Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Fakultät für Informatik
Facoltà di Scienze e Tecnologie informatiche
Faculty of Computer Science

Bachelor in Computer Science

Bachelor Thesis

Distributed Particle Swarm Optimization: Design of Synchronous and Asynchronous Algorithms for Optimization Problems at the Edge

Candidate: Riccardo Busetti

Supervisor: Prof. Claus Pahl

July 22, 2022

Abstract

The growth and wide adoption of edge computing have introduced several issues such as load balancing, resource provisioning, and workload placement which are all developing as optimization problems. An algorithm, in particular, was successful in solving some of these optimization problems, namely PSO. Particle Swarm Optimization (PSO) is a nature-inspired stochastic optimization algorithm that emulates the behavior of bird flocks, whose objective is to iteratively improve the solution of a problem over a given objective. The execution of PSO at the edge would result in the offload of resource-intensive computational tasks from the cloud to the edge, leading to more efficient use of existing resources. However, it would also introduce several performance and fault tolerance challenges due to the resource-constrained environment with a high probability of faults. Given the benefits and the challenges above, the need for a new PSO algorithm to be efficiently executed in an edge network arises. This thesis introduces multiple distributed synchronous and asynchronous variants of the PSO algorithm implemented with the Kotlin programming language, built on the Apache Spark distributed computing framework and Kubernetes container orchestration platform. These algorithm variants aim to address the performance and fault tolerance problems introduced by the execution in an edge network. Moreover, they want to provide a greater level of scalability. By designing a PSO algorithm that distributes the load across multiple executor nodes, effectively realizing both coarse- and fine-grained parallelism, we can significantly increase the algorithms' performance, fault tolerance, and scalability. The experimental results show the benefits of the proposed distributed variants of the PSO algorithm versus a baseline, the traditional PSO, testing different values of specific configuration parameters of the algorithms.

Sommario

La crescita e l'ampia adozione dell'edge computing hanno introdotto diversi problemi come il bilanciamento del carico, il provisioning delle risorse e la distribuzione del carico di lavoro, che si sviluppano tutti come problemi di ottimizzazione. Un algoritmo, in particolare, ha avuto successo nel risolvere alcuni di questi problemi, cioè PSO. Particle Swarm Optimization (PSO) è un algoritmo di ottimizzazione stocastica ispirato alla natura, che emula il comportamento degli stormi di uccelli, il cui obiettivo è quello di migliorare iterativamente la soluzione ad un problema su un dato obiettivo. L'esecuzione di PSO sull'edge comporterebbe l'offload di compiti computazionali ad alta intensità dal cloud all'edge, portando ad un uso più efficiente delle risorse esistenti. Tuttavia, introdurrebbe anche diverse sfide relative alle prestazioni e alla tolleranza ai guasti, a causa dell'ambiente con risorse limitate e ad alta probabilità di guasti. Dati i benefici e le sfide di cui sopra, nasce la necessità di un nuovo algoritmo PSO in grado di essere eseguito in modo efficiente in una rete edge. In questa tesi, introduciamo più varianti distribuite, sincrone e asincrone, dell'algoritmo PSO implementato con il linguaggio di programmazione Kotlin e costruito sopra il framework di calcolo distribuito Apache Spark e la piattaforma di orchestrazione di container Kubernetes. Queste varianti dell'algoritmo mirano ad affrontare i problemi di performance e tolleranza ai guasti introdotti dall'esecuzione in una rete edge, inoltre vogliono fornire un maggiore livello di scalabilità. Progettando un algoritmo PSO che distribuisce il carico su più nodi esecutori, realizzando efficacemente sia il parallelismo a grana grossa che a grana fine, possiamo ottenere un aumento significativo delle prestazioni, oltre ad una migliore tolleranza ai guasti e maggiore scalabilità. I risultati sperimentali mostrano i benefici delle varianti distribuite proposte rispetto a una linea di base, il PSO tradizionale, testando parametri di configurazione specifici degli algoritmi con diversi valori.

Kurzfassung

Das Wachstum und die breite Einführung von Edge Computing haben verschiedene Probleme wie Lastausgleich, Ressourcenbereitstellung und Arbeitslastplatzierung mit sich gebracht, die sich alle als Optimierungsprobleme entwickeln. Ein Algorithmus war bei der Lösung einiger dieser Optimierungsprobleme besonders erfolgreich, nämlich PSO. Particle Swarm Optimization (PSO) ist ein von der Natur inspirierter stochastischer Optimierungsalgorithmus, der das Verhalten von Vogelschwärmen nachahmt und dessen Ziel es ist, die Lösung eines Problems iterativ über ein vorgegebenes Ziel zu verbessern. Die Ausführung von PSO am Edge würde dazu führen, dass ressourcenintensive Rechenaufgaben von der Cloud zum Edge verlagert werden, was zu einer effizienteren Nutzung der vorhandenen Ressourcen führt. Aufgrund der ressourcenbeschränkten Umgebung mit einer hohen Fehlerwahrscheinlichkeit würde dies jedoch auch einige Herausforderungen in Bezug auf Leistung und Fehlertoleranz mit sich bringen. In Anbetracht der oben genannten Vorteile und Herausforderungen ergibt sich der Bedarf an einem neuen PSO-Algorithmus, der in einem Edge-Netzwerk effizient ausgeführt werden kann. In dieser Arbeit stellen wir mehrere verteilte synchrone und asynchrone Varianten des PSO-Algorithmus vor, die mit der Programmiersprache Kotlin implementiert wurden und auf dem Apache Spark-Framework für verteiltes Rechnen und der Kubernetes-Container-Orchestrierungsplattform aufbauen. Diese Varianten des Algorithmus zielen darauf ab, die Leistungs- und Fehlertoleranzprobleme zu lösen, die durch die Ausführung in einem Edge-Netzwerk entstehen, und sollen darüber hinaus ein höheres Maß an Skalierbarkeit bieten. Durch die Entwicklung eines PSO-Algorithmus, der die Last auf mehrere Ausführungsknoten verteilt und sowohl grob- als auch feinkörnige Parallelität effektiv realisiert, können wir eine erhebliche Leistungssteigerung, aber auch mehr Fehlertoleranz und Skalierbarkeit erreichen. Die experimentellen Ergebnisse zeigen die Vorteile der vorgeschlagenen verteilten Varianten des PSO-Algorithmus im Vergleich zu einer Grundlinie, dem traditionellen PSO, wobei verschiedene Werte spezifischer Konfigurationsparameter der Algorithmen getestet werden.

Acknowledgements

I would like to express my gratitude to my primary supervisor, Prof. Claus Pahl, who guided me throughout this project and gave essential feedback during the writing of this thesis. I want also to thank Dr. Nabil El Ioni for his dedication in helping me overcome technical and organizational issues that arose during the research. Thanks, for giving me the chance to work on an interesting project in the field of distributed systems and optimization algorithms.

I wish to extend my special thanks to my family that consistently supported me during this chapter of my life and played a critical role in my personal growth. I wanted to thank my parents that were there to help me whenever I needed them but most importantly provided me with the determination and ambition that drive me daily. A special thanks goes to my grandparents with whom I spent most of my childhood and that raised me with strong values and education. I want to dedicate this thesis to them as a gift for all of the time that they devoted to me in order to form a professional but most importantly a human being.

In conclusion, I would like to thank my girlfriend, friends, ex-colleagues, and all the people I met along this journey that contributed to making it one of the most important and incisive periods of my life.

Contents

List of Figures

Chapter 1

Introduction

This chapter introduces the main objectives of the thesis and relevant background information.

1.1 Context

Cloud computing is unequivocally the backbone of a large fraction of existing systems. It provides access to computational resources and data storage to many mobile, embedded, and Internet-of-Things (IoT) devices [1]. The current architecture for cloud computing revolves around a set of powerful computing and storage resources housed in data centers managed by public or private organizations. The emergence of SaaS has considerably increased the growth of cloud computing, which offers customers the ability to pay a fee for a service that conceals the complexity derived from the management of cloud infrastructure. For example, Google Cloud Platform (GCP), Amazon Web Services (AWS), and Microsoft Azure are widespread Cloud Service Providers (CSP).

As the number of IoT and similar devices grows [2], the demand for cloud services will increase. This increase in demand will introduce several scalability problems such as bandwidth limitations, load balancing issues, and geographic partitioning of data. Altogether this results in a drop in Quality of Service (QoS) that affects all the applications running on the cloud.

Several solutions have been proposed to tackle the issue of high demand. However, one, in particular, is gaining much attention. Researchers have introduced a new paradigm called edge computing, aiming at reducing the strain on cloud resources by pushing the compute and storage resources as close as possible to the edge [1]. The edge comprises several heterogeneous components, like raspberry pis, gateways, micro-servers, and personal computers. Each component is classified as an edge node in the network and can run applications close to the data source. This proximity to the source results in a decrease in latency time, congestion, and, most notably, an improvement in the Quality of Experience (QoE) [3].

Unlike cloud data centers, edge devices are geographically distributed, constrained by available resources, and highly dynamic. These characteristics carry jointly several issues. Some of these issues are developing as optimization problems [4] involving load balancing, resource provisioning, and workload placement. The workload placement problem, in particular, is formulated with the question “how to efficiently deploy services on available edge nodes” [5] and is an excellent example of an optimization problem in the domain of edge computing.

As of now, several attempts have been made at optimization problems involving edge computing, and the usage of meta-heuristics nature-inspired methods such as Particle Swarm Optimization (PSO) showed promising results in work [6]. The problem with existing PSO-based solutions is that they are designed to run on single machines, which is problematic on edge devices due to their limited resources and reliability. These limitations are especially evident when dealing with significant optimization problems executed in an edge network. Hence, if we want to improve the performance but also scalability and fault tolerance of the PSO algorithm at the edge, there is the need to horizontally scale the computations to combine resources of multiple edge nodes into one transparent distributed system.

1.2 Motivation

Particle Swarm Optimization (PSO) is a nature-inspired stochastic optimization algorithm emulating the intelligent collective behavior of animals, which involves cooperation to find the solution to an optimization problem. Since its presentation in 1995, it has experienced a multitude of enhancements due to the broad attention received in recent years [7]. These enhancements gave birth to variants of the PSO algorithm, such as Discrete PSO, Binary PSO, and Multi-Objective PSO, which have shown the potential and flexibility of the algorithm.

The continuous improvements of the algorithm increased its computational complexity. However, the most significant factor that impacts the complexity, measured by elapsed time, remains the problem encoding. A problem encoding refers to representing an optimization problem as a PSO-solvable problem, which includes the design of a fitness function and the definition of sets of constraints representing the solution space for the problem. Encodings are so impactful on performance that there are many benchmark encodings such as De Jong’s function 1, Axis Parallel Hyper-Ellipsoid, and Schwefel’s Problem in the literature [8]. These encodings are complex by design because their goal is to evaluate different evolutionary algorithms on performance and accuracy. In addition, they show how quickly the complexity of the PSO algorithm can increase if the encoding involves a complex fitness function, a high number of dimensions, and/or several expensive computations to be performed on the particles.

In the context of edge computing, a meaningful example of PSO application is the work done in [6], which uses a Binary Multi-Objective PSO (BMOPSO) algorithm with a matrix-based encoding to solve the workload placement problem. In the encoding, each particle’s position and velocity are represented as matrices that encode the placement of a specific module on a fog node. This optimization problem would benefit from being solved directly in an edge network. However, in the encoding, several operations on matrices are performed, which lead to an average asymptotic complexity of $O(n^2)$. The quadratic complexity overgrows and becomes a bottleneck when the problem dimensionality increases slightly. In addition, the encoding is open to several refinements, which will most likely introduce an additional increase in complexity (e.g., the addition of more constraints, usage of continuous values, dimensionality increase).

In general, if we decide to leverage edge computing to offload the work from the cloud to the edge, we also expect to be able to solve optimization problems with the PSO algorithm directly in an edge network, with all the benefits and challenges involved. The main challenges of this approach are:

- **Lack of computational resources**: unlike the cloud, in an edge computing environment, the resources are costly and limited. The scarcity of resources becomes a significant bottleneck, especially when running large optimization problems on single nodes. Therefore there is the need to optimize the available resources by horizontally scaling computations to as many nodes as possible, leveraging their combined computational power.
- **Higher probability of faults**: in the distributed and dynamic edge computing environment, edge nodes are more subject to runtime failures [9]. These failures can be problematic if happening in the middle of a long-running computation because they will effectively cancel the current progress and all of its results. In addition, running the algorithm on a single node introduces a single point of failure, which is always to avoid when fault tolerance is a critical requirement. Consequently, there is the need to leverage multiple nodes to increase the degree of fault tolerance in the algorithm.

The growing complexity of optimization problems, the lack of computational resources, and the higher probability of faults in an edge network altogether create the need to design a variant of the PSO algorithm that can combine the computational power of multiple edge nodes and offer a resilient mechanism in the case of nodes failures.

1.3 Objective

The current research gap and source of this thesis proposal are related to the problem of designing and implementing distributed synchronous and asynchronous variants of the PSO algorithm to reduce the execution time but also enhance fault tolerance and improve scalability.

The work on parallel PSO algorithms has already been done in the literature to alleviate the high computational cost associated with the algorithm. Most of these parallel algorithms are not distributed, meaning they consist of multiple processors that communicate with each other using shared memory instead of numerous computers communicating over the network [10]. Moreover, a good number of parallel implementations of the PSO algorithm present in the literature are based on a synchronous implementation (e.g., [11]) where all particles within an iteration are evaluated before the next iteration is started. This implementation can easily lead to poor parallel performance in multiple cases, particularly when there is an imbalance between the cluster's resources. To solve this issue, the work in [10] proposed a distributed asynchronous PSO algorithm based on Message Passing Interface (MPI) but did not take into account fault tolerance and scalability, which are critical requirements in an edge computing environment.

An attractive property of the PSO algorithm is that it has a strong distributed ability because the algorithm is essentially the evolutionary swarm algorithm [7]. This property ideally suits a coarse-grained parallel implementation on a parallel or distributed computing network. However, the distribution of the algorithm on multiple nodes over the network introduces several new challenges discussed in the solution chapter.

The objective of this thesis is to understand whether distributed synchronous and asynchronous variants of the PSO algorithm can be designed and implemented with the following goals:

- **Better performance**: the algorithm should have better performance, measured as elapsed time, by leveraging the multiple computing nodes of the cluster. This reduction in elapsed time should be inversely proportional to the number of nodes in the cluster. The more nodes in the cluster, the faster the algorithm should work.
- **More fault tolerance**: the algorithm should tolerate node failures and act upon them to preserve the current state and progress. The definition of tolerating node failures refers to the ability of the system to detect when a failure has occurred and to react by trying to carry on the computation, with the only possible observed issue being an additional delay.
- **Flexible scalability**: the algorithm should be able to scale horizontally, coping with the addition or removal of nodes from the cluster. The elastic scaling property is helpful in an edge computing environment where there could be a need to optimize the available resources for specific optimization problems.

The core research objective is to evaluate and assess whether distributed synchronous and asynchronous implementations of the PSO algorithm bring the benefits mentioned above over the traditional PSO algorithm, especially in an edge computing environment.

1.4 Approach

The distributed PSO algorithm will be proposed in multiple variants, namely Spark Distributed Synchronous PSO with Local Update (SDSPSO with LU), Spark Distributed Synchronous PSO with Distributed Update (SDSPSO with DU), and Spark Distributed Asynchronous PSO (SDAPSO). These variants differ in their inner workings and are proposed to establish multiple approaches to the same problem instead of a one-size-fits-all solution. The algorithms are developed with the Kotlin programming language, the Apache Spark distributed computing framework, and the Kubernetes container orchestration platform to create performant, fault-tolerant, and scalable variants of the PSO algorithm. Thanks to Apache Spark's distributed memory abstraction, namely Resilient Distributed Dataset (RDD), the algorithms will benefit from in-memory, fault-tolerant, parallel and efficient transformations of the particles. Together with the container-based orchestration offered by Kubernetes and the Kotlin programming language, such characteristics will enable the algorithm to reach the forenamed goals with a simple, portable, and extensible architecture that can be easily maintained. In the end, the distributed algorithms will be evaluated by considering the algorithms' performance in terms of elapsed time. Moreover, fault tolerance and scalability will be considered and assessed. In conclusion, the focus of this research is to design, implement and evaluate distributed variants of the PSO algorithm that could be executed on a cluster of edge nodes, considering all the strengths and weaknesses typical of edge computing environments.

1.5 Structure of the Thesis

The thesis is organized as follows:

- **Chapter 1** presents the context, the motivation, the objective of the thesis, and the approach to achieve it.
- **Chapter 2** discusses the necessary background on swarm optimization, edge computing, Internet-of-Things, distributed systems, Apache Spark, and Kotlin coroutines.
- **Chapter 3** discusses the problem statement.
- **Chapter 4** presents the proposed solution to the problem.
- **Chapter 5** discusses the results of the proposed solution with performance evaluations and platforms assessment.
- **Chapter 6** summarizes the current research project, together with a perspective for future work, and concludes the present dissertation.
- **Appendix A** presents the pseudocode of the algorithms, their simplified implementation in Kotlin, and a sample configuration file for the Spark Operator.

Chapter 2

Background

This chapter explains acronyms and concepts for a complete understanding of this research.

2.1 Swarm Optimization

2.1.1 General Overview

Swarm Intelligence (SI) attempts to design multi-agent systems that take inspiration from the collective behavior of social animals/insects. The core idea of these systems is that they can accomplish complex tasks with cooperation, even though the single individuals belonging to the group are relatively non-sophisticated [12]. SI algorithms are decentralized and self-organized, which turns out to be especially good for problems with dynamic properties, incomplete information, and limited computational capabilities [12].

The landscape of SI has been expanding rapidly due to the performance and simplicity of the algorithms. These algorithms rely on different intelligent behaviors inspired by birds, ants, bats, and pigeons. Some examples of existing SI-based algorithms that are considered ready to use include Ant Colony Optimization (ACO), Artificial Bee Colony (ABC), Bacterial Foraging Optimization (BFOA), and Particle Swarm Optimization (PSO) [12]. The readiness means that these algorithms are frequently used and standardized by the scientific community due to their wide adoption across many problems.

2.1.2 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a nature-inspired stochastic optimization algorithm proposed in 1995 by Kennedy and Eberhart in [13], which is motivated by the intelligent collective behavior of animals like flocks of birds. The authors of the algorithm followed the five basic principles of swarm intelligence during the design, which include:

1. **Proximity principle**: the population should be able to carry out simple space and time computations.

2. **Quality principle**: the population should be able to react to quality factors in the environment.

3. **Diverse response principle**: the population should not commit its activities along excessively narrow channels.

4. **Stability principle**: the population should not change its mode of behavior every time the environment changes.

5. **Adaptability principle**: the population must be able to change behavior mode when it is worth the computational price.

PSO has received broad attention in recent years mainly due to its advantages which become essential when applied in specific domains. These benefits include the excellent robustness and adaptability, the strong distributed ability due to the independence between particles, the quick convergence to the optimization value, and the easiness of hybridization with other algorithms [7].

The PSO algorithm revolves around a population composed of particles. A particle is an abstraction created by the designers to represent an entity that can move with a given velocity and acceleration. It represents a point in the cartesian coordinate system characterized by a randomly assigned position and velocity. Each particle in the swarm keeps track of its best personal position and best global position, the latter representing the best position found by any particle belonging to the swarm. The "best" position can be understood together with the fitness function we want to optimize. The term "best" refers to the minimum or maximum value found during the evaluation of the particle's position with the given fitness function. PSO advances in iterations, during which the particles are evaluated against their fitness function, and their best personal and global positions are consequently updated. Before the end of each iteration, the particles' velocities and positions are updated with specific formulas. The stopping condition for the algorithm is generally defined as the number of iterations. However, it may also use other conditions (e.g., the specific value found, elapsed time).

The flow of the algorithm can be easily modeled with a flowchart, as shown in Figure 2.1.

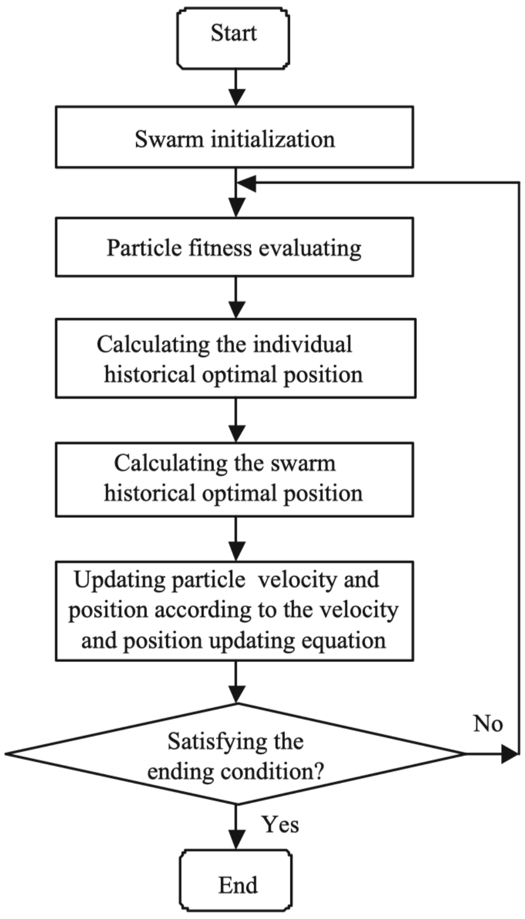

Start
Swarm initialization
Particle fitness evaluating
Calculating the individual historical optimal position
Calculating the swarm historical optimal position
Updating particle velocity and position according to the velocity and position updating equation
Satisfying the ending condition?
No
Yes
End

Figure 2.1: Flowchart of the particle swarm optimization algorithm. [7]

The work in [7] provides a fairly complete mathematical presentation of the PSO algorithm in the continuous space coordinate system. It starts by assuming the swarm size to be N, each particle's vector to be in a D-dimensional space, and the fitness function to be f. Then, the algorithm is characterized by:

- The i-th particle position vector $X_i = (x_{i1}, x_{i2}, \cdots, x_{id}, \cdots, x_{iD})$
- The i-th particle velocity vector $V_i = (v_{i1}, v_{i2}, \cdots, v_{id}, \cdots, v_{iD})$
- The i-th particle personal best position vector $P_i = (p_{i1}, p_{i2}, \cdots, p_{id}, \cdots, p_{iD})$
- The best global position vector $P_g = (p_{g1}, p_{g2}, \cdots, p_{gd}, \cdots, p_{gD})$

Taking into consideration a minimization problem as an example, the best position vectors are updated using the following formula, where t is the iteration:

$$p^d_{i,t+1} = \begin{cases} x^d_{i,t+1}\text{ , if } f(X_{i,t+1}) < f(P_{i,t}) \\ p^d_{i,t}\text{ , otherwise} \end{cases} \tag{2.1}$$

The update formula for each particle's velocity and position is denoted as follows, where c_1, c_2 are acceleration coefficients expressing confidence and $rand$ is a randomly generated number that controls the stochastic influence of the cognitive and social components:

$$v_{i,t+1}^{d} = v_{i,t}^{d} + c_1 * rand * (p_{i,t}^{d} - x_{i,t}^{d}) + c_2 * rand * (p_{g,t}^{d} - x_{i,t}^{d}) \tag{2.2}$$

$$x_{i,t+1}^{d} = x_{i,t}^{d} + v_{i,t+1}^{d} \tag{2.3}$$

Figure 2.2 shows how the velocity and position update formulas work by emphasizing the different influence sources of a particle.

