support double-precision floating-point arithmetic, we opt for an easy-changing precision strategy. This involves using type synonyms to represent the floating-point type used in the library.

In certain scenarios, it is often advantageous to utilize single-precision floating-point arithmetic due to specific requirements or considerations. For instance, if the level of precision beyond a certain number of decimal places is not crucial to the application or if excessive decimal accuracy is unnecessary, opting for single-precision floating-point computations can be preferable. By focusing on single-floating computations, we can effectively strike a balance between computational efficiency and the level of precision required for the task at hand.

To sum up, the main design choices of the library are *modularity* and keeping all the computations inside the Acc type constructor. These design choices allow us to create a library that is easy to use and extend, the computations are as fast as possible, and it is easy to integrate with other code.

3.2 Dependencies

In this section, we will discuss the dependencies of the library. It is important to note that Accelerate implements a modular design. Consequently, each backend is implemented as a separate package, and several packages offer supplementary functionality.

The principal packages required by our library are:

- accelerate: Contains the core functionality of the library. It defines the types constructors, synonyms, and functions used to represent the HOAS (Higher-Order Abstract Syntax) representation of the computations.
- mwc-random-accelerate: Provides a high-quality random number generator that can be employed within the Accelerate context.
- sfc-random-accelerate: Provides a random number generator suitable for use within the Accelerate context.
- containers-accelerate: Offers several data structures, such as maps and sets, which can be utilized within the Accelerate context. Furthermore, it provides a few sorting algorithms which can be employed within the Accelerate context.

We will discuss later on how we extended some of these packages to provide additional functionality or to fix some issues that were present in the original packages.

Moreover, to test the library, we need to include some backends. The backends used for testing are:

- accelerate-llvm-native: This backend compiles the computations to LLVM and then to native code.
- accelerate-llvm-ptx: This backend compiles the computations to LLVM and then to CUDA code.

3.3 Random number generation

Random number generation is a crucial part of the library, as it is used in all algorithms. As such, it is important to have a good random number generator. This is why it is meritorious enough to have its own section. In addition, it is worth discussing random number generation in Haskell in general, as some issues need to be addressed.

3.3.1 Random number generation in Haskell

As mentioned in Section 1.2, Haskell is a pure functional programming language. This means that functions are pure, i.e. they do not have side effects. This is a good thing, as it makes the code easier to reason about. However, it also means that we cannot use the same approach to random number generation as we would in an imperative language.

In functional languages, side effects are usually handled by using monads. A monad is a type that represents a computation. It is a way of sequencing computations. The main difficulty of working with monads is that once we are inside a monad, we cannot get out of it easily. As random number generation in computers is usually done by reading from a file that contains a source of entropy (e.g. /dev/random on Linux), we need to be able to get the random numbers out of the IO monad (IO stands for Input/Output) to use them in our computations.

In the following subsections, the approach to random number generation within the Accelerate library will be discussed. Then, we will discuss the approach used in the library.

3.3.2 Random number generation in Shizen

The Accelerate library has various modules that provide random number generation. The most important ones are mwc-random-accelerate and sfc-random-accelerate. Both generate Accelerate arrays filled with high-quality random numbers, but they use different algorithms to do so and have different strengths and weaknesses.

The primary advantage of mwc-random-accelerate lies in its ability to generate high-quality random numbers without the need to handle seed generation intricacies. However, it has two significant drawbacks. Firstly, it generates random numbers on the CPU and then transfers them to the GPU, resulting in reduced efficiency due to data transfers between the two. Secondly, it returns the Array of random numbers within the IO monad, making it inconvenient for direct use in computations since extracting the random numbers from the IO monad would involve unsafe operations that may lead to undefined behavior.

In contrast, the key strength of sfc-random-accelerate is its capability to generate random numbers directly on the GPU, making it more efficient than mwc-random-accelerate. However, it has a major drawback: it lacks support for generating a suitable seed for the random number generator. Another issue is that it does not support generating random numbers within the Exp context. Fortunately, there is a fork of the library available (found here) which does support generating random numbers within the Exp context, despite the absence of such support in the official version of the library. The change is relatively simple, as it only

involves changing the type of the generator, from a data type to a type synonym.

At this juncture, it is worth noting that the two libraries are not mutually exclusive. In fact, they complement each other. The mwc-random-accelerate library is better at generating the seed for the random number generator, while the sfc-random-accelerate library is better at generating the random numbers themselves. This is why we decided to use both in our library.

There is one more issue that needs to be addressed. The sfc-random-accelerate library does not support generating normal random numbers, nor does generate uniform random numbers within some range. This is why we extended the library to support generating normal random numbers using the Box-Muller transform, as it is the most efficient way of generating normal random numbers [5]. The extension is available in the following repository.

3.4 Basic types, data structures, and functions

In this section, we will discuss the basic types, functions, and data structures used in the library. These are defined in the Shizen.Types and Shizen.Utils modules. We will start by discussing the basic types and type classes, and then we will discuss the functions and data structures.

3.4.1 Basic types and data structures

The library extensively utilizes modularity, employing type classes to abstract even the most fundamental types. Among these types, the basic ones represent positions in the search space and the boundaries of an n-dimensional problem. This gives us two main type classes:

- Boundaries a: Represents the boundaries of an n-dimensional problem. This type class contains a function that constructs the boundaries of a problem given a bound in one dimension, replicated n times. This function is used to create the boundaries of a problem in a polymorphic way. The only requirement is that the type a implements the Elt type class, which is used internally by the Accelerate library to represent types that can be used as array elements.
- Position p b: This type class contains all the basic functions needed to work with positions in the search space. These functions include computing the distance between two positions, mapping a function over a position, projections, etc. The type p represents the type of the position, while the type b represents the type of the boundaries of the problem. The only requirement for the type p is that it implements the Elt type class, while the type b must implement the Boundaries type class.

The signature of the Position type class initially poses a significant challenge. In Haskell, inferring the type of the boundaries from the type of the position in a polymorphic manner is not straightforward. However, in our specific scenario, the type of the boundaries is closely tied to the type of the position. So, how can we overcome this problem? The solution lies in utilizing type classes with functional dependencies, as outlined by Jones [17]. By employing this language extension, the compiler becomes capable of deducing the type of the boundaries based on the type of the position, and vice versa.

We establish a relationship between the type of the position and the type of the boundaries by employing functional dependency notation, denoted as p → b. This signifies that for each position type p, there exists only one corresponding boundary type b that can be utilized for defining the problem's boundaries. This approach enables us to achieve polymorphism in this particular context.

Implementation details of the instances

Let's deep dive into the implementation of the specific instances of these type classes. For simplicity, at first, we defined a type synonym for the bound of a single dimension, called Bound. This type synonym is defined as a tuple of two elements, representing the lower and upper bounds of the dimension. Thus, the type of the boundaries of an n-dimensional problem is defined as a tuple of n Bounds.

In practice, we don't declare this as vanilla Haskell tuples. Instead, we utilize a more versatile form of tuples known as GADTs (Generalized Algebraic Data Types). This allows us to define functions with more advanced type behavior [30]. Moreover, in the implementations of these types, we leverage the power of modularity by exploiting the trivial isomorphism between tuples and nested tuples:

$$x_1 \times (x_2 \times x_3) \cong (x_1 \times x_2) \times x_3 \cong x_1 \times x_2 \times x_3$$

which allows us to define the positions and boundaries of higher-dimensional problems in terms of lower-dimensional ones. For instance, the boundaries of a 9-dimensional problem are defined as a triple of 3-dimensional boundaries, which are defined as regular triples of Bounds. Symmetrically, the instances of the Position type class are defined in terms of lower-dimensional positions.

The recursive nature of this approach is immensely beneficial when incorporating the operations of the type class. It allows us to repeatedly invoke the same function for every dimension involved. Returning to the previous example, when implementing the fromBound function for a 9-dimensional problem, we can invoke the same function for a 3-dimensional problem three times in a recursive manner, and then combine the results into a triple.

This approach enables us to handle various dimensions without writing redun-

dant and error-prone code, and also reduces the likelihood of mistakes, as we rely on a smaller number of well-tested instances. It is also worth noting that this technique simplifies the implementation of the library and allows users to extend it with their own instances easily.

At this juncture, a few inquiries may arise: why not utilize a vector or a list instead of a tuple? The explanations for these queries are straightforward. Firstly, the Accelerate library does not support nested vectors, as it aims to avoid nested parallelism. Moreover, lists are not instances of the Elt type class (representing types of array elements), given that they lack a fixed size. Consequently, tuples serve as a viable alternative to vectors within the confines of the Accelerate library

Finally, we define a type synonym for a point $p \in \mathbb{R}^{n+1}$, called Point p, which is defined as a tuple of n real numbers and a fitness value. This type synonym is quite useful, as in many cases we need to store the fitness value along with the position.

3.4.2 Functions

The library also contains a few utility functions that are used throughout the library. These functions are defined in the Shizen.Utils module. The most important of these functions is createGenerator, which provides the functionality discussed in Section 3.3. It generates a vector full of entropy and the only argument of this function is the size of it. The rest of the functions are used for abstracting some common operations, like calculating the minimum of a vector.

3.5 Algorithms

In this section, we will discuss the algorithms implemented in the library. All of the algorithms follow the same structure. They are implemented as a function that takes the problem to solve, the number of iterations, and the parameters of the algorithm, and returns the best solution found.

