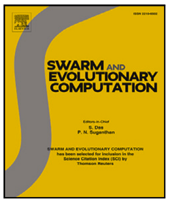
Swarm and Evolutionary Computation 85 (2024) 101483

Contents lists available at ScienceDirect

Swarm and Evolutionary Computation

journal homepage: www.elsevier.com/locate/swevo



Population based metaheuristics in Spark: Towards a general framework using PSO as a case study

Xoán C. Pardo [a,*], Patricia González [a], Julio R. Banga [b], Ramón Doallo [a]

[a] CITIC, Computer Architecture Group, Universidade da Coruña, Spain
[b] CSIC, Computational Biology Lab, Spain

ARTICLE INFO

Keywords:
Population based metaheuristics
Distributed metaheuristics
Metaheuristic optimization framework
Particle swarm optimization
Apache Spark

ABSTRACT

Over the years metaheuristics have been successfully applied to optimization problems in many real-world applications. The increasing complexity and scale of the problems addressed has posed new challenges to researchers in the field. The application of distributed metaheuristics is a common approach to speed up the time to solution or improve its quality by leveraging traditional parallel programming models on platforms like multicore processors or computer clusters. More recently, the emergence of Cloud Computing and new programming models and frameworks for Big Data has facilitated access to an unprecedented amount of computational resources, which led to a growing interest in optimization frameworks that support the development and execution of distributed metaheuristics taking advantage of this potential. In this paper, we present the current status of development of one such framework that aims to provide support for the application of distributed population-based metaheuristics to the global optimization of large-scale problems in Spark. The framework provides a reduced set of abstractions to represent the general structure of population-based metaheuristics as templates and strategies to particularize them into concrete metaheuristics, as well as other nice features like out of the box implementations of the most common distributed models, full configurability through a human-friendly format, and the possibility of rapid prototyping and testing metaheuristics in the Spark shell. To validate the approach, a template for Particle Swarm Optimization (PSO) was implemented as a proof of concept, which includes strategies for instantiating different variants of the algorithm, configurable topologies, and sequential and distributed execution models.

1. Introduction

Optimization problems arise in almost every discipline, including engineering, economics, social sciences, computational biology, business management or logistics, only to cite a few. Different optimization methods have been applied during decades to solve increasingly large and complex problems. Among these methods, metaheuristics [1] have gained more popularity over exact methods in hard optimization problems due to their potential to achieve satisfactory solutions at an affordable cost when exact methods fail to find an optimal solution. Even so, there are still many challenging problems for which metaheuristics are not able to reach satisfactory results within a reasonable time due to their complexity or size. High-dimensional search spaces with a large number of local optima or fitness functions with an extremely high computational cost are typical of these problems. The development of distributed metaheuristics to reduce the time needed to obtain a solution or to improve its quality, using traditional parallel programming models such as OMP or MPI to take advantage of the computational resources available on platforms such as multicore processors or computer clusters, has been a common approach to overcome the limitations of the applicability of metaheuristics to large-scale problems. Furthermore, in recent years, the emergence of new computing platforms, such as GPUs or the Cloud, has aroused interest and led to new developments in order to take greater advantage of the unprecedented availability of computing resources.

However, implementing scalable and efficient distributed metaheuristics is a non-trivial task. Fully exploiting available computational resources requires knowledge of complex hardware architectures and software platforms, sometimes requiring to carefully make trade-offs among different performance metrics. Moreover, the underlying platform partially influences the implementation, conditioning its performance, scalability and fault-tolerance [2]. In this context, the availability of frameworks that provide customizable distributed implementations of the most popular metaheuristics is helpful for practitioners with different expertise, knowledge of the underlying platform

* Corresponding author.
E-mail address: xoan.pardo@udc.gal (X.C. Pardo).

https://doi.org/10.1016/j.swevo.2024.101483
Received 16 October 2023; Received in revised form 21 December 2023; Accepted 17 January 2024
Available online 23 January 2024

and programming skills. But, although there are many frameworks for metaheuristics available [3–7], the few existent studies, as far as we know, that have compared them [8–10] show that there are significant differences in the level of support for distribution, if any, and the execution performance they provide.

Furthermore, although the availability of new frameworks for Big Data [11], has opened the possibility for solving optimization problems at an unprecedented scale, there has not yet been much work on frameworks to support the development and efficient execution of distributed metaheuristics on these platforms. So far, most proposals have been ad hoc distributed implementations of specific metaheuristics applied to particular problems. In this respect, the situation resembles when the first frameworks supporting the execution of metaheuristics on traditional parallel infrastructures were proposed, and it reproduces many of the pitfalls that have been identified as inhibitors for reproducibility and progress in the field [12]: duplication of effort and siloing of research, tendency to re-implement metaheuristics from scratch because it is difficult to reuse existing implementations, or difficulty to independently replicate the results of other proposals, which is fundamental for the comparison between different approaches, because algorithm descriptions in many papers are not precise enough and public access to the source code of the implementations or experimental data is rarely available.

In this paper, we present the current status in the development of a framework to support the development and efficient execution of distributed population-based metaheuristics in Spark [13]. The motivation of this proposal comes from our own interest in the optimization of large-scale problems from Computational Biology in distributed frameworks such as Hadoop (the most popular MapReduce [14] framework) or Spark. After having implemented and evaluated different distributed population-based metaheuristics on these frameworks [15–19], we decided to start the development of the Metaheuristic Optimization Framework (MOF) we introduce in this paper to facilitate the development of new metaheuristics and improve the automation and reproducibility of our experiments. From the two main Big Data frameworks known for having support for efficient iterative algorithms, i.e. Spark and Flink [20], we selected Spark because it is the most mature and widely adopted. Spark facilitates the development of scalable distributed algorithms through features such as high-level programming abstractions and built-in support for data distribution, fault tolerance and load balancing [18,19]. To validate the approach, Particle Swarm Optimization (PSO) was used as a proof of concept. The result is a template for PSO with support to the instantiation of different variants of the algorithm and configurable topologies, which can be executed on Spark clusters using various sequential or distributed execution models. The source code is available in a public repository [21].

It should be noted that our proposal is focused on population-based metaheuristics applied to Large-Scale Global Optimization (LSGO) problems as defined in [22], and that, due to the early stage of development, it does not support many of the features that would be expected in a MOF [10], although it already provides some nice features such as a human-friendly format for the configuration of experiments, problems and metaheuristics, the possibility of rapid prototyping and testing metaheuristics in the Spark shell or the availability of LSGO benchmarks out of the box.

The main contributions of this paper are:

- The initial design of a framework to support the development and efficient execution of population-based metaheuristics on Spark clusters is presented. The design of the framework aims at providing a reduced set of abstractions that capture the common structure of population-based metaheuristics as general templates. From the general templates, specific metaheuristics are instantiated by defining the concrete abstractions to model the population and the strategies applied by the metaheuristic. Strategies can be reused between metaheuristics, enforcing code reusability.
- The proposed framework provides implementations of the most common distributed execution models. A detailed discussion of these implementations in Spark and their efficiency, supported by experiments, is provided.
- A template for PSO was defined using the abstractions of the framework to validate the approach. In addition, several strategies have been implemented to support the instantiation of different variants of PSO, including configurable topologies, whose genericity and correctness have been experimentally proven. Furthermore, the PSO instances can run on Spark clusters using any of the distributed execution models provided by the framework. To the best of our knowledge, no other implementation of a PSO template supports all of these features together.

The remainder of this paper is organized as follows: Section 2 is a summary of the related work; the design and main abstractions provided by the proposed framework are described in Section 3; Section 4 introduces the PSO template that was implemented to validate the approach; in Section 5 the results of the experimental evaluation are presented; and Section 6 gives some remarks to conclude the paper.

2. Related work

This section summarizes the main findings in the literature related to the proposal of this paper. To provide a general context, the section starts by referring to some recent reviews on metaheuristics, their parallelization and supporting frameworks. Then, the main proposals on the implementation of population-based metaheuristics in distributed frameworks for Big Data like Hadoop or Spark are included. The section ends by commenting on the MOFs proposed to support the development and execution of population-based metaheuristics in distributed frameworks for Big Data.