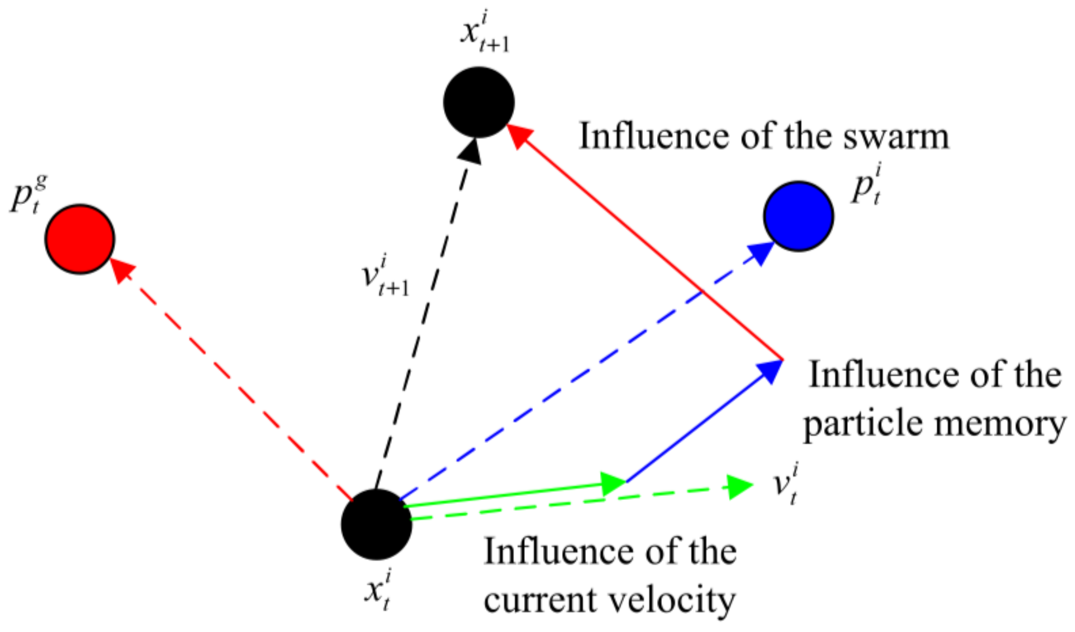


Figure 2.2: Iteration scheme of the particles. [7]

2.1.3 PSO Improvements

The PSO algorithm has undergone a multitude of enhancements in the literature. Some of these improvements involve the adoption of multi-sub-populations where the swarm is divided into several sub-swarms that share information with each other, the improvement of the selection strategy for particle learning objects, the modification of the particle's update formula, and the combination of PSO with other search techniques [7]. The previous improvements resulted in several variants of the PSO algorithm, such as the Discrete PSO, Dynamic PSO, and Multi-Objective PSO.

2.1.4 Synchronous vs. Asynchronous PSO

The PSO algorithm's effectiveness depends not only on introducing new parameters or on the focus on solving specific types of problems such as multi-objective optimization. Indeed, the particles' update sequence also affects the performance and accuracy of the algorithm because the new velocity and position depend on the order in which the best personal and global positions are evaluated [14]. The PSO algorithm has two primary variants that differ in the update sequence of the particles' velocities and positions. These variants are the Synchronous PSO (SPSO) and Asynchronous PSO (APSO).

The synchronous PSO is the standard variant of the algorithm and revolves around the concept of iterations. Each iteration is composed of four main steps: the first step is to evaluate the fitness function with each particle's position as input, the second step is the evaluation of the best personal position for every particle, the third step is the determination of the best global position among all the particles, and the fourth step is the update of each particle's velocity and position based on the best positions computed earlier. Once all the steps are concluded, a new iteration will start with each particle's newly updated positions and velocities.

The asynchronous PSO is slightly different from the synchronous variant. The main difference lies in the fact that the particles are updated based on the current state of the swarm in the asynchronous variant. Each particle's velocity and position are updated as soon as its fitness function is evaluated, thus considering the best global position found until that point. This mechanism creates total independence between particles that are moved with the information available at the evaluation time.

The main strength of SPSO is in exploiting the information, meaning that we always have to wait for each particle to conclude exploring, leading up to a possible better global position. On the other hand, the main strength of APSO is that particles are updated using partial information, leading to more substantial exploration.

2.2 Edge Computing

Where Cloud Computing (CC) is a computing paradigm that delivers services to the end-users with computing resources distributed in data centers collocated in several parts of the world, Edge Computing (EC) focuses on moving the computations, data, applications, and services away from centralized servers to the edge of a network. The proximity permits content providers and application developers to offer services closer to their users [15]. OpenEdge Computing defines edge computing as computation done at the network's edge through small data centers close to users [16].

The primary characteristics of edge computing are the low-latency, real-time access to network information, and high bandwidth. These qualities allow for several compelling applications and use cases, such as location services, augmented reality, video analytics, and data caching. For the reasons mentioned above, this new paradigm is becoming an exciting business sector for many vendors, third parties, and operators [15].

2.2.1 Where is the Edge?

The term edge is used in various papers about edge computing, cloudlets, fog computing, and mist computing, often creating confusion. The edge in the telecommunications industry usually refers to 4G/5G base stations, RANs (Radio Access Network), and ISP (Internet

Service Provider) access/edge networks [17].

In simpler terms, the edge is the immediate first hop from the IoT devices (not the IoT nodes themselves), such as the WiFi access points or gateways [17].

A graphical representation of the edge's location is visible in Figure 2.3.

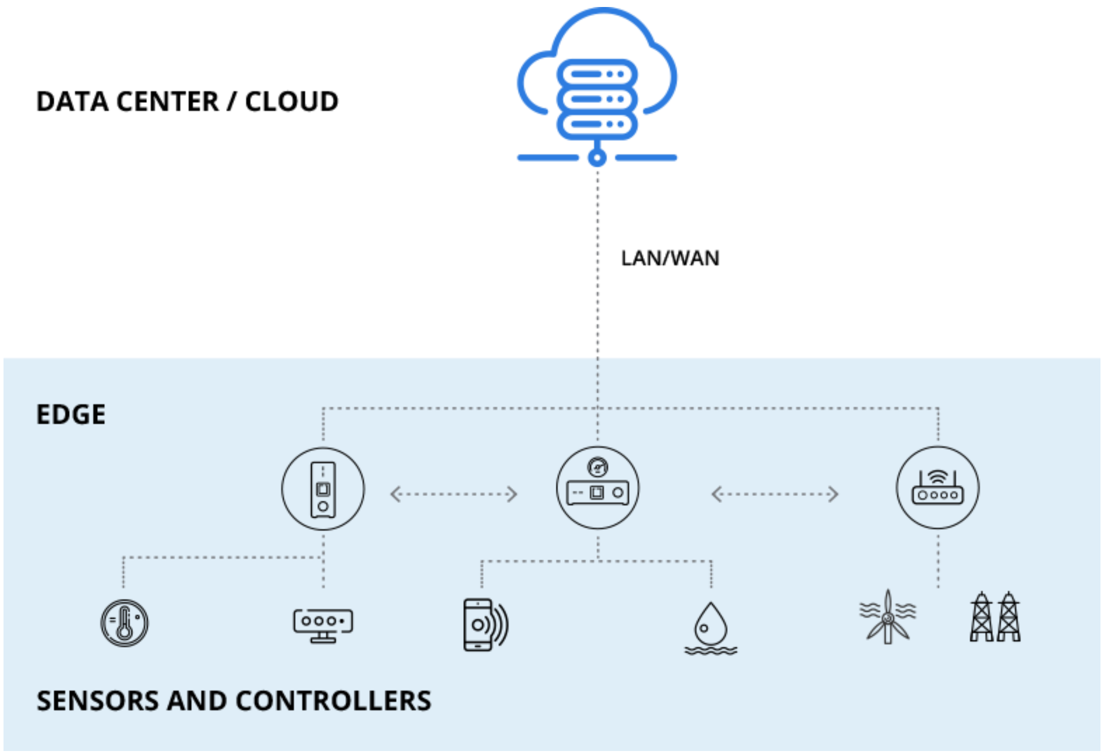


Figure 2.3: Representation of the edge that partitions cloud and IoT devices. [18]

2.2.2 Architecture

The architecture of edge computing is composed of a homogeneous set of edge nodes and devices able to perform many computing tasks (e.g., data processing, temporarily storing, devices management, workload offloading). The main objectives behind using edge nodes are to reduce network latency and traffic between end devices and the cloud. The reduction of network traffic is achieved by offloading as many tasks as possible from cloud data centers to edge nodes.

All of the edge nodes can interconnect and communicate locally, effectively creating an edge network that handles most of the tasks in a distributed manner. In addition, edge devices connect to cloud data centers by a core network to send preprocessed information that might need further processing by more powerful specialized systems [19].

A diagram of the edge computing architecture is available in Figure 2.4.

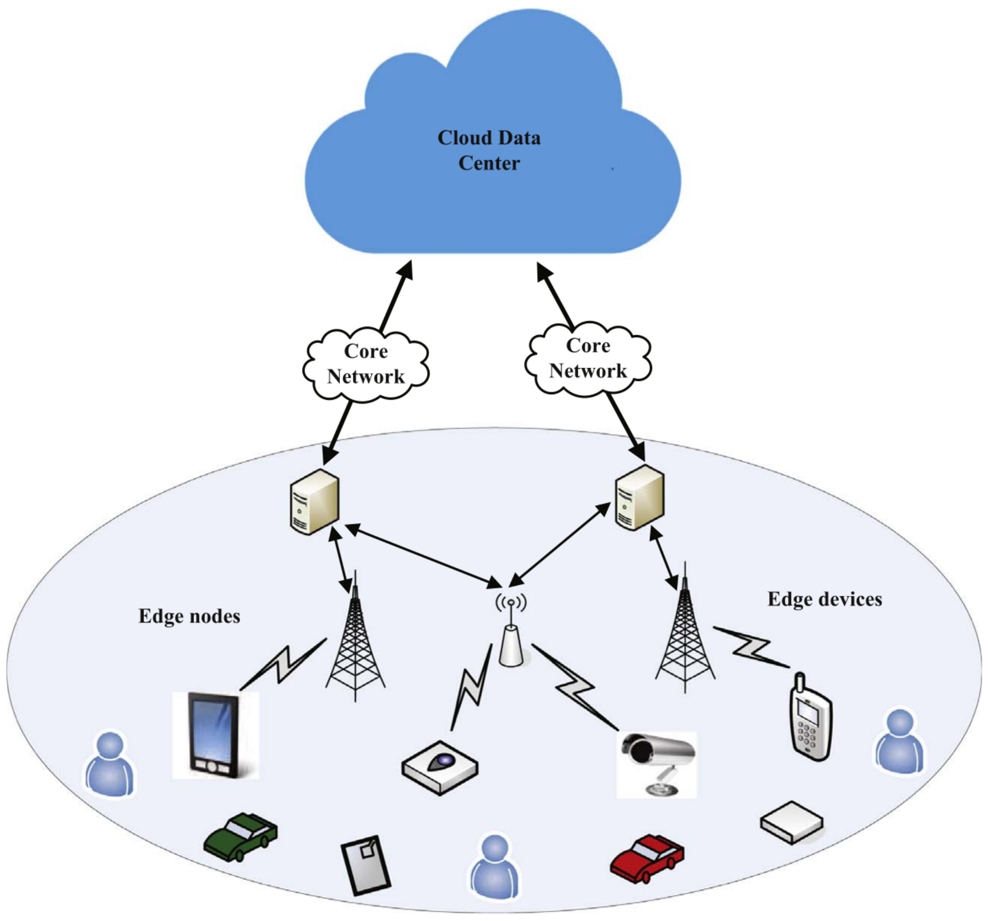


Figure 2.4: Architecture of edge computing. [19]

2.2.3 Characteristics

Edge and cloud computing share several similarities. However, as discussed in [15], edge computing has some characteristics that make it different. The following characteristics are unique to edge computing and are responsible for its widespread use:

- **Dense geographical distribution**: edge computing brings several services closer to the user by deploying numerous computing platforms in the edge networks.
- **Mobility support**: edge computing supports mobility with the Locator ID Separation Protocol (LISP) that is used to communicate directly with mobile devices. The LISP protocol is a network-layer-based protocol that separates IP addresses into two new numbering spaces: Endpoint Identifiers (EIDs) and Routing Locators (RLOCs) [20].
- **Location awareness**: edge computing is location-aware, which allows mobile users to access services from the edge server closest to their physical location.
- **Proximity**: edge computing offers computation resources and services in the proximity of the users in order to enhance their Quality of Experience (QoE).

- **Low latency**: edge computing provides computation resources and services closer to the users, which reduces the latency while accessing the services.
- **Context-awareness**: edge computing supports context-awareness to offer users services that are aware of their context. These services incorporate information about the user to become more valuable.
- **Heterogeneity**: edge computing is heterogeneous by design because it is composed of a wide variety of platforms, architectures, infrastructures, computing, and communication technologies. Even though this characteristic seems to be an advantage, heterogeneity is a significant issue that renders edge computing difficult to deploy successfully.

2.2.4 Related Computing Paradigms

In the latest years, several new computing paradigms have emerged, benefiting the current and future landscape of connected devices. All these paradigms have multiple things in common and are often confused. Hence there is a need to differentiate them to have a comprehensive global understanding of their properties.

The survey [17] contains a clear overview of several computing paradigms, some of which are briefly described in the following subsections and are shown in Figure 2.5.

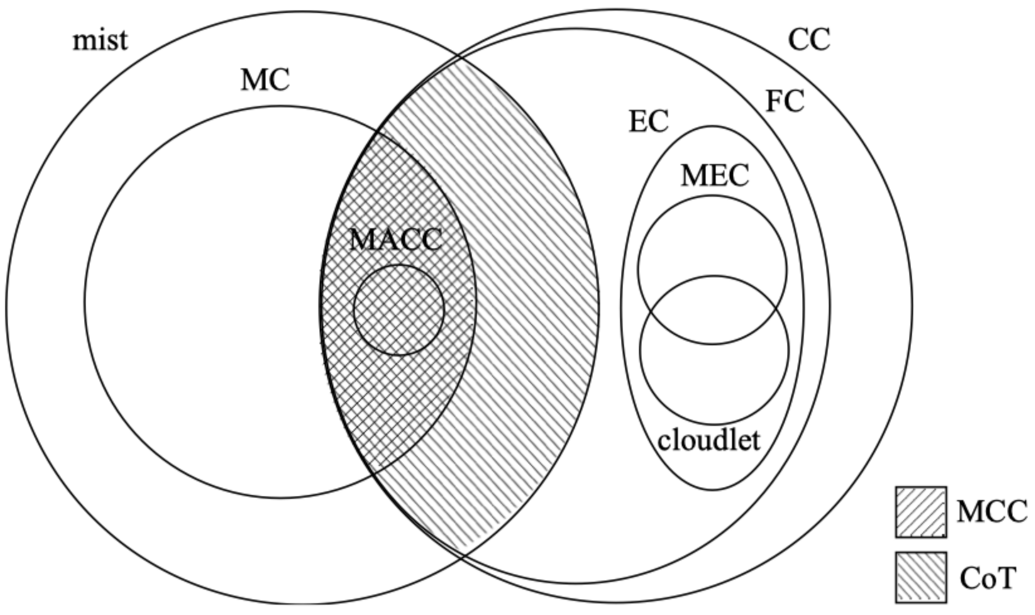


Figure 2.5: Classification of fog computing and its related computing paradigms. [17]

Cloud Computing

Cloud Computing has seen tremendous growth in the latest years in computing, storage, and networking infrastructure. The main benefits and reasons for the extensive adoption of the cloud are the performance and transparency offered to end-user. The National Institute of Standards and Technology (NIST) defines cloud computing as a model that promotes ubiquitous, on-demand network access to shared computing resources [21].

Cloud data centers are at the core of cloud computing. They are made up of large pools of highly accessible virtualized resources. The virtualized nature of the infrastructure enables a dynamic reconfiguration of the resources that allows for scalable workloads to be supported. This flexibility in the reconfiguration is why pay-as-you-go cloud services have emerged recently [22].

Fog Computing

Fog Computing (FC) enables computing, storage, and data management to be deployed close to IoT devices. This immediacy allows data management to occur in the cloud and along the IoT-to-Cloud path, thus distributing the data processing across multiple nodes and networks that can perform operations on the data.

The OpenFog Consortium defines in [16] fog computing as "a horizontal system-level architecture that distributes computing, storage, control, and networking functions closer to the users along a cloud-to-thing continuum."

2.3 Internet of Things

The term Internet-of-Things (IoT) was first introduced in 1999 by Kevin Ashton due to an interesting observation [23]. The Internet is almost wholly dependent on human beings for information. Most of the data currently on the Internet has been captured and created by humans by typing, talking, and taking pictures. Kevin emphasized how leveraging people to populate the data on the Internet was a bad idea due to their inherent problems with attention, accuracy, and time, which could have led to the spread of low-quality data on the network. Kevin's vision was to empower computers with their means of gathering information through RFID and sensors technology. The usage of several sensors would have allowed computers to observe, identify, and understand the world around them.

Today, there has been much progress, and the Internet-of-Things has boomed. This boom was mainly due to the explosive growth of smartphones and tablet PCs, which brought to 50 Billion the number of connected devices available in the world. IoT is becoming immensely important because it is the first natural evolution of the Internet, which will lead to revolutionary applications such as connected cars, smart cities, smart homes, connected health, and much more [2].

2.3.1 Architecture

The IoT is composed of sensor devices, objects, communication infrastructure, and processing units that can be placed in the cloud or even edge networks. Each object in an IoT network is equipped with a Radio-Frequency IDentification (RFID) tag, which other devices can sense [24]. All devices in an IoT network need to be connected through scalable, interoperable, reliable, and secure network architecture. Generally, the architecture of IoT is divided into five layers which are shown in Figure 2.6:

1. **Perception layer**: this layer is responsible for collecting information from sensors and passing this information to the network layer.
2. **Network layer**: this layer transfers the information from the perception layer to the middleware layer via wired or wireless transmission mediums.
3. **Middleware layer**: this layer is responsible for service management. Hence it performs information processing and makes automatic decisions based on the results of the computations.
4. **Application layer**: this layer enables the creation of applications based on the information processed by the middleware layer.
5. **Business layer**: this layer is responsible for managing the applications and services running on the IoT network.

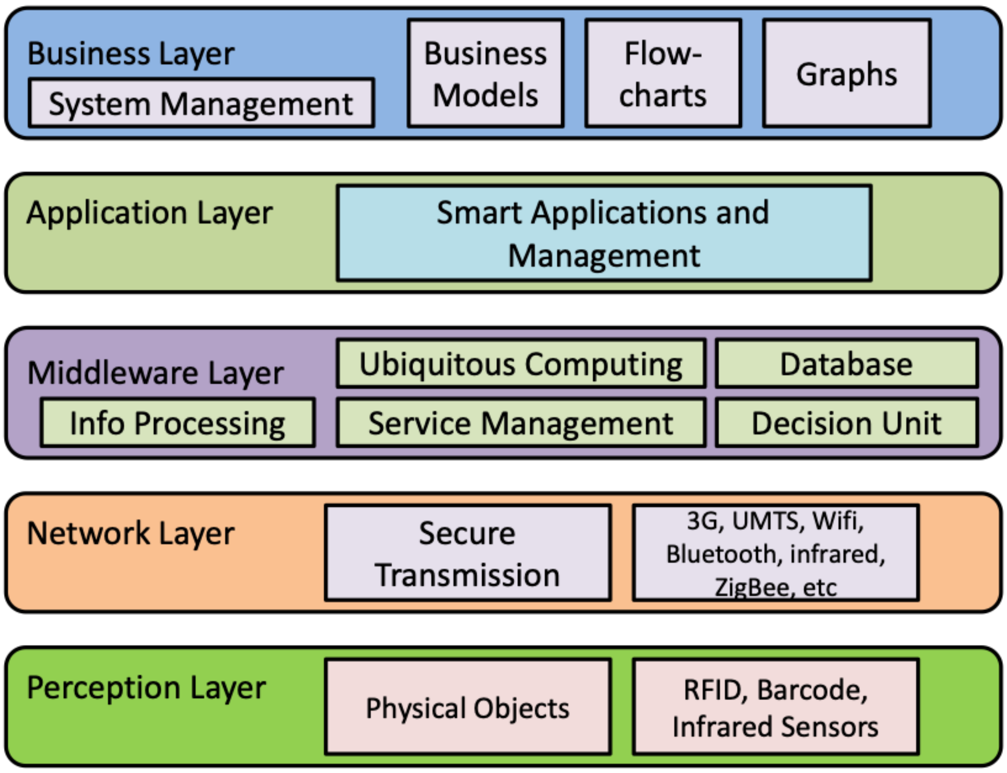


Figure 2.6: IoT layered architecture. [24]

2.4 Distributed Systems

A distributed system is a computing environment in which various components are spread across multiple computers on a network. Each computer in a distributed system is called a node and can be any computing device (e.g., desktop computer, server, mobile device, internet-connected car). These devices split up the work, coordinating their efforts to complete the job more efficiently than if a single device had been responsible for the task.

A distributed system is built for many reasons. We want to have better performance, reliability, solve bigger problems, and/or the system we are building is inherently distributed. However, several problems related to building a distributed system come from the fact that communication may fail without us knowing it, any node may crash or deviate from its normal behavior, and all of the aforementioned issues might happen non-deterministically.

The work in [25] introduces several concepts of distributed systems in a comprehensive and coherent form. The most important concepts are introduced with some simplifications in the following sections to give the reader a high-level overview of what a distributed system is and its main components.

2.4.1 Concurrency

Concurrency is a property of a system representing the fact that multiple activities are executed simultaneously. According to Van Roy in [26], concurrency describes a program having "several independent activities, each of which executes at its own pace." In addition, the activities may perform some interaction among them which can take place in different environments, such as single-core processors, multi-core processors, multi-processors, or even on multiple machines as part of a distributed system.

2.4.2 Models of Distributed Systems

The selection of a system model is vital when designing any distributed algorithm because it helps us clarify the assumptions before starting the design. System models take into consideration many perspectives such as possible faults (e.g., network, node, timing), available network links (e.g., reliable links, fair-loss links, arbitrary links), potential node failures (e.g., crash-stop, crash-recovery, byzantine), and timing assumptions (e.g., synchronous, asynchronous, partially synchronous).

2.4.3 Fault Tolerance and High Availability

The main reason for building a distributed system is to achieve higher reliability than using a single computer. From a business point of view, availability translates to more opportunities. Therefore it is crucial to have high availability.

The availability of a service is typically measured in its ability to respond correctly to requests within a specific time. Typically availability is called uptime, which is the fraction of time a service is functioning correctly. This fraction is computed considering a year (e.g., 99% (Two nines) = 3.7 days/year the system is down).

A fault like a node crash or network interruption is a common cause of unavailability. If we want to increase availability, we can reduce the frequency of faults or design a system that can continue working even if some of its components are faulty. The latter approach is called fault tolerance. Many distributed systems take it instead of buying higher-quality hardware in the hope of better availability.

2.4.4 Broadcast Protocols

Broadcasting is a type of communication involving one sender and a group of recipients, including the sender itself. It is used frequently in distributed systems for many tasks such as replication and shuffling. Reliable broadcast algorithms are the most used because they implement message retransmissions and include:

- **FIFO broadcast**: if m_1 and m_2 are broadcast by the same node, and broadcast(m_1) $\rightarrow broadcast$(m$_2$), then m_1 must be delivered before m_2.
- **Causal broadcast**: if broadcast(m_1) $\rightarrow broadcast$(m$_2$), then m_1 must be delivered before m_2.
- **Total order broadcast**: if m_1 is delivered before m_2 on one node, then m_1 must be delivered before m_2 on all nodes.
- **FIFO-total order broadcast**: a combination of FIFO broadcast and total order broadcast.

2.4.5 Replication

Replication implies maintaining a copy of the same data on multiple nodes called replicas. Fault tolerance is achieved mainly via replication. If one replica becomes faulty, we can continue accessing the copies of the data on another replica.