Every algorithm stores something similar to a solution archive, with some differences, i.e. the pheromone table in ACO, the swarm in PSO, etc. This archive is implemented as an Acc (Vector a), where a is the type of the values stored. This is where the magic of the Accelerate library happens. Every time we operate on the archive, we are executing in parallel the same operation on every element of the archive.

```
ghci> run $ scanl1 (+) (fromList (Z:.3) [1,2,3])
Vector (Z:.3) [1,3,6]
```

Listing 3.1: Example of the scanl1 function. The first element of the vector is used as the initial value.

3.5.1 Ant colony optimization

The first algorithm we implemented was Ant Colony Optimization (ACO). The implementation of this algorithm is in the Shizen.AntColony module. This module contains two submodules, one with the types, and one for the algorithm itself. We will start by discussing the types, and then we will discuss the algorithm.

Types

The parallel implementation of ACO requires the same data structure as the sequential implementation, i.e. a solution archive that works as a pheromone table. This data structure is implemented as an Acc (Vector (Point p)), where p is the type of the position. This data structure is used to store the best solutions found by the ants. The best solution is the first element of the vector. This is illustrated in Table 2.1.

This module also contains a submodule with another type class that extends Position. This type class is called AntPosition p b, and it contains only one function: updatePosition. This function is used to update the position of an ant, i.e. making a Gaussian sampling around the selected position. All the instances of the type class Position are also instances of this type class.

Implementation details

The implementation of the algorithm seems straightforward, but some details need to be taken into account. This section will address the most significant ones.

Efficient parallel implementation of the roulette wheel selection

The first one is the way we select m elements from the archive, given a vector representing the probabilities. The first thing we need to do is to convert the probabilities into a vector of cumulative probabilities. This is done by using the scanl1 function, which is similar to the scanl function, but instead of using an initial value, it uses the first element of the vector. This is illustrated in Listing 3.1.

Once we have the vector of cumulative probabilities, we need to generate m

random numbers between 0 and 1, and then we need to find the index of the first element of the vector that is greater than the random number. This idea is simple to implement sequentially, but it is not so simple to implement in a parallel fashion without having nested parallelism.

The parallel implementation of this selection process draws inspiration from the matrix-vector product parallel implementation discussed in Appendix A.2. The idea behind this implementation is to make copies of the vector of cumulative probabilities and the vector of random numbers. These copies are processed in parallel, this way we can bypass the nested parallelism limitation.

The process now is simple. To find the index of the first element greater than the random number, we compare the random number with all the elements of the vector of cumulative probabilities at the same time, and then we retrieve the index of the first element that is greater than the random number. A detailed explanation of this process is provided in Appendix A.3.

Efficient parallel implementation of the Gaussian sampling

At this point, we have a vector of positions that need to be updated. To update a position, we need to make a Gaussian sampling around it. To do this, we need to compute first the standard deviation as defined in Section 2.2.1. Again, we have a context in which parallelizing without nested parallelism is not straightforward.

However, we can use the same approach as before. To compute the standard deviation, we need to compute the mean of the differences between the positions in the archive and the selected position. To do this, we make copies of the vector of positions and the selected position. Then, we process these copies in parallel, computing for each position the standard deviation, as defined in Section 2.2.1. Thus, we can now compute the Gaussian sampling, i.e. generate a random number from a normal distribution with mean the reference position, and standard deviation as computed before in parallel without worrying about nested parallelism. A detailed explanation of this process is provided in Appendix A.4.

Sorting the archive

The last detail we need to address is the sorting of the archive. The archive needs to be sorted after each iteration, so the best solution is always the first element of the vector. This is done by using the `sort` function from the `containers-accelerate` library. This library implements parallel versions of *merge sort* and *quicksort*.

As the archive is maintained sorted, at each iteration we only need to sort the m ants created and then insert them in the correct position. This is done by using

the insertion_sort function, which is implemented for the merge sort algorithm.

3.5.2 Particle swarm optimization

The implementation of the Particle Swarm Optimization (PSO) algorithm is in the Shizen.ParticleSwarm module. This module contains the implementation of the PSO algorithm, as well as the implementation of the Particle type, which is used to represent the particles of the swarm.

Types

The first thing we need to do is to define the only type used in this module, which is the Particle type. This type is defined as a tuple of 4 elements. The first 3 elements are of type Point p, representing the current position and its fitness, the best position found by the particle and its fitness, and the best position found so far by the swarm with its fitness as well. The last element is the velocity of the particle, which is of type p.

Implementation details

The implementation of the algorithm is straightforward and it follows the description of the algorithm in Section 2.3.1. There was no need to use any special technique to implement this algorithm.

3.5.3 Differential evolution

The implementation of the Differential Evolution (DE) algorithm is in the Shizen.DifferentialEvolution module. This module contains two submodules, one with the types required, and another one with the algorithm itself.

Types

As in the ACO algorithm, the data structure used is a vector of Point p. Moreover, this module also contains the definition of another extension of the Position type class, which is the DEPosition type class. This type class defines the function required by the DE algorithm to work. This function is updatePosition, which is used to mutate a position and then do a crossover between the mutated position and the original position.

Implementation details

The implementation of the algorithm is also straightforward and it follows the description of the algorithm in Section 2.4.1. There was no need to use any special technique to implement this algorithm.

Chapter 4

Results and Analysis

This chapter is divided into two sections. In the first section, we will discuss the problems we encountered during the development of the project, and how we solved them (if we did). In the second section, we will present the results obtained from the implementation and parallelization of the bio-inspired algorithms. Finally, we will compare the results obtained from the bio-inspired algorithms with the results obtained from the traditional imperative approaches to parallel computing.

4.1 Encountered problems

During the development of the project, we encountered several problems that delayed the completion of the project. In this section, we will describe the most important ones.

4.1.1 Bug in the mergesort of containers-accelerate

Background and context

As mentioned in section 3.2, the containers-accelerate package provides an implementation of some container types, such as maps and sets, for the Accelerate library. However, in our project, we required it primarily for conducting sorting operations on vectors when implementing the aco algorithm.

As explained in section 2.2, the aco algorithm uses a vector to represent the solution archive, and it needs to be sorted to represent the pheromone trails. As the archive is sorted partially, we just need to sort the new elements and add them in the correct position. With this information, the mergesort algorithm is the best choice for this task.

Occurrence and manifestation

The problem manifested itself when testing the aco algorithm. As shown in the section 4.2, the firsts benchmark functions we tested had their minimum at the origin. When we tested them, the algorithm always converged to the minimum value with a very high precision and in a very short time. This was not the expected behavior, as some of the functions have many local minima, and the algorithm should most likely converge to one of them, not to the global minimum.

This strange behavior made us think that there was a problem with the implementation of the aco algorithm. The testing continued and we discovered a problem that was even more strange. When we tested a benchmark function with a negative minimum value, the algorithm escaped from the bounds of the function and converged to $\pm\infty$. This behavior convinced us that the problem was in Shizen.Types module, where the fixBounds function is defined.

Nevertheless, despite extensive debugging and efforts to isolate the issue, we could not identify any flaws within the fixBounds function. Consequently, we opted to reevaluate the aco algorithm. Initially, we believed the problem stemmed from the aco algorithm itself. However, through thorough debugging and with the invaluable assistance of Javier Lobillo, whose creative thinking played a pivotal role in resolving this matter, we discovered that the actual issue resided within the mergesort algorithm.

Impact and consequences

The impact of this problem was an important delay in the development of the project. We spent a lot of time trying to find the problem in the aco algorithm, and we did not realize that the problem was in the mergesort algorithm until the end of the project. We had to solve the problem ourselves, which also took a lot of time.

In addition, the problem with the mergesort algorithm made us lose confidence in the containers-accelerate package, which made us analyze the implementation of the package in detail. Unfortunately, this setback resulted in time constraints that forced us to expedite the development of the remaining algorithms. As a consequence, we were unable to implement several algorithms that we had initially planned.

Attempts at resolution

The first attempt to solve the problem was to open an issue in the containers-accelerate repository (here is the link of the issue). Even though the maintainer of the package answered us, he could not find the problem, and then he did not

answer us anymore.

After that, we decided to make a *fork* of the package and try to fix the problem ourselves. We spent a lot of time trying to understand the implementation of the package, and we finally found the problem.

The algorithm works by splitting the vector into smaller vectors, sorting them using insertion sort, and then merging them. This last step works by reordering the indices of the elements of the vector, which is done by the scatter function. This function has undefined behavior when the indices are repeated. This fact made us think that the problem was in the function which computes the new indices of the elements of the vector. After analyzing the implementation, we found that at an edge case, the function was generating repeated indices, and with a single line of code, we solved the problem.

4.1.2 Internal error in package Accelerate I

Background and context

As discussed in section 3.2, the accelerate-llvm-ptx library provides a backend for running computations on NVIDIA GPUs. This library is crucial for our project, as the GPU was the main target of our project. However, due to hardware limitations, we did not have access to a GPU until the beginning of April, which made us unable to test the algorithms on the GPU until a moment when a big change in the approach of the project was not possible.

Occurrence and manifestation

The problem manifested itself when we tried to run the aco algorithm on the GPU. The algorithm was working on the CPU, but when we tried to run it on the GPU, we got an internal error, which is shown in the Listing 4.1.