2.1. Metaheuristics

Research on metaheuristics has been extensively reported in the literature. Some recent surveys can be found in [1,23–27]. Also, the parallelization of metaheuristics has been studied extensively in the last decade [28] with the aim of reducing the time needed to solve larger problem instances and improve the robustness of sequential versions. The following are some recent reviews about the parallelization of different types of metaheuristics: in [29] the main concepts and general strategies for the design of parallel metaheuristics are presented; a new integrative framework of parallel computational optimization across optimization problems, algorithms and application domains is presented in [30]; a conceptual framework and the most important models for distributed evolutionary computing (dEC) are presented and analyzed in [2]; a systematic review on dEC and a new taxonomy for distributed optimization problems is presented in [31]; recent advances in parallel genetic algorithms (PGAs) are reported in [32] and a systematic survey on parallel PSO algorithms can be found in [33].

Metaheuristic Optimization Frameworks (MOFs) have numerous advantages for practitioners and researches in the field of optimization, such as including default implementations of the most popular metaheuristics, support for evaluating and comparing different methods, ease of adapting or developing metaheuristics for particular problems, or advanced features such as support for hybridization and parallel and distributed execution capabilities. Although there are many MOFs reported in the literature, e.g. HeuristicLab [3], ECJ [4], ParadisEO [5], DEAP [6], jMetal [7], to the best of our knowledge there are only a few studies that compare them. The most prominent ones – in chronological order – are: (i) a comparative study of ten MOFs over a set of 271 features grouped into 30 characteristics and 6 areas of interest is presented in [8]. Among the conclusions, a significant lack of support for parallel and distributed computing capabilities and the need for wider implementation of Software Engineering best practices were identified;

Table 1
Summary of related work in Section 2.2 by metaheuristic and framework.

Metaheuristic	Hadoop	Spark
PSO	[34–38]	[39,39–48]
GA	[49,49,50,50–52]	[53–57]
DE	[58]	[15,18,59,60]
WOA	[61,62]	[62–64]
ACO	[65]	[66]
CS	[67]	[68]
GSO	[69]	[70]
Other metaheuristics	BA [71]	eSS [19], ALO [72], SCA [73], ABC [74,75], CRO [76]
Hybrid approaches	GA-PSO [77]	ACO-GA [78], PSO-K-means [79–82]

(ii) the review in [9] extended that of [8] by performing a comparative analysis of 25 MOFs over a set of 22 characteristics grouped into 4 categories, focusing particularly on issues related to the use of multi-agent structures in the development of hybrid metaheuristics, such as the level of support for hybridization, cooperation and parallelism. From the results of the analysis, it is concluded that there are still important gaps to be filled in the development of MOFs, and with respect to support for parallel and distributed computing, the analysis shows that there are significant differences in the level of support, if any, that the frameworks provide; and (iii) a comparative study of 10 MOFs that provide support for multi-objective optimization over a set of features organized around seven characteristics is presented in [10]. Notably, two of the characteristics were assessed by performing an analysis of code metrics and a series of experiments. The experimental comparison found significant differences between the MOFs in performance under demanding configurations, both in terms of execution time and memory usage. Regarding support for parallel and distributed execution, the analysis shows that most MOFs only parallelize the evaluation of solutions and only two provide support for both of the most common distributed models, i.e., the master–worker model and the island-based model.

2.2. Population-based metaheuristics in Big Data frameworks

The design and development of parallel metaheuristics using programming models and frameworks for Big Data, such as Hadoop and Spark, is receiving increasing attention in recent years. Some of the main proposals related to population-based metaheuristics are collected in this section. A summary of the references by metaheuristic and framework is shown in Table 1.

MapReduce (MR) was the first programming model successfully and widely applied for Big Data applications. The parallelization of PSO in MR was proposed in [34], where the constricted version of PSO was implemented following a synchronous approach that preserves the semantics of the sequential PSO. In the mapping phase the updated velocity, position and best position of each particle is obtained and in the reduce phase, all the swarm information is collected and the global best updated. The scalability of the proposal was demonstrated using the problem of training a radial basis function network (RDF) with up to 256 processors. The main drawback is the overhead introduced to update the global best of each particle, because the particle has to store a list with the identifiers of all its neighbors and a message has to be emitted for each of them, i.e. N*N messages for a swarm of size N with global topology. A cooperative PSO is implemented in [35] which outperforms significantly the sequential version of the algorithm on both time and quality of solution. Moreover, it also has better performance on several problems, and a significant advantage on time, over two algorithms of the CEC 2013 competition with which it is compared. A distributed quantum-behaved PSO is proposed in [36] showing better experimental results on both quality of solution and time cost than the sequential version. Also, clustering algorithms based on PSO have been proposed in [37,38] showing to scale well with datasets of increasing size.

Different approaches to implement Genetic Algorithms (GA) in MR can be found in [49–51]. In [49] the evaluation of the population fitness is performed in the map function whereas the selection and recombination operations are applied in the reduce function. According to [50], GAs cannot be easily parallelized using MR due to their specific characteristics, so they propose to incorporate a hierarchical reduction phase to overcome this problem, however, the results shown little scalability. Also, in [52] the performance of three different models of PGAs – global, grid and island – were assessed on a Hadoop cluster, showing that the island model outperforms the others.

Other population-based metaheuristics that were also implemented using MR are: Differential Evolution (DE) [58], Ant Colony Optimization (ACO) [65], Whale Optimization Algorithm (WOA) [61], Cuckoo Search (CS) [67], Glowworm Swarm Optimization (GSO) [69], Parallel Bat Algorithm (PBA) applied to solve clustering problems [71] or a hybrid GA-PSO to infer large networks of genes [77].

One of the drawbacks of MapReduce when implementing iterative algorithms, such as population-based metaheuristics, is that it is a batch-oriented programming model which introduces I/O overhead penalties between iterations. Spark adds support for streaming applications and improves the performance of iterative algorithms through in-memory computing [83]. Thus, the number of population-based metaheuristics implemented in Spark has seen an increase in recent years. In [39] a parallel PSO is implemented and tested, both in Spark and Hadoop, with a real world use case of energy optimization in buildings. The design of the parallel PSO follows a map-reduce scheme computing the velocity update, position update, fitness evaluation and local best update of each particle in the map and the global best update in the reduce. Experimental results show that both platforms could handle the optimization problem but Spark is faster than Hadoop while having the same accuracy. In [40] a parallel PSO is proposed and compared with a parallel dynamic programming algorithm using the cascade eight-reservoir system in the Yuanshui basin in China as a testbed. As in [39] the map operator is used to evaluate the fitness function and update the position and velocity of particles. Simulation experiments were performed on the Alibaba Cloud Computing platform showing that the computational efficiency is influenced by the number of particles, whereas a small amount is used, increasing the number of cores will increase the overhead of Spark tasks and the execution time does not always decrease. In [41] a Cooperative co-Evolution PSO for solving high-dimensional problems which combines global and local versions of PSO is proposed. Dynamic grouping and computing multiple algorithms at the same node are adopted to increase the degree of parallelism, reduce the computation time and improve the algorithm efficiency. Experimental evaluation shows that the proposal outperforms other approaches for some of the benchmark functions used. In [42] a distributed implementation of the quantum-behaved PSO is proposed. The same idea is then reused in the quantum-behaved cooperative evolutionary PSO proposed in [43]. In this work, the population is initialized using an opposition-based learning scheme, particles are updated using a quantum attractor and a parallel search is performed distributing the population in sub-populations and partitioning the search space in sub-spaces. The proposal is compared with the distributed PSOs implemented in [44] concluding that it can be applied to address the complexity of LSGO problems because it improves the search efficiency and shows high scalability. In [45] an island model of PSO is implemented and hybridized with an evolutionary strategy of DE. Experimental evaluation shows that the proposal outperforms other approaches on the set of benchmark functions tested. In [46] a distributed training algorithm based on PSO for a type of Recurrent Neural Networks is proposed, and in [47] a parallel PSO is applied to the efficient utilization of water resources among reservoirs in Jinsha River basin.

Also several proposals for applying PSO to clustering problems using Spark can be found in the literature, for example: a parallel PSO clustering algorithm for a learning analytics system where the arrival of data is continuous in [48]; PSO has been combined with K-means in [79–81]; a classification algorithm which uses PSO to find the optimal centroid for each target label in [84]; or a feature selection algorithm using Binary PSO integrated with a combination of PSO and K-means for effective clustering in cancer prognosis in [82].