For replication to be successful, most of the operations happening in a distributed system should be idempotent. If they are applied multiple times, the final result remains unchanged. Idempotency is defined mathematically as $f(x) = f(f(x))$ and helps to deal with reconciliation errors that may occur during replication.

2.5 Apache Spark

2.5.1 General Overview

A popular model of cluster computing widely adopted in the literature and industry revolves around parallel computations executed on clusters of unreliable machines by systems that provide data locality, fault tolerance, and load balancing. The most common framework that implemented this model is MapReduce, developed by Google [27]. According to the authors of Spark, the main issue with MapReduce and similar frameworks is the acyclic data flow model around which they are built [28]. The acyclic nature of the data flow is not suitable for many popular applications which reuse data across multiple parallel operations. This problem led to the creation of a new cluster computing framework called Spark, a scalable, fault-tolerant, and high-performance solution for executing distributed tasks on clusters of machines [28].

The main benefit of Spark over existing alternatives is the usage of a distributed memory abstraction called Resilient Distributed Dataset (RDD), a read-only collection of objects partitioned across a set of machines. The main benefit of RDDs is that they are in-memory by default, which translates to performance improvements in the order of magnitude if compared to traditional disk-based approaches [29].

As discussed by the authors in [28], Spark is beneficial in iterative jobs, which reuse a working set of data across multiple parallel operations (e.g., machine learning algorithms, optimization algorithms).

2.5.2 Programming Model

Spark is based on a programming model which provides two main abstractions for parallel programming: Resilient Distributed Datasets (RDDs) and parallel transformations on these datasets. In addition, it supports two restricted types of shared variables that enable an efficient implementation of particular use-cases in Spark.

These features of the Spark programming model have been used extensively during the design of the proposed algorithms. They are going to be illustrated in full detail in the solution chapter.

Resilient Distributed Dataset

The Resilient Distributed Dataset (RDD) is a distributed memory abstraction that allows the execution of in-memory computations on large clusters in a fault-tolerant manner. Behind the creation of RDDs, there was the need to reuse data between computations, a common use case for algorithms such as PageRank, K-means clustering, and logistic regression. The reuse of data could have only been done with existing frameworks by writing it to stable external storage, hence introducing a significant delay. As a result, RDDs can obtain 20x faster performance than Hadoop for iterative applications [29].

RDDs aim at providing fault tolerance efficiently with the use of coarse-grained transformations (e.g., `map`, `filter`, and `join`) that are applied in parallel to all the partitions of the RDD [29]. The immutable nature of RDDs and the determinism of the parallel transformations facilitate the debugging process and significantly streamline the design of resiliency mechanisms.

RDDs provide fault tolerance by creating the lineage, which is a log of all the transformations used to build a dataset. Hence, if a partition or more is lost, the RDD has enough information to recompute the partition by accessing and replaying its lineage. Essentially, a program cannot reference an RDD that it cannot reconstruct after a failure [29].

The programming interface of Spark appears limited at first due to its immutable nature and coarse-grained transformations. Nonetheless, usage and experience have shown that this programming model is suitable for many applications and cluster programming models (e.g., MapReduce, DryadLINQ, SQL, Pregel). The composable nature of the transformations and an idiomatic language such as Kotlin encourage the developers to write high-quality, straightforward code. An example of estimating the number π with Kotlin and Spark is shown in Figure 2.7.

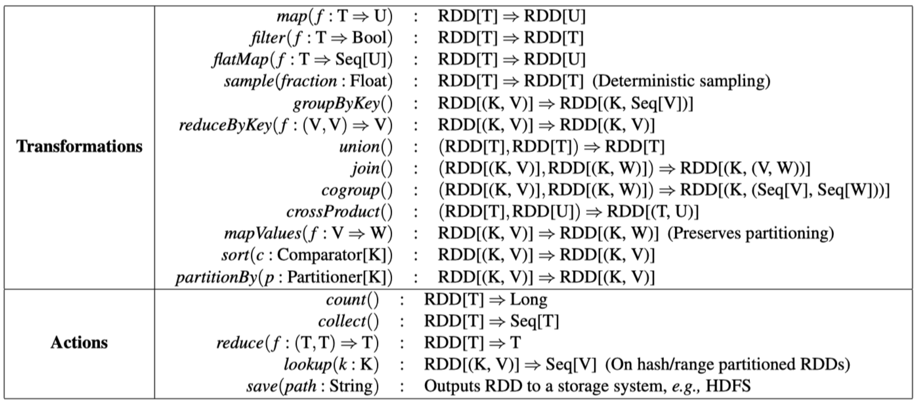
```scala
val n = 1000
val count: Long = sc
    .parallelize((1..n).toList())
    .filter {
        val x = Math.random()
        val y = Math.random()
        x * x + y * y < 1
    }
    .count()

println("Pi is roughly \${4.0 * count / n}")
```

Figure 2.7: Estimation of the number π with Apache Spark.

Spark permits two additional aspects of the RDD to be handled, such as persistence and partitioning [29]. When an RDD is persisted, each node stores any partition it has computed and can reuse them for later operations. RDDs are persisted by default in memory. However, the user can call the `persist` method to persist an RDD with a specific strategy (e.g., hybrid, disk, replicated). For example, in hybrid mode, the data will be spilled to disk only when there is not enough main memory. The following valuable component for working with RDDs is partitioning, which can be tuned to fit specific use-cases (e.g., change the number of partitions, partition elements by key). The effect of partitioning on the algorithm's performance can be particularly noticeable. For example, if we use too few partitions, our worker nodes could sit idle. In contrast, if we use too many partitions, we could introduce a considerable overhead in the scheduling performed by Spark.

Spark offers a wide range of transformations and actions that can be performed on RDDs. The former contains lazy operations that define a new RDD (e.g., `map`, `filter`), while the latter contains operations that launch computations in order to return a result or write to external storage (e.g., `collect`, `reduce`). There are several transformations and actions available, some of them are shown in Figure 2.8.



Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] $\Rightarrow$ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$sample(fraction : \mathrm{Float})$	:	RDD[T] $\Rightarrow$ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V},\mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) $\Rightarrow$ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) $\Rightarrow$ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
Actions	$count()$	:	RDD[T] $\Rightarrow$ Long
	$collect()$	:	RDD[T] $\Rightarrow$ Seq[T]
	$reduce(f : (\mathrm{T},\mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] $\Rightarrow$ T
	$lookup(k : \mathrm{K})$	:	RDD[(K, V)] $\Rightarrow$ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Figure 2.8: Transformations and actions available on RDDs in Spark. [29]

The transformations are *lazy* because they are not evaluated until an action is called on the RDD. This aspect opens several opportunities for optimization because Spark can have a high-level overview of all the transformations we want to perform before executing them.

At the center of any optimization in Spark, the scheduler is responsible whenever a user runs an action to create a Directed Acyclic Graph (DAG) of stages to execute. The DAG is created by inspecting the lineage graph of the RDDs and deriving from that a set of stages. Each stage contains as many operations as possible and is delimited by the usage of shuffle operations (e.g., `groupWith`, `join`). In general, we can say that a job is launched when an action is called and is divided into multiple stages. Each of the stages contains a set of parallel tasks executed one per partition.

Shared Variables

The Spark API hides most of the complexity with a programming model similar to Java Streams, in which there are a series of transformations applied to the data. The problem is that the implementation is entirely different underneath, causing frequently unexpected behavior. When programmers use Spark and pass closures to operations such as `map` or `filter`, there is always a concern related to the variables used within the closure. In that case, Spark sends separate copies of all the variables referenced by a closure to all the worker nodes executing that particular transformation. However, no updates to the variables performed on the remote machines are propagated back to the driver program. This design decision is mainly for efficiency reasons because general read-write shared variables are costly and inefficient. Instead, Spark provides two types of shared variables that aim at supporting common usage patterns [28]:

- **Broadcast variables**: when you often reuse a heavy piece of data across multiple operations, it becomes pretty inefficient to resend it for each closure's execution. Data propagation becomes even worse when dealing with large clusters, leading to high network usage and congestion. Broadcast variables are read-only variables that are efficiently broadcasted to each worker node and cached in the main memory for faster access. The main benefit is that these variables do not need to be sent for each closure. Instead, they are sent asynchronously and used throughout the life-cycle of the Spark application. In the end, each of these variables may also be unpersisted in order to free up some memory.

- **Accumulators**: it is common for Spark jobs to increment shared variables such as a counter. However, this cannot be achieved simply by incrementing a global variable from within a closure for a previously explained reason. Accumulators aim to solve this problem by offering a variable that can be only "added" through associative and commutative operations. These properties allow for an efficient implementation of this type of shared variable but restrict its usage possibilities.

2.5.3 Architecture

The architecture of Spark is based on an independent set of processes on a cluster coordinated by the driver program. The driver is a critical component of Spark because it is responsible for tasks such as maintaining the `SparkContext`, scheduling jobs on worker nodes, and managing the DAG of operations. In order to run on a cluster, the `SparkContext` must connect to a cluster manager such as standalone, Mesos, YARN, or Kubernetes. Spark is agnostic to the underlying cluster manager, so deciding which cluster manager to use is entirely up to the developer. The only prerequisite for Spark to work is to acquire executor processes that can communicate with each other [30].

The architecture of Spark is shown in Figure 2.9.

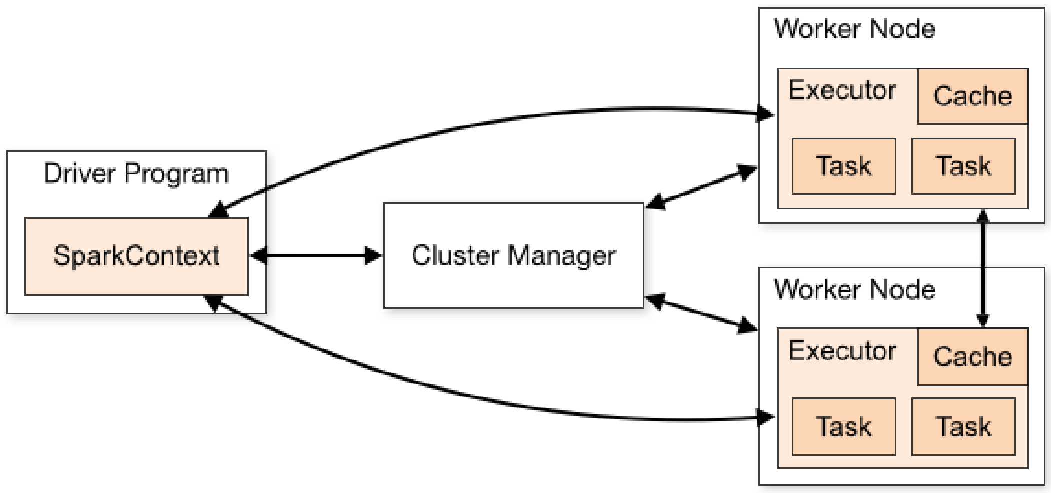


Figure 2.9: Spark architecture. [30]

Spark is resilient to worker failures by default as long as the driver program remains active. This characteristic introduces a single point of failure in case the driver crashes because the current state of the Spark application will be lost, together with all the results of the jobs that were executing or have finished execution. There are solutions to this problem, such as using ZooKeeper to maintain the shared state of the driver across multiple replicated machines. However, the solutions depend on the cluster manager used and are currently being proposed by individual contributors and companies.

2.5.4 The Kubernetes Cluster Manager

Spark supports several cluster managers such as standalone, Apache Mesos, Hadoop YARN, and Kubernetes. The latter is the cluster manager of choice for deploying the algorithms proposed in this thesis. The primary motivation behind the choice of Kubernetes is the extraordinary growth of containerized applications, which are becoming wildly popular due to their use of containers. Containers encapsulate autonomous execution environments running on logical resources and provide better performance than Virtual Machines (VMs). The main benefits of containers include modularity, portability, scalability, and maintainability because they can be updated, changed, or removed without much friction.

Kubernetes is an open-source system for automating containerized applications' deployment, scaling, and management. A Kubernetes cluster consists of worker machines (at least one), called nodes, that run containerized applications. Kubernetes revolves around core components that handle resource management, load balancing, and autoscaling [31]:

- **Pods**: pods are the basic scheduling unit in Kubernetes. They group containerized applications and are guaranteed to be allocated on the same machine with the ability to share resources. Each pod is assigned specific resource requirements that must be met by Kubernetes while deploying pods on nodes. Additionally, pods allow direct communication between containers belonging to the same pod.

- **Services**: services are components that act as simple load balancers for pods. They group several pods and abstract them as individual entities. The grouping allows a single entry-point to be defined and accessible by internal or external consumers. In addition, it enables the scaling of all the back-end containers, depending on the requirements.

- **Autoscalers**: autoscalers are responsible for making sure the number of pods in the cluster corresponds to the number of pods in the configuration. A service always has one autoscaler responsible for handling horizontal scalability, therefore increasing availability.

- **Nodes**: nodes are components of the cluster that run the Kubernetes platform. There are two types of nodes, namely master and worker nodes. The master node implements the server, which exposes the Kubernetes API and performs scheduling and orchestration decisions. The worker node hosts pods and exposes a set of resource capabilities that are considered for any scheduling decision.

- **Scheduler**: the scheduler is responsible for assigning pods to specific nodes in the cluster, taking into account all the requirements and resource capabilities of the nodes. In order to make all of these decisions, the scheduler needs access to information such as the total capacity of each note and the already allocated resources.

2.6 Kotlin and Coroutines

2.6.1 The Kotlin Language

Kotlin is a statically typed programming that is concise, safe, and interoperable with Java. It provides many ways to reuse code between multiple platforms for effective programming [32]. The language has been developed by JetBrains, the renowned Czech company behind the IDEs like IntelliJ IDEA and PyCharm. Since its first release, the growth pace and adoption of the language have continued to increase, making Kotlin the fastest evolving language on GitHub.

The growth of Kotlin and its developer satisfaction have been significant for several reasons. The first and foremost reason is that it is a modern programming language built by a company specializing in IDEs that understands other programming languages and their problems pretty well. The experience of the designers has led to a continuous improvement of the language, which included major requested features such as the support for multiple platforms and the introduction of asynchronous programming with coroutines.

The verbosity of the Java language compared to Kotlin is also one of the main motivations for many developers to switch. For example, if we want to sum integers from 1 to x in Java, we would do it like in Figure 2.10.

```java
int x = 100;
int sum = 0;
for (int i = 1; i <= x; i++) {
    sum += i;
}
```

Figure 2.10: Summing integers from 1 to x in Java.

In Kotlin, we can achieve the same thing with the code in Figure 2.11.

```kotlin
val x = 100
val sum = (1..x).sum()
```

Figure 2.11: Summing integers from 1 to x in Kotlin.

In addition to the streamlined syntax, Kotlin has an attractive feature-set that contains null-safety, inline functions, extension functions, type-safe builders, and destructuring declarations to make it one of the best and most exciting programming languages available to date. Another appealing characteristic of the language is its standard library, which contains the living essentials of the language. It is also updated frequently with new functions (e.g., higher-order functions, extension functions, utilities, and functions for JDK interoperability).

All of the benefits above and positive past experiences with the language led to the decision of Kotlin to implement the algorithms proposed in this thesis.

2.6.2 Kotlin Coroutines

Coroutines are components of a computer program that generalize subroutines by allowing their execution to be suspended and resumed. The suspension and resumption characteristics allow coroutines to do non-preemptive multitasking. In coroutines, there is no context switch when running from one process to another. For this reason, there is the need to yield control to other coroutines voluntarily via a symmetric yielding mechanism (e.g., coroutine A calls coroutine B, and then coroutine B yields to A).

Kotlin introduced in version 1.1 the first implementation of coroutines, which are designed for asynchronous and non-blocking programming. The exciting part of Kotlin's implementation resides in the two different levels at which they have been integrated into the language. First, the coroutines have been supported at the language level with a compiler implementation. However, most of the functionality is moved to libraries [33]. The implementation of most high-level coroutine-enabled primitives is contained in the coroutines library, which can be updated independently from the language, resulting in quicker bug fixes and faster development.

Structured Concurrency

In Kotlin, all coroutines follow the structured concurrency principle. A new coroutine can be launched only within a specific scope, thus delimiting its lifetime. Having a structure in the concurrency allows for managing multiple coroutines and avoids lost or leaking coroutines. For example, in Kotlin, the outer level scope cannot complete until all its child coroutines have been completed [33].

An example of a simple program using coroutines, which shows their structured nature, is available in Figure 2.12.

```
fun main() = runBlocking {
    launch { doWorld() } // A new coroutine is launched.
    println("Hello") // The print statement is executed before the other print statement.
}

suspend fun doWorld() {
    delay(1000L) // The coroutine is suspended and resumes after 1000ms.
    println("World!") // The print statement is executed after the resumption.
}
```

Figure 2.12: Example of Kotlin coroutines showing structured concurrency.

Suspending Functions

Central to coroutines in Kotlin is the concept of suspending functions that are marked with the `suspend` modifier. A suspending function is a particular function that can be called only from a coroutine or another suspending function. A call to a suspending function creates and starts a coroutine. From the developer's perspective, suspending functions are like normal functions. However, they can suspend and resume the execution of the calling coroutine.

Kotlin automatically passes a `Continuation<in T>` as a parameter to the function in order to implement the suspending behavior. The continuation has a `resumeWith(result: Result<T>)` method that is automatically called by the coroutine when the execution is concluded in order to resume from the calling coroutine. The return type of the suspending function becomes the `T` parameter of the continuation.

Suppose we want to call a suspending function from a non-suspending context. In that case, Kotlin provides several "coroutine builders" that are required as most platforms are unaware of coroutines. The role of builders is to abstract away the complexity of managing coroutines and their lifecycle. Some builders include `launch`, `runBlocking`, `async` [34].

Coroutine Dispatchers

Coroutines are distinct from threads because they are executed on threads and are interleaved with respect to their suspension points. In order to enable the execution of coroutines on different threads, Kotlin includes the concept of coroutine dispatchers. A dispatcher determines which thread(s) the corresponding coroutine uses for its execution (e.g., specific thread, thread pool, unconfined) [33].

Chapter 3

Problem Statement

This chapter presents the problem we want to address and the research questions we want to answer.

3.1 Context

The growth of edge computing has introduced several challenges for different research areas. One of the primary challenges is solving optimization problems that include optimizing offloading strategies, workload placement, load balancing, performance management, and more. The landscape of optimization algorithms is vast. However, an algorithm has gained much attention in recent years. The algorithm is Particle Swarm Optimization (PSO), a nature-inspired stochastic optimization method well known for its effectiveness in addressing several types of optimization problems. PSO has been applied to several edge computing-related problems like performance management [35], computation offloading [36], and workload placement [6] in the literature. The central concern of the forenamed research is the lack of focus on performance, fault tolerance, and scalability of the PSO algorithm, which are all essential factors to consider when executing the algorithm in an edge network. The papers instead focus on developing a problem encoding, that is, representing a specific domain problem into a PSO-solvable problem.

It becomes crucial to move as many computations as possible to the edge in edge computing. Otherwise, they would be forced to the cloud, making the edge network less valuable. The execution of the PSO algorithm at the edge brings several challenges, the major one being creating a performant, fault-tolerant, and scalable algorithm, all within stringent constraints. The complexity of creating the algorithm comes from the lack of computational resources and the high likelihood of failures typical of edge networks. These problems become even more evident when running optimization algorithms that are likely to grow in complexity, such as PSO. Indeed, the complexity of PSO can overgrow depending, especially on the problem encoding, due to the fitness function complexity, the number of constraints, the dimensionality of the problem, and the data structures used.

Considering all the previous concerns, our problem is the low performance, lack of scalability, and absence of fault tolerance in the traditional PSO algorithm when executed in an edge computing environment.

3.2 Problem Details

3.2.1 Research Questions

We have a few research questions for this thesis that we want to answer, whose goal is to guide us during the research. The questions that we asked ourselves are:

- Does it make sense to distribute the computations of the PSO algorithm across multiple machines?
- Is it feasible to make a distributed algorithm using existing technologies without implementing everything from scratch?
- Is it possible to find technologies that will provide a good balance between performance, fault tolerance, and scalability?
- Is it doable to design distributed synchronous and asynchronous variants of the PSO algorithm?
- What are the strengths and weaknesses of the synchronous design compared to the asynchronous and vice versa?
- Will the distributed algorithms perform worse than the traditional PSO in certain conditions?

3.2.2 Particle Swarm Optimization Complexity

The PSO algorithm has attracted much attention mainly because of its ability to solve many problems in different domains. However, it suffers from issues related to high computational complexity. This performance bottleneck restricts the domain of applications in which the algorithm can be used, for example, when constraints on computational resources are needed [37].

The number of computations required for completing an algorithm iteration is the sum of all the calculations for evaluating the fitness function and updating the velocity and position of each particle in the swarm. These computations are directly proportional to the number of iterations. The more iterations we have, the more work must be accomplished. The number of iterations becomes a more significant problem knowing that the standard PSO algorithm does not reduce the number of calculations required during the execution. For example, using the sphere function defined as $\sum_{i=1}^{D} x_i^2$, assuming M particles with D dimensions and N iterations, we can approximately compute the complexity of the algorithm. For each particle, we will perform D multiplications N times, which for M particles results in $D \times N \times M$ multiplications. This simple example does not consider all the details of the algorithm. However, it shows how quickly the complexity can grow even with a trivial fitness function with linear asymptotic complexity.

The problem encoding is how we represent a domain problem as a PSO-solvable problem. The definition of a problem encoding directly impacts the algorithm's performance because it involves a fitness function, set of constraints, and data structures to represent the problem itself. All of these elements will inevitably increase the complexity of the algorithm.

As discussed in the introduction, encodings are so impactful on performance that in the literature, there exist many encodings to benchmark the implementation of the algorithm. From this, it may seem that the main improvement that can be made to the algorithm is to tweak its problem encoding. However, this is far from the truth. There are several research papers such as [37] which aim to improve the PSO algorithm from both performance and accuracy perspectives. All of this without considering the problem encoding, but instead by reworking the algorithm's fundamentals or introducing new concepts.

3.2.3 Particle Swarm Optimization at the Edge

One of the main rationales behind edge computing is the possibility of pushing as many computations as possible to the edge to reduce the strain on the cloud infrastructure. The execution of optimization algorithms to improve resource utilization and workload placement in an edge network is a critical task that must be executed in the proximity of the edge devices. The nearby execution of the algorithms brings several improvements, such as reduced latency and less network congestion. However, it also brings unique challenges due to the lack of computational resources and a higher probability of faults.

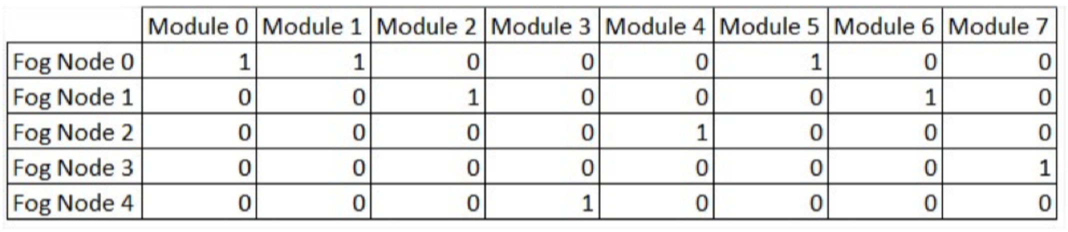
The challenges become even more apparent when working with optimization algorithms such as PSO, which is known to be expensive to run, even in non-edge environments. The main challenges that can arise when running the PSO algorithm at the edge are the following:

- **Low computational capabilities**: each node in an edge network is relatively slow, meaning that it has low computational capabilities. The lack of computational power is a big problem for PSO, which is an algorithm that requires a certain number of resources in order for it to be executed moderately fast. The algorithm's execution time becomes even more critical in a time-sensitive domain like workload placement, where the placement decisions need to be determined in a small amount of time.
- **Faults during the execution**: in an edge network, the probability of faults is higher than in a standard cloud infrastructure. The higher probability is mainly because cloud environments use different techniques to reduce the likelihood of failures, such as data sharding, Uninterruptible Power Supply (UPS), and highly-reliable hardware. These techniques are expensive and inefficient because they leverage redundancy and advanced hardware to reduce the probability of failures. Edge networks cannot implement these resiliency mechanisms due to the aforementioned reasons. Therefore the likelihood of node failures is, in general, higher. A failure can be pretty impactful when running the PSO algorithm on a single node because it will inevitably invalidate all the progress made during the optimization. The progress invalidation becomes especially problematic in the case of significant optimization problems because it will require a re-execution of the algorithm, resulting in a noticeable delay.