Impact and consequences

This problem had a big impact on the development of the project. We had to stop the development of the project for a few weeks, trying to isolate the problem and find a solution or a workaround. All of these efforts were unsuccessful, and we lost a lot of time and motivation. There were a few reasons to think that this was the only problem and that maybe the library was not ready for production.

```
*** Internal error in package accelerate ***
*** Please submit a bug report at
↪   https://github.com/AccelerateHS/accelerate/issues

CUDA Exception: misaligned address

CallStack (from HasCallStack):
  internalError: Data.Array.Accelerate.LLVM.PTX.State:53:9
```

Listing 4.1: Message provided by the accelerate-llvm-ptx backend when the internal error "misaligned address" occurred.

Attempts at resolution

The first attempt to solve the problem was to open an issue in the accelerate repository (here is the link of the issue)). One of the maintainers of the package answered us, and he told us that they have had a similar problem with a large project. Unfortunately, they weren't able to find the problem and neither were we.

After that, we tried to isolate the problem by creating a minimal example that reproduced the problem, but we were unable to do it. We also tried to run the algorithm on a different GPU, but the problem persisted. The time was running out, and we had to find a solution. We decided to change the approach of the thesis and focus only on parallelizing the algorithms on the CPU.

4.1.3 Internal error in package Accelerate II

Background and context

Once we decided to focus only on parallelizing the algorithms on the CPU, we started the development of some algorithms. We started with the pso algorithm, which was the simplest one. We were able to parallelize the algorithm without any problems. Since the bug in the aco algorithm was still present, we decided to try to run the pso algorithm on the GPU. We expected to get the same internal error, but surprisingly, the algorithm worked correctly.

Occurrence and manifestation

After the first success, we had to reevaluate the situation. We decided to analyze each algorithm taking into account where it was going to be executed, i.e. aco on the CPU and pso on both. We started to test the pso in higher dimensions, and we found that the algorithm does not work with a 30-dimensional problem.

```
*** Internal error in package accelerate ***
*** Please submit a bug report at
↪   https://github.com/AccelerateHS/accelerate/issues

CUDA Exception: invalid argument

CallStack (from HasCallStack):
  internalError: Data.Array.Accelerate.LLVM.PTX.State:53:9
```

Listing 4.2: Message provided by the accelerate-llvm-ptx backend when the internal error "invalid argument" occurred.

We were able to run the algorithm up to 29 dimensions, but when we tried to run it with 30 dimensions, we got an internal error, which is shown in the Listing 4.2.

Impact and consequences

At this point, we were very confused. We were able to run the algorithm on the GPU with 29 dimensions, but not with 30. Moreover, we were not sure if the problem was in the accelerate library or in the GPU. Again, this problem had an impact on the development of the project, since we had to stop the development.

Attempts at resolution

The first attempt to solve the problem was trying to understand why the algorithm was working with 29 dimensions but not with 30. We analyzed the memory usage of the algorithm, and we found that the memory usage was normal and that the algorithm was not using more memory than the GPU had.

After that, we tried to isolate the problem by creating a minimal example that reproduced the problem, but we were unable to do it. We also tried to run the algorithm on a different GPU, but the problem persisted. The workaround that we came up with was to test the algorithm just with 29 dimensions.

4.2 Performance Evaluation

In this section, we are going to evaluate the performance of the algorithms implemented in this thesis, in terms of the quality of the solutions and the execution time.

Test function	S	f_{min}
$f_1(x) = \sum_{i=1}^{n} x_i^2$	$[-100, 100]^n$	0
$f_2(x) = \sum_{i=1}^{n} \lvert x_i \rvert + \Pi_{i=1}^{n} \lvert x_i \rvert$	$[-10, 10]^n$	0
$f_{10}(x) = -20 \cdot \exp[-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}] - \exp[\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)] + 20 + e$	$[-32, 32]^n$	0

Table 4.1: Benchmark functions used in this thesis. n is the dimension of the problem, S is the search space, and f_{min} is the global minimum.

4.2.1 Benchmark functions

To test the algorithms, we need to use a set of benchmark functions. These functions are used to evaluate the performance of the algorithms. The benchmark functions used in this thesis are a subset of the benchmark functions defined in [41]. The functions are divided into two groups: unimodal and multimodal. The unimodal functions have only one global optimum, while the multimodal functions have multiple local minima, making them more difficult to optimize.

The functions selected for this thesis are shown in Table 4.1. These functions provide a good representation of the problems that the algorithms are going to solve. These functions are analyzed in [41], however, we are going to provide a brief description of each function.

Sphere model

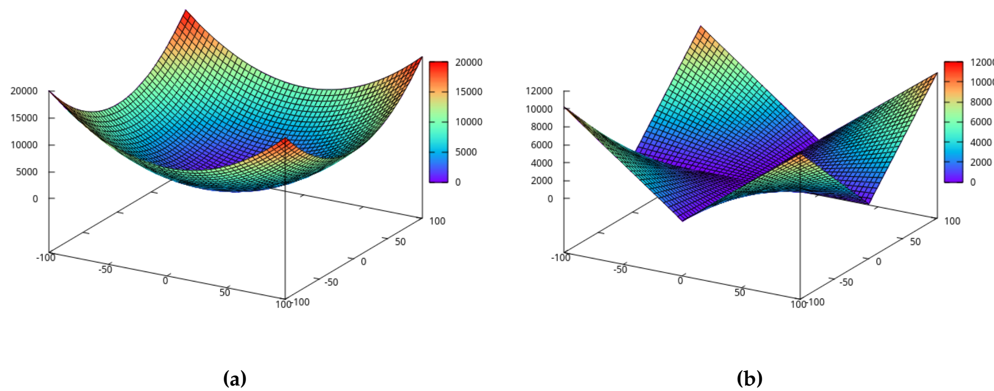
The first function that we are going to analyze is the n-dimensional sphere function. This function is defined in Table 4.1 as f_1. This function is quite simple to optimize, and every meaningful optimization algorithm should be able to find the global minimum.

This function has n local minima, and the global minimum is located at the origin. The function is convex, and it is unimodal. A graphical representation of the function is shown in Figure 4.1a.

Schwefel's Problem 2.22

The second function that we are going to analyze is Schwefel's Problem 2.22. This function is defined in Table 4.1 as f_2. Again, this function is quite simple to optimize, and every meaningful optimization algorithm should be able to find the global minimum.

The function is convex, and unimodal, and the global minimum is located at the origin. A graphical representation in two dimensions of the function is shown in Figure 4.1b.



(a) (b)

Figure 4.1: Graphical representation in two dimensions of the unimodal functions used in this thesis. **(a)** Sphere function. **(b)** Schwefel's Problem 2.22.

Ackley's function

The third function that we are going to analyze is Ackley's function. This function is defined in Table 4.1 as f_{10}. This function is widely used in the literature to test the performance of optimization algorithms. It was proposed by David Ackley in 1987 [1]. This function is more difficult to optimize than the previous ones, and it makes a difference between good and bad optimization algorithms.

The function is characterized by having a large number of local minima, and a single global minimum. The function is multimodal, and it is not convex. A graphical representation in two dimensions of the function is shown in Figure 4.2. The graphical representation helps to understand the behavior of the function, it has a "flat" area on the outer part and a "hole" in the center. The "flat" area is a manner of speaking since the function has many local optima in that area.

4.2.2 Test environment

The algorithms were tested on a computer with the following specifications:

- OS: Manjaro Linux x86_64
- Kernel: 6.1.26-1-MANJARO
- Shell: zsh 5.9
- CPU: Intel i7-4790 (8) @ 4.000GHz
- GPU: NVIDIA GeForce GTX 760 OEM
- RAM: 15949MiB

The chosen test environment provided a capable system for evaluating the algorithms. The Manjaro Linux operating system offered a stable and efficient platform, while the Intel i7-4790 CPU provided ample processing power with its

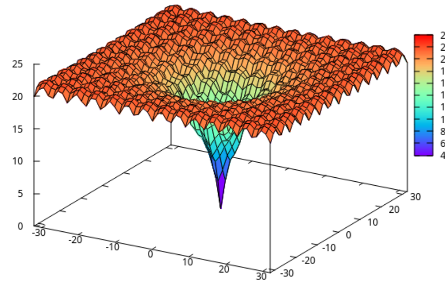


Figure 4.2: Graphical representation in two dimensions of the multimodal function used in this thesis. Ackley's function.

eight cores. The NVIDIA GeForce GTX 760 OEM GPU added further computational capabilities, which were used to accelerate the algorithms that supported GPU processing. With 15949MiB of RAM available, the system had sufficient memory to handle the algorithms and any associated data sets.

It's worth noting that the choice of the test environment might influence the performance and behavior of the algorithms. Different operating systems, hardware configurations, and software versions could yield varying results. Therefore, the results obtained in this specific test environment may not be directly applicable to other systems or configurations. However, the information provided here serves as a reference for understanding the computing resources used during the evaluation of the algorithms.

4.2.3 How tests were performed

The algorithms were tested with different dimensions, iterations, and population sizes. The dimensions were fixed to $n = 5$, $n = 10$, $n = 20$, and $n = 29$. The rest of the parameters were varied depending on the algorithm. Furthermore, each algorithm has a set of parameters that can be tuned. The parameters were set to the values shown in Section 2.1.