There have been several proposals for implementing GAs in Spark. In [53] the population is distributed to the workers and a map function is used to apply the genetic operations and evaluation, then a reduce function is used to aggregate most promising individuals of the sub-populations when the termination condition is met. In [54] the evaluation is performed in parallel in the map function and then all the population is collected and the genetic operators applied to the whole population at once. GAs applied to a pairwise test suite generation problem were implemented in [55] following a map-reduce scheme. The fitness of the chromosomes is evaluated in the map function and aggregated in the reduce function to obtain the input for the selection operator. More recently, a detailed critical review of some previous proposals is presented in [56] and a new GA distributed architecture is proposed and implemented both in MR and Spark. Experimental results show that the proposed architecture outperforms significantly the other previous approaches, especially for LSGO problems. Also, two versions of parallel GAs, the master–worker and island models, to solve large dimensional classifier problems using Spark were proposed in [57].

DE was implemented in [85]. In [15] a sequential and two Spark implementations of the distributed master–worker and island-based models were implemented and compared. In [18] MR and Spark implementations of DE are compared concluding that the Spark implementations outperform MR in this kind of iterative algorithms. An enhanced DE for LSGO problems based on the use of a ring topology and CUDE as the internal optimizer is proposed in [59], which has been extended in [60] with the proposal of a novel grouping topology model that uses five DE variants as internal optimizers.

Other population-based metaheuristics that were implemented in Spark are: the MAX-MIN variant of ACO [66]; sequential, MR and Spark implementations of ACO are compared and an hybridization of ACO and GA is proposed to avoid premature converge to local optima in [78]; an enhanced Scatter Search (eSS) [19]; WOA [63]; sequential, MR and Spark implementations of WOA are compared in [62] showing that the Spark implementation successfully handles higher-complexity problems than the others; also a feature selection method based on distributed WOA is proposed in [64]; Spark and MR implementations of GSO are compared in [70]; Ant Lion Optimizer (ALO) [72]; Sine Cosine Algorithm (SCA) [73]; a multi-objective Artificial Bee Colony (ABC) [74]; a multi-swarm ABC based on clustering which has demonstrated excellent performance in medical image registration tests is proposed in [75]; CS were applied in [68] to solve the crew scheduling problem; and the Coral Reflection Optimization (CRO) in [76] to solve the Job Shop Scheduling Problem.

2.3. MOFs in Big Data frameworks

To the best of our knowledge, there are only a few proposals of MOFs that support the development and execution of parallel metaheuristics in Big Data frameworks. ECJ+Hadoop [86] is an enhancement to ECJ based on MapReduce for deploying massive runs of evolutionary algorithms on Hadoop clusters. The proposal implements the simplest form of the master–worker distribution model in which the fitness evaluation of individuals is distributed by means of a map evaluator.

jMetalSP [87] was originally conceived as a framework that combined the jMetal multi-objective MOF with Spark. Since then, it has evolved into a framework which supports dynamic multi-objective optimization using jMetal with different streaming engines such as Spark, Flink or Kafka [88]. The framework is implemented in Java and the source code is freely available on a Github repository. The support to parallelize metaheuristics with Spark was added to jMetalSP in [89]. jMetal provides an interface which defines an evaluate method to encapsulate the evaluation of solutions and two implementations of this interface: sequential and multi-threaded. Metaheuristics using this interface are agnostic of the concrete implementation used to evaluate the solutions. The approach used in jMetalSP was to provide a new implementation of this interface that uses Spark RDDs (*Resilient Distributed Datasets*) to distribute the evaluation of solutions to worker nodes. However, this approach has some limitations: (i) it is an implementation of the simplest form of the master–worker distribution model in which only the evaluation of the solutions is parallelized; and (ii) the objective function (i.e. the evaluation method of the problem in jMetal) used to evaluate the solutions must not modify variables outside the scope of the RDD containing the list of solutions to be evaluated.

A high level description of a unified framework to explain how to parallelize data clustering algorithms based on population-based metaheuristics for cloud computing platforms was presented in [66]. The framework represents population-based metaheuristics in terms of three general operators: (i) *transition*, used to adjust the searched solution; (ii) *evaluation*, used to evaluate the objective function; and (iii) *determination*, used to decide the search directions for the following iterations; and assign the transition and evaluation operators to the map function and the determination operator to the reduce function. Using the proposed framework several data clustering algorithms such as k-means, genetic k-means, and PSO were implemented. Implementations were written in Python and Java and evaluated with eleven datasets both on a stand-alone system and Spark. Neither details about the implementations, nor a repository with source code, are provided. Regarding PSO, the original proposal of [90] is cited, but no details are given on whether any enhancements, variants or topology support were included. Although in theory the framework could be used for any population-based metaheuristic, it is very general and follow a map-reduce scheme that does not take into account enhancements like other models (e.g. cooperative islands), variants, hybridization or topologies.

A framework to support the development of parallel population-based metaheuristics – more specifically, parallel evolutionary algorithms (PEAs) – in Spark is proposed in [44]. This work has two main contributions: (i) a quantitative and experimental analysis of the master–worker model based on Amdahl's law is performed to explain when and why this model could work well for PEAs; and (ii) a software framework is developed in which parallel versions of three different PSO variants are implemented in a unified way. The framework is implemented in Scala combining functional and object oriented approaches and the source code was freely distributed in a Github repository, although the development seems to have been discontinued. Running an optimization problem is configured by means of three types of configuration classes: (i) test parameters (e.g. number of runs, random seeds); (ii) objective function parameters (e.g. function dimension, search bounds); and (iii) algorithm parameters (e.g. population size, maximum number of fitness evaluations) which can be further customized for each algorithm by subclassing. All PSO variants are implemented in a unified way, encapsulating the implementation of each variant in a class that defines an *optimize* method with a common signature, i.e. it takes the objective function as input and returns the optimization result as output. The optimization method of the algorithm, the objective function and the test parameters are passed to the main class in charge of running the optimization problem at hand. Although this proposal is a first step in – what we think – is the right direction in the development of a framework to support population-based metaheuristics in Spark, it shows some drawbacks: (i) only one distribution model is implemented, i.e. the simplest version of the master–worker model which distributes the evaluations of the objective function to workers; (ii) although different PSO variants are implemented in a unified way, this is just a convention followed by the authors, which is not enforced in the framework using any of the mechanisms provided by Scala (e.g. inheritance from abstract base classes or mixin composition using traits); and (iii) the three PSO variants implemented have a similar structure and share much of the same code, however no mechanisms are provided by the framework to model that common structure or reusing code between implementations.

HyperSpark, a MOF that supports the design and execution of parallel metaheuristics on a Spark cluster is proposed in [91]. The framework is implemented in Scala and the source code is freely available on a Github repository, although the development seems to have been discontinued. The framework is composed of two main components: (i) a conceptual architecture inspired by other MOFs like jMetal, which provides the abstract entities (i.e. *Problem*, *Algorithm*, *Solution*) to represent any arbitrary combination of a problem representation, metaheuristic algorithm, and solution encoding; and (2) an execution workflow which supports the iterative execution of user-provided algorithms as independent parallel tasks, with or without cooperation between them, on a Spark cluster. The workflow is composed of several steps that iteratively split the problem into different sub-problems (e.g. different regions of the solution space or different objective functions), distribute the execution of the user-provided algorithms assigned to each sub-problem to the available computational nodes, aggregate the algorithm outcomes and use them to feedback the process until the stopping condition is met. Although the workflow iterations are implemented with a fixed map-reduce structure, the user can configure some steps and parameters such as the splitting of the problem, the assignment of algorithms to sub-problems, the aggregation function, the collaboration strategy used to distribute the results of an iteration between the algorithm instances, the number of iterations or the stopping condition, among others.

HyperSpark shares most of its design principles (i.e. ease-of-use, configurability, flexibility, extensibility) with the proposal presented in this paper, although with a different approach. HyperSpark follows what could be called a top-down approach by providing a high-level execution workflow for metaheuristic instances, but without being concerned about the internal parallelism of individual metaheuristics, nor providing mechanisms for code reuse between them. Whereas our proposal follows a bottom-up approach based on the use of strategies to enforce code reuse between metaheuristics and which supports various parallel models at different levels, e.g. strategy or metaheuristic instance. In our opinion, the following are some drawbacks of the HyperSpark approach: (i) the framework implements a fixed map-reduce distribution model and the parallelism is at the level of metaheuristic instances. No support or guidance for the parallel implementation of individual metaheuristics is provided, being it under the user's responsibility; (ii) no mechanisms are provided to enforce reusing code between metaheuristics. Only a limited form of code reuse based on single inheritance with method overriding is used in the implementation of some of the metaheuristics included with the framework. Moreover, due to the strong dependencies between the entities of the conceptual model, the types of the problem and solution encoding are hard-coded in the implementation of metaheuristics, preventing them to be reused between different types of problems; and (iii) although the framework is implemented in Scala, its design and implementation is highly biased towards an object oriented approach and procedural style that does not take advantage of the functional capabilities of the language.