3.2.4 PSO Encoding for Workload Placement

An example of PSO used in the context of edge computing is the workload placement problem that has been tackled in [6]. The work in [6] demonstrated encouraging results like a reduction in the number of iterations needed to obtain a satisfactory result compared to other solutions. The algorithm used in the forenamed work is a variation of the traditional PSO algorithm, namely Binary Multi-Objective Particle Swarm Optimization (BMOPSO). BMOPSO is a variant of the PSO that uses a discrete set of values $(0, 1)$ and searches for a balance between multiple conflicting objectives while optimizing the fitness function.

The problem encoding of BMOPSO uses a matrix data structure D to represent each particle's position. The matrix comprises a set of fog nodes F and a set of modules M. Each cell in the matrix $D[f][m]$ for some module f and fog node m contains either 0 or 1 given the binary nature of the algorithm. The 1 signals that the module m should be placed on the fog node f, whereas the 0 signals the opposite. A graphical representation of the matrix data structure used in the encoding is represented in Figure 3.1.



	Module 0	Module 1	Module 2	Module 3	Module 4	Module 5	Module 6	Module 7
Fog Node 0	1	1	0	0	0	1	0	0
Fog Node 1	0	0	1	0	0	0	1	0
Fog Node 2	0	0	0	0	1	0	0	0
Fog Node 3	0	0	0	0	0	0	0	1
Fog Node 4	0	0	0	1	0	0	0	0

Figure 3.1: Placement representation. [6]

The fitness function s considers multiple parameters such as RAM, latency, and CPU usage over fog nodes. The result of the fitness function is a measure of the maximum resource requirements needed to run a set of modules on any one of the fog nodes. The goal of the optimization is to minimize the maximum value to find the most efficient placement of the modules on the available fog nodes.

The asymptotic complexity of the algorithm can be approximated to $O(n^2)$ if we assume a similar number of fog nodes and modules. The quadratic complexity is not problematic for minor optimization problems. However, when the problem size increases, several scalability issues may arise (e.g., slow execution time, node failure). These issues are even more evident in an edge environment where the computations could take a significant amount of time and the nodes are more subject to failures due to resource constraints. In addition, it is essential to consider that the problem encoding could be tweaked with a likely growth in complexity due to an increase in constraints and/or improvement of the fitness function.

The BMOPSO algorithm has been shown to require a significant amount of time to obtain an optimal placement response, which is a problem for delay-sensitive applications such as workload placement. Moreover, the optimization problem is solvable when enough time and resources are available, which is often not the case in an edge environment. The work in [6] is a good example and motivation for this thesis because it focused on designing a problem encoding instead of also considering the performance, fault tolerance, and scalability of the PSO algorithm. The problems above will be tackled in this thesis, with a core focus on the performance of the algorithms in a resource-constrained environment like an edge network.

Chapter 4

Problem Solution

This chapter comprehensively introduces the suitability of PSO to be distributed, the challenges involved in designing distributed algorithms, and the design decisions behind the proposed algorithms.

4.1 PSO Suitability

The PSO algorithm has several benefits, and one of the most impactful for our work is its strong distributed ability. This ability comes from the fact that the algorithm revolves around a set of particles that are mostly independent. This independence proves very beneficial in designing a single-machine parallel version of the algorithm. However, when distributing the computation over multiple machines, new challenges arise that are not strictly related to PSO. A distributed algorithm generally uses two or more computers communicating over an unreliable network to achieve a common objective. In the case of PSO would be to find the minimum/maximum of a specific fitness function. Theoretically, a distributed algorithm is significantly more fault-tolerant and powerful than a centralized algorithm running on a single machine. However, the performance of the distributed algorithm can severely degrade if not properly designed and can become even worse than the non-distributed variant [38]. The significant problems that can emerge when distributing the algorithm are related to the communication and execution cost (e.g., synchronization time, delayed data transfer, lost and corrupted packages, lost connection) and may outweigh the benefits offered by the distribution of the algorithm.

We analyzed the basic PSO algorithm to find out how it could be parallelized. PSO is suited for coarse-grained parallelization, in which the swarm is split into multiple large sub-swarms that are evaluated in parallel either on the same multi-processor machine or in a distributed system composed of multiple nodes. Indeed, for each algorithm iteration, all the particles are independent and can be quickly evaluated in parallel [10]. In the work [11], a parallel implementation of PSO was proposed using a coarse decomposition scheme. The algorithm performed the fitness evaluations concurrently on a parallel machine. Both [10] and [11] proposed the fitness function evaluation as the parallelization point because each particle could evaluate its fitness function independently. We have also identified that the fitness function evaluation was the most computationally intensive part of the algorithm for many problem encodings. For example, in [11] a medium-scale biomechanical system identification problem has been tackled, in which the fitness function evaluation took approximately 1 minute. It becomes clear that executing the traditional sequential PSO algorithm would require significant time for a satisfactory result. We decided to proceed and parallelize the algorithm around the fitness function evaluation for these reasons. However, we encountered another problem related to the performance of the traditional synchronous PSO algorithm.

The synchronous PSO algorithm can be implemented with coarse-grained parallelism. However, it presents a problem of efficiency because it is nearly impossible to keep all nodes working towards the end of each iteration before starting the next iteration [10]. For example, we could have a faster machine in the cluster that computes a subset of particles in half the time compared to the other nodes. Hence, it will remain idle until all the other nodes have completed their execution, resulting in a severe efficiency problem that can be solved with the usage of fine-grained parallelism and the asynchronous variant of the PSO algorithm. In fine-grained parallelism, the algorithm will be broken into many small tasks that are executed in parallel, which is necessary but not sufficient to make the algorithm more efficient. The necessary modification is to make the algorithm asynchronous. In the asynchronous algorithm, we will update each particle's velocity and position right after its function evaluation instead of updating the velocity and position only after all the particles have been evaluated. This behavior change will result in a continuous evaluation of particles that are no more tied to a specific iteration. Therefore they are entirely independent and can be rescheduled on less busy nodes for their subsequent evaluation. The difference between synchronous and asynchronous PSO is also described comprehensively in [14] and in the background chapter of this thesis.

The proposed solutions for the distributed PSO algorithm involve designing and implementing two variants of the distributed PSO algorithm, namely synchronous and asynchronous. These variants will be first presented with a higher-level description of their control flow. Then the actual implementations will be comprehensively explained.

4.2 Distributed Synchronous and Asynchronous PSO

After identifying the parallelization point, we started the high-level design of two variants of the distributed PSO algorithm without considering the implementation details but rather the execution flow. The design of the two variants aimed at exploring several possibilities instead of searching for a one-size-fits-all solution. In the background chapter, we introduced the significant dissimilarities between the synchronous and asynchronous PSO, which eventually came down to their different handling of the particle's velocity and position updates. There is the need to synchronize all the particles around iterations in the synchronous PSO. In contrast, each particle can move right after its fitness function has been evaluated in the asynchronous PSO. In the context of distributed algorithms choosing the timing assumptions and distributed computing paradigm is critical because it will inevitably affect the design and performance of the algorithm. For our algorithms, we decided to opt for the master/slave paradigm in which there are two types of nodes; the master node, which is responsible for coordinating the algorithm, and the slave node, which is responsible for carrying out computations dispatched by the master. The choice of this paradigm was particularly suitable for PSO because a master node can control the flow of the algorithm, and the slave nodes can carry out the function evaluations.

Returning back to the PSO algorithm's fundamentals, we introduced that PSO is characterized by the following components:

- The i-th particle position vector X_i
- The i-th particle velocity vector V_i
- The i-th particle personal best position vector P_i
- The best global position vector P_g
- The fitness function f

All of the components above are discussed more in detail in the background chapter. They are also summarized here to simplify the understanding of the upcoming algorithms.

4.2.1 Distributed Synchronous PSO (DSPSO)

Our distributed synchronous PSO follows the master/slave paradigm in which the master node holds the state of the entire swarm and dispatches the particles on each slave node for evaluation. The master also performs all the synchronization needed to manage the iterations specific to the synchronous PSO.

The tasks performed by the master and slave nodes are as follows:

Master node

1. Initializes the problem encoding parameters, positions, and velocities;
2. Initializes the state of the swarm, including current iteration and received particles;
3. Starts the iteration by distributing all the particles to the available slave nodes;
4. Waits and receives back all the particles with their function evaluation result and best personal position P_i;
5. Computes for each incoming particle the best global position P_g until all particles have been received;
6. Updates the velocity V_i and position X_i vectors of each particle i based on the best personal P_i and global position P_g found by all the particles;
7. Goes back to step 3 if the last iteration is not reached;
8. Returns the best global position P_g.

Slave node

1. Waits for a particle from the master node;
2. Evaluates the fitness function f and updates the personal best position P_i;

3. Sends the evaluated particle back to the master node;

4. Goes back to step 1 if the master is not finished.

A high-level overview of the flow of the algorithm is depicted in Figure 4.1.

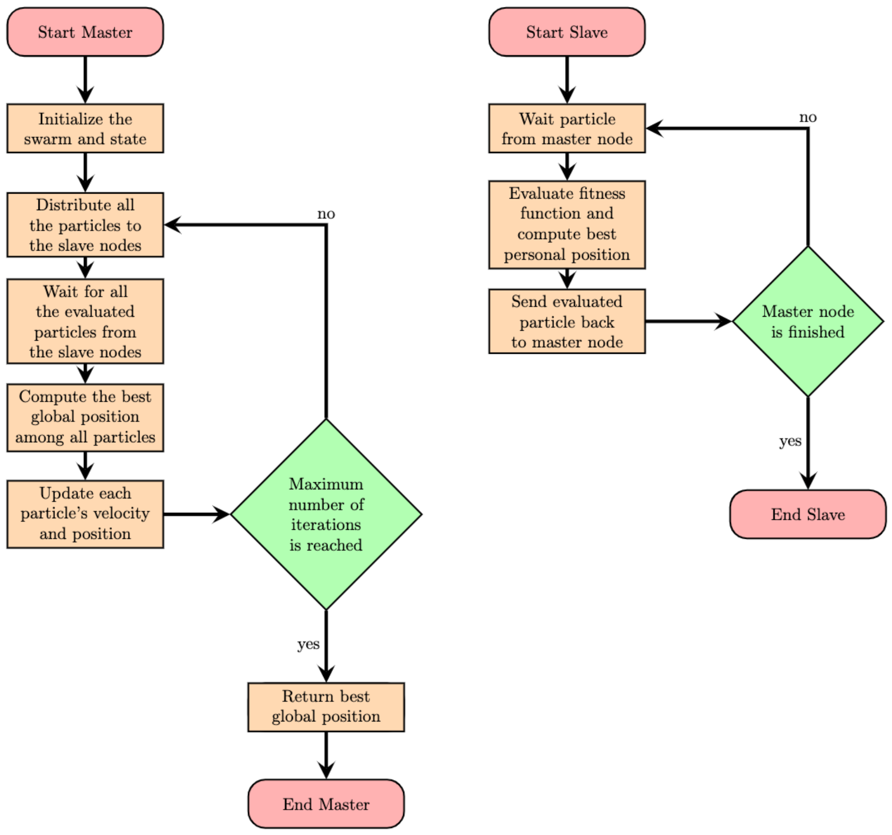

Start Master
Initialize the swarm and state
Distribute all the particles to the slave nodes
Wait for all the evaluated particles from the slave nodes
Compute the best global position among all particles
Update each particle's velocity and position
Maximum number of iterations is reached
no
yes
Return best global position
End Master
Start Slave
Wait particle from master node
Evaluate fitness function and compute best personal position
Send evaluated particle back to master node
Master node is finished
no
yes
End Slave

Figure 4.1: Flowchart of master and slave in DSPSO.

4.2.2 Distributed Asynchronous PSO (DAPSO)

Our distributed asynchronous PSO follows the master/slave paradigm similar to the synchronous variant. However, it differs in the synchronization part because there are no iterations that link together all the particles. In the asynchronous PSO, each particle is evaluated and moved independently from the others, increasing their independence. The tasks performed by the master and slave nodes are as follows:

Master node

1. Initializes the problem encoding parameters, positions, and velocities;
2. Initializes the state of the swarm, including a queue of particles to send to slave nodes;
3. Loads the initial particles into the queue;
4. Distributes the particles from the queue to the available slave nodes;
5. Waits and receives back each particle with its fitness function evaluation result and best personal position P_i;
6. Updates the best global position P_g based on each incoming particle;
7. Updates the velocity V_i and position P_i vectors of each incoming particle i based on the best personal P_i and global position P_g found until that point;
8. Pushes the particle back into the queue and goes back to step 4 if the stopping condition is not met;
9. Returns the best global position P_g.

Slave node

1. Waits for a particle from the master node;
2. Evaluates the fitness function f and updates the personal best position P_i;
3. Sends the evaluated particle back to the master node;
4. Goes back to step 1 if the master is not finished.

A high-level overview of the flow of the algorithm is depicted in Figure 4.2.

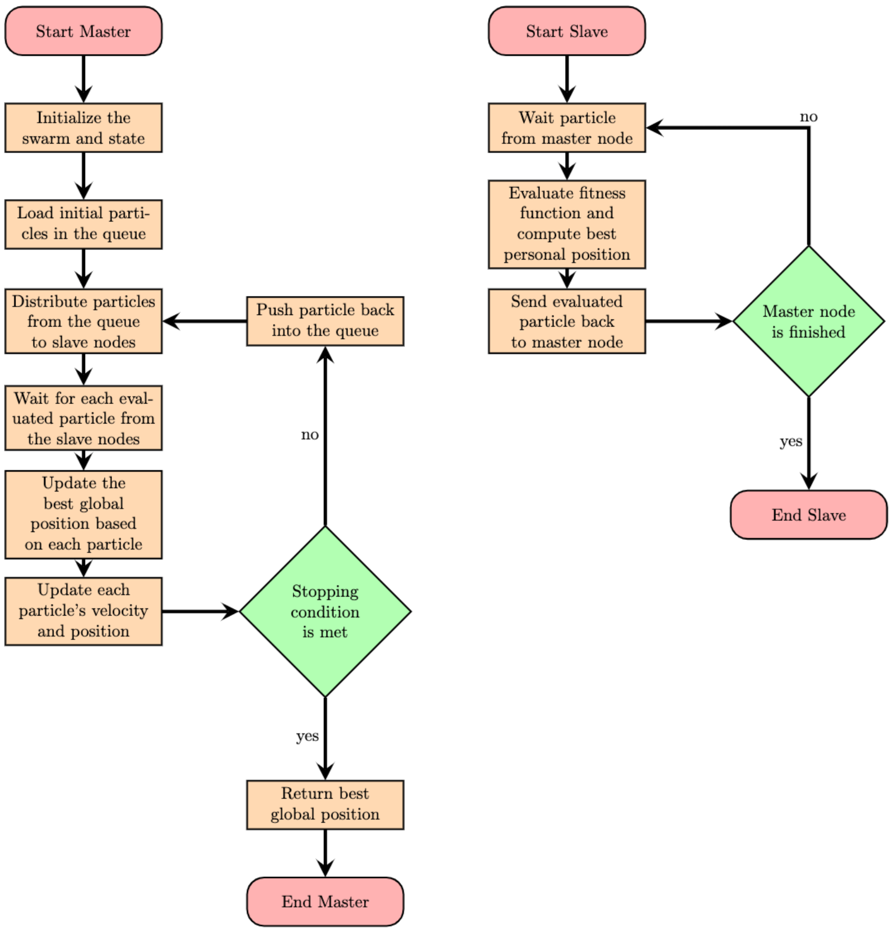

Start Master
Initialize the swarm and state
Load initial particles in the queue
Distribute particles from the queue to slave nodes
Push particle back into the queue
Wait for each evaluated particle from the slave nodes
Update the best global position based on each particle
Update each particle's velocity and position
Stopping condition is met
no
yes
Return best global position
End Master
Start Slave
Wait particle from master node
Evaluate fitness function and compute best personal position
Send evaluated particle back to master node
Master node is finished
no
yes
End Slave

Figure 4.2: Flowchart of master and slave in DAPSO.

4.2.3 Discussion

Even if the two algorithms look pretty similar from their high-level flow, they are pretty different from an execution flow and accuracy perspective.

The synchronous algorithm requires the master to wait for all particles to be evaluated before starting a new iteration, which results in some slave nodes being idle, especially if the load on the cluster is imbalanced (e.g., some slave nodes might finish before others). Moreover, it allows for faster convergence because the whole swarm will converge faster towards the best global position found by any of the particles in the previous iteration.

This behavior is in stark contrast with the asynchronous algorithm, which updates the velocity and position of each particle as soon as its fitness function has been evaluated, effectively using the best global position available until that point. This characteristic results in better usage of cluster resources and more exploration by the swarm, which can be positive or negative depending on the optimization problem's domain.

4.3 Distributed Programming

4.3.1 Designing a Distributed Algorithm

Creating a distributed algorithm is complex and demands meticulous assumptions to be made during the design. If not designed adequately, the algorithm's performance can deteriorate quickly, thus leading to worse performance than the non-distributed counterpart. Indeed, two high costs of any distributed algorithm are identified in the execution and communication time [38]. In order to minimize these two costs, four aspects should be considered while conceiving the algorithm [38]:

1. **Fault tolerance**: in a distributed system, non-deterministic failures can occur and disrupt the execution. For this reason, the algorithm should be able to react to lost connections, corrupted packages, and other possible errors.

2. **Synchronous and asynchronous**: timing assumptions are necessary while designing any distributed algorithm. Generally, synchronous algorithms are simpler to design because all the nodes are synchronized. In contrast, there is no minimum waiting time for synchronization between nodes in the asynchronous model, thus making the design more complicated. *It is important to note that the synchronous and asynchronous terms mean different things in distributed systems and PSO.*

3. **Partitioning and load balancing**: because of the different characteristics of the nodes in a distributed system, it is vital to distribute the load across the nodes evenly. The even distribution requires proper partitioning and balancing of the load, which might also involve redistributing the load dynamically based on the system's state.

4. **Paradigms**: there are four types of paradigms that are typically used when implementing distributed algorithms, which include master-slave, island, diffusion, and hierarchical hybrid model. These models have their characteristics and best use-cases, which must be carefully considered.

4.3.2 Custom vs. Ready Distributed Computing Framework

Considering the intricacies of designing and implementing a distributed algorithm, we identified two options for proceeding.

The first option was to design and implement the entire distributed algorithm from scratch, which meant designing a distributed computing platform on which the algorithm would run and designing the PSO algorithm to run on this platform. This option was the best in terms of performance. However, it required a significant amount of work on the core components of the distributed system, such as communication, fault tolerance, elasticity, and load balancing. These aspects required plenty of work which was definitely out of scope for the objective of this thesis. Moreover, building a custom solution meant it was more challenging to set up and provision the algorithm for external users.

The second and more feasible option was to research a distributed computing framework that provided several guarantees and features out of the box. These guarantees allowed the algorithm to be designed on top of an existing framework without worrying about most of the complexities involved. Moreover, using a preexisting framework allowed the algorithm to be more easily understood, deployed, and maintained, thanks to the use of existing well-documented technologies.

4.3.3 The Choice of Apache Spark

We decided to opt for the second option and started to search existing distributed computing frameworks. The metrics we decided for evaluating the frameworks were popularity, simplicity, performance, and extensibility. After some research, we have selected three candidates which included Hadoop MapReduce, Ray, and Apache Spark.

The first alternative, Hadoop, was the most popular solution. However, it had two major problems. It leveraged a distributed file system that slowed all operations in the orders of magnitude and, most notably, modeled computations around *map-reduce* phases, which did not fit the PSO algorithm (the PSO algorithm did not need any reduce phase).

The second alternative was Ray, a relatively new framework built to parallelize most Python/Java workloads with a flexible API. Ray had numerous primitives for fine-grained control of the distributed computations. However, this would have increased the complexity and extensibility of the solution.

The last framework and our final choice was Apache Spark. Spark is a popular distributed computing framework widely used for large-scale data processing and can be easily adapted to several use-cases. We decided to opt for Spark because it was popular in the industry. It offered a remarkably intuitive programming model based on transformations, actions, and shared variables. It provided excellent performance because in-memory data structures such as RDDs were reasonably straightforward to extend and maintain. Another practical reason for choosing Spark was its support for Kotlin, a programming language that we decided to use to implement the algorithms. In the end, Spark also satisfied the four essential aspects of any distributed algorithm in the following way:

1. **Fault tolerance**: Spark keeps track of the lineage of operations used to construct an RDD in order to react to any fault in the system. Thanks to the lineage, it can recompute any lost partition, effectively making the algorithm fault-tolerant. The only part of Spark that is exposed to faults is the driver program, which must be kept alive for the system to work.

2. **Synchronous and asynchronous**: Spark is based on a synchronous model, in which the driver program and the executors are all synchronized between each other. The synchronization is required because each executor works only on a partition of the data and needs to synchronize with other executors for operations such as collecting and shuffling.

3. **Partitioning and load balancing**: Spark employs a distributed data abstraction called Resilient Distributed Dataset (RDD). An RDD is effectively a collection of elements partitioned across multiple executor nodes that will independently perform deterministic transformations on the data. The load balancing is made by defining the number of partitions in which the collection must be split. Typically the number of partitions is defined statically during the creation of the RDD. However, it can be changed after creation with additional overhead.

4. **Paradigms**: Spark is based on a master/slave paradigm, in which the driver is the master that coordinates the system, and the executors are the slaves that execute the tasks. The master/slave paradigm allows Spark to efficiently distribute the load among the available executors and simplifies external cluster managers' support.

Apache Spark is a popular framework that has many features. However, it does come with some weaknesses. The most noticeable problem is the overhead introduced by Spark itself, which is related to its complex inner workings and operations (e.g., job scheduling, DAG creation, resiliency mechanisms). The other issue is related to the manual optimizations that should be done when implementing efficient algorithms on Spark. These optimizations must be done manually by tweaking the size of partitions, caching strategies, and more. Nonetheless, the benefits of simplicity, performance, fault tolerance, and scalability show that Spark is a valid alternative to hand-made solutions.

In conclusion, the choice of Apache Spark came down to the reasons above and proved to be suitable for our use case. However, it required a lengthy and complex configuration process, especially when using Kubernetes.

4.4 Spark Distributed Synchronous PSO (SDSPSO)

4.4.1 Phases of the Algorithm

The synchronous algorithm proposed in this thesis is the Spark Distributed Synchronous PSO (SDSPSO), a synchronous PSO designed to be distributed with Apache Spark. The SDSPSO algorithm realizes coarse-grained parallelization in which each executor node in the cluster will compute a large task composed of multiple particles whose fitness functions will be evaluated.

The algorithm is divided into three sequential phases: the initialization phase, the fitness evaluation phase, and the velocity/position evaluation phase.

The pseudocode for both variants of the SDSPSO algorithm is shown in Figure A.2 and Figure A.3, where I is the number of iterations, P is the number of particles, N is the number of fog nodes, and M is the number of modules.