The algorithms were run 50 times for each combination of parameters. The compilation of the code incurred a runtime overhead, which was accounted for in all the runs. It is important to clarify that the implementation of the algorithm is the same for all the runs. The only difference between the runs is the backend used. The backends are either CPU or GPU. For testing the sequential version of the algorithms, the CPU backend includes a parameter that enforces execution within a single thread.

The results were averaged, and the standard deviation was calculated. Moreover, the best solution found by the algorithm was recorded. The best solution is

defined as the solution with the lowest value. Moreover, the time required to find the best solution was recorded. The time was measured in seconds.

4.2.4 Performance of the pso algorithm

In this section, we are going to analyze the performance of the pso algorithm. This subsection is divided into two parts. First, we are going to analyze the quality of the solutions. Second, we are going to analyze the execution time.

Quality of the solutions

The function's value is used to measure the quality of the solutions. A lower value indicates a better solution. The results can be seen in Table 4.2.

Based on the results, it is evident that the pso algorithm can successfully converge to the global optima in almost all cases. The only exception is Ackley's function with low dimensions, where the algorithm becomes trapped in a local optimum. However, in all other cases, the algorithm is capable of converging to the global optima.

A noteworthy observation is that the algorithm performs better when the number of particles is increased. This outcome was anticipated, as a larger number of particles enables the algorithm to explore the search space more efficiently. This trend is consistent across all the functions.

Additionally, the quality of the solutions remains consistent regardless of changes in the backend. This outcome is expected since the quality of the solutions is independent of the backend utilized.

Execution time

The execution time of the pso algorithm is shown in Table 4.3. The results were surprisingly, as the GPU was slower than the CPU, and the multicore version was slower than the singlecore version. This will be discussed in more detail in Section 4.2.8.

Function	Backend	Measure	n=5	n=10	n=20	n=29
		Best sol.	9.085e-8	5.914e-5	1.312e-7	1.646e-10
	GPU	Mean	7.416e-6	1.538e-3	1.181e-5	3.226e-9
		Stdev	1.650e-5	1.869e-3	1.825e-5	3.458e-9
	CPU	Best sol.	2.506e-8	1.970e-5	2.106e-7	3.719e-10
	Multicore	Mean	4.015e-5	2.474e-3	8.906e-6	6.269e-9
f_1		Stdev	1.437e-4	8.097e-3	1.613e-5	7.679e-9
	CPU	Best sol.	1.019e-7	1.033e-5	6.494e-7	2.725e-10
	Singlecore	Mean	3.074e-5	1.391e-3	8.901e-6	4.467e-9
		Stdev	9.085e-5	2.345e-3	1.270e-5	3.745e-9
		Iterations	28	56	200	400
		Population	256	512	1024	2048
		Best sol.	2.813e-18	4.942e-9	4.958e-4	5.392e-3
	GPU	Mean	1.091e-16	1.207e-6	6.627e-3	2.202e-2
		Stdev	1.434e-16	2.627e-6	6.472e-3	2.044e-2
	CPU	Best sol.	1.712e-18	2.729e-9	7.575e-4	3.419e-3
	Multicore	Mean	1.273e-16	4.467e-7	4.650e-3	1.919e-2
f_2		Stdev	2.776e-16	8.215e-7	3.358e-3	1.196e-2
	CPU	Best sol.	3.663e-18	1.508e-9	1.104e-3	6.214e-3
	Singlecore	Mean	9.804e-17	1.699e-6	5.130e-3	1.930e-2
		Stdev	1.365e-16	5.010e-6	4.594e-3	1.053e-2
		Iterations	100	100	100	100
		Population	15000	15000	15000	15000
		Best sol.	1.537e+0	1.323e+0	7.706e-1	8.945e-2
	GPU	Mean	1.537e+0	1.323e+0	7.706e-1	9.059e-2
		Stdev	2.156e-15	1.857e-11	2.815e-5	9.782e-4
	CPU	Best sol.	1.537e+0	1.323e+0	7.706e-1	8.932e-2
	Multicore	Mean	1.537e+0	1.323e+0	7.706e-1	9.020e-2
f_{10}		Stdev	2.697e-15	3.436e-11	4.765e-5	5.376e-4
	CPU	Best sol.	1.537e+0	1.323e+0	7.706e-1	8.918e-2
	Singlecore	Mean	1.537e+0	1.323e+0	7.706e-1	9.003e-2
		Stdev	2.961e-15	1.520e-11	3.787e-5	6.367e-4
		Iterations	100	100	100	100
		Population	15000	15000	15000	15000

Table 4.2: Quality of the solutions obtained by the pso algorithm. The table shows the best solution, the mean, and the standard deviation of the solutions obtained by the algorithm for each function, backend, and dimension.

Function	Backend	Measure	n=5	n=10	n=20	n=29
f_1	GPU	Best time	4.500e-1	6.700e-1	2.530e+0	5.980e+0
		Mean	4.570e-1	6.835e-1	2.573e+0	6.120e+0
		Stdev	7.327e-3	1.814e-2	2.408e-2	8.553e-2
	CPU Multicore	Best time	3.900e-1	5.300e-1	1.670e+0	3.640e+0
		Mean	3.980e-1	5.465e-1	1.702e+0	3.713e+0
		Stdev	3.350e-2	4.133e-2	7.806e-2	1.024e-1
	CPU Singlecore	Best time	3.600e-1	5.000e-1	1.620e+0	3.520e+0
		Mean	3.755e-1	5.135e-1	1.643e+0	3.542e+0
		Stdev	8.256e-3	7.452e-3	1.302e-2	1.309e-2
		Iterations	28	56	200	400
		Population	256	512	1024	2048
f_2	GPU	Best time	5.900e-1	8.000e-1	2.040e+0	3.640e+0
		Mean	6.085e-1	8.275e-1	2.083e+0	3.731e+0
		Stdev	1.137e-2	1.888e-2	2.003e-2	4.388e-2
	CPU Multicore	Best time	5.200e-1	7.700e-1	2.050e+0	3.760e+0
		Mean	5.365e-1	7.825e-1	2.074e+0	3.817e+0
		Stdev	9.333e-3	1.070e-2	1.899e-2	3.420e-2
	CPU Singlecore	Best time	5.600e-1	8.800e-1	2.370e+0	4.190e+0
		Mean	5.660e-1	8.915e-1	2.401e+0	4.215e+0
		Stdev	5.026e-3	8.127e-3	1.774e-2	1.356e-2
		Iterations	100	100	100	100
		Population	15000	15000	15000	15000
f_{10}	GPU	Best time	5.900e-1	8.300e-1	2.120e+0	3.850e+0
		Mean	6.080e-1	8.610e-1	2.180e+0	3.962e+0
		Stdev	1.196e-2	6.632e-2	1.333e-1	1.927e-1
	CPU Multicore	Best time	5.800e-1	8.700e-1	2.340e+0	4.260e+0
		Mean	5.920e-1	8.970e-1	2.354e+0	4.321e+0
		Stdev	6.959e-3	1.525e-2	9.445e-3	3.740e-2
	CPU Singlecore	Best time	7.200e-1	8.700e-1	2.320e+0	4.230e+0
		Mean	7.245e-1	9.010e-1	2.362e+0	4.304e+0
		Stdev	6.048e-3	1.944e-2	3.669e-2	5.326e-2
		Iterations	100	100	100	100
		Population	15000	15000	15000	15000

Table 4.3: Execution time of the `pso` algorithm. The table shows the best time, the mean, and the standard deviation of the execution time of the algorithm for each function, backend, and dimension.

4.2.5 Performance of the de algorithm

In this section, we will discuss the performance of the de algorithm. As in the previous section, we will first discuss the quality of the solutions found by the algorithm, and then we will discuss its execution time.

Quality of the solutions

Similar to the previous algorithm, this particular algorithm demonstrated the ability to locate the global minimum of functions f_1, f_2, and f_{10} in most cases. However, the same exception occurred with the function f_{10} in lower dimensions, where the algorithm became trapped in a local minimum. The outcomes of the algorithm can be found in Table 4.4.

Given that this algorithm focuses primarily on exploitation, it is unsurprising that its performance improves with increasing iterations. The table confirms this observation, as higher dimensions necessitated an increase in the number of iterations to maintain the same solution quality. In this algorithm, it is important to highlight that the population size is not large compared to the previous algorithm. This is because the current algorithm operates by improving the existing population. Consequently, having a larger population could potentially result in slower convergence.

Once again, the algorithm yielded comparable results across different backends, demonstrating no significant discrepancies.

Execution time

As we can see in Table 4.5, the execution time of the de algorithm is slower in the GPU than in the CPU. This behavior was not expected and the possible causes are discussed in Section 4.2.8.