3. The framework

This section introduces the design of the proposed software framework and the main features it provides for the implementation and execution of parallel population-based metaheuristics.

The approach is based on the observation that population-based metaheuristics such as, for example, Evolutionary Algorithms (EAs), Swarm-based metaheuristics (e.g. PSO), Differential Evolution (DE), or Scatter Search (SS) share a common structure in which an initial population of candidate solutions to a given optimization problem is repeatedly evolved by applying a composition of different intensification and diversification strategies until a termination condition is met. So they can be abstracted, from an algorithmic point of view, as general templates that can be particularized through the composition of reusable strategies for a wide class of optimization problems. The representation of metaheuristics as general abstractions that can be particularized through concrete instances of abstract components is an approach that has been followed, from diverse perspectives, in different proposals found in the literature, for example: software design patterns [92], grammar-guided genetic programming [93], parallel solvers specification languages [94] or generalized metaheuristic mathematical models [95].

The framework is being developed using Scala [96] as the programming language. Scala was chosen because is a concise, very expressive high-level language that combines object-oriented and functional approaches, it is the programming language of distributed frameworks like Spark or Flink, and it is highly interoperable with Java. The framework functionality is built upon a reduced set of abstractions to support the implementation of general algorithm templates and reusable strategies to instantiate different variants, including parallel, of metaheuristics. It leverages mainly a functional approach based on the use of *higher-order functions* (i.e. functions that take other functions as parameters and, possibly, return new functions created dynamically) combined with some of the object-oriented features provided by Scala (e.g. inheritance using traits and classes, mixin compositions) to implement well-known software design patterns such as *Template Method*, *Composite*, *Strategy* or *Dependency Injection*.

The status of the framework presented here is an evolution of some of the ideas already put in practice in [19], in which a general template for the enhanced SS metaheuristic was developed from which different instances of the algorithm, including a parallel version, could be instantiated. For simplicity, in the following, some details of the actual implementation have been omitted or slightly changed in the source code listings and UML diagrams. Interested readers are referred to the original source code that is publicly available in the companion repository [21].

3.1. Population, state and evolution

At the core of the framework there are the abstractions to represent a population and its evolution in the context of an optimization problem. A population is implemented as a collection of individuals. Each individual represents a solution in the search space of the problem at hand which holds two pieces of information: (i) the position vector of the solution in the search space, i.e. a collection of D – the dimension of the problem – real values, that is named *Element* in the framework; and (ii) a fitness value. As it can be seen in Fig. 1 the definition of the *Individual* class provides a minimum set of operations to evaluate the fitness, improve the solution, i.e. apply an intensification strategy, and bound the solution to search space limits. All these methods have been defined as *higher-order functions* so the concrete fitness, improvement or bounding functions to apply are passed as input arguments to the method call. Moreover, to support applying different strategies for the generation of the initial solutions, a factory method (i.e. *apply*) with a parameter for the generation function is defined in the *Individual* companion object.

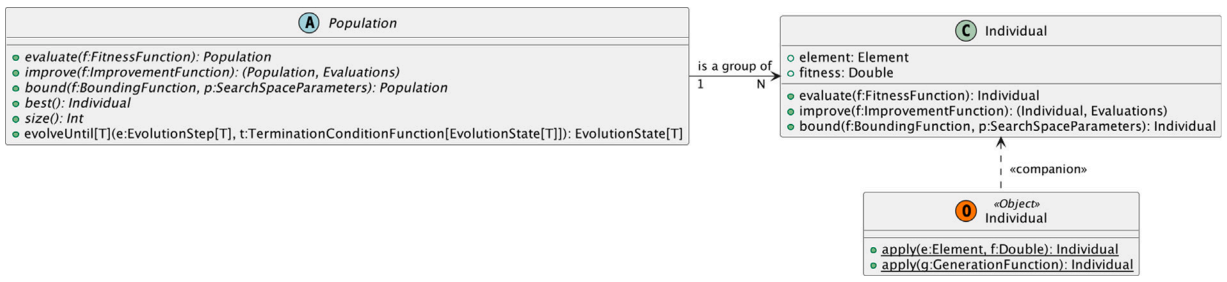

Population
evaluate(f:FitnessFunction): Population
improve(f:ImprovementFunction): (Population, Evaluations)
bound(f:BoundingFunction, p:SearchSpaceParameters): Population
best(): Individual
size(): Int
evolveUntil[T](e:EvolutionStep[T], t:TerminationConditionFunction[EvolutionState[T]]): EvolutionState[T]
is a group of
1
N
Individual
element: Element
fitness: Double
evaluate(f:FitnessFunction): Individual
improve(f:ImprovementFunction): (Individual, Evaluations)
bound(f:BoundingFunction, p:SearchSpaceParameters): Individual
«companion»
«Object»
Individual
apply(e:Element, f:Double): Individual
apply(g:GenerationFunction): Individual

Fig. 1. Individual and Population class diagram. The Scala *companion object* concept is represented as a class tagged with the *Object* stereotype, which contains only static methods, with a dependency tagged with the *companion* stereotype to its companion class.

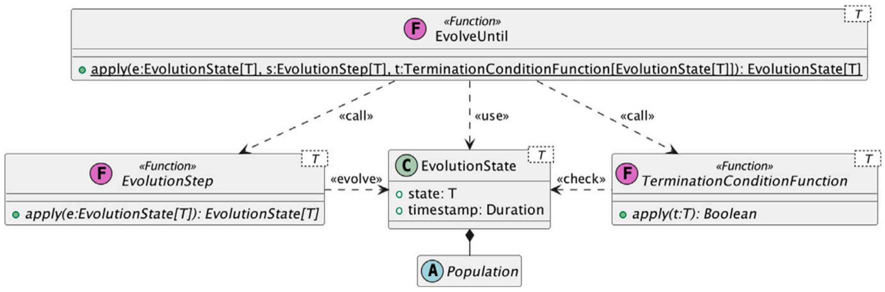

«Function»
EvolveUntil
T
apply(e:EvolutionState[T], s:EvolutionStep[T], t:TerminationConditionFunction[EvolutionState[T]]): EvolutionState[T]
«call»
«use»
«call»
«Function»
EvolutionStep
apply(e:EvolutionState[T]): EvolutionState[T]
«evolve»
EvolutionState
state: T
timestamp: Duration
«check»
«Function»
TerminationConditionFunction
apply(t:T): Boolean
Population

Fig. 2. Class diagram of the generic types to model states and evolutions. Generic functions are represented as generic classes tagged with the *Function* stereotype.

The *Population* abstract class defines, at the population level, the same methods as *Individual*. Besides, it provides additional methods to get the best individual, the size of the population and to evolve the population, as it is shown in Fig. 1. The *evolveUntil* method was also defined as a higher-order function, with the evolution function and termination condition to apply in the evolution of the population as input parameters. Moreover, it is a generic function parameterized by the type of the evolution state that is propagated between evolution iterations, which allows evolution functions with different state representations to be used to evolve the population. The default implementation of the method – not shown in the figure – initializes an evolution state with the initial population and calls *EvolveUntil* (Fig. 2), the generic function provided by the framework to evolve an state until a termination condition is met, as it is explained later. The reason *Population* is defined as an abstract class is that two different population states, grouped and distributed, are distinguished by the framework and implemented differently by subclassing *Population*. As it will be explained in Section 3.4, this is used to support the parallel implementation of population-based metaheuristics in the framework.

Population-based metaheuristics are iterative methods that propagate state information between iterations during the evolution of a population. Therefore, the framework provides generic types and functions to support the representation of a state and its evolution. These generic definitions form the basis for the implementation of population-based metaheuristics as general templates from which specific instances are obtained by particularizing the generic types. As it is shown in Fig. 2, an *EvolutionState* is represented as a generic class that contains, at least, the population being evolved and a *timestamp* field to account for the evolution elapsed time. *EvolutionStep* and *TerminationConditionFunction* are the generic types for functions that, respectively, evolve an state and check for the fulfillment of a termination condition. *EvolveUntil* is the higher-order function that implements the generic iterative process followed by any population-based evolutionary method to evolve an initial state, by repeatedly applying an evolution function, until a termination condition is met. The function pseudocode is shown in algorithm 1. The initial state is passed as input argument and the evolved state is returned as output.