Initialization

In the initialization phase, the algorithm randomly initializes the array of particles, creates the best global position accumulator, and initializes the fitness function parameters' broadcast variable. The particles track their current/best position and are all generated in memory without additional external storage. The whole algorithm is completely in-memory for performance reasons. The best global position accumulator is a shared variable that has been implemented to track the best global position across multiple executor nodes. The broadcast variable is used to send and cache read-only parameters used by the executors while evaluating the function.

Fitness Evaluation

The fitness evaluation phase is the most intricate because it is the parallelized core part of the algorithm. The evaluation of the fitness function is distributed across multiple executor nodes working in parallel by leveraging the independence during evaluation. The array of randomly initialized particles is parallelized by Spark via the `parallelize` method in several partitions that are equivalent to the number of cores in the cluster (e.g., two nodes with two cores each will result in four cores, which is four partitions). The array of particles is then mapped with the `map` transformation that takes an RDD of type T and creates a new RDD of type E by applying a closure λ. The closure λ passed to `map` is responsible for evaluating the fitness function of the particle, computing the best global and personal positions, and returning the transformed particle. The closure λ closes over two essential parameters: the accumulator for the best global position and the broadcast variable containing the fitness function parameters.

The accumulator is a shared variable used to compute the best global position. It has been implemented to decrease the communication overhead by reducing the number of times the data is sent between nodes in the cluster. The accumulator developed for this algorithm possesses the associativity and commutativity properties required by Spark and another essential property, idempotence. An accumulator in Spark is updated via the `add` method, which is responsible for adding a value to the accumulator. Because the `add` method will be called by each executor in a non-deterministic order, the associativity and commutativity properties must hold for the data structure used. Otherwise, some inconsistencies might arise. An important thing to note is that the behavior of Spark will be distinct depending on where we call the `add` method. If we call the `add` method from within an action, we have the guarantee that the method will be called only once for each particle. In contrast, if we call it from within a transformation, the `add` method may be called multiple times (e.g., due to node failures). Having idempotency in the accumulator allows the `add` method to be called multiple times with the same parameters without resulting in a diverse final state. The accumulator is copied to each component of the cluster, the driver, and executor nodes. Once the array of particles is parallelized, the empty accumulator is sent to each executor node, which will independently execute the `map` transformation on each particle of the partition and call the accumulator's `add` method. The call to the `add` method will update the local copy of the accumulator, effectively tracking the best global position of the sub-swarm in a specific node in the cluster. We can compute the best global position independently for each partition because, in the evaluation phase, we do not have a dependence between particles. However, for the next phase, we will need to obtain first the best global position among all the sub-swarms. The evaluation of the best global position is achieved after all the particles are collected back into the driver's main memory via the `collect` action. Right after the collection, the `value` method is called on the accumulator, resulting in a merge between all the accumulators from the executors and the driver, effectively deriving the best global position of the entire swarm. The pseudocode that defines the essential functions of the accumulator is shown in Figure A.4.

The additional essential parameter of the closure λ is the broadcast variable, which contains read-only data required during the fitness function evaluation. The broadcast variable has been used to speed up the transfer of data from the driver to the executors and reduce the amount of data exchanged between the two entities. Indeed, the broadcast variable is broadcasted efficiently to each executor before starting the iterations and is kept on the executors during the whole execution of the algorithm.

Velocity and Position Update

The last phase of the algorithm is the velocity/position update phase, in which the velocity and position of each particle are updated according to the traditional PSO formula. This phase has been implemented in two variants, namely local and distributed:

- **Local Update**: the local variant performs the update of the particles entirely in the driver program, which results in faster performance; however, it does decrease the fault tolerance because no resilient executors are used.
- **Distributed Update**: the distributed variant performs the update of the particles by parallelizing the collection of evaluated particles and by calling the `map` transformation on each particle. This implementation is comparable but more straightforward than the evaluation phase because it only requires the best global position to update each particle's velocity and position. The best global position used in the `map` is sent as a broadcast variable to all the executors in order to reduce communication overhead. It is unpersisted before the iteration ends to free up some space. The distributed update allows for a more fault-tolerant algorithm. However, it does introduce some overhead which is especially noticeable for simpler optimization problems.

4.4.2 Conclusion

Once the velocity and position of each particle have been updated, the algorithm will start a new iteration from the evaluation phase, which will evaluate the fitness function of the updated particles again and consequently call the subsequent phases. This process repeats until the number of iterations is reached, or another stopping condition is met. Once terminated, the algorithm will return the best global position.

The implementation of the SDSPSO algorithm can be found in the GitHub repository (`https://github.com/iambriccardo/thesis-algorithms`) and the simplified code in Figure A.5, located in the appendix.

4.5 Spark Distributed Asynchronous PSO (SDAPSO)

The asynchronous algorithm proposed in this thesis is the Spark Distributed Asynchronous PSO (SDAPSO), which is an asynchronous PSO designed to be distributed with Apache Spark. The SDAPSO algorithm realizes both coarse-grained and fine-grained parallelization depending on the configuration parameters, which will make the algorithm more flexible for specific use cases. SDAPSO differs from existing implementations such as the asynchronous PSO [10] because it does not use the parallel scheme of Message Passing Interface (MPI) and is designed around a completely different programming model (offered by Spark). However, it uses the same master/slave paradigm.

The pseudocode for the SDAPSO algorithm is shown in Figure A.6, where I is the number of iterations, P is the number of particles, N is the number of fog nodes, M is the number of modules, and S is the SuperRDD size.

4.5.1 Challenges with the Asynchronous Design

SDAPSO is built on top of Kotlin coroutines, which allow the concurrent execution of multiple Spark jobs on the cluster. In the case of SDAPSO, a Spark job is a parallel computation consisting of multiple tasks executed on the cluster. Each job contains a group of particles whose fitness function is evaluated by the executor nodes. The use of coroutines requires a more careful design of the algorithm fundamentals, especially shared mutable structures such as the best global position.

The asynchronous algorithm is also more complicated than its synchronous counterpart because the programming model of Apache Spark is not suitable for asynchronous use-cases. However, it allows multiple jobs to be executed concurrently on the same cluster. This aspect of Spark required an implementation involving new ideas to balance efficiency and asynchrony.

We wanted to create two coroutines during the first draft of the algorithm. The first was going to schedule Spark jobs, each with an RDD that contained a single particle, and run the fitness evaluation on the cluster, while the second was going to collect the result of each job asynchronously, update the particle's velocity/position locally and send back the particle as a new Spark job. This implementation realized complete asynchrony because each particle was wholly independent of the others and was evaluated as soon as an executor was free. Nonetheless, this design was relatively inefficient because it created several Spark jobs that were executed only for a few milliseconds. Many Spark jobs resulted in significant overhead, especially noticeable on simpler optimization problems. For the reasons above, we had to think about possible ways of reducing the number of jobs while keeping a good level of asynchrony.

4.5.2 SuperRDD Abstraction

In order to solve the efficiency issue, we designed and implemented an abstraction called SuperRDD, which is essentially a collection of particles that are all dependent on each other and are executed on the cluster as a single RDD belonging to the same Spark job.

The idea of SuperRDD arose when thinking that we could have improved the algorithm's efficiency by reducing its asynchrony, that in the context of Apache Spark, it meant grouping together multiple particles under a single RDD that was evaluated by the cluster. The usage of SuperRDDs resulted in fewer Spark jobs to be scheduled but also less asynchrony, as the particles belonging to the same SuperRDD must have waited for all the other particles to be evaluated.

For more flexibility, the size of a SuperRDD can be defined in the interval $[1, n]$ where 1 means complete asynchrony and n no asynchrony at all (n is the number of particles). A SuperRDD of size 1 effectively resembles the first inefficient idea of having one Spark job per particle, where a SuperRDD of size n behaves similarly to the synchronous PSO, in which each particle must wait for all the other particles to be evaluated before proceeding to update its velocity and position. The main difference between SDSPSO and SDAPSO with SuperRDDs of size n is how particles are updated. In the former, we update the particles' velocities and positions after a single best global is computed among all the particles, which results in all the particles updating their velocities and positions with the same best global position. In the latter, we update the velocities and positions with the best global position found until that point. Thus the particles belonging to the same SuperRDD might be updated with different best global positions.

4.5.3 Components of the Algorithm

Due to the concurrent nature of the implementation, it is not possible to strictly define the phases of the algorithm. However, we can identify its major components, including a set of channels, the aggregator, the producer, and the consumer. These components are running concurrently on Kotlin coroutines and are represented at a higher level in Figure 4.3.

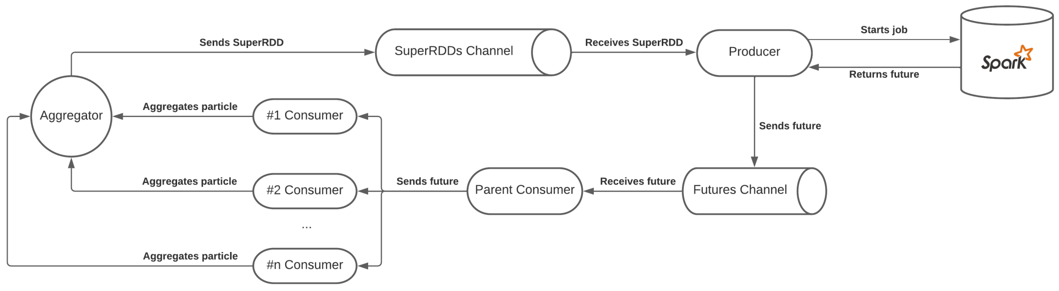


Figure 4.3: Diagram of the components in the SDAPSO algorithm.

Figure 4.3 does not depict the concurrency of the system. However, the aggregator, the consumers, and the producer all work concurrently with respect to each other. Their behavior is synchronized with the use of multiple suspension points that are activated when calling the `send` and `receive` primitives of the respective channel. It is important to note that the coroutines used in the algorithm are dispatched to different threads by the coroutine dispatchers depending on their work.

Initialization

The algorithm is initialized similarly to SDSPSO. The only difference is the absence of an accumulator and the existence of a set of channels and an aggregator, which are all created during initialization. All the randomly initialized particles are sent to the aggregator, which will be responsible for creating SuperRDDs of a given size. The algorithm also performs an automatic adaptation of the final number of particles based on the SuperRDD size and an initial number of particles (e.g., ten initial particles and six particles per SuperRDD will result in twelve final particles). This adaptation makes the number of particles a multiple of the SuperRDD size, resulting in a more straightforward design of the aggregator, which will not deal with undersized SuperRDDs.

The formula that computes the optimal number of particles is the following, where n is the initial number of particles, s is the SuperRDD size, and n^I is the final number of particles:

$$n^I = n + \begin{cases} 0, \text{ if } n \bmod s = 0 \\ s - (n \bmod s)\ , \text{ otherwise} \end{cases} \qquad \text{where } 0 \leq s \leq n \tag{4.1}$$

Set of Channels

The set of channels contains two channels used for synchronization and communication between the producer and consumer coroutines. The first channel is responsible for carrying the SuperRDDs ready to be executed on the cluster. The second is responsible for maintaining references to the launched Spark jobs using Java's futures.

A channel in Kotlin is similar to a blocking queue in Java. It is defined as `Channel<T>` where `T` is the type of the elements in the channel [39]. Channels are used in the SDAPSO algorithm as buffers with the added benefit of working with suspending primitives like `send` and `receive`. These primitives allow the calling coroutines to suspend their execution when calling these methods in case of an empty or full buffer, effectively avoiding busy waiting and improving thread utilization.

Aggregator

The aggregator is a fundamental component of the algorithm because it is responsible for constructing SuperRDDs of a given size. An aggregator exposes the method `aggregate` which accepts a particle and aggregates it with the existing particles. When the number of particles in the aggregator reaches the size of the SuperRDD, the aggregator will construct a SuperRDD and send it to the producer of Spark jobs, which will, in turn, create an RDD that will be submitted to the cluster.

The aggregator is implemented with an actor. An entity made up of a coroutine, an encapsulated state, and a channel to communicate with other coroutines [40]. The usage of an actor simplifies the management of a shared mutable state, which is always a complex problem to solve in concurrent programs because it must consider several aspects such as synchronization of access, performance, and thread-safety. Actors have also been used to manage the algorithm's state, which includes the best global position.

Producer

The producer is a coroutine responsible for executing SuperRDDs on the Spark cluster. It waits for new SuperRDDs to be received over the respective channel. Once received, it will parallelize the array of particles in the SuperRDD, effectively creating an RDD. The parallelization is done through the `parallelize` method with several partitions that are equivalent to the number of cores in the cluster.

The array of particles is then mapped with the `map` transformation that takes an RDD of type T and creates a new RDD of type E by applying a closure λ. In this algorithm, the closure λ is responsible for evaluating the fitness function of the particle, computing the best personal position, and returning the transformed particle. Differently from the SDSPSO algorithm, the closure λ of the `map` closes over one parameter, namely the broadcast variable containing the fitness function parameters. The absence of the accumulator is related to the fact that the algorithm does not need to compute the best global position after all the particles have been evaluated. Instead, it computes it directly after each particle of each SuperRDD is received.

Another significant difference from the SDSPSO is that SDAPSO uses the `collectAsync` action, which is a non-blocking variant of the `collect` action that returns a `Future<T>` object that contains a computation that will be completed in the future. The result of type `T` will be returned once the Spark job finishes and can be obtained by calling the blocking method `get` on the future object. The future returned by the `collectAsync` action will be sent in the futures channels in order for it to be collected by the consuming coroutines.

Consumer

The consumer is a coroutine responsible for collecting a precise number of futures from the respective channel. The number of futures to be collected is:

$$r = \frac{n \times i}{s} \tag{4.2}$$

where r is the number of futures collected, n is the number of particles, i is the number of iterations, and s is the SuperRDD size.

The idea of the formula 4.2 is to obtain a similar number of particle evaluations to the synchronous variant of the algorithm, even though there is no concept of iteration in the asynchronous algorithm. For example, ten particles and ten iterations will result in one hundred evaluations which for a SuperRDD of size five will result in twenty SuperRDDs to be evaluated. This formula has been designed to have a stopping condition that would allow us to compare the synchronous and asynchronous algorithms more easily. Nevertheless, the stopping condition can be changed easily depending on the needs.

The collection of the futures will be accomplished in parallel by a series of consuming child coroutines created by the parent consumer coroutine running on the main thread. The parent will dispatch r child coroutines with the `Dispatchers.IO` dispatcher and will wait for all of them to finish before ending. The `Dispatchers.IO` will schedule the child coroutines on a thread pool specialized for IO intensive operations, resulting in a parallel consumption of the r futures. For each received future, the respective child coroutine will call the `get` method and block until the Spark job tied to that future has been completed. After completion, the child coroutine iterates for each particle transformed by the Spark job computes the best global position, updates each particle's velocity/position, and sends each particle back to the aggregator by calling the `aggregate` method.

4.5.4 Conclusion

After r futures are consumed, the algorithm will terminate by cleaning up the remaining futures and coroutines. The cleanup is done to avoid leaking futures scheduled by previous child coroutines but are not consumed because r futures have already been handled. Once the cleanup is completed, the algorithm will return the best global position.

The implementation of the SDAPSO algorithm can be found in the GitHub repository (`https://github.com/iambriccardo/thesis-algorithms`) and the simplified code in Figure A.7, located in the appendix.

4.6 Integration with Apache Spark

The design of the algorithms has been done by leveraging Apache Spark's API for Java, which is interoperable with Kotlin and contains all the required abstractions to work with the programming model of Spark. Some of these abstractions include all of the transformations (e.g., `map`, `filter`), actions (e.g., `collect`) and most importantly the `SparkContext`. The `SparkContext` represents the connection to a Spark cluster and can be used to create RDDs, accumulators, and broadcast variables on that cluster.

In order to run an application on Spark, two essential things are needed: the first is an instance of Apache Spark running either on a single or on multiple machines. The second is a `.jar` file containing the code of Spark's application. In our case, the Spark application is a `.jar` that contains all the algorithms proposed in this thesis and can change the running algorithm depending on configuration parameters.

Once the previous requirements are met, the Spark application can be submitted. Even though we used Kubernetes with Spark Operator in our evaluation, which abstracts the submission of Spark applications, for this chapter, we will explain how to run a Spark application with the default method. To submit an application to Spark, one can use the built-in script called `spark-submit.sh` which takes as input the main class of the driver program, the `.jar` file containing the code of the application and other configuration parameters that are specific to Spark (e.g., connection type, deploy mode, memory configurations). The script will automatically launch the Spark driver. Through the cluster manager, it will request a specific number of executors to which it will dispatch tasks generated by the Spark application. In our case, the tasks will perform the fitness evaluation and the velocity/position update (the latter only in the case of SDSPSO with DU). An example usage of `spark-submit` is shown in Figure 4.4.

```
spark-submit --class MainKt --master local[*] dpso.jar
```

Figure 4.4: Example usage of `spark-submit` with a `.jar` file.

While submitting the Spark application, two different deployment modes can be chosen: *cluster mode* or *client mode*. The difference is that the driver program will be allocated in any of the worker machines available in the cluster with the cluster mode. In contrast, in the client mode, the driver will be allocated in the machine submitting via `spark-submit`. The client mode generally allows easier debugging due to the local execution. However, it is not recommended for production use.

In conclusion, the steps for executing the algorithms are:

1. Setup a running instance of Apache Spark.
2. Compile a `.jar` file containing the code of the Spark application.
3. Run the `spark-submit` command with specific configuration parameters.
4. Monitor the application's execution through the Spark dashboard.
5. Obtain the results at the end of the execution.

A high-level diagram of the whole architecture is shown in Figure 4.5.

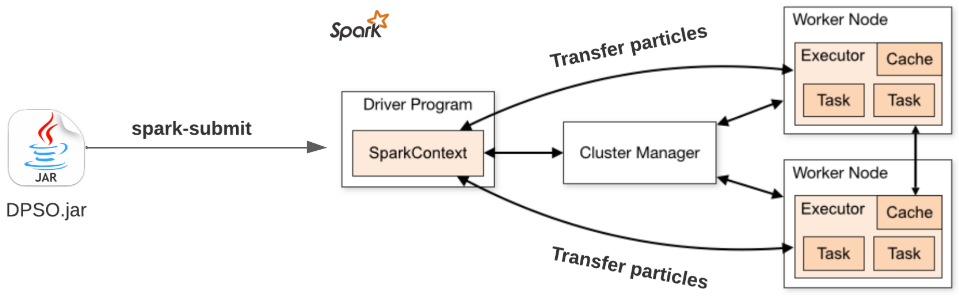


Figure 4.5: How the `.jar` containing the algorithms is submitted to Spark.

Chapter 5

Evaluation and Discussion

This chapter explains the evaluation of the distributed algorithms we have developed. The focus of the evaluation is the performance of the algorithms in terms of elapsed time. However, we also considered the fault tolerance and scalability of the algorithms via an assessment of the platforms used.

5.1 Cluster Setup

In order to provide a production-like execution environment for running the evaluations, we needed to explore possible alternatives and how to implement the most suitable for our use case. We executed the algorithms with a local version of Spark during the development for simplicity reasons. This approach used the standalone cluster manager directly on the host OS. However, it would not scale well in a production environment.

As explained in the background chapter, we decided to use the Kubernetes cluster manager for running Spark. We will explain in this chapter how the cluster has been set up to run the evaluations. It is important to note that the cluster setup for this evaluation is virtualized. Thus it is different from a possible real-world application on a cluster of edge nodes. The difference will be mainly in topology and configuration parameters. However, the usage of Kubernetes and Spark Operator would remain unchanged.

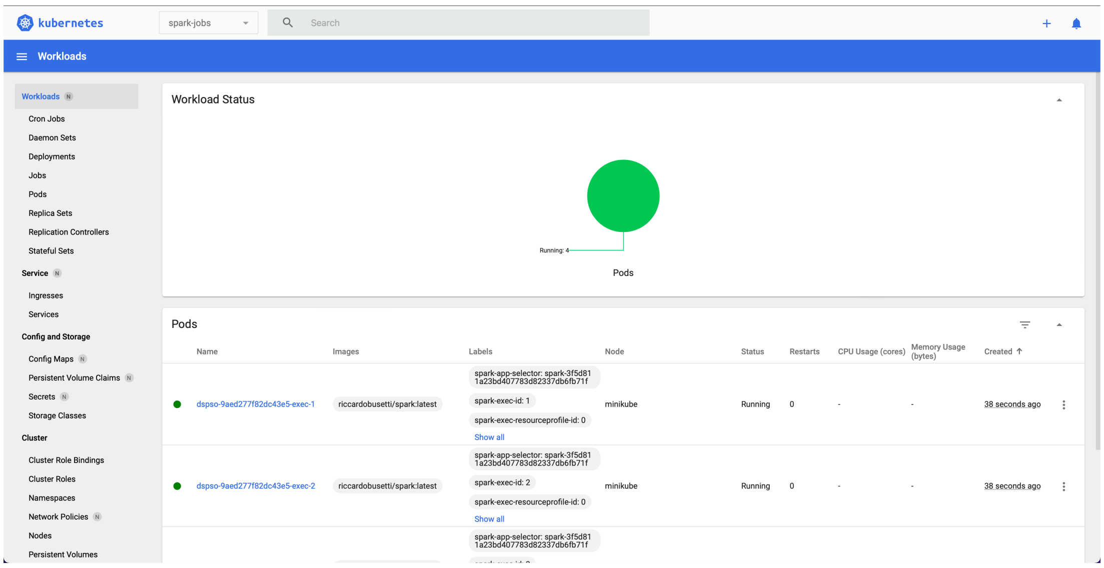
5.1.1 Kubernetes Cluster

The evaluation of the distributed algorithms has been accomplished by executing the algorithm in a virtualized Kubernetes cluster. The usage of Kubernetes was motivated in the background chapter. It resulted in the best choice in terms of simplicity, scalability, and extensibility. Working with containers facilitated the deployment of the algorithms. Moreover, abstractions such as pods streamlined the management of the applications running within the cluster. For example, actions such as starting and destroying running applications were as simple as running a single command in the shell.

In addition to the benefits above, Kubernetes had two other features that have been implemented into the cluster and ultimately influenced our decision:

- **Persistent volumes**: in order to store data in persistent storage, Kubernetes offers an API that enables safe and easy access to specific volumes. These volumes are an abstraction over the actual storage, that can be a centralized file system, a distributed file system, or even a network file system. Each volume has a specific size and can only be accessed via a persistent volume claim, a claim made by a pod that specifies the characteristics of the volume it needs (e.g., size required, type of volume). In our algorithms, we have used persistent volumes to write the results to the file system of the master running the driver program.

- **Dashboard**: observability is a fundamental attribute of any distributed system, and this also applies to SDSPSO and SDAPSO. Running the algorithms in a cluster demands a general overview of the progress, possible problems, and even critical errors. Kubernetes offers an intuitive dashboard whose goal is to get a holistic view of the system while giving the ability to dig as deep as accessing the shell of each running container. In our algorithms, we have used the dashboard extensively to monitor the progress and possible failures of the algorithms. A screenshot of the Kubernetes dashboard can be seen in Figure 5.1.



Pods

	Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↑
●	dspso-9aed277f82dc43e5-exec-1	riccardobusetti/spark:latest	spark-app-selector: spark-3f5d81 1a23bd407783d82337db6fb71f spark-exec-id: 1 spark-exec-resourceprofile-id: 0 Show all	minikube	Running	0	-	-	38 seconds ago
●	dspso-9aed277f82dc43e5-exec-2	riccardobusetti/spark:latest	spark-app-selector: spark-3f5d81 1a23bd407783d82337db6fb71f spark-exec-id: 2 spark-exec-resourceprofile-id: 0 Show all	minikube	Running	0	-	-	38 seconds ago
			spark-app-selector: spark-3f5d81 1a23bd407783d82337db6fb71f						

Figure 5.1: Kubernetes dashboard running the SDSPSO algorithm.