Function	Backend	Measure	n=5	n=10	n=20	n=29
f_1	GPU	Best sol.	7.124e-15	7.296e-14	5.344e-7	1.350e-6
		Mean	2.345e-14	2.078e-13	8.504e-7	2.018e-6
		Stdev	9.917e-15	8.879e-14	2.114e-7	3.886e-7
	CPU Multicore	Best sol.	4.923e-15	5.191e-9	4.481e-7	1.253e-6
		Mean	2.802e-14	1.059e-8	8.927e-7	2.016e-6
		Stdev	1.769e-14	4.102e-9	2.707e-7	3.860e-7
	CPU Singlecore	Best sol.	6.554e-15	6.289e-9	6.953e-7	1.182e-6
		Mean	2.466e-14	1.133e-8	9.746e-7	1.911e-6
		Stdev	1.255e-14	3.073e-9	1.785e-7	5.222e-7
		Iterations	300	600	1400	3000
		Population	300	300	300	300
f_2	GPU	Best sol.	2.463e-12	5.549e-9	6.486e-5	1.127e-4
		Mean	5.432e-12	1.288e-8	8.640e-5	1.403e-4
		Stdev	2.289e-12	3.441e-9	1.279e-5	1.657e-5
	CPU Multicore	Best sol.	2.393e-12	7.319e-9	5.725e-5	1.000e-4
		Mean	5.218e-12	1.512e-8	8.232e-5	1.367e-4
		Stdev	1.472e-12	3.417e-9	1.234e-5	2.080e-5
	CPU Singlecore	Best sol.	2.11e-12	6.451e-9	6.575e-5	1.206e-4
		Mean	4.83e-12	1.313e-8	8.487e-5	1.402e-4
		Stdev	1.71e-12	3.489e-9	1.339e-5	1.572e-5
		Iterations	300	600	1400	3000
		Population	300	300	300	300
f_{10}	GPU	Best sol.	1.537e+0	1.323e+0	7.706e-1	8.912e-2
		Mean	1.537e+0	1.323e+0	7.706e-1	8.912e-2
		Stdev	1.689e-11	1.395e-8	4.035e-6	9.953e-8
	CPU Multicore	Best sol.	1.537e+0	1.323e+0	7.708e-1	8.912e-2
		Mean	1.537e+0	1.323e+0	7.708e-1	8.912e-2
		Stdev	9.223e-9	1.878e-8	2.078e-5	1.003e-7
	CPU Singlecore	Best sol.	1.537e+0	1.323e+0	7.707e-1	8.912e-2
		Mean	1.537e+0	1.323e+0	7.708e-1	8.912e-2
		Stdev	1.347e-8	2.164e-8	2.752e-5	6.485e-8
		Iterations	300	600	1400	10000
		Population	300	300	300	300

Table 4.4: Quality of the solutions of the de algorithm. The table shows the best solution, the mean, and the standard deviation of the solutions of the algorithm for each function, backend, and dimension.

Function	Backend	Measure	n=5	n=10	n=20	n=29
		Best time	5.000e-1	8.500e-1	2.060e+0	4.880e+0
	GPU	Mean	5.160e-1	8.770e-1	2.111e+0	5.004e+0
		Stdev	1.875e-2	6.258e-2	6.409e-2	6.451e-2
	CPU	Best time	4.000e-1	5.300e-1	1.310e+0	2.670e+0
	Multicore	Mean	4.120e-1	5.430e-1	1.496e+0	3.059e+0
f_1		Stdev	5.231e-3	6.569e-3	1.432e-1	2.284e-1
	CPU	Best time	3.500e-1	4.600e-1	1.060e+0	2.250e+0
	Singlecore	Mean	3.585e-1	4.715e-1	1.077e+0	2.265e+0
		Stdev	5.871e-3	5.871e-3	1.410e-2	1.469e-2
		Iterations	300	600	1400	3000
		Population	300	300	300	300
		Best time	5.000e-1	8.000e-1	2.210e+0	4.900e+0
	GPU	Mean	5.100e-1	8.185e-1	2.277e+0	5.009e+0
		Stdev	7.947e-3	2.134e-2	3.466e-2	5.600e-2
	CPU	Best time	4.000e-1	5.700e-1	1.350e+0	2.610e+0
	Multicore	Mean	4.095e-1	5.890e-1	1.542e+0	3.143e+0
f_2		Stdev	5.104e-3	5.525e-3	1.410e-1	2.913e-1
	CPU	Best time	3.500e-1	4.700e-1	1.080e+0	2.190e+0
	Singlecore	Mean	3.585e-1	4.885e-1	1.101e+0	2.216e+0
		Stdev	6.708e-3	8.127e-3	1.518e-2	2.010e-2
		Iterations	300	600	1400	3000
		Population	300	300	300	300
		Best time	5.400e-1	8.500e-1	2.200e+0	1.308e+1
	GPU	Mean	5.585e-1	8.715e-1	2.288e+0	1.354e+1
		Stdev	1.599e-2	2.870e-2	2.628e-2	2.635e-1
	CPU	Best time	3.900e-1	6.300e-1	1.510e+0	7.470e+0
	Multicore	Mean	4.005e-1	6.420e-1	1.891e+0	1.148e+1
f_{10}		Stdev	3.940e-3	1.240e-2	2.533e-1	1.621e+0
	CPU	Best time	3.600e-1	5.100e-1	1.210e+0	5.380e+0
	Singlecore	Mean	3.715e-1	5.260e-1	1.233e+0	5.418e+0
		Stdev	4.894e-3	8.208e-3	2.296e-2	2.375e-2
		Iterations	300	600	1400	10000
		Population	300	300	300	300

Table 4.5: Execution time of the de algorithm. The table shows the best solution, the mean, and the standard deviation of the execution time of the algorithm for each function, backend, and dimension.

4.2.6 Performance of the aco algorithm

This section presents the performance of the aco algorithm. As in the previous algorithms, this section is divided into two parts. The first part presents the quality of the solutions obtained by the algorithm, and the second part presents the execution time of the algorithm.

Quality of the solutions

The quality of solutions obtained using the aco algorithm is presented in Table 4.6[1]. However, the results depicted in the table are disappointing, as the algorithm fails to produce even a satisfactory solution for any of the functions. Moreover, the best solution found by the algorithm significantly deviates from the global optimum. This is the reason why we stopped the analysis since the results are not worth presenting.

These findings are unexpected considering the widespread recognition of the aco algorithm and its successful application to various optimization problems, especially in the domain of discrete optimization. However, when implementing the continuous version of the algorithm based on the studies conducted by Socha [35][34], we encountered contrasting results. The only divergence between our implementation and the referenced studies is the introduction of parallelism.

Function	Backend	Measure	n=5	n=10	n=20	n=29
		Best sol.	-	-	-	-
	GPU	Mean	-	-	-	-
		Stdev	-	-	-	-
	CPU	Best sol.	4.529e+2	2.758e+3	2.040e+4	4.280e+4
	Multicore	Mean	1.188e+3	7.479e+3	2.829e+4	4.871e+4
f_1		Stdev	3.897e+2	1.607e+3	3.342e+3	3.823e+3
	CPU	Best sol.	4.696e+2	4.238e+3	2.416e+4	4.035e+4
	Singlecore	Mean	1.335e+3	7.572e+3	2.850e+4	4.917e+4
		Stdev	5.926e+2	1.824e+3	2.187e+3	3.762e+3
		Iterations	100	100	100	100
		Population	800	800	800	800

Table 4.6: Quality of the solutions of the aco algorithm. The table shows the best solution, the mean, and the standard deviation of the quality of the solutions of the algorithm the f_1 function, CPU backends, and some dimensions.

[1] The results for the GPU backend are not presented in the table, as the algorithm failed to run, as explained in Section 4.1.2.

Execution time

The execution time of the aco algorithm is presented in Table 4.7. For completeness, the execution time of the algorithm is also presented, even though the algorithm failed to produce any solution.

Function	Backend	Measure	n=5	n=10	n=20	n=29
		Best time	-	-	-	-
	GPU	Mean	-	-	-	-
		Stdev	-	-	-	-
	CPU	Best time	6.400e-1	7.700e-1	1.460e+0	2.250e+0
	Multicore	Mean	6.465e-1	7.870e-1	1.478e+0	2.274e+0
f_1		Stdev	6.708e-3	8.013e-3	1.387e-2	1.392e-2
	CPU	Best time	4.900e-1	6.600e-1	1.470e+0	2.330e+0
	Singlecore	Mean	4.990e-1	6.675e-1	1.494e+0	2.353e+0
		Stdev	6.407e-3	7.864e-3	2.038e-2	1.743e-2
		Iterations	100	100	100	100
		Population	800	800	800	800

Table 4.7: Execution time of the aco algorithm. The table shows the best solution, the mean, and the standard deviation of the execution time of the algorithm the f_1 function, CPU backends, and some dimensions.

4.2.7 Comparative with imperative implementations

In this section, we compare the performance of the functional implementations with the performance of imperative implementations of the same algorithms. Specifically, we compare our implementations with the implementations of the same algorithms written directly in CUDA C. The reference imperative implementations are provided by Iván Cañaveral [6] (code available at ivanCanaveral/GMEL).

As expected, the imperative implementations outperform the functional implementations in all cases. The performance difference is more pronounced in the GPU backend, where the imperative implementations outperform our functional implementations by a factor of 165.5x in the pso algorithm.

Even though we expected the imperative implementations to be faster than the functional implementations, we did not expect such a large difference in performance. We attribute this large difference in performance mainly to the overhead introduced by the runtime compilation.

Nevertheless, we must note that the benefits of functional programming are not limited to performance. The functional implementations are more concise and easier to understand than the imperative implementations. Furthermore, the functional implementations are more flexible, as they can be executed in different backends without any modification. In contrast, the imperative implementations

are tied to the GPU backend, and they cannot be executed in different backends without significant modifications.

4.2.8 Discussion

In the following section, we will examine the findings presented in the preceding sections. Initially, we will delve into the reasons behind the single-core backend's ability to outperform both the multicore and GPU backends. Subsequently, we will explore the factors contributing to the slowness of the algorithms in general, and, finally, we will discuss the failure of the `aco` algorithm in producing any viable solution.