Algorithm 1 The general algorithm of a state evolution

```
Inputs: state (the state to be evolved); evolve (an evolution function);
converged (a termination condition)
Output: the evolved state
1: if $converged(state)$ then
2:     state
3: else
4:     evolveUntil(evolve(state))                    ▷ recursive call
5: end if
```

3.2. Algorithms and strategies

The core functionality of the framework also provides abstractions for the implementation of general algorithm templates and reusable strategies. The common structure of population-based metaheuristics was captured as a general algorithm template (algorithm 2) applying the *template method* design pattern. The template was implemented using mixin composition and the generic definitions introduced in Section 3.1, as it is shown in Fig. 3 and listing 1. With this template, a particular population-based metaheuristic will only have to extend the base algorithm and provide the implementation of the three functions used as input parameters in the call to the *EvolveUntil* (algorithm 1) generic evolution function: (i) the function to generate the initial state containing the population to be evolved; (ii) the function to evolve the state; and (iii) the termination condition. Note that each of these functions can be itself implemented as a general template that uses *higher-order functions*, so concrete instances of the algorithm can be as complex as needed.

To illustrate how this general approach is applied to get a concrete metaheuristic, we will use the PG method of algorithm 2 as an example. Algorithm 3 shows a possible general algorithm for the PG method, whose purpose is to provide the initial set of solutions (initial population) that will then be iteratively evolved. To accomplish that, three methods are involved: (i) the *generation method* (GM) to generate

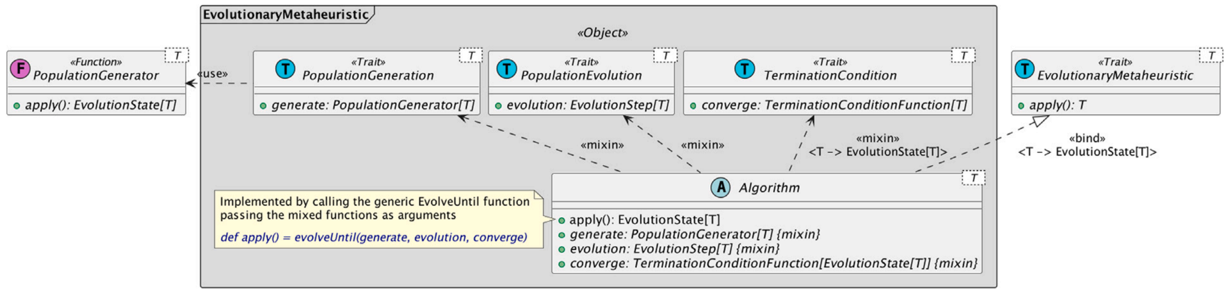

EvolutionaryMetaheuristic
«Object»
«Function» PopulationGenerator
apply(): EvolutionState[T]
«use»
«Trait» PopulationGeneration
generate: PopulationGenerator[T]
«Trait» PopulationEvolution
evolution: EvolutionStep[T]
«Trait» TerminationCondition
converge: TerminationConditionFunction[T]
«Trait» EvolutionaryMetaheuristic
apply(): T
«mixin»
«mixin»
«mixin» <T -> EvolutionState[T]>
«bind» <T -> EvolutionState[T]>
Algorithm
apply(): EvolutionState[T]
generate: PopulationGenerator[T] {mixin}
evolution: EvolutionStep[T] {mixin}
converge: TerminationConditionFunction[EvolutionState[T]] {mixin}
Implemented by calling the generic EvolveUntil function passing the mixed functions as arguments
def apply() = evolveUntil(generate, evolution, converge)

Fig. 3. Class diagram of the generic types to model the algorithm template of population-based metaheuristics. Scala concepts are represented in the following manner: *traits* as generic classes tagged with the *Trait* stereotype, *mixins* as dependencies tagged with the *mixin* stereotype and mixed operations indicated with the *mixin* property, and the *object definition* as a frame tagged with the *Object* stereotype.

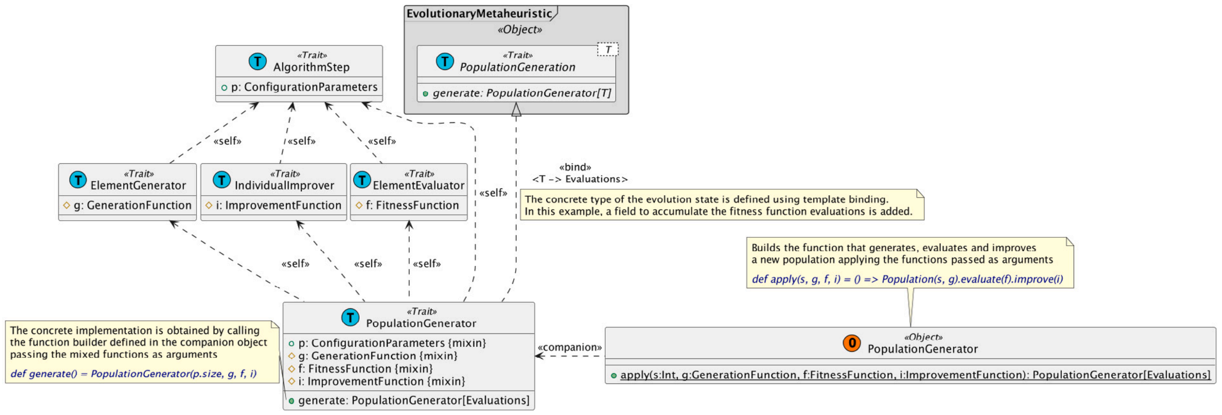

EvolutionaryMetaheuristic
«Object»
«Trait» PopulationGeneration
generate: PopulationGenerator[T]
«Trait» AlgorithmStep
p: ConfigurationParameters
«self»
«self»
«self»
«self»
«bind» <T -> Evaluations>
The concrete type of the evolution state is defined using template binding. In this example, a field to accumulate the fitness function evaluations is added.
«Trait» ElementGenerator
g: GenerationFunction
«Trait» IndividualImprover
i: ImprovementFunction
«Trait» ElementEvaluator
f: FitnessFunction
«self»
«self»
«self»
Builds the function that generates, evaluates and improves a new population applying the functions passed as arguments
def apply(s, g, f, i) = () => Population(s, g).evaluate(f).improve(i)
The concrete implementation is obtained by calling the function builder defined in the companion object passing the mixed functions as arguments
def generate() = PopulationGenerator(p.size, g, f, i)
«Trait» PopulationGenerator
p: ConfigurationParameters {mixin}
g: GenerationFunction {mixin}
f: FitnessFunction {mixin}
i: ImprovementFunction {mixin}
generate: PopulationGenerator[Evaluations]
«companion»
«Object» PopulationGenerator
apply(s:Int, g:GenerationFunction, f:FitnessFunction, i:ImprovementFunction): PopulationGenerator[Evaluations]

Fig. 4. Class diagram of the PG method. The Scala *self-type* concept is represented as a dependency tagged with the *self* stereotype, and mixed attributes are indicated using the *mixin* modifier.

Algorithm 2 The general algorithm of a population-based metaheuristic

```
Inputs: PG (population generation method); PE (population evolution method); TC (termination condition)
Output: the best solution found
1: population = apply PG
2: while ¬TC do
3:     apply PE to population
4: end while
5: return the best individual in population
```

Algorithm 3 A general algorithm for the population generation method (PG)

```
Inputs: GM (generation method); IM (improvement method); EM (evaluation method)
Output: the initial population evaluated and improved
1: population = apply GM
2: evaluated = apply EM to evaluate the solutions in population
3: improved = apply IM to evaluated using EM to evaluate the improved solutions
4: return improved
```

a collection of diverse solutions; (ii) the *improvement method* (IM) to apply an intensification strategy to the initial solutions; and (iii) the *evaluation method* (EM) to evaluate the fitness of both the initial and improved solutions. To obtain a specific instance of the PG method, implementations of the GM, IM and EM methods must be provided as inputs to the algorithm.

```scala
type PopulationGenerator[T] = () => EvolutionState[T]

trait EvolutionaryMetaheuristic[T] {
  def apply():T
}

object EvolutionaryMetaheuristic {
  // a population generator
  trait PopulationGeneration[T] {
    def generate:PopulationGenerator[T]
  }
  // an evolution step
  trait PopulationEvolution[T] {
    def evolution:EvolutionStep[T]
  }
  // a termination condition
  trait TerminationCondition[T] {
    implicit def converge:TerminationConditionFunction[T]
  }
  // the general algorithm of an evolutionary
  // metaheuristic
  abstract class Algorithm[T]
    extends EvolutionaryMetaheuristic[EvolutionState[T]]
      with PopulationGeneration[T]
      with PopulationEvolution[T]
      with TerminationCondition[EvolutionState[T]] {
    // implemented by calling the generic evolveUntil
    // function
    override def apply() = generate evolveUntil evolution
  }
}
```

Listing 1: Implementation of the general algorithm of a population-based metaheuristic.