5.1.2 Minikube

In order to run a virtualized Kubernetes cluster, we chose to use minikube, which implements a local Kubernetes cluster on macOS, Linux, and Windows. Minikube's primary goals are to be the best tool for local Kubernetes application development and to support all Kubernetes features that fit.

Minikube runs a single-node Kubernetes cluster on a personal computer. However, it supports multiple virtual nodes within the cluster. In the context of Apache Spark, Kubernetes can deploy multiple pods in the same virtual node, which all act as independent executors from Spark's perspective. Our evaluation decided to use only one virtual node in minikube due to some problems regarding volume mounts that are not supported for multiple virtual nodes. The ending result is similar because, with multiple virtual nodes, the only difference is that pods might be scheduled on different nodes. However, it is the same from Spark's perspective because Spark sees pods as executor nodes, and these pods might even be scheduled on the same virtual node. This difference must be considered in a real cluster because of the communication overhead introduced by running multiple pods on different nodes.

5.1.3 Spark Operator

Spark has added the support of Kubernetes since version 2.3, which uses the native Kubernetes scheduler. The traditional way of using Spark with Kubernetes is to use the `spark-submit` command to submit a Spark application to a Kubernetes cluster. In this case, Spark will create the driver pod, which will create the executor pods. Once the execution is finished, the executor pods will be destroyed, and the driver will remain alive. This mechanism is depicted in Figure 5.2.

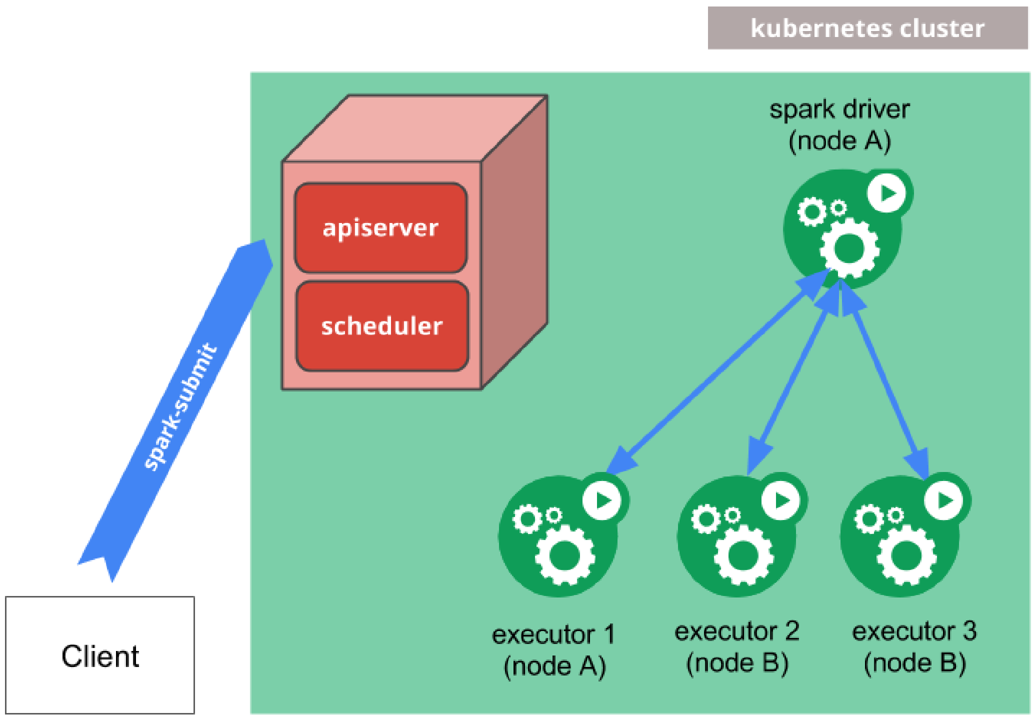


Figure 5.2: Architecture of Apache Spark with Kubernetes and `spark-submit`. [41]

The process of submitting the Spark application via `spark-submit` is relatively straightforward. However, it offers an unnatural interface to Kubernetes because the whole interaction is abstracted behind the command and does not allow any Kubernetes-specific configuration. We were not convinced by the default Kubernetes integration, therefore we decided to search for alternatives and found an attractive one designed by Google.

The alternative is a project called Spark on K8S Operator and was initially developed by the Google Cloud Platform team, which decided to open source the project later on

GitHub. Spark Operator implements the operator pattern that encapsulates the main logic of running Spark applications in custom resources and defines custom controllers that operate on those custom resources. The Operator offers a declarative API by combining custom resource definitions with custom controllers. This API allows the developer to declare the Spark application's desired state, such as volume mounts, number of cores per driver/executor, and number of executor pods. The declaration of the state, as mentioned earlier, is done through a `.yaml` file, which is submitted and converted to a Spark Application Object. The Operator handles this object to achieve and maintain the previously declared state. Underneath, the Operator is using `spark-submit` to submit the application to the apiserver. Therefore the behavior from that point on will be the same. The inner workings of Spark Operator are shown in Figure 5.3.

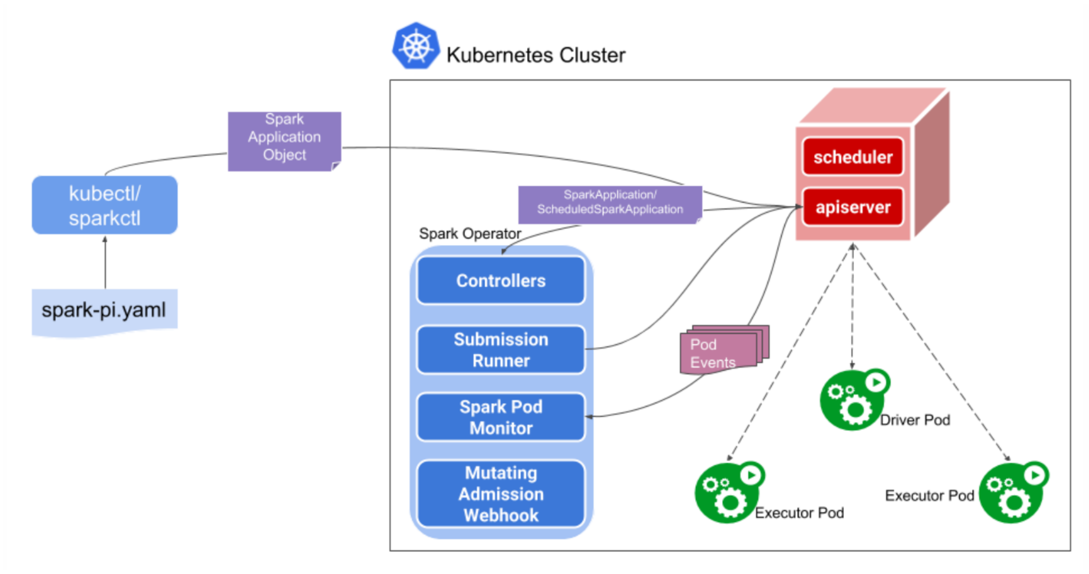
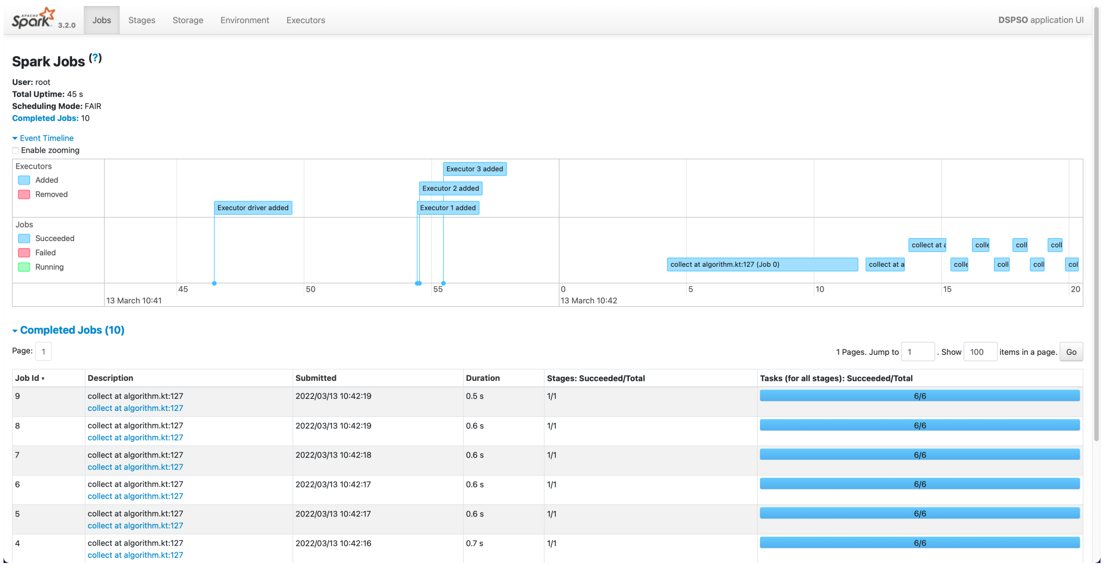


Figure 5.3: Spark Operator architecture. [42]

The choice of Spark Operator has enabled us to effortlessly send several Spark applications to the Kubernetes cluster, which proved to be essential while performing different evaluations of multiple algorithms. However, the whole setup of Spark Operator resulted in a significant amount of work due to the issues that arose because of the lack of documentation and support.

In order to monitor the Spark application, we also exposed the Spark UI to the outside via Kubernetes proxy. The proxy enables the binding between the localhost address to the apiserver of Kubernetes. The access to the dashboard allowed us to monitor the progress of the algorithm. Also, it helped identify possible performance bottlenecks in the Spark jobs. A screenshot of the Spark dashboard can be seen in Figure 5.4



Spark Jobs (?)

User: root
Total Uptime: 45 s
Scheduling Mode: FAIR
Completed Jobs: 10

Completed Jobs (10)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
9	collect at algorithm.kt:127 collect at algorithm.kt:127	2022/03/13 10:42:19	0.5 s	1/1	6/6
8	collect at algorithm.kt:127 collect at algorithm.kt:127	2022/03/13 10:42:19	0.6 s	1/1	6/6
7	collect at algorithm.kt:127 collect at algorithm.kt:127	2022/03/13 10:42:18	0.6 s	1/1	6/6
6	collect at algorithm.kt:127 collect at algorithm.kt:127	2022/03/13 10:42:17	0.6 s	1/1	6/6
5	collect at algorithm.kt:127 collect at algorithm.kt:127	2022/03/13 10:42:17	0.6 s	1/1	6/6
4	collect at algorithm.kt:127 collect at algorithm.kt:127	2022/03/13 10:42:16	0.7 s	1/1	6/6

Figure 5.4: Spark dashboard running the SDSPSO algorithm.

5.1.4 Docker Image

In order to run the Spark application on Kubernetes, there was the need to create a Docker image of Spark. Fortunately, Spark provides a script for automatically building the Spark Docker image. However, the Docker image also requires a `.jar` file that contains the code of the Spark application we want to run. There are two ways of accessing the `.jar` file; the first is to include the `.jar` into the Docker image, and the second is to load the `.jar` from an external directory outside of the container. Due to the complexities involved with mounting volumes and the static nature of the `.jar`, we decided to opt for the first solution. We built a script that automatically created a fat `.jar` with all the required dependencies, built the Spark Docker image with the inclusion of the `.jar` and later pushed the image on the Docker Hub registry.

The `.jar` file contained all the code for the algorithms that we evaluated, namely traditional PSO, SDSPSO with local velocity/position update (LU), SDSPSO with distributed velocity/position update (DU), and SDAPSO. In addition, it supported several parameters that could be tweaked to apply changes to the application without rebuilding the Docker image (e.g., type of the algorithm, iterations, particles, result path).

In addition to Spark's Docker image, we had also to build our own Docker image for the Spark Operator. Our image was created because of some problems regarding the existing pre-built images by Google, which were not up-to-date and not functioning correctly.

5.1.5 Cluster Resources

The configuration of the experimental cluster is shown in Figure 5.5. It has been perfected after a series of tests. The Kubernetes cluster was provisioned on a MacBook Pro 2017 using minikube. The configuration of minikube was set to use eight virtual CPUs and 8192MB of RAM to fully utilize the power of the machine hosting the virtual cluster.

Element	Value
System	2,8 GHz Quad-Core Intel Core i7
Memory	16GB
Operating System	macOS 12 Monterey
Minikube CPUs	8
Minikube RAM	8GB
Minikube Nodes	1

Figure 5.5: Resources of the experimental cluster.

5.1.6 Cluster Topology

The cluster topology is composed of a set of four pods that are scheduled on a single minikube virtual node. One pod is used as the driver and has one core, whereas the remaining three pods are executor nodes with two cores each. It is essential to understand that in Kubernetes, one CPU is the same as one core. Therefore in total, we have used seven CPUs/cores. We could not utilize all eight cores provisioned because the remaining core was taken by the pods concurrently running, such as the Spark Operator, Kubernetes dashboard, and other supporting services. An overview of the cluster topology is available in Figure 5.6.

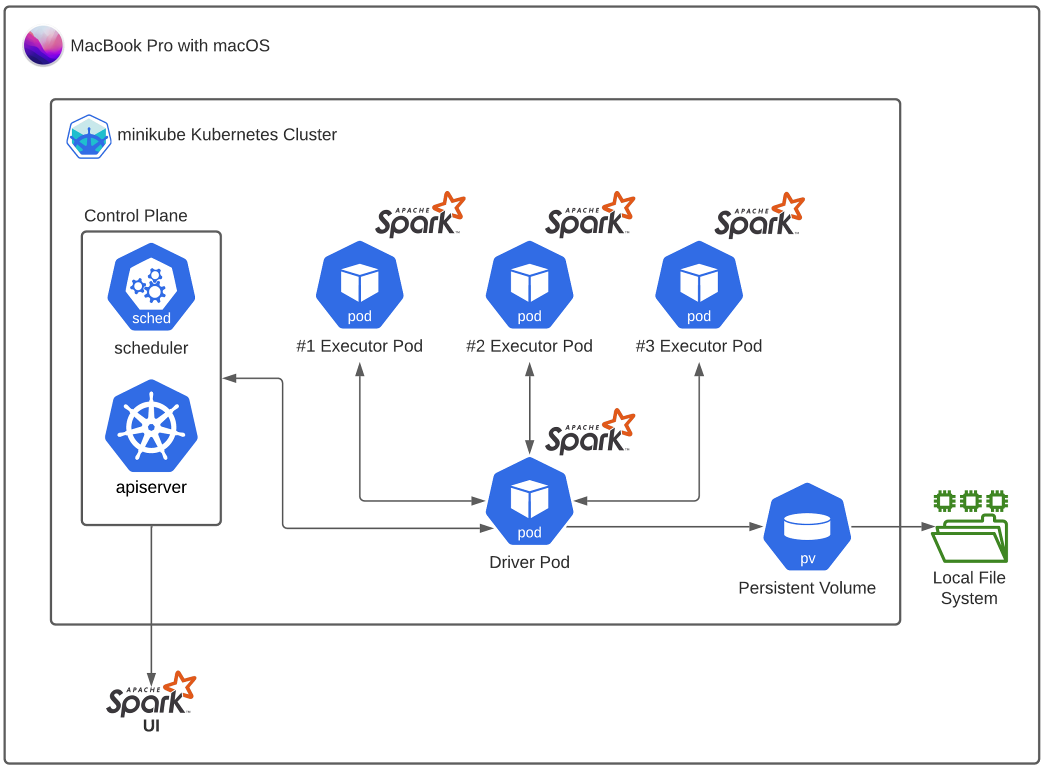


Figure 5.6: Architecture of the cluster setup for the evaluation.

The decision to have multiple cores and fewer executor nodes was motivated considering that, in this case, broadcast variables are copied to fewer executors. The copying to fewer executors happens because if multiple partitions are executed on different cores of the same machine, then Spark will perform multi-threading. Thus the broadcast variable will be shared among multiple tasks executed on different threads of the same machine. The benefit of sharing a broadcast variable between threads is more noticeable when the broadcast variable's content is significant and occupies a non-negligible amount of memory. In addition, a broadcast variable must be serialized and deserialized during transfer between the driver and executors, which is an expensive operation that depends on the size of the data to be transferred and the network speed.

5.2 Evaluation Methodology

5.2.1 Evaluation Goals

The goal of the evaluation is to understand whether the distributed algorithms are more performant than the non-distributed counterpart. The evaluation of the performance has been done by considering the elapsed time needed for the execution. This metric has been chosen because the time taken to find an optimal value is critical for most of the use cases employing the PSO algorithm.

5.2.2 Evaluation Environment

All the tests were performed on the MacBook Pro by trying to reduce as much as possible the interference of other processes concurrently running in the system. The interference reduction was achieved by shutting down most of the CPU/IO-consuming processes and cleaning up the RAM after each run. Even though this solution was not perfect, it helped reduce noise within the performance measurements.

In addition, to make the measurements more accurate, we decided to implement a benchmarking abstraction that used the *monotonic clock*. When using a monotonic clock we are guaranteed that the time always moves forward and will not be impacted by variations in time like the clock skew. Instead, suppose we decided to use the standard wall clock. In that case, we could have ended up in situations in which the machine's clock could have jumped forward or backward depending on the time received from the NTP (Network Time Protocol) server.

5.2.3 Evaluated Algorithms

We evaluated and compared the traditional PSO, SDSPSO with local velocity/position update (SDSPSO with LU), SDSPSO with distributed velocity/position update (SDSPSO with DU), and SDAPSO. To test the performance of the algorithms, several scenarios have been developed, in which we decided to increase only one of the algorithm's parameters and keep the other unchanged (e.g., increase only the number of particles). The increase of only one parameter was made to more clearly understand the bottlenecks of the algorithms and how the algorithms react to an increase in complexity on certain parts.

5.2.4 Problem Encoding Used

To test the performance of the proposed algorithms, we decided to implement into our distributed algorithms the workload placement problem encoding from [6]. The encoding implementation required only an adaptation of the data structures, fitness function, and particles' velocity/position update formulas. Most importantly, it did not involve any change in the high-level flow of the distributed algorithms. We decided to use this problem encoding for the evaluation because it is a meaningful application of the distributed PSO in an edge computing environment. The workload placement problem is a significant challenge in edge computing, and for performance reasons, it should be solved right at the edge. The execution of the optimization problem at the edge will bring benefits such as reduced delay and network congestion, which are fundamental parameters in edge computing.

The problem encoding implemented uses a matrix data structure D, which is composed of a set of fog nodes F and a set of modules M. Each cell in the matrix $D[f][m]$ for some module f and fog node m contains either 0 or 1 given the binary nature of the algorithm. The 1 signals that the module m should be placed on the fog node f, whereas the 0 signals the opposite. The fitness function minimizes the maximum number of resources needed by a set of modules if deployed on a specific fog node. Overall, the algorithm's complexity can be approximated to $O(n^2)$ where n is the number of fog nodes and modules. The quadratic complexity becomes problematic for a high number of n, which is likely in the case of large edge networks.

5.3 Evaluation Results

The evaluation that we decided to perform was about the performance of the algorithms. The measure of performance that we chose to analyze was related to the elapsed time spent by the algorithm to perform the optimization problem until the stopping condition was met. The elapsed time is different from the convergence speed, another performance metric representing how quickly the optimization algorithm converges towards an optimal value. Another measure of performance could have been the quality of solutions, but this was not the goal of this thesis since we started our designs to improve the speed of the traditional PSO algorithm.

It is necessary to understand that for the distributed algorithms, we did not consider the time taken by Spark to startup and connect. Instead, we did consider the initialization phase, which involved the setup and propagation of shared variables, which is not a trivial operation for Spark.

Based on the observations we made during the first run of the algorithms, the complexity of the encoding used was not complicated enough to obtain significant results. Indeed, the distributed algorithms were performing worse than the non-distributed counterpart. The slower performance was already expected during the first drafts of the distributed algorithms because the overhead introduced by Spark is reasonably evident for simpler optimization problems. Due to the low complexity of the encoding, we experimented by simulating a longer-running fitness function evaluation with the introduction of an artificial delay. After some experimentation, we decided to introduce a 50ms delay in the fitness function, in addition to the normal computation, which simulated an average fitness function evaluation duration. The problem with this delay is that it will be the same even if the cluster has nodes with different computational capabilities, which is not the case for our evaluation but should be noticed. The delay was also introduced because we wanted the fitness function evaluation to be several orders of magnitude slower than all the other parts of the algorithm, which was not the case for the problem encoding that we implemented for the evaluation.

In conclusion, because the distributed algorithms are parallelized around the fitness function evaluation, it becomes less convenient to use them if the fitness evaluation is not the most time-consuming part of the optimization (by a significant margin). In that case, we will only gain fault tolerance and scalability, but we will get a penalty in performance.

5.3.1 Particles Increase Benchmark

The first evaluation considered the four algorithms executed with an increasing number of particles and the other parameters fixed. This experiment aimed to understand the effect of increasing only the number of particles because each algorithm handles particles differently, and we wanted to check if these technical differences impact performance.

The experiment setup was the following:

- # particles = 10 (rounded 12), 50 (rounded 60), 100 (rounded 108), 200 (rounded 200), 500 (rounded 504), 1000 (rounded 1000)
- # iterations = 10
- # fog nodes = 10
- # modules = 10
- 50ms fitness function artificial delay
- SuperRDD size = 6 (for # particles ≤ 10), 12 (for # particles ≥ 50)

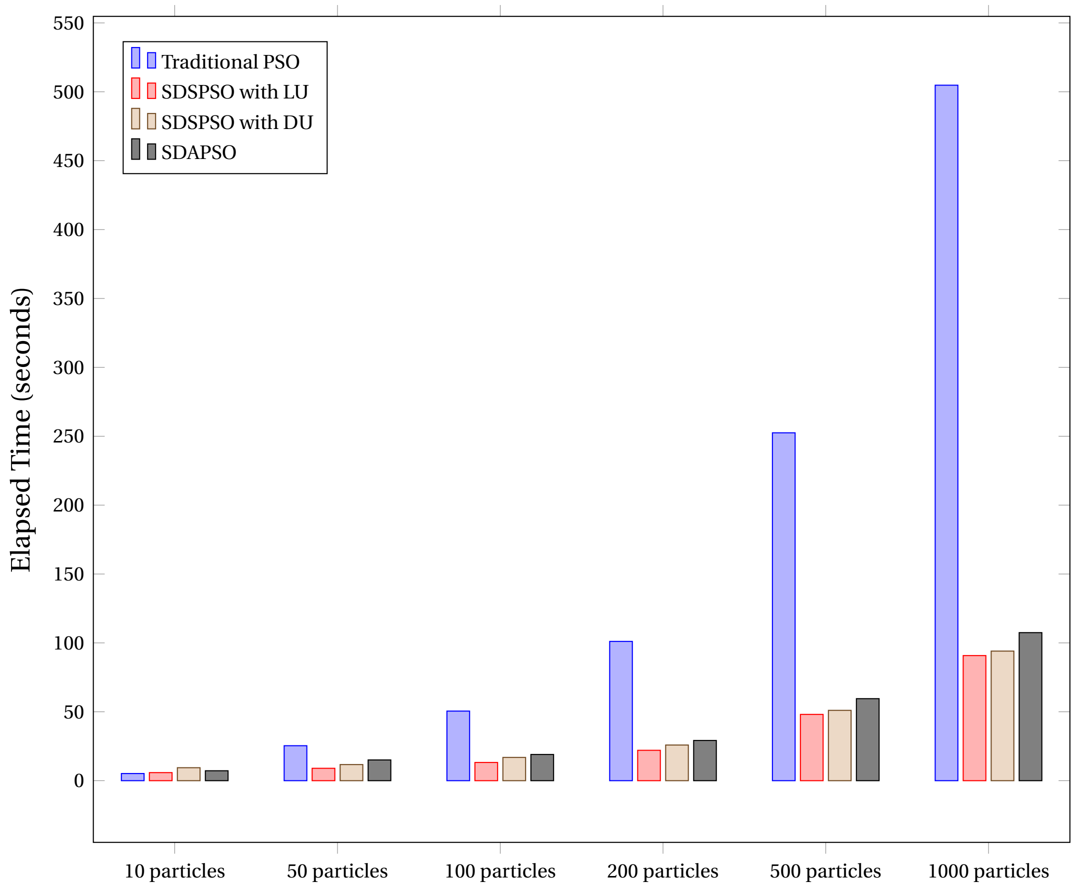


Figure 5.7: Performance evaluation with change in the # of particles.