Single-core backend outperforms multicore and GPU backends

To our surprise, the single-core backend consistently outperforms both the multicore and GPU backends in all algorithms, contrary to expectations given the parallelization capabilities of the latter two. We attribute this unexpected result to the overhead introduced by parallelization. In the multicore backend, inter-thread communication contributes to the overhead, while in the GPU backend, communication between the host and the device is the primary source. In contrast, the single-core backend operates without such communication overhead, enabling smooth execution of the algorithm.

We speculate that larger datasets could potentially improve the performance of the multicore and GPU backends by allowing the overhead to be amortized through the increased computational load. Regrettably, we lack the necessary resources to empirically test this hypothesis.

Furthermore, as discussed in Section 1.1.2, it is worth noting that GPUs typically possess a greater number of cores compared to CPUs. However, the cores in GPUs are designed to be simpler compared to CPU cores. Consequently, the individual power of GPU cores is comparatively lower than that of CPU cores. This distinction in power could potentially account for the performance disparity observed between the single-core and GPU backends. The complex nature of data representation within the algorithms might be a contributing factor as well. It is conceivable that the internal data type representation is excessively intricate for the GPU cores to efficiently handle.

Backends are slower than expected

The execution time of all backends takes much longer than expected, mainly because of the additional time needed for runtime compilation. To avoid this delay,

we can use a different function called runQ instead of run to run the algorithm. This technique is known as "ahead-of-time" compilation.

By using runQ, the necessary code for executing the Accelerate computation is generated, compiled, and linked into the final program during the compilation phase of Haskell. This approach eliminates the extra time introduced by runtime compilation. Additionally, the generated code is specifically compiled for the default GPU architecture, and all the required data is included directly in the program.

However, due to the random nature of our algorithms, we cannot use runQ to execute them. These algorithms rely on a random number generator's seed, which cannot be included as part of the program. Furthermore, we don't know the size of the input data at the time of compilation, which is another requirement for using runQ. As a result, we are forced to use run instead of runQ to execute our algorithms in this situation.

ACO algorithm fails to produce any solution

The aco algorithm proves to be ineffective in generating solutions for any of the functions. In our previous discussion on the algorithm, we emphasized its reliance on the pheromone representation of the problem.

Although the pheromone representation proposed by Socha [35] is suitable for continuous optimization problems, it falls short when applied to high-dimensional problems. This representation lacks a clear strategy, offering neither exploration nor exploitation. The algorithm generates new solutions in the vicinity of the initial random ones, but it quickly becomes trapped. Rather than being stuck in a local optimum, the main issue arises when the diversity of solutions in the archive diminishes after a certain number of iterations. Consequently, the algorithm struggles to find new solutions, impeding its effectiveness.

Even if the algorithm were able to find good solutions, it would still be a poor choice for GPGPU computing, since the intricate nature of it would make it slower than other algorithms that rely on simpler operations, such as the pso algorithm or the de algorithm.

Chapter 5

Conclusions and Future Work

In this chapter, we present the conclusions of this thesis giving a summary of the work done and the results obtained. We also discuss future work that can be done to improve the work done in this thesis.

5.1 Conclusions

In the first place, a deep review of the literature on bio-inspired algorithms, functional programming, and the Accelerate library was conducted. This review allowed us to understand the state of the art in the field, as well as the advantages of functional programming for parallel computing. By doing this, we were able to fulfill the objectives **O1.1** and **O1.2**. Moreover, when reviewing the literature on Accelerate, we were able to understand some advanced features of Haskell, such as some language extensions and how to hide computations behind types. This allowed us to fulfill the objectives **O1.3** and **O1.4**.

Once the review was completed and the necessary knowledge was acquired, the algorithms were implemented in Haskell and parallelized using the Accelerate library. This process took longer than expected, encountering several problems and bugs that were not anticipated. Nevertheless, we were able to contribute to the development of the library by reporting these bugs and proposing solutions when possible. Regardless of the problems encountered, the implementation was completed and the objective **O2** was fulfilled.

Subsequently, some literature on how to analyze the performance of algorithms was reviewed. This allowed us to understand how to measure the overall performance of the algorithms, as well as the quality of the results obtained. By doing this, we were able to fulfill the objective **O3.1**. Then, we proceeded to measure the execution time and quality of the results obtained by the parallelized algorithms. This allowed us to fulfill the objective **O3.2**.

Finally, the performance of the parallelized algorithms was analyzed and

compared with traditional imperative approaches to parallel computing, as well as sequential functional implementations of the algorithms. This allowed us to fulfill the objectives **O3.3** and **O3.4**, which were the last objectives of this thesis.

The results obtained were not as expected, as the parallelized algorithms were slower than the sequential implementations and much slower than the imperative approaches. Nevertheless, we were able to identify possible causes for this behavior and contribute to the research on the topic by discussing these causes.

Based on the work conducted in this thesis, we can draw several conclusions. Firstly, the Accelerate library is still under development and has some bugs that need to be fixed, making it unsuitable for use in real-world applications at the moment. However, it shows promise and can be a good starting point for future research. It's tempting to give up on using Accelerate due to the problems encountered, but it's important to remember that these issues can only be discovered by actually using the library. So, researchers should continue utilizing and improving the library, as its true potential can only be realized through active usage and contributions.

Despite the challenges faced, it is important to highlight the undeniable significance of functional programming in parallel computing. Functional programming enables us to write code that is more comprehensible and easier to analyze. This becomes apparent when examining the implementations of the algorithms presented in this thesis.

Furthermore, it is crucial to promote research in this field as it can contribute to the advancement of superior tools for parallel computing. These tools are not only valuable for solving optimization problems but also for efficiently addressing various other challenges we encounter in modern times.

Secondly, we found that the Ant Colony Optimization (ACO) algorithm is not well-suited for continuous optimizations, especially when using the pheromone representation described in [34]. Even though the algorithm can find solutions of similar quality to sequential implementations, it takes much longer to execute. This is an important lesson we learned: to achieve good performance on the GPU, algorithms should be based on simple and independent operations.

In conclusion, we can say that the objectives of this thesis were fulfilled, even though the results obtained were not as expected. Nevertheless, we were able to contribute to the research on the topic by identifying possible causes for the poor performance of the parallelized algorithms. We also contributed to the development of the Accelerate library by reporting bugs and proposing solutions when possible. Finally, we were able to demonstrate the advantages of functional programming for parallel computing by implementing these algorithms in Haskell.

5.2 Future work

As mentioned in the previous section, there is still much work to be done in this field. In this section, we discuss some of the possible future work that can be done to improve the work done in this thesis.

To begin with, the ongoing development of the Accelerate library has revealed the presence of certain bugs that require resolution. Consequently, future efforts should concentrate on enhancing the library by rectifying these bugs and adding new features. This will end up benefiting the research on the topic, as well as the development of real-world applications.

Moreover, the algorithms presented in this thesis can be used as a starting point for future research and development. Even though they are not as fast as traditional imperative approaches, some of them can be used in real-world scenarios.

Implementations, although they are correct, there is still room for improvement. This can be achieved by trying and testing different approaches in data representation and parallelization. The main example of this is the Ant Colony Optimization (ACO) algorithm, which can be potentially improved by using a different pheromone representation. Moreover, the other algorithms can also be improved by trying different approaches, and new algorithms can be implemented based on the ideas and work done in this thesis.

To conclude, we can say that there is still much work to be done in this field, and we believe that the work done in this thesis can be used as a starting point for everyone interested in this topic.

Appendices

Appendix A

Avoiding nested data-parallelism

In this appendix, we will explore the concept of nested data parallelism, methods to circumvent it, and the significance of doing so. While avoiding nested data parallelism may not always be simple, particularly in the context of the Accelerate library, we will delve into specific scenarios where this challenge arises and provide strategies to overcome it.

A.1 Understanding nested data-parallelism

As its name suggests, nested data parallelism occurs when we try to instantiate a collective operation inside another collective operation. The debate about whether nested data parallelism should be allowed or not is still open. There are some supporters, such as [8] [4], who argue that nested data parallelism can be tamed by the compiler, leading to a more expressive programming model. On the other hand, there are several reasons why nested data parallelism should be avoided. In this section, we will explore some of these reasons.

In Section 1.3, we have observed that the collective operations are contained within the `Acc` type. Let's deepen our understanding of this type. Each collective operation is formed by several scalar operations, which are the ones that are executed in parallel. These scalar operations are embedded in the `Exp` type. When a value is of type `Exp a`, it is unable to generate additional collective operations. This design choice plays a critical role in enforcing *not*-nested parallelism.

Nevertheless, there are some *backdoors* that allow us to bypass this restriction. The main one is the function `the`, which allows us to retrieve a value of the type `Exp a` from a value of type `Acc (Scalar a)`. If we are not careful, this can lead to nested data parallelism errors that are not detected by the compiler, but rather, by the runtime system.

A.1.1 Why is nested data-parallelism a problem?

Although the restriction of nested data parallelism may initially seem burdensome for programmers, it offers several benefits. By imposing this restriction, both the programmer and the compiler can gain a clearer understanding of the parallel behavior of the program. This, in turn, enables finer-grained parallelism and opens up opportunities for more aggressive compiler optimizations.