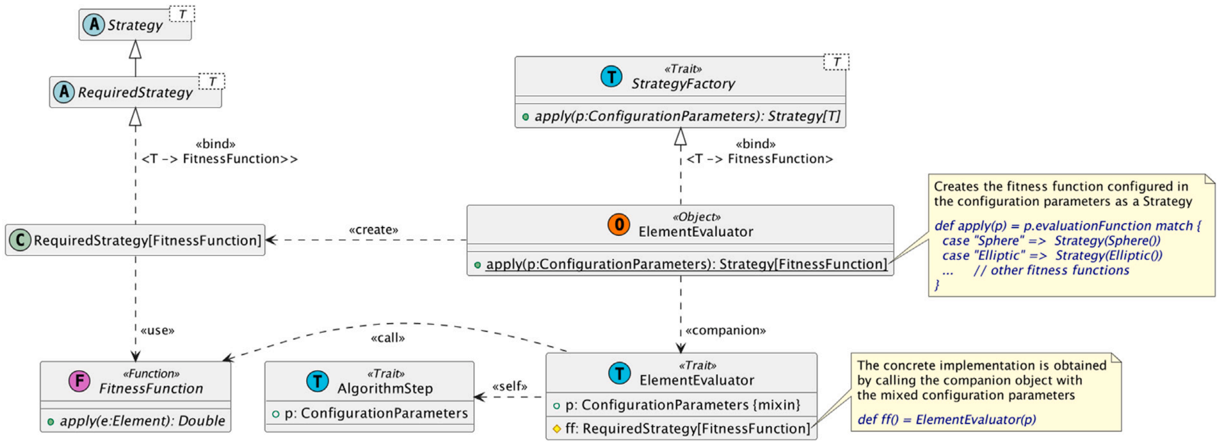

Strategy
T
RequiredStrategy
«bind» <T -> FitnessFunction>>
RequiredStrategy[FitnessFunction]
«create»
«use»
«Trait» StrategyFactory
apply(p:ConfigurationParameters): Strategy[T]
«bind» <T -> FitnessFunction>
«Object» ElementEvaluator
apply(p:ConfigurationParameters): Strategy[FitnessFunction]
Creates the fitness function configured in the configuration parameters as a Strategy
def apply(p) = p.evaluationFunction match {
case "Sphere" => Strategy(Sphere())
case "Elliptic" => Strategy(Elliptic())
... // other fitness functions
}
«companion»
«call»
«Function» FitnessFunction
apply(e:Element): Double
«Trait» AlgorithmStep
p: ConfigurationParameters
«self»
«Trait» ElementEvaluator
p: ConfigurationParameters {mixin}
ff: RequiredStrategy[FitnessFunction]
The concrete implementation is obtained by calling the companion object with the mixed configuration parameters
def ff() = ElementEvaluator(p)

Fig. 5. Class diagram of the EM method.

Fig. 4 shows how the PG method would be added to the framework, using again a *higher-order function* to apply the *template method* design pattern. PG is defined as a trait *PopulationGenerator* that implements the *generate* method defined in the *PopulationGeneration* trait of *EvolutionaryMetaheuristic*. The dependencies of PG on the GM, IM and EM methods are resolved by mixing the traits that implement the methods using self-types (i.e. a form of *Dependency Injection* in Scala). The same approach is also followed when mixing *AlgorithmStep*, a trait provided by the framework that can be used to bring configuration parameters into the context of other types, to access the population size configured in the configuration parameters. The concrete implementation of the *generate* method is obtained by delegating its construction to the function builder (i.e. a *higher-order function* that returns a function) defined in the companion object, passing the mixed methods and parameters as arguments.

To support the variety of implementations that a method could have and enforce reusability of method implementations between metaheuristics, the framework also defines abstractions to apply the *Strategy* design pattern. Generic types for optional and required strategies, and a generic strategy factory are provided. Fig. 5 shows how this approach would be applied for the implementation of the EM method of algorithm 3. The method is defined as a trait *ElementEvaluator* with an *AlgorithmStep* self-type. *AlgorithmStep* and *AlgorithmTemplate* – not shown in the figure – are traits provided by the framework to implement the *Composite* design pattern through mixin composition. Any trait with an *AlgorithmStep* self-type can be mixed into another *AlgorithmStep* or *AlgorithmTemplate*. They are therefore a way to reuse algorithm steps between algorithms and, in addition, they are also used to enable access to configuration parameters, as we have seen in the example of Fig. 4.

ElementEvaluator declares a fitness function as a required strategy. Fitness functions are instances of *FitnessFunction*, and have an input parameter of type *Element* (i.e. a solution in the search space) and an output parameter of type *Double* (i.e. the fitness value). Function types are fundamental to the composition and implementation of strategies because they define the signatures that strategy variants must adhere to and that the algorithm steps that mix them must use. The instantiation of the fitness function is delegated to the *ElementEvaluator* companion object, which is a specialization of the generic *StrategyFactory* that applies pattern matching on the fitness function name configured in the configuration parameters to select the fitness function to create.

The set of abstractions explained so far is enough to provide a concrete implementation to the general algorithm of a population-based metaheuristic of Fig. 3, as it is shown in the example of listing 2 and Fig. 6. The *Algorithm* of the concrete metaheuristic is defined by extending *EvolutionaryMetaheuristic.Algorithm* and mixing in the traits with the three functions required by the generic evolution function implemented by *EvolutionaryMetaheuristic.Algorithm* (i.e. *generate*, *evolution* and *converge*). For simplicity, in the example is assumed that these functions have already been implemented using the strategy approach described earlier in this section. Therefore, only the parts of their trait declarations and dependencies needed for the explanation are included in listing 2 and Fig. 6. An example of how the concrete metaheuristic would be instantiated and executed is shown in listing 3. The algorithm instance is configured with the configuration parameters passed at construction time and the result is a state that contains the evolved population, the number of evaluations of the fitness function performed and the evolution elapsed time.

3.3. Configuration and properties

The configuration of a metaheuristic and its strategies require some configuration parameters to be provided as input at construction time. The framework is very flexible in supporting the loading of configuration parameters from files, URLs, or *classpath* in a combination of three different formats: Java properties, JSON, and HOCON, i.e. a human-friendly JSON superset with nice features such as comments, file includes or variable substitution.

The current format of the configuration has three sections: (i) *execution context*, optional section with parameters for sequential – used by default – or Spark execution; (ii) *libraries*, optional section with parameters to load external libraries that provide access to existent problems (e.g. models in Computational Systems Biology) or methods (e.g. local solvers) usually written in languages like Fortran or C; and (iii) *experiments*, with parameters for configuring one or more executions. For each experiment it can be specified, between others, the number of runs, the population size, the algorithm and strategies to use and their specific parameters, the fitness function, the search space dimension and bounds and the termination condition.

While configuration parameters provide the initial static information needed to configure metaheuristics, the behavior of a metaheuristic also depends on accessing and modifying information that changes dynamically during execution, i.e. what has been described as *environmental state* in [97]. The framework provides two ways for making that information accessible to a metaheuristic and its strategies: (i) the evolution state (Section 3.1), which is passed on during the evolution through the strategies that are part of the metaheuristic; and (ii) a properties store, which can be mixed in any algorithm or strategy in the same way as other dependencies.