The results from Figure 5.7 indicate that for 10 particles, the distributed algorithms all perform marginally worse than the traditional PSO algorithm. This performance difference was expected since the overhead introduced by Spark is mainly noticeable for small optimization problems. On the other hand, when the number of particles increases, the distributed algorithms become significantly faster than the traditional PSO. On average, the speedup obtained is 5x, which is correlated to the number of CPUs used, in this case, six (three pods with two cores each). This correlation emerges because if we have x amount of work and we distribute it over n executors, we will obtain, in the best case scenario, a uniform distribution of $\frac{x}{n}$ amount of work per executor.

The differences between the distributed algorithms are more difficult to notice. However, in general, between SDSPSO with LU and DU, there is a small performance gain for the LU variant due to the in-memory update of particles, which is faster despite being less fault-tolerant. The SDAPSO algorithm turns out to be the slowest of the distributed algorithms because the synchronous algorithms produce one job or two jobs per iteration, depending on the variant. In contrast, the asynchronous algorithm produces a number of jobs directly proportional to the number of particles and iterations, thus introducing additional overhead. An important thing to note is that SDAPSO used more particles than the other algorithms due to the rounding performed by the algorithm.

5.3.2 Iterations Increase Benchmark

The second evaluation considered the four algorithms executed with an increasing number of iterations and the other parameters fixed. The goal of this test was to check whether more Spark jobs generated by the synchronous algorithms will impact their performance. This impact on performance could happen because we create one or two jobs per iteration in the synchronous algorithms, ultimately resulting in more Spark jobs.

The experiment setup was the following:

- # particles = 20 (rounded 24)
- # iterations = 10, 50, 100, 200, 500
- # fog nodes = 10
- # modules = 10
- 50ms fitness function artificial delay
- SuperRDD size = 6

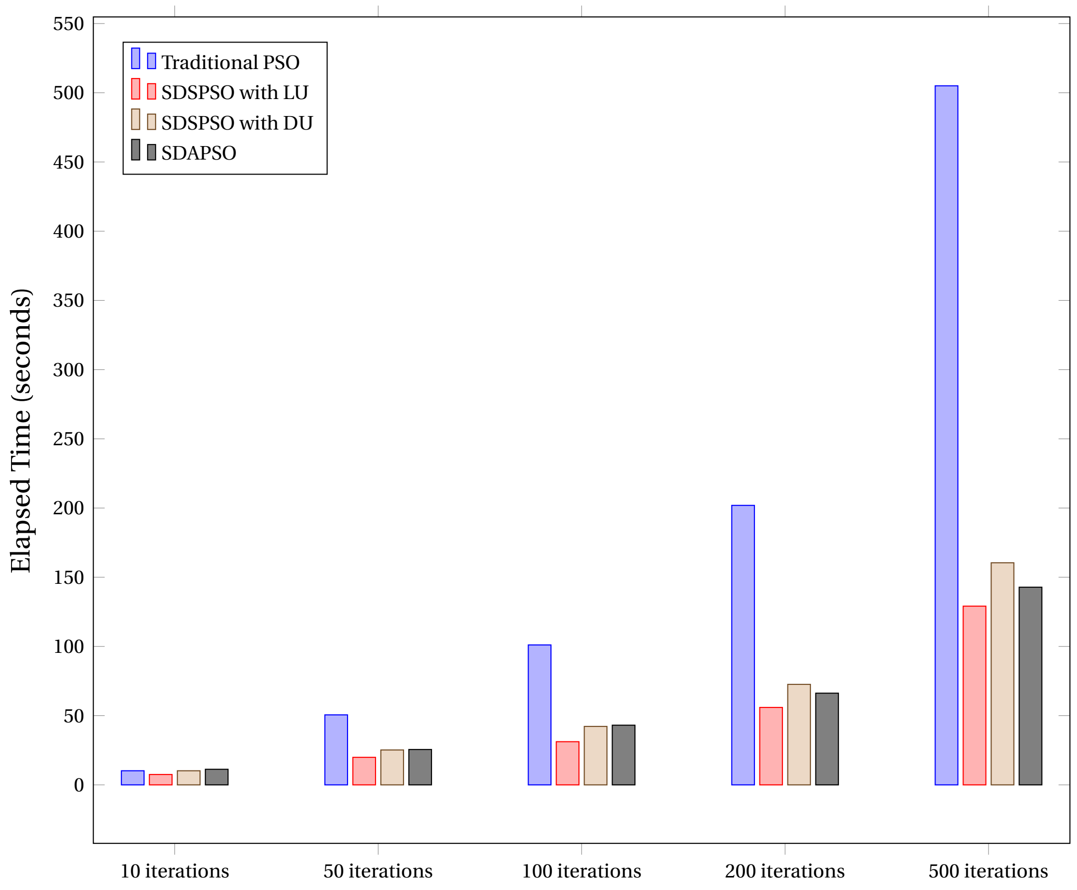


Figure 5.8: Performance evaluation with change in the # of iterations.

The results from Figure 5.8 exhibit similar behavior to the increase in the number of particles. However, a slight difference can be detected. The difference is related to the reduced performance of SDSPSO with DU compared to SDAPSO. This difference in performance was not the case before. However, the number of Spark jobs increased significantly with the increase of iterations. For example, with 500 iterations, the number of Spark jobs reached 1000 in the case of SDSPSO with DU, whereas before, with 10 iterations was only 20 jobs. This performance difference is not significant but repeatedly shows how the overhead of Spark must be taken into consideration while selecting the algorithm.

In this test, we obtained, on average, a 5x speedup in the distributed algorithms compared to the traditional PSO, which is a comparable result to the test executed before. A similar speedup demonstrates how the algorithms perform equally well, irrespective of increased parameters.

Also, in this evaluation, we must consider the rounding of particles performed by SDAPSO, which will result in more particles to be evaluated.

5.3.3 Dimensionality Increase Benchmark

The third evaluation considered the four algorithms executed with an increasing number of fog nodes and modules. This test aimed to understand if the increase in problem dimensionality affected the algorithm, primarily because broadcast variables and accumulators propagate data tied to the problem's dimensionality. For example, the accumulator keeps track of the best global position, including the entire placements matrix and the error.

The experiment setup was the following:

- # particles = 20 (rounded 24)
- # iterations = 10
- # fog nodes = 5, 10, 20, 50, 70, 100, 200
- # modules = 5, 10, 20, 50, 70, 100, 200
- 50ms fitness function artificial delay
- SuperRDD size = 6

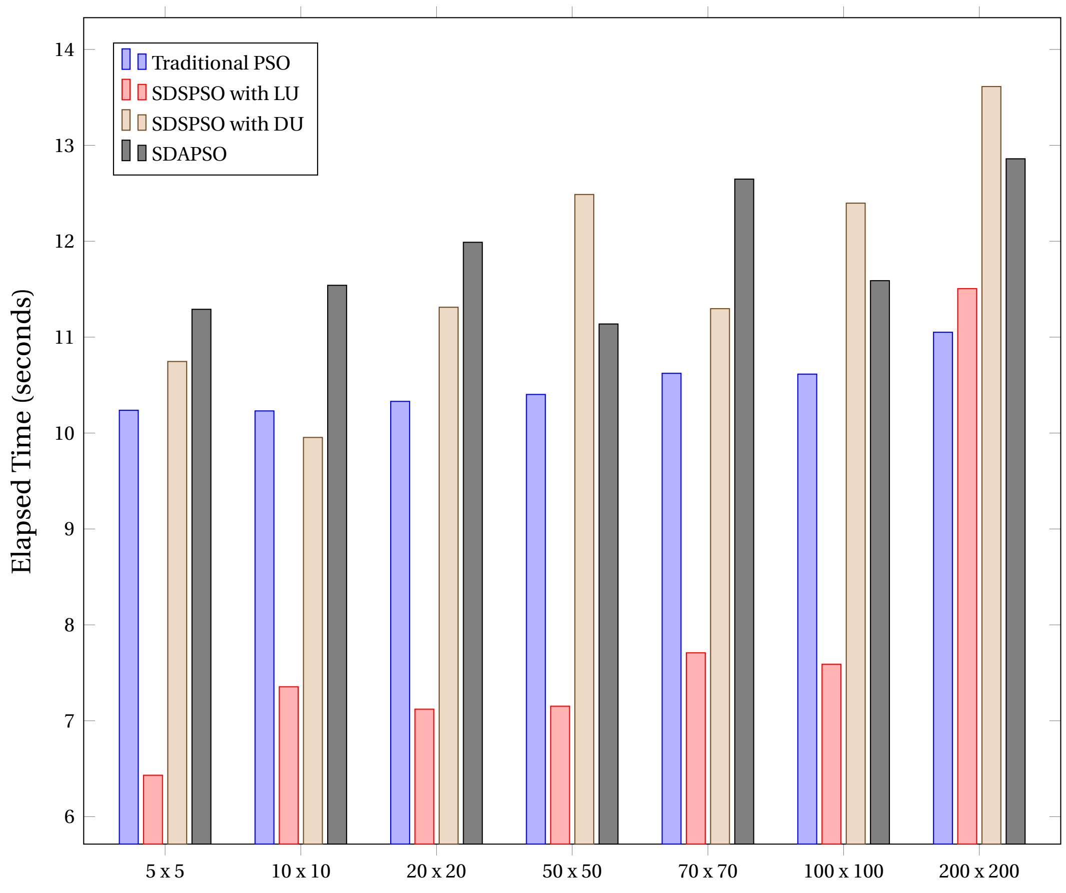


Figure 5.9: Performance evaluation with change in the # of fog nodes and modules.

The results from Figure 5.9 display that the number of fog nodes and modules does, in general, slightly impact the performance of all the algorithms. The impact on performance is more noticeable in the distributed algorithms due to their usage of shared variables and continuous data transfer between nodes, which result in regular serialization and deserialization under the hood. The impact of serialization is more noticeable in the 200 x 200 test, where we can see that the distributed algorithms start to perform worse than the non-distributed counterpart, which is a sign that shows that the overhead of the data transfer is affecting performance.

There is a slight increase in elapsed time across all the algorithms because each phase of the algorithm will take more time due to the quadratically increasing number of combinations to account for. However, it is essential to note that due to the noise within the measurements, some differences in this graph might be coincidental, and because of the scale of the graph, they seem more significant than they are.

The number of particles is also rounded for the SDAPSO algorithm for this evaluation. It is more evident in this experiment due to the minor difference in elapsed time between the algorithms.

5.4 Platforms Assessment

Together with evaluating performance, we had to assess the two other essential requirements that we initially set as goals for the proposed algorithms: fault tolerance and scalability. These two requirements were provided by the platforms that we carefully chose to build and run our algorithms on. We thought they should be assessed in a different section. The platforms we will assess are Apache Spark and Kubernetes, which, as we will see, proved to be good choices to meet our requirements.

5.4.1 Fault Tolerance Assessments

Fault tolerance is defined as the ability of a system to continue working even if some of its components are faulty [25]. In running optimization problems at the edge, it becomes essential to have a fault-tolerant algorithm that can continue its work in case of faults. However, fault tolerance evaluation is rather challenging to do because it does not involve a straightforward metric like elapsed time. Instead, it requires careful understanding of the system's behavior under specific circumstances, such as the failure of a node or an unexpected reduction in network performance. In our case, because we designed the distributed algorithms on top of Apache Spark and Kubernetes, we could leverage their behavior and guarantees to understand the fault tolerance of the algorithms.

Apache Spark Fault Tolerance

Apache Spark provides fault tolerance with Resilient Distributed Datasets (RDDs), extensively used in the algorithms we proposed in this thesis. RDDs are immutable data structures that keep track of their deterministic transformations in the lineage, which is a data structure that enables Spark to reconstruct the RDD in the case it is lost [29]. The RDD, or even a partition, can be lost if an executor node has failed. The lost partition(s) can be recomputed from the original dataset in this situation. The only way to recover the data is to get it again from the source. When a failure occurs, the cluster manager will find out that an executor node is dead, and it will try to reschedule the RDD transformation on another node in the cluster.

A problem with Spark is that in case a failure occurs in the node running the driver program, then the whole state of the application will be lost, resulting in a complete failure of the optimization algorithm. There are solutions to the driver failure, including replicating the driver's state via ZooKeeper. However, they were not implemented in this thesis.

Kubernetes Fault Tolerance

Kubernetes does not offer fault tolerance mechanisms but helps with the system's reliability by providing several features. The control plane of Kubernetes contains the vital components of the platform (e.g., apiserver, scheduler, etcd) that can make global decisions about the cluster, detect, and respond to cluster events. For example, suppose an executor pod crashes. In that case, the control plane will be notified about the failure and reschedule a new pod to be allocated on one of the available nodes in the cluster. However, if a node belonging to the cluster crashes, it may contain multiple executor pods, which will all crash and be rescheduled on another available node by Kubernetes. A failure will result in the cluster operating in a degraded state, leading to new resource constraints.

When running Apache Spark on Kubernetes, an executor's failure could mean that the pod crashed, but the node where it is scheduled is running, or the node hosting the pod is crashed. From Spark's perspective, the vital thing is to have enough executor nodes to transform and shuffle the RDDs. Without executor nodes, the optimization algorithm cannot proceed, thus resulting in a significant or infinite delay if no more executors are provided.

A problem with Kubernetes, which is similar to Spark, is that the node running the control plane is subject to failures, which will destroy the cluster state in case of failure. Also, in this case, we can use a similar technique as with Apache Spark. We can replicate the control plane across multiple nodes so that in case of failure, we can switch to the other node entirely.

5.4.2 Scalability Assessments

Scalability is defined as the ability of a system to handle a growing load by increasing the system resources to provide an equally good service with fewer or more requests. Connected to scalability, elasticity is also an essential property for distributed algorithms. Elasticity extends the definition of scalability by introducing the system's ability to scale down based on the load, effectively creating an elastic system that can increase or decrease its resources based on the demand.

In the case of our distributed algorithms, we have a remarkably elastic system because we can dynamically provide the cluster with the resources that might be needed to carry out the optimization problem within time constraints. Elasticity is provided by Kubernetes, the cluster manager on which Spark is running. It is responsible for managing the driver and executor pods needed by Spark to execute the algorithms. Thanks to the declarative nature of Spark Operator and Kubernetes, we can customize the resources needed by the algorithms based on the cluster topology we have configured. For example, we could have a cluster of five Kubernetes nodes. However, we might need only three pods that can be scheduled on any available nodes by the Kubernetes scheduler.

This elasticity opens up many possibilities, including the provisioning of an edge subnet that is configured as a Kubernetes cluster. Besides the one hosting the Kubernetes control plane, each node in the subnet could execute the Kubernetes client and other independent services concurrently, effectively utilizing efficiently the resources offered by each edge node. When we want to solve an optimization problem, we declare the required resources that we expect we would need, and we submit the application to the Kubernetes cluster. The Kubernetes scheduler will then try to automatically match the current state of the cluster to the declared state of the application.

An example of the Spark Operator declarative `.yaml` configuration can be seen in Figure A.1, located in the appendix, in which we are specifying that we want the driver with one core and three executors with two cores each.

In conclusion, it is clear how effortlessly we can provide the distributed algorithms with the required resources without touching any of the code. In addition to the simplicity of scaling up and down the algorithm, it is also very portable due to the usage of Docker containers. Indeed, the algorithms can be run from any Kubernetes cluster. The only requirement is that the Spark Operator is already installed on the cluster and the Docker image is available.

5.5 Discussion

In conclusion, we wanted to briefly discuss several aspects of the distributed algorithms we have designed and implemented to have a basis for reasoning about possible future improvements.

5.5.1 Further Properties

The main goal of this thesis is to create a variant of the PSO algorithm that is faster but also more fault-tolerant and scalable when executed in an edge computing environment. This objective has been reached, as seen in the previous paragraphs. However, we have also considered other essential properties that characterize the proposed algorithms:

- **Simplicity**: simplicity is the quality of being easy to understand, and even though this property might seem trivial, it is not when designing algorithms. We designed the algorithms with the Kotlin programming language. We tried to build a set of abstractions, using the language's features, that separates the flow of the algorithm from the fitness function evaluation and velocity/position update, which are the most likely to be changed when an encoding is implemented into the algorithm. Therefore we wanted to keep them separated from the main control logic.

- **Portability**: portability is the ability of software to be transferred from one machine to the other. This property is crucial for an algorithm that is meant to be distributed in an edge network, mainly because of the heterogeneous nature of edge nodes. The proposed algorithms are highly portable because they use popular technologies, such as Apache Spark and Kubernetes, supported by many different Operating Systems (OSs). In addition, the use of containers enables the algorithms to be deployed with all of the required dependencies already built into the image, thus streamlining the deployment process.

- **Extensibility**: extensibility is the quality of being designed to allow the addition of new capabilities. This property is strictly connected to simplicity because a simple algorithm is easier to extend. The PSO algorithm is easy to hybridize with other algorithms [7]. The hybridization becomes even more straightforward for possible distributed variants thanks to the Apache Spark programming model.

5.5.2 Discussion for SDSPSO

The SDSPSO algorithm has been designed to tackle the performance problem of the traditional PSO but also the lack of fault tolerance and scalability. In this paragraph, both LU and DU variants will be considered as one. If some details apply only to one of the variants, it will be specifically written.

The algorithm's performance is enhanced with the usage of multiple executor nodes that can evaluate the fitness function in parallel. The performance benefits will be more evident for more complex optimization problems because the overhead introduced by Spark is fairly impactful on simpler problems, leading to potentially slower performance, as shown in the experiments. It is important to note that the increase in performance will be highly dependent on the cluster configuration, which includes the number of executor nodes, the communication channel, the partitioning scheme, and the computational resources of each node. Tweaking the configuration does not imply that the more nodes and partitions we add, the better the algorithm performs, simply because the overhead introduced by Spark will increase. Thus, there must always be a balance between all the algorithm parameters.

An important consideration about performance is that the problem encoding used with SDSPSO must be designed taking into consideration the communication cost, which is significantly more than the traditional PSO algorithm. The most impactful improvement would be to reduce as much as possible the size and amount of data propagated through the cluster (e.g., simpler data structures, more efficient serialization).

The fault tolerance of the algorithm is improved with respect to the non-distributed variant because of the resiliency offered by Spark's RDDs. The RDDs created for the fitness function evaluation and velocity/position update can be recomputed if any failures occur in the executors. The re-execution of the former will produce deterministic results. In contrast, the re-execution of the latter will produce non-deterministic results due to the stochastic parameters used in the velocity and position update formulas. This non-determinism does not lead to any problems. However, it will result in different positioning of certain particles in the swarm. In addition, the DU variant of the algorithm is more fault-tolerant but less performant because of the added collection phase. Therefore, it can be used in contexts where the velocity/position update is more computationally intensive and requires more tolerance to failures.

The only point of failure of the algorithm is the driver program, which manages the whole lifecycle of the algorithm, including its state. Spark does not implement any resiliency mechanisms for the driver program. Therefore, resiliency must be achieved at the node level, the node running the driver program. The failure of the driver program will result in a complete loss of all the data. Thus a solution would be to execute the driver program on a replicated node that can be swapped in case of failures.

The algorithm's scalability is another essential non-functional requirement that SDSPSO achieves. The algorithm can be executed on a cluster that can be scaled depending on several parameters such as network congestion, nodes availability, and problem complexity. For example, we could have ten edge devices that are available to be used for Spark, but they can also do other work in the meanwhile. Before starting the optimization algorithm, we could prepare the cluster using only three nodes for simple optimization problems or scale it to ten for more complex problems. This elasticity allows the best possible usage of existing resources and could be implemented dynamically by observing the state of the edge devices and the complexity of the optimization problem to be executed.

5.5.3 Discussion for SDAPSO

The SDAPSO algorithm has also been designed to tackle the performance problem, lack of fault tolerance, and scalability of the traditional PSO while executed in an edge computing environment. The primary considerations of the SDSPSO algorithm also apply to SDAPSO. However, some additional reflections must be made.

The performance of the asynchronous algorithm is heavily dependent on the SuperRDD size, which means that a smaller SuperRDD will result in the worst performance but more asynchrony. In contrast, a bigger SuperRDD will result in better performance but worst asynchrony. The choice of the size of the SuperRDD strongly depends on the cluster resources, mainly because the asynchronous algorithm is expected to perform better if compared to the synchronous variant when executed on an imbalanced cluster. This performance difference is evident because, in the case of an imbalanced cluster, some nodes might stay idle while waiting for slower nodes to finish the fitness evaluation of each particle in their partition. For this reason, the more the cluster is imbalanced, the fewer particles should be set within a SuperRDD.

In our experiments, we used a balanced virtual cluster, which was not the best environment to show the performance of the SDAPSO algorithm. However, it showed how similar the performance is between all the distributed algorithms in a balanced environment, which is undoubtedly advantageous when deciding which algorithm to use.

The fault tolerance of the algorithm is also on par with the SDSPSO. However, SDAPSO performs more complex computations on the driver program, which might increase the likelihood of driver failure. As we discussed in the consideration for the SDSPSO, the driver's fault tolerance could be eventually improved with a replication of the driver's state across multiple machines. However, this is out of scope for this thesis as it will require significant work from an infrastructure point of view.

The algorithm's scalability remains the same as the SDSPSO algorithm because they both leverage a horizontally scalable architecture built on top of Spark and Kubernetes.

5.5.4 Comparing Synchronous and Asynchronous Designs

In conclusion, the synchronous and asynchronous designs have proved to be reasonable solutions during our evaluations. However, they have slight differences that will play an important role when deciding on the algorithm to use. Due to the primarily non-deterministic nature of distributed systems, it will be challenging to decide upfront which algorithm to use. Therefore real-world testing should be performed.

The synchronous algorithm is simpler and more efficient because it runs a single Spark job per iteration (two in the case of SDSPSO with DU), which translates to less overhead by Spark. After all, it will need to perform fewer optimizations, DAG management, and lineage maintenance. Nevertheless, SDSPSO uses one more shared variable in contrast to SDAPSO, which is the accumulator used to keep track of each best "local" position of each subswarm and compute the final best global position. The usage of an accumulator is undoubtedly an additional overhead. However, in our tests, we did not find it noticeably influential on the performance. SDSPSO is also simpler than the asynchronous design because it is based on strict iterations and clearly defined sequences of operations that do not require any tricky implementation. The simplicity aspect, as stated earlier, is an essential property of both algorithms, especially considering that they need to be adapted each time a new problem encoding needs to be implemented.

On the other hand, the asynchronous algorithm is more complex and, in general, less efficient than the synchronous counterpart. However, it is flexible and suitable for specific use cases. Given that SDAPSO has near to complete independence between particles, depending on the configured SuperRDD size, it can work well in clusters where the resources are imbalanced. An imbalanced cluster is characterized by a heterogeneous set of nodes with different computational capabilities and, therefore, will take different amounts of time to perform the same tasks dispatched by Spark. With the asynchronous algorithm, we can realize fine-grained parallelism, using a small SuperRDD size to have several Spark jobs with fewer particles executed concurrently. When faced with multiple concurrent jobs, Spark will automatically try to optimize as much as possible the available cluster resources. For example, if a node is idle because it is faster than the others, then Spark will start a task that might belong to a different Spark job on the idle node. A problem of SDAPSO, besides the overhead introduced by the high number of Spark jobs, is the complexity of the implementation. The asynchronous algorithm uses a multitude of concurrent primitives and techniques to manage multiple jobs and shared mutable states. Therefore it is more complex to work on and requires a careful modification of the core components.