One advantage of adhering to the restriction is that the single-threaded programming model closely resembles sequential programming. This similarity makes it easier for programmers to comprehend and reason about the code. The familiar programming paradigm allows developers to leverage their existing knowledge and skills in sequential programming, potentially reducing the learning curve associated with parallel programming.

Additionally, the restriction on nested data parallelism provides the compiler with greater insights into the program's behavior. The compiler can more accurately analyze and optimize the code, as it can make assumptions about the parallel execution of operations within the data structures. This increased predictability allows for more radical and effective compiler optimizations, leading to potential performance improvements.

In summary, although the restriction on nested data parallelism may impose some limitations, it ultimately offers advantages such as finer-grained parallelism, enhanced compiler optimizations, and a programming model that aligns with sequential programming, making it easier for programmers to understand and reason about their code.

A.2 Avoiding nested data-parallelism in Accelerate

In this section, we will explore the strategy followed in this work to avoid nested data parallelism. This strategy is based on the following example, which illustrates the problem and the solution.

A.2.1 Example of nested data parallelism

A classic example of this practice is trying to implement a matrix-vector multiplication using a parallel function that calculates the dot product of two vectors. As it is well known, matrix-vector multiplication is defined as the dot product of each row of the matrix with the vector. However, if we try to implement this function naively, we will end up with nested data parallelism. This is because the dot product function is defined as a collective operation, and we are trying to use

```
ghci> run $ fold1 (+) (fromList (Z:.3) [1,2,3])
Vector (Z:.1) [6]
```

Listing A.1: Example of the fold1 function. The function argument needs to be associative.

it inside another collective operation.

The solution to this issue is to make a copy of the vector for each row of the matrix. In this approach, we replicate the vector n times and then follow these steps:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix} \times \begin{pmatrix} x_1 & x_1 & \cdots & x_1 \\ x_2 & x_2 & \cdots & x_2 \\ \vdots & \vdots & \ddots & \vdots \\ x_m & x_m & \cdots & x_m \end{pmatrix}^T \rightarrow \begin{pmatrix} a_{11}x_1 & a_{12}x_2 & \cdots & a_{1m}x_m \\ a_{12}x_2 & a_{22}x_2 & \cdots & a_{2m}x_m \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1}x_1 & a_{n2}x_2 & \cdots & a_{nm}x_m \end{pmatrix}$$

where a_{ij} is the element in the i-th row and j-th column of the matrix, x_i is the i-th element of the vector, $_ \times _$ represents a pair constructor, and $\rightarrow$ represents the application of the zipWith function, with the multiplication function as the first argument, and the two matrices as the second argument. The result of this operation is a matrix where each element is the product of the elements with matching indices of the two input matrices.

Finally, we need to sum the elements of each row of the resulting matrix. This can be done by taking advantage of the fact that the fold1 function is *rank-polymorphic*, meaning that it can be applied to arrays of any rank, and every time it is applied, it reduces the innermost[1] dimension of the array by applying the given function. This gives us the result of the matrix-vector product. An example of this operation is shown in Listing A.1.

This approach leverages parallelism to execute all operations simultaneously, eliminating the need for nested parallelism. Once we know this technique, we can apply it to other scenarios where nested data parallelism arises.

Remark A.2.1. An important fact is that when we replicate the vector, we are not creating a new matrix in memory. Instead, it will be directly fused into the operation which consumes it.

[1]When applying this function to a vector, the result is a scalar. When applying it to a matrix, the result is a vector, and the function is applied to each row of the matrix.

A.3 Efficient parallel implementation of the Roulette Wheel Selection

In this section, we will explore the implementation of the roulette wheel selection algorithm using Accelerate. This algorithm is used in some metaheuristics to select the individuals that will be used to generate the next population.

For this algorithm, we assume that we have a list of k individuals and a discrete probability distribution, where each individual has a probability associated with it. Moreover, we assume that we have m uniform random numbers between 0 and 1, since we want to select m individuals from the list.

First, we replicate the random numbers vector $r = (r_1, \ldots, r_m)$ k times, and the vector of cumulative probabilities $d = (d_1, \ldots, d_k)$ m times. Next, we apply the zipWith function, using the $<$ function as the first argument and the two vectors as the second argument. This operation yields a matrix where each element is a boolean value indicating whether the corresponding element in the random numbers vector is less than the corresponding element in the vector of cumulative probabilities.

$$\begin{pmatrix} r_1 & r_2 & \cdots & r_m \\ r_1 & r_2 & \cdots & r_m \\ \vdots & \vdots & \ddots & \vdots \\ r_1 & r_2 & \cdots & r_m \end{pmatrix} \times \begin{pmatrix} d_1 & d_1 & \cdots & d_1 \\ d_2 & d_2 & \cdots & d_2 \\ \vdots & \vdots & \ddots & \vdots \\ d_k & d_k & \cdots & d_k \end{pmatrix} \to \begin{pmatrix} r_1 < d_1 & r_2 < d_1 & \cdots & r_m < d_1 \\ r_1 < d_2 & r_2 < d_2 & \cdots & r_m < d_2 \\ \vdots & \vdots & \ddots & \vdots \\ r_1 < d_k & r_2 < d_k & \cdots & r_m < d_k \end{pmatrix}$$

We then apply the indexed function to the resulting matrix, which converts it into a matrix of pairs. Each pair consists of the index of the element in the matrix as the first component and the value of the element as the second component.

Next, we define an associative function:

$$f((i_1, b_1), (i_2, b_2)) = \begin{cases} (i_1, b_1) & \text{if } b_1 \text{ is true} \\ (i_2, b_2) & \text{if } b_1 \text{ is false} \end{cases}$$

By using this function as an input for the fold1 function, we obtain a vector of size m containing the indices of the first element of each row in the matrix that evaluates to true. These indices represent the elements of the archive that need to be selected. Finally, we can use the map function to select these elements in parallel.

A.4 Efficient parallel implementation of the Gaussian sampling

In this section, we will explore the implementation of the Gaussian sampling algorithm using Accelerate. This algorithm is used in the implementation of the Ant Colony Optimization algorithm, which is described in Section **??**.

For this algorithm, we assume that we have a list of k positions and a list of m selected positions, each of which can be seen as a vector of n elements. To perform the Gaussian sampling around the selected positions, we proceed similarly to the roulette wheel selection algorithm. First, we replicate the selected positions $r = (r_1, \ldots, r_m)$ k times, and the vector all positions $p = (p_1, \ldots, p_k)$ m times. Then, we apply the zipWith function, using the difference function as the first argument and the two matrices as the second and third arguments, respectively.

$$\begin{pmatrix} p_1 & p_2 & \cdots & p_k \\ p_1 & p_2 & \cdots & p_k \\ \vdots & \vdots & \ddots & \vdots \\ p_1 & p_2 & \cdots & p_k \end{pmatrix} \times \begin{pmatrix} r_1 & r_1 & \cdots & r_1 \\ r_2 & r_2 & \cdots & r_2 \\ \vdots & \vdots & \ddots & \vdots \\ r_m & r_m & \cdots & r_m \end{pmatrix} \rightarrow \begin{pmatrix} p_1 - r_1 & p_2 - r_1 & \cdots & p_k - r_1 \\ p_1 - r_2 & p_2 - r_2 & \cdots & p_k - r_2 \\ \vdots & \vdots & \ddots & \vdots \\ p_1 - r_m & p_2 - r_m & \cdots & p_k - r_m \end{pmatrix}$$

where $_ - _$ is sugar for the difference function. This operation is part of the Position type class, and it is implemented as a vector difference, i.e.

$$r_i - p_j = (r_{i1} - p_{j1}, \ldots, r_{in} - p_{jn}).$$

Now, we can use another operation of the Position type class, called pmap, which applies a function over all slots of a position. In this case, we map the function pmap (abs) over the resulting matrix. This gives us a matrix where each element is the absolute value of the difference between the corresponding elements of the two input matrices.

$$\begin{pmatrix} |p_1 - r_1| & |p_2 - r_1| & \cdots & |p_k - r_1| \\ |p_1 - r_2| & |p_2 - r_2| & \cdots & |p_k - r_2| \\ \vdots & \vdots & \ddots & \vdots \\ |p_1 - r_m| & |p_2 - r_m| & \cdots & |p_k - r_m| \end{pmatrix}$$

where in this case, $|_|$ means pointwise absolute value, i.e. $|p_i| = (|p_{i1}|, \ldots, |p_{in}|)$.

We can use the same trick as before, i.e. perform a fold1 operation on the resulting matrix, using the sum function as the first argument. This gives us a vector of size m containing the sum of the differences of each row of the matrix. over the vector of sums. This gives us a vector of size m

$$s = (s_1, \ldots, s_m), \text{ where } s_i = \sum_{j=1}^{k} |p_j - r_i|$$

where the sum is a vector sum. Finally, we can map the function `pmap (/k - 1)` (we do not include the case where $p_i = r_i$) over the vector of sums, s. This gives us a vector of size m where each element is a position that contains the standard deviations of the reference position with the same index. If we call this vector d, then we have that $d_{ij} = \sigma_j$ of the reference position r_i.

At this point, it is easy to perform the Gaussian sampling around the reference positions, since we have computed standard deviations for each dimension, and we use the reference position as the mean.

Bibliography

[1] D Ackley. *A connectionist machine for genetic hillclimbing Kluwer Acad*. 1987 (cited on page 49).

[2] 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, pages 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. URL: https://doi.org/10.1145/1465482.1465560 (cited on page 6).