3.4. Sequential and parallel implementations

As it has been explained in Section 3.2, the framework provides support for algorithm templates and strategies with different implementations. Which one to use is configured in the configuration parameters

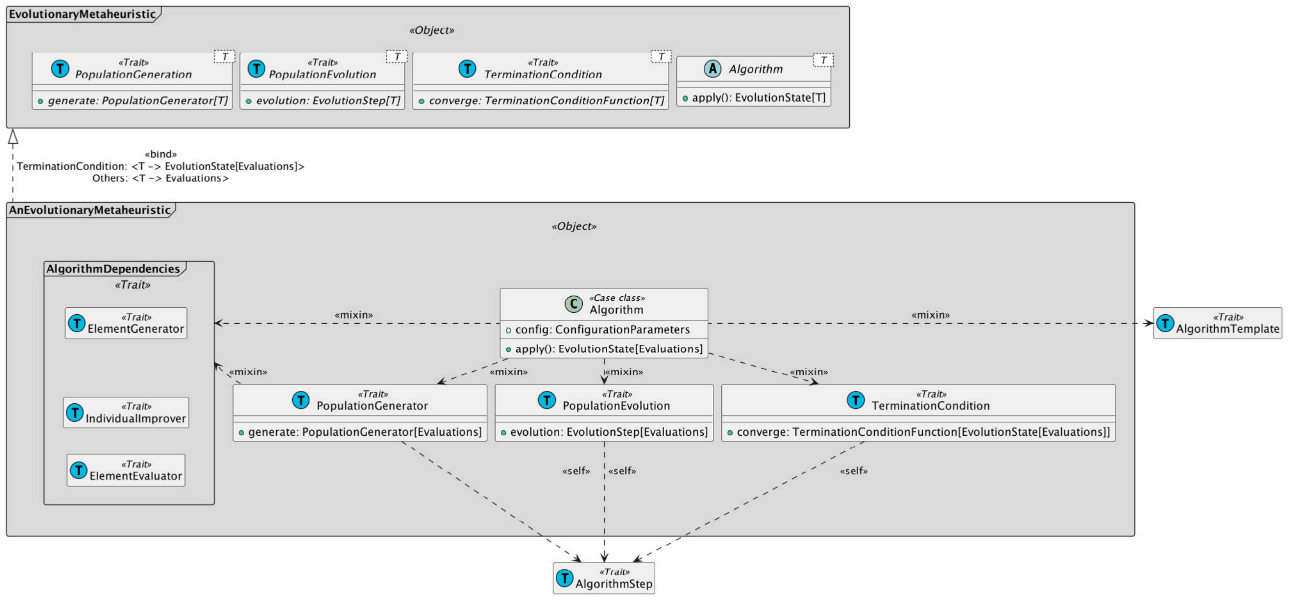

EvolutionaryMetaheuristic
«Object»
«Trait» PopulationGeneration
generate: PopulationGenerator[T]
«Trait» PopulationEvolution
evolution: EvolutionStep[T]
«Trait» TerminationCondition
converge: TerminationConditionFunction[T]
Algorithm
apply(): EvolutionState[T]
«bind»
TerminationCondition: <T -> EvolutionState[Evaluations]>
Others: <T -> Evaluations>
AnEvolutionaryMetaheuristic
«Object»
AlgorithmDependencies
«Trait»
«Trait» ElementGenerator
«Trait» IndividualImprover
«Trait» ElementEvaluator
«Case class» Algorithm
config: ConfigurationParameters
apply(): EvolutionState[Evaluations]
«mixin»
«Trait» AlgorithmTemplate
«Trait» PopulationGenerator
generate: PopulationGenerator[Evaluations]
«Trait» PopulationEvolution
evolution: EvolutionStep[Evaluations]
«Trait» TerminationCondition
converge: TerminationConditionFunction[EvolutionState[Evaluations]]
«self»
«Trait» AlgorithmStep

Fig. 6. Class diagram of an implementation of the algorithm template of Fig. 3. Only the *PopulationGenerator* dependencies explained in the example of Fig. 4 are shown. For clarity, they are included in a separate trait *AlgorithmDependencies*.

and instantiated at metaheuristic construction time. The same approach is followed to support sequential and parallel implementations of an algorithm or strategy. An example of how it would be applied to have two implementations of *PopulationGenerator*, a sequential and a master–worker parallel version, is shown in Fig. 7. A function builder is defined for each implementation (*Sequential* and *MasterWorker* objects in Fig. 7(a)), and the proper version is chosen at construction time using pattern matching on the execution context configured in the configuration parameters (Fig. 7(b)). The only difference between the two implementations is that in the parallel version, after the population is generated, it is distributed before it is evaluated and improved.

This similarity between the implementations is possible because *Population* was defined as an abstract class (Fig. 1) which can be in one of two different states, grouped – the initial state – or distributed. Both states are implemented differently by subclassing *Population*, and *group* and *distribute* operations are available to switch between states. Fig. 8 shows an example of how the *evaluate* operation is implemented in two *Population* subclasses: (i) a grouped population that stores the individuals using a Scala collection; and (2) a distributed population that stores the individuals using a Spark RDD. Both implementations evaluate the population by delegating the evaluation to the population individuals, using the collection *map* in the grouped population and the RDD *map* in the distributed population.

3.5. *Base evolutionary metaheuristic*

Because similar population generation methods and termination conditions are used by many population-based metaheuristics, to facilitate the development of new metaheuristics, the framework provides a *BaseEvolutionaryMetaheuristic* with some common functionalities already implemented. Metaheuristics which reuse this *BaseEvolutionaryMetaheuristic* will have the following features:

- An evolution state comprised of the population being evolved, the number of generations, the number of fitness evaluations performed, and the evolution time elapsed.
- Access to the configuration parameters and the properties store (Section 3.3).
- Sequential and parallel implementations of a population generation method that can be configured with different strategies for the generation and evaluation of the individuals. A random generation strategy is used by default and fitness functions from some common benchmarks are included out of the box [98].
- A configurable termination condition based on a combination of different criteria, i.e. number of fitness function evaluations, evolution elapsed time, quality of the best solution found so far, number of generations and number of evolutions without improvement.
- Logging of meaningful information at each iteration and at the end of the evolution.
- *Hook methods* at some points of the evolution (e.g. after generating the initial population, at the beginning and at the end of each iteration) which can be overridden to add additional features such as custom logging, checkpointing the evolution state, etc. without having to modify the metaheuristic logic.

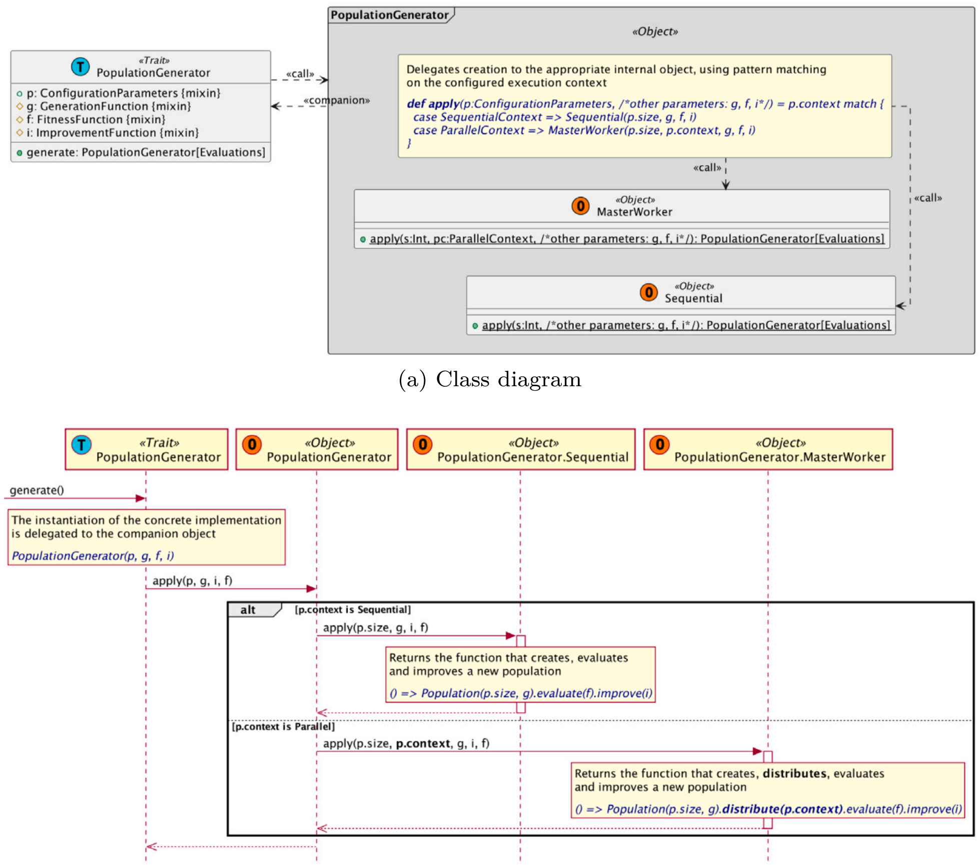
This common functionality is expected to be sufficient for many metaheuristics that will only have to provide the implementation of the function to evolve the population (Fig. 3).

4. A general template for PSO

To validate the approach, this section presents the implementation of a general template for PSO developed using the abstractions provided by the framework explained in Section 3. PSO was selected as a proof of concept because it is one of the most popular metaheuristics found in the literature, which has been applied to all kinds of complex optimization problems in different fields, and that, although it has a simple algorithmic description, it also has a huge number of variants and improvements proposed over the years. The PSO template includes support to configurable topologies and strategies to instantiate different variants of the algorithm, which can be executed on Spark clusters using the sequential or parallel models provided by the framework. To the best of our knowledge, there is no other PSO template that supports all these features together.

4.1. *The PSO algorithm*

PSO is an stochastic optimization algorithm originally proposed in [90] that is inspired by the collective behavior observed in some animal groups, such as flocks of birds or schools of fish, in which members cooperate in foraging, changing their behavior based on their own experiences and those of other members. The algorithm represents the set of candidate solutions as a swarm of particles that


PopulationGenerator
«Object»
«Trait» PopulationGenerator
p: ConfigurationParameters {mixin}
g: GenerationFunction {mixin}
f: FitnessFunction {mixin}
i: ImprovementFunction {mixin}
generate: PopulationGenerator[Evaluations]
«call»
«companion»
Delegates creation to the appropriate internal object, using pattern matching on the configured execution context
def apply(p:ConfigurationParameters, /*other parameters: g, f, i*/) = p.context match {
case SequentialContext => Sequential(p.size, g, f, i)
case ParallelContext => MasterWorker(p.size, p.context, g, f, i)
}
«Object» MasterWorker
apply(s:Int, pc:ParallelContext, /*other parameters: g, f, i*/): PopulationGenerator[Evaluations]
«Object» Sequential
apply(s:Int, /*other parameters: g, f, i*/): PopulationGenerator[Evaluations]
«Trait» PopulationGenerator
«Object» PopulationGenerator
«Object» PopulationGenerator.Sequential
«Object» PopulationGenerator.MasterWorker
generate()
The instantiation of the concrete implementation is delegated to the companion object
PopulationGenerator(p, g, f, i)
apply(p, g, i, f)
alt [p.context is Sequential]
apply(p.size, g, i, f)
Returns the function that creates, evaluates and improves a new population
() => Population(p.size, g).evaluate(f).improve(i)
[p.context is Parallel]
apply(p.size, p.context, g, i, f)
Returns the function that creates, distributes, evaluates and improves a new population
() => Population(p.size, g).distribute(p.context).evaluate(f).improve(i)
(a) Class diagram

(b) Sequence diagram

Fig. 7. Sequential and Master–Worker implementations of the PG method of Fig. 4.

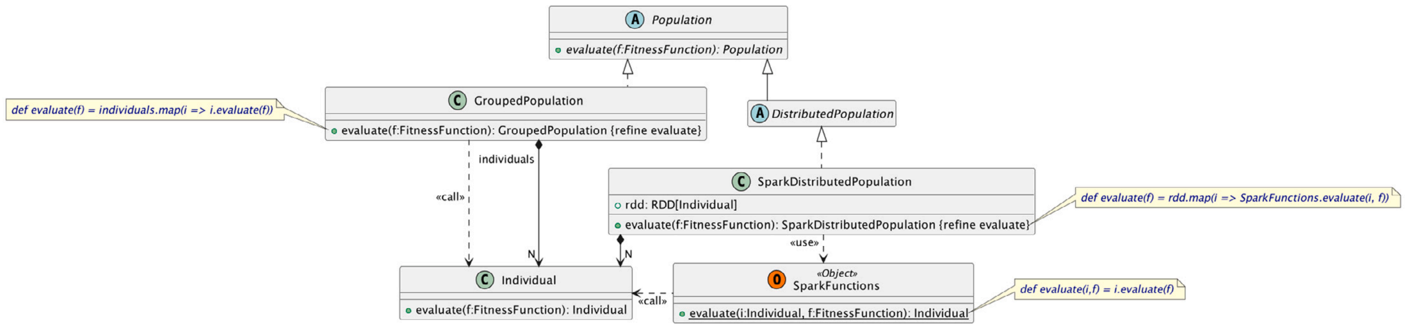

Population
evaluate(f:FitnessFunction): Population
GroupedPopulation
evaluate(f:FitnessFunction): GroupedPopulation {refine evaluate}
def evaluate(f) = individuals.map(i => i.evaluate(f))
DistributedPopulation
SparkDistributedPopulation
rdd: RDD[Individual]
evaluate(f:FitnessFunction): SparkDistributedPopulation {refine evaluate}
def evaluate(f) = rdd.map(i => SparkFunctions.evaluate(i, f))
individuals
«call»
N
«use»
Individual
evaluate(f:FitnessFunction): Individual
«Object» SparkFunctions
evaluate(i:Individual, f:FitnessFunction): Individual
def evaluate(i,f) = i.evaluate(f)

Fig. 8. Class diagram of the implementation of the *evaluate* operation in the grouped and distributed states of *Population*. As recommended in the Spark documentation, the function to be distributed to Spark executors to evaluate the individuals is implemented in a separate object.

traverse a multidimensional search space seeking the optimum. During the traversal, the position and velocity of each particle are updated considering both its own trajectory and those of the other particles in the swarm. Better solutions are progressively found by iteratively updating and evaluating the particles, which eventually will lead the swarm towards the global optimum.

Algorithm 4 shows the pseudocode of the PSO algorithm used as a basis in this work for the development of the general template. Its main steps are the following:

- *Initializing the swarm*. The first step of the algorithm initializes the N particles of the swarm. Each particle stores its current position, velocity, fitness, and its best historical position and fitness. Note that, as this work is aimed at continuous optimization problems, position and velocity are assumed to be D-dimensional vectors of real values. At the time of creation, the position and velocity are initialized and the objective function is evaluated to obtain the initial fitness value. Also, the particle best is initialized with the initial position and fitness.
- *Moving the swarm*. This is the step that provides the algorithm with its optimization capability. The position and velocity of each particle are updated based on its own trajectory, as well as the trajectory of its neighbors.
- *Evaluating the swarm*. In this step the particles are evaluated individually, i.e., the fitness value is calculated evaluating the objective function at the current position. Furthermore, the historical best of the particle is updated if the best fitness is improved.

The move and evaluate steps are repeated until the termination condition is fulfilled. In the synchronous version of the algorithm, all

```
object AnEvolutionaryMetaheuristic {

 trait PopulationGenerator
 extends EvolutionaryMetaheuristic.PopulationGeneration[
    Evaluations] {
   self:AlgorithmStep
     with ElementGenerator
     with IndividualImprover
     with ElementEvaluator =>
   override
    def generate:PopulationGenerator[Evaluations] = {
      /* concrete implementation */
    }
 }

 trait PopulationEvolution
 extends EvolutionaryMetaheuristic.PopulationEvolution[
    Evaluations] {
   self:AlgorithmStep
     with ... =>  // other dependencies
   override
    def evolution:EvolutionStep[Evaluations] = {
      /* concrete implementation */
    }
 }

 trait TerminationCondition
 extends EvolutionaryMetaheuristic.TerminationCondition[
    EvolutionState[Evaluations]] {
   self:AlgorithmStep
     with ... => // other dependencies
   override
    implicit def converge:TerminationConditionFunction[
     EvolutionState[Evaluations]] = {
      /* concrete implementation */
    }
 }

 // dependencies to mix the traits in the algorithm
 trait AlgorithmDependencies
 extends ElementGenerator // PopulationGenerator
   with IndividualImprover
   with ElementEvaluator
   with ...                // other dependencies

 case class Algorithm(
   implicit override val config:ConfigurationParameters)
 extends EvolutionaryMetaheuristic.Algorithm[Evaluations]
   with AlgorithmTemplate
   with AlgorithmDependencies
   with PopulationGenerator
   with PopulationEvolution
   with TerminationCondition
}
```

Listing 2: Implementation of a population-based metaheuristic.

```
// create a metaheuristic with default configuration
val config = ConfigurationParameters()
val metaheuristic =
      AnEvolutionaryMetaheuristic.Algorithm(config)
// apply the metaheuristic and get the best solution
val (population,evaluations,duration) = metaheuristic()
val best = population.best
```

Listing 3: Instantiation and execution of a metaheuristic.

particles are updated at once in each iteration, while in the asynchronous version one particle is updated at a time. The particle with the best historical fitness is returned as the best solution found.

In PSO the neighborhoods are determined by the social topology, which dictates which particles share information during the search. The two most common topologies are: (i) *global best (gbest)*, in which each particle is connected to all other particles; and (ii) *local best (lbest)