In conclusion, both variants of the distributed PSO algorithm have advantages and disadvantages. However, due to the multitude of configuration parameters available (e.g., SuperRDD size, number of partitions, nodes in the cluster, cores per node), it is difficult to define the "best" one strictly; therefore, only a real production experience will be able to identify the most suitable variant for a specific use case.

Chapter 6

Conclusions and Future Work

This chapter concludes the thesis with some considerations and discusses the future work related to the algorithms proposed in this research.

6.1 Conclusions

To conclude this thesis, we want to sum up what was achieved. In this research, we proposed two distributed variants of the PSO algorithm implemented on top of Apache Spark, namely SDSPSO and SDAPSO. The SDSPSO algorithm was also proposed in 2 sub-variants, the SD-SPSO with local and (LU) distributed update (DU), to further explore possible variations in the design.

We compared the algorithms above and the traditional PSO by running experiments on a virtual Kubernetes cluster deployed on a MacBook Pro. The objective of the experiments was to analyze the performance of the algorithms in terms of elapsed time, which is an important metric to consider, especially when the optimization problem is executed on resource-constrained nodes in an edge network. In order to study more comprehensively the performance of the algorithms, we decided to perform multiple experiments, each with an increase of a specific PSO parameter (e.g., particles, iterations, dimensionality). The experiments allowed a deeper understanding of the algorithms' behaviors and exposed possible bottlenecks in their implementation. At the end of the experiments, we demonstrated that the distributed algorithms resulted to be more performant than the traditional PSO, obtaining, on average, a 5x speedup. However, even if the distributed algorithms performed better, it was not always the case, especially when the optimization problems were simpler. In that case, the traditional PSO algorithm performed better in elapsed time. However, it did not provide any fault tolerance and scalability. The aforementioned characteristics were also evaluated as part of this research but with a different methodology as opposed to performance evaluations. Due to the multitude of different ways for measuring fault tolerance and scalability, we chose to evaluate these characteristics based on the guarantees and features offered by the platforms on which the distributed algorithms were designed and run.

The algorithms proposed in this thesis demonstrated how complex and error-prone the design of distributed algorithms could be. In addition, they highlighted how using existing distributed computing platforms can significantly simplify the design of the algorithms and enable more portability of the solutions thanks to the use of well-known technologies. The distributed algorithms can provide better performance if appropriately designed, even if executed on a platform with a specific programming model that will constrain several optimization possibilities. In the case of this thesis, Apache Spark did not allow for fine-grained optimizations that a custom-made solution could have offered. However, the trade-off between performance and simplicity/portability resulted in a meaningful choice.

In conclusion, the distributed algorithms proposed in this thesis provide a performant, fault-tolerant, and scalable solution to executing PSO optimization problems in resource-constrained environments like edge networks. The experiments show that the distributed algorithms perform better when optimization problems are larger and more complex. However, they are slower when simpler problems are executed. In addition, the distributed algorithms provide better fault tolerance and scalability with respect to the traditional PSO, which ultimately comes down to more reliability and extensibility when running the algorithms in an unpredictable environment like an edge network.

6.2 Future Work

In future work, we plan to improve the algorithms' performance by tuning Apache Spark and exploring new ideas for designing the algorithms on different platforms. Both of the directions above are independent and aim to provide two new starting points for future work connected to this research.

The tuning of Apache Spark is a complex process that requires extensive experience with the platform and has major optimization points that can be tweaked. The first improvement relates to data serialization, which plays a vital role in any distributed algorithm. A possible modification is the use of more efficient serialization mechanisms, such as the Kyro serializer, which promises a speed up with respect to the Java default serialization mechanism of about 10x. However, it does not support all the serializable types. The second improvement is memory usage, which is especially critical when running the algorithms on resource-constrained devices like edge nodes. The main aspects of memory usage in Spark are the amount of memory used by a specific object, the cost of accessing that specific object, and because Spark runs in the JVM, the overhead introduced by the garbage collector. The previous aspects can all be tweaked with better design of the data structures, garbage collection tuning, and in general, a reduction of the space used by the data. Additionally, other improvements can be made on Spark, such as parameter tuning, change in the level of parallelism, and consideration of data locality. However, each of these points would require a thesis on its own.

Exploring other distributed computing platforms is also something we would like to do because it helps identify the most suitable one for an edge computing environment. Evaluating platforms by looking at their inner workings and documentation is not enough. Instead, we should try to implement the distributed algorithms on those platforms to measure the performance, fault tolerance, and scalability effectively. For example, we could design the distributed algorithms on platforms such as Ray and compare their performance with the algorithms implemented on top of Apache Spark.

Appendix A

Code Excerpts

This appendix presents the pseudocode of the algorithms, their simplified implementation in Kotlin, and a sample `.yaml` configuration file for the Spark Operator.

```yaml
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
  name: sdspso-du
  namespace: spark-jobs
spec:
  type: Scala
  mode: cluster
  image: "riccardobusetti/spark:latest"
  mainClass: MainKt
  mainApplicationFile: "local:///opt/spark/jars/dpso.jar"
  arguments:
    - "--sync"
    ...
  sparkVersion: "3.1.1"
  restartPolicy:
    type: Never
  volumes:
    - name: spark-data
      persistentVolumeClaim:
        claimName: local-pvc
  driver:
    cores: 1
    volumeMounts:
      - name: "spark-data"
        mountPath: "/spark-volume"
  executor:
    cores: 2
    instances: 3
```

Figure A.1: Example of Spark Operator declarative configuration.

```
procedure SDSPSO-LU($I$, $P$, $N$, $M$)
    $sc \leftarrow$ INITSPARK()
    $acc \leftarrow sc$.NEWBESTGLOBALACCUMULATOR(($null, \infty$))
    $broad \leftarrow sc$.NEWBROADCASTVARIABLE([$N$][$M$])

    $ps \leftarrow$ INITPARTICLES($P, N, M$)
    $bg \leftarrow (null, \infty)$
    $i \leftarrow 0$

    while $i < I$ do
        $\lambda \leftarrow$ FITNESSEVAL($broad, acc$)
        $ps \leftarrow sc$.PARALLELIZE($ps$).MAP($\lambda$).COLLECT()
        $bg \leftarrow acc$.VALUE()

        for all $par \in ps$ do
            $par$.UPDATEVELOCITY($bg$)
            $par$.UPDATEPOSITION()
        end for

        $i \leftarrow i + 1$
    end while

    return $bg$
end procedure

procedure FITNESSEVAL($broad$, $acc$) return closure
    procedure CALL($par$)
        $pos \leftarrow par$.POSITION()
        $var \leftarrow broad$.VALUE()
        $err \leftarrow$ FITNESS($pos, var$)
        $par$.UPDATEBESTPERSONAL($pos, err$)
        $acc$.ADD($pos, err$)
        return $par$
    end procedure
end procedure
```

Figure A.2: Pseudocode of the SDSPSO algorithm with Local Update (LU).

```
procedure SDSPSO-DU($I$, $P$, $N$, $M$)
    $sc \leftarrow$ INITSPARK()
    $acc \leftarrow sc$.NEWBESTGLOBALACCUMULATOR(($null$,$\infty$))
    $broad \leftarrow sc$.NEWBROADCASTVARIABLE([$N$][$M$])

    $ps \leftarrow$ INITPARTICLES($P, N, M$)
    $bg \leftarrow (null, \infty)$
    $i \leftarrow 0$

    while $i < I$ do
        $\lambda_1 \leftarrow$ FITNESSEVAL($broad, acc$)
        $ps \leftarrow sc$.PARALLELIZE($ps$).MAP($\lambda_1$).COLLECT()
        $bg \leftarrow acc$.VALUE()

        $bgbroad \leftarrow sc$.NEWBROADCASTVARIABLE($bg$)
        $\lambda_2 \leftarrow$ POSEVAL($bgbroad$)
        $ps \leftarrow sc$.PARALLELIZE($ps$).MAP($\lambda_2$).COLLECT()

        $i \leftarrow i + 1$
    end while

    return $bg$
end procedure

procedure FITNESSEVAL($broad$, $acc$) return closure
    procedure CALL($par$)
        $pos \leftarrow par$.POSITION()
        $var \leftarrow broad$.VALUE()
        $err \leftarrow$ FITNESS($pos, var$)
        $par$.UPDATEBESTPERSONAL($pos, err$)
        $acc$.ADD($pos, err$)
        return $par$
    end procedure
end procedure

procedure POSEVAL($bgbroad$) return closure
    procedure CALL($par$)
        $bg \leftarrow bgbroad$.VALUE()
        $par$.UPDATEVELOCITY($bg$)
        $par$.UPDATEPOSITION()
        return $par$
    end procedure
end procedure
```

Figure A.3: Pseudocode of the SDSPSO algorithm with Distributed Update (DU).

```
$bg \leftarrow (null, \infty)$

procedure ADD($obg$)
    if $obg[1] < bg[1]$ then                    ▷ $bg[n-1]$ takes the $n$-th value from the tuple
        $bg \leftarrow obg$
    end if
end procedure

procedure MERGE($oa$)
    $obg \leftarrow oa$.VALUE()
    ADD($obg$)
end procedure

procedure VALUE
    return $bg$
end procedure
```

Figure A.4: Pseudocode of the best global position accumulator in Spark.

```kotlin
fun sdspso(config: Configuration, sc: JavaSparkContext)
: Tuple2<PosPlacementMatrix?, Double?> {
    ...
    val bestGlobalPositionAccumulator = PositionAccumulator(
        MutablePosition.BestPosition()
    )
    sc.sc().register(
        bestGlobalPositionAccumulator,
        "BestGlobalPositionAccumulator"
    )
    var bestGlobalPosition: Tuple2<PosPlacementMatrix?, Double?> = Tuple2(
        null, null
    )
    val runtimesMatrixBroadcast = sc.broadcast(
        randomRuntime(nFogNodes, nModules)
    )
    var inputParticles: List<Particle> = randomPlacement(
        nParticles, nFogNodes, nModules
    )

    repeat(nIterations) {
        inputParticles = sc
            .parallelize(inputParticles)
            .map(
                FitnessEvaluation(
                    FitnessFunction.PlacementCost(),
                    runtimesMatrixBroadcast,
                    bestGlobalPositionAccumulator,
                    config.fitnessEvalDelay.toLong()
                )
            )
            .collect()

        bestGlobalPosition = bestGlobalPositionAccumulator.value()

        inputParticles = if (config.distributedPosEval) {
            val bestGlobalPositionBroadcast = sc.broadcast(bestGlobalPosition)
            val particles = sc.parallelize(inputParticles)
                .map(PositionEvaluation(bestGlobalPositionBroadcast))
                .collect()

            bestGlobalPositionBroadcast.unpersist()

            particles
        } else {
            inputParticles.map { particle ->
                particle.updateVelocity(bestGlobalPosition._1)
                particle.updatePosition()
                particle
            }
        }
    }

    return bestGlobalPosition
}
```

Figure A.5: Simplified implementation of the SDSPSO algorithm in Kotlin.

```
procedure SDAPSO($I$, $P$, $N$, $M$, $S$)
    $sc \leftarrow$ INITSPARK()
    $broad \leftarrow sc$.NEWBROADCASTVARIABLE([$N$][$M$])

    $ps \leftarrow$ INITPARTICLES($P, N, M$)
    $bg \leftarrow (null, \infty)$

    $srch \leftarrow$ NEWCHANNEL()
    $fuch \leftarrow$ NEWCHANNEL()
    $aggr \leftarrow$ AGGREGATOR($S, srch$)

    for $par \in ps$ do
        $aggr$.AGGREGATE($par$)
    end for

    for async $sr \in srch$ do                                  ▷ Non-blocking for
        $\lambda \leftarrow$ FITNESSEVAL($broad$)
        $psfu \leftarrow sc$.PARALLELIZE($ps$).MAP($\lambda$).COLLECTASYNC()
        $fuch$.SEND($psfu$)
    end for

    $i \leftarrow 0$
    $I \leftarrow \frac{I \times P}{S}$

    while async $i < I$ do                                      ▷ Non-blocking while
        $sr \leftarrow fuch$.RECEIVE()

        for $par \in sr$ do
            $pos \leftarrow par$.POSITION()
            $err \leftarrow par$.ERROR()
            if $err < bg[1]$ then
                $bg \leftarrow (pos, err)$
            end if
            $par$.UPDATEVELOCITY($bg$)
            $par$.UPDATEPOSITION()
            $aggr$.AGGREGATE($par$)
        end for

        $i \leftarrow i + 1$
    end while

    wait $i == I$                        ▷ Wait until $I$ superRDDs have been evaluated

    return $bg$
end procedure

procedure FITNESSEVAL($broad$) return closure
    procedure CALL($par$)
        $pos \leftarrow par$.POSITION()
        $var \leftarrow broad$.VALUE()
        $err \leftarrow$ FITNESS($pos, var$)
        $par$.UPDATEBESTPERSONAL($pos, err$)
        return $par$
    end procedure
end procedure
```

Figure A.6: Pseudocode of the SDAPSO algorithm.

```kotlin
fun sdapso(config: Configuration, sc: JavaSparkContext)
: Tuple2<PosPlacementMatrix?, Double?> = runBlocking {
    ...
    val nFillingParticles =
        if (nBaseParticles % superRDDSize == 0) 0
        else (superRDDSize - (nBaseParticles % superRDDSize))
    val nParticles = nBaseParticles + nFillingParticles
    val runtimesMatrixBroadcast = sc.broadcast(randomRuntime(nFogNodes, nModules))
    val inputParticles = randomPlacement(nParticles, nFogNodes, nModules)
    val superRDDChannel = Channel<SuperRDD>()
    val aggregator = superRDDAggregator(superRDDSize, superRDDChannel)
    inputParticles.forEach { aggregator.aggregate(it) }
    val futuresChannel = Channel<JavaFutureAction<List<Particle>>>()
    val stateActor = stateActor()

    val producer = launch {
        for (superRDD in superRDDChannel) {
            val future = sc.parallelize(superRDD.particles)
                .map(AsyncFitnessEvaluation(
                    FitnessFunction.PlacementCost(),
                    runtimesMatrixBroadcast,
                    config.fitnessEvalDelay.toLong())
                )
                .collectAsync()
            futuresChannel.send(future)
        }
    }

    val consumer = launch {
        val jobs = mutableListOf<Job>()
        repeat((nParticles * nIterations) / superRDDSize) {
            val job = launch(Dispatchers.IO) {
                futuresChannel.receive().get().forEach { particle ->
                    stateActor.mutate { state ->
                        state.mutateBestGlobalPosition(
                            Tuple2(particle.position, particle.error)
                        )
                    }
                    val state = stateActor.snapshot()
                    particle.updateVelocity(state.bestGlobalPosition.position())
                    particle.updatePosition()
                    aggregator.aggregate(particle)
                }
            }
            jobs.add(job)
        }
        jobs.joinAll()
        ...
    }

    ...
    return@runBlocking Tuple2(
        state.bestGlobalPosition.position(),
        state.bestGlobalPosition.error()
    )
}
```

Figure A.7: Simplified implementation of the SDAPSO algorithm in Kotlin.

Bibliography

[1] Khoi D. Hoang et al. *New Distributed Constraint Satisfaction Algorithms for Load Balancing in Edge Computing : A Feasibility Study*. en. 2019. URL: `https://bit.ly/3yyJFtU`.

[2] Dave Evans. “How the Next Evolution of the Internet Is Changing Everything”. en. In: *undefined* (2011). URL: `https://bit.ly/3wjECLg`.

[3] Redowan Mahmud, Kotagiri Ramamohanarao, and Rajkumar Buyya. “Application Management in Fog Computing Environments: A Taxonomy, Review and Future Directions”. In: *ACM Computing Surveys* 53.4 (July 2020), 88:1–88:43. ISSN: 0360-0300. URL: `https://bit.ly/3FDJVth`.

[4] Flavio Bonomi et al. “Fog computing and its role in the internet of things”. In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. MCC ’12. New York, NY, USA: Association for Computing Machinery, Aug. 2012, pp. 13–16. ISBN: 9781450315197. URL: `https://bit.ly/397rVLv`.

[5] Farah Aït Salaht, Frédéric Desprez, and Adrien Lebre. “An Overview of Service Placement Problem in Fog and Edge Computing”. In: *ACM Computing Surveys* 53.3 (June 2020), 65:1–65:35. ISSN: 0360-0300. URL: `https://bit.ly/3LWJw7A`.

[6] Oscar Ricardo Cuellar Rodriguez et al. “Improvement of Edge Computing Workload Placement using Multi Objective Particle Swarm Optimization”. In: *2021 8th International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. Gandia, Spain: IEEE, Dec. 2021, pp. 1–8. ISBN: 9781665458689. URL: `https://bit.ly/3MO1ztN`.

[7] Dongshu Wang, Dapei Tan, and Lei Liu. “Particle swarm optimization algorithm: an overview”. en. In: *Soft Computing* 22.2 (Jan. 2018), pp. 387–408. ISSN: 1432-7643, 1433-7479. URL: `https://bit.ly/3M4zAZG`.

[8] Hasan Ozcan. “Comparison Of Particle Swarm And Differential Evolution Optimization Algorithms Considering Various Benchmark Functions”. en. In: 20.4 (2017). URL: `https://bit.ly/39aI96V` (visited on 03/30/2022).

[9] Wei Du et al. “Fault-Tolerating Edge Computing with Server Redundancy Based on a Variant of Group Degree Centrality”. en. In: *Service-Oriented Computing*. Ed. by Eleanna Kafeza et al. Cham: Springer International Publishing, 2020, pp. 198–214. ISBN: 9783030653101.

[10] Gerhard Venter and Jaroslaw Sobieszczanski-Sobieski. "Parallel Particle Swarm Optimization Algorithm Accelerated by Asynchronous Evaluations". en. In: *Journal of Aerospace Computing, Information, and Communication* 3.3 (Mar. 2006), pp. 123–137. ISSN: 1542-9423. URL: `https://bit.ly/3kTNmT6`.

[11] J. F. Schutte et al. "Parallel global optimization with the particle swarm algorithm". en. In: *International Journal for Numerical Methods in Engineering* 61.13 (Dec. 2004), pp. 2296–2315. ISSN: 0029-5981, 1097-0207. URL: `https://bit.ly/37v0N8T`.

[12] Ouarda Zedadra et al. "Swarm intelligence-based algorithms within IoT-based systems: A review". en. In: *Journal of Parallel and Distributed Computing* 122 (Dec. 2018), pp. 173–187. ISSN: 07437315. URL: `https://bit.ly/38fI01P`.

[13] J. Kennedy and R. Eberhart. "Particle swarm optimization". In: *Proceedings of ICNN'95 - International Conference on Neural Networks*. Vol. 4. Perth, WA, Australia: IEEE, 1995, pp. 1942–1948. ISBN: 9780780327689. URL: `https://bit.ly/3N0HnrF`.

[14] Nor Azlina Ab Aziz et al. "A Synchronous-Asynchronous Particle Swarm Optimisation Algorithm". en. In: *The Scientific World Journal* 2014 (2014), pp. 1–17. ISSN: 2356-6140, 1537-744X. URL: `https://bit.ly/3PmQi91`.

[15] Wazir Zada Khan et al. "Edge computing: A survey". en. In: *Future Generation Computer Systems* 97 (Aug. 2019), pp. 219–235. ISSN: 0167-739X. URL: `https://bit.ly/3w0VIy0`.

[16] OpenFog Consortium Architecture Working Group. *OpenFog Reference Architecture for Fog Computing*. URL: `https://bit.ly/3vYj1sK`.

[17] Ashkan Yousefpour et al. "All one needs to know about fog computing and related edge computing paradigms: A complete survey". en. In: *Journal of Systems Architecture* 98 (Sept. 2019), pp. 289–330. ISSN: 1383-7621. URL: `https://bit.ly/3sMqppt`.

[18] *Hazelcast Edge Computing*. URL: `https://bit.ly/38mECSG`.

[19] Pengfei Hu et al. "Survey on fog computing: architecture, key technologies, applications and open issues". en. In: *Journal of Network and Computer Applications* 98 (Nov. 2017), pp. 27–42. ISSN: 1084-8045. URL: `https://bit.ly/38avpx3`.

[20] Internet Engineering Task Force. *The Locator/ID Separation Protocol (LISP)*. URL: `https://bit.ly/38cj7UT`.

[21] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. URL: `https://bit.ly/37zgqfG`.

[22] Luis M. Vaquero et al. "A break in the clouds: towards a cloud definition". In: *ACM SIGCOMM Computer Communication Review* 39.1 (Dec. 2009), pp. 50–55. ISSN: 0146-4833. URL: `https://bit.ly/3P7MVCx`.

[23] Kevin Ashton. *The 'Internet of Things' Thing*. URL: `https://bit.ly/38iDZd1`.

[24] "Future Internet: The Internet of Things Architecture, Possible Applications and Key Challenges". en. In: *ResearchGate*. URL: `https://bit.ly/3kW2vmR`.

[25] Martin Kleppmann. *Distributed Systems*. URL: `https://bit.ly/3KZ6v0G`.

[26] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. en. Cambridge, MA, USA: MIT Press, Feb. 2004. ISBN: 9780262220699.

[27] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: simplified data processing on large clusters”. en. In: *Communications of the ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782, 1557-7317. URL: `https://bit.ly/3spYPxW`.

[28] Matei Zaharia et al. “Spark: cluster computing with working sets”. In: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. HotCloud’10. USA: USENIX Association, June 2010, p. 10.

[29] Matei Zaharia et al. “Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing”. In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. NSDI’12. USA: USENIX Association, Apr. 2012, p. 2.

[30] *Spark Programming Guide*. URL: `https://bit.ly/3sr51Wl`.

[31] Gianluca Turin et al. “A Formal Model of the Kubernetes Container Framework”. en. In: *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2020, pp. 558–577. ISBN: 9783030613624.

[32] *Kotlin Docs*. URL: `https://bit.ly/3KYkofy`.

[33] *Kotlin Coroutines Docs*. URL: `https://bit.ly/3M5n10k`.

[34] *Kotlin Coroutines Specification*. URL: `https://bit.ly/3so1wzY`.

[35] Shelernaz Azimi, Claus Pahl, and Mirsaeid Shirvani. “Particle Swarm Optimization for Performance Management in Multi-cluster IoT Edge Architectures:” in: *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. Prague, Czech Republic: SCITEPRESS - Science and Technology Publications, 2020, pp. 328–337. ISBN: 9789897584244. URL: `https://bit.ly/3L1KxKa`.

[36] Aichuan Li, Lin Li, and Shujuan Yi. “Computation Offloading Strategy for IoT Using Improved Particle Swarm Algorithm in Edge Computing”. en. In: *Wireless Communications and Mobile Computing* 2022 (Feb. 2022). Ed. by Xin Ning, pp. 1–9. ISSN: 1530-8677, 1530-8669. URL: `https://bit.ly/3N2Aztv`.

[37] Muhammad Saqib Sohail et al. “Low-Complexity Particle Swarm Optimization for Time-Critical Applications”. In: *arXiv:1401.0546 [cs]* (Jan. 2014). arXiv: 1401.0546. URL: `https://bit.ly/3Fz0DK9`.

[38] Liu Dasheng. “Multi-Objective Particle Swarm Optimization: Algorithms and Applications”. PhD thesis. 2008. URL: `https://bit.ly/3Fy0uXB`.

[39] *Kotlin Channels Docs*. URL: `https://bit.ly/3spZcIQ`.

[40] *Shared Mutable State and Concurrency*. URL: `https://bit.ly/3L19W6I`.

[41] *Spark on Kubernetes*. URL: `https://bit.ly/3L19S6Y`.

[42] *Spark Operator Architecture*. URL: `https://bit.ly/3Fufd5G`.