[3] Mauro Birattari et al. "A Racing Algorithm for Configuring Metaheuristics." In: *Gecco*. Volume 2. 2002. Citeseer. 2002 (cited on page 18).

[4] Guy E Blelloch. *NESL: a nested data parallel language*. Carnegie Mellon Univ., 1992 (cited on page 67).

[5] G. E. P. Box and Mervin E. Muller. "A Note on the Generation of Random Normal Deviates". In: *The Annals of Mathematical Statistics* 29.2 (1958), pages 610–611. DOI: 10.1214/aoms/1177706645. URL: https://doi.org/10.1214/aoms/1177706645 (cited on page 35).

[6] Iván Ca, Fernando Rubio, et al. "Dealing with Swarm Intelligence on GPUs". In: *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*. IEEE. 2019, pages 2192–2197 (cited on page 58).

[7] Manuel M T Chakravarty et al. "Accelerating Haskell array codes with multicore GPUs". In: *DAMP '11: The 6th workshop on Declarative Aspects of Multicore Programming*. ACM, Jan. 2011 (cited on page 13).

[8] Manuel MT Chakravarty et al. "Nepal—nested data parallelism in Haskell". In: *Euro-Par 2001 Parallel Processing: 7th International Euro-Par Conference Manchester, UK, August 28–31, 2001 Proceedings 7*. Springer. 2001, pages 524–534 (cited on page 67).

[9] Robert Clifton-Everest et al. "Embedding Foreign Code". In: *PADL '14: The 16th International Symposium on Practical Aspects of Declarative Languages*. LNCS. Springer-Verlag, Jan. 2014.

[10] Robert Clifton-Everest et al. "Streaming Irregular Arrays". In: *Haskell '17: The 10th ACM SIGPLAN Symposium on Haskell*. ACM, Sept. 2017, pages 174–185.

[11] Murray Cole. “Algorithmic Skeletons”. In: *Research Directions in Parallel Functional Programming*. Edited by Kevin Hammond and Greg Michaelson. London: Springer London, 1999, pages 289–303. ISBN: 978-1-4471-0841-2. DOI: 10.1007/978-1-4471-0841-2_13. URL: https://doi.org/10.1007/978-1-4471-0841-2_13 (cited on page 13).

[12] Steven P Coy et al. “Using experimental design to find effective parameter settings for heuristics”. In: *Journal of Heuristics* 7 (2001), pages 77–97 (cited on page 18).

[13] J.-L. Deneubourg et al. “The self-organizing exploratory pattern of the argentine ant”. In: *Journal of Insect Behavior* 3.2 (Mar. 1990), pages 159–168. ISSN: 1572-8889. DOI: 10.1007/BF01417909. URL: https://doi.org/10.1007/BF01417909 (cited on page 19).

[14] M. DORIGO. “Optimization, Learning and Natural Algorithms”. In: *Ph.D. Thesis, Politecnico di Milano* (1992). URL: https://cir.nii.ac.jp/crid/1573950400977139328 (cited on page 18).

[15] Russ C Eberhart and Yuhui Shi. “Comparing inertia weights and constriction factors in particle swarm optimization”. In: *Proceedings of the 2000 congress on evolutionary computation. CEC00 (Cat. No. 00TH8512)*. Volume 1. IEEE. 2000, pages 84–88 (cited on page 24).

[16] Michael Guntsch and Martin Middendorf. “A Population Based Approach for ACO”. In: *Applications of Evolutionary Computing*. Edited by Stefano Cagnoni et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pages 72–81. ISBN: 978-3-540-46004-6.

[17] Mark P Jones. “Type classes with functional dependencies”. In: *Programming Languages and Systems: 9th European Symposium on Programming, ESOP 2000 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000 Berlin, Germany, March 25–April 2, 2000 Proceedings*. Springer. 2000, pages 230–244 (cited on page 36).

[18] J Kennedy, RC Eberhart, and Y Shi. “Swarm Intelligence Morgan Kaufmann Publishers”. In: *San Francisco* (2001).

[19] James Kennedy and Russell Eberhart. “Particle swarm optimization”. In: *Proceedings of ICNN’95-international conference on neural networks*. Volume 4. IEEE. 1995, pages 1942–1948 (cited on page 23).

[20] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. “Parallel Functional Programming in Eden”. In: *Journal of Functional Programming* 15.3 (2005), pages 431–475 (cited on page 12).

[21] Geoffrey Mainland. “Why It’s Nice to be Quoted: Quasiquoting for Haskell”. In: Sept. 2007, pages 73–82. DOI: 10.1145/1291201.1291211 (cited on page 14).

[22] Trevor L. McDonell. *GitHub - tmcdonell/mwc-random-accelerate: Generate Accelerate arrays filled with high quality pseudorandom numbers — github.com*. `https://github.com/tmcdonell/mwc-random-accelerate`. 2017.

[23] Trevor L. McDonell. *GitHub - tmcdonell/sfc-random-accelerate — github.com*. `https://github.com/tmcdonell/sfc-random-accelerate`. 2020.

[24] Trevor L. McDonell et al. "Optimising Purely Functional GPU Programs". In: *ICFP '13: The 18th ACM SIGPLAN International Conference on Functional Programming*. ACM, Sept. 2013.

[25] Trevor L. McDonell et al. "Type-safe Runtime Code Generation: Accelerate to LLVM". In: *Haskell '15: The 8th ACM SIGPLAN Symposium on Haskell*. ACM, Sept. 2015, pages 201–212.

[26] NVIDIA. *CUDA LLVM Compiler*. URL: `https://developer.nvidia.com/cuda-llvm-compiler` (cited on page 14).

[27] Varun Kumar Ojha, Ajith Abraham, and Václav Snášel. "ACO for continuous function optimization: A performance analysis". In: *2014 14th International Conference on Intelligent Systems Design and Applications*. 2014, pages 145–150. DOI: `10.1109/ISDA.2014.7066253` (cited on pages 20, 21).

[28] Magnus Erik Hvass Pedersen. "Tuning & simplifying heuristical optimization". PhD thesis. University of Southampton, 2010 (cited on page 24).

[29] Magnus Erik Hvass Pedersen and Andrew J Chipperfield. "Simplifying particle swarm optimization". In: *Applied Soft Computing* 10.2 (2010), pages 618–628.

[30] Simon Peyton Jones et al. "Simple unification-based type inference for GADTs". In: *ACM SIGPLAN Notices* 41.9 (2006), pages 50–61 (cited on page 36).

[31] Yuhui Shi and Russell Eberhart. "A modified particle swarm optimizer". In: *1998 IEEE international conference on evolutionary computation proceedings. IEEE world congress on computational intelligence (Cat. No. 98TH8360)*. IEEE. 1998, pages 69–73 (cited on page 23).

[32] Yuhui Shi and Russell C. Eberhart. "Parameter selection in particle swarm optimization". In: *Evolutionary Programming VII*. Edited by V. W. Porto et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pages 591–600. ISBN: 978-3-540-68515-9 (cited on page 24).

[33] Daniel L. Slotnick, W. Carl Borck, and Robert C. McReynolds. "The SOLOMON Computer". In: *Proceedings of the December 4-6, 1962, Fall Joint Computer Conference*. AFIPS '62 (Fall). Philadelphia, Pennsylvania: Association for Computing Machinery, 1962, pages 97–107. ISBN: 9781450378796. DOI: 10.1145/1461518.1461528. URL: https://doi.org/10.1145/1461518.1461528 (cited on page 6).

[34] Krzysztof Socha and Christian Blum. "An ant colony optimization algorithm for continuous optimization: application to feed-forward neural network training". In: *Neural Computing and Applications* 16.3 (May 2007), pages 235–247. ISSN: 1433-3058. DOI: 10.1007/s00521-007-0084-z. URL: https://doi.org/10.1007/s00521-007-0084-z (cited on pages 18, 57, 62).

[35] Krzysztof Socha and Marco Dorigo. "Ant colony optimization for continuous domains". In: *European Journal of Operational Research* 185.3 (2008), pages 1155–1173. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2006.06.046. URL: https://www.sciencedirect.com/science/article/pii/S0377221706006333 (cited on pages 18, 19, 57, 60).

[36] Herbert Spencer. "Principles of biology, vol. 1 London". In: *UK: Williams and Norgate [Google Scholar]* (1864) (cited on page 26).

[37] R Storn et al. "Proceedings of North American Fuzzy Information Processing". In: (1996).

[38] Rainer Storn and Kenneth Price. "Differential evolution-a simple and efficient heuristic for global optimization over continuous spaces". In: *Journal of global optimization* 11.4 (1997), page 341 (cited on pages 25, 27).

[39] Bo Joel Svensson et al. "Converting Data-Parallelism to Task-Parallelism by Rewrites". In: *FHPC '15: The 4th ACM SIGPLAN Workshop on Functional High-Performance Computing*. ACM, Sept. 2015, pages 12–22.

[40] Mojtaba Taherkhani and Reza Safabakhsh. "A novel stability-based adaptive inertia weight for particle swarm optimization". In: *Applied Soft Computing* 38 (2016), pages 281–295 (cited on page 24).

[41] Xin Yao, Yong Liu, and Guangming Lin. "Evolutionary programming made faster". In: *IEEE Transactions on Evolutionary computation* 3.2 (1999), pages 82–102 (cited on page 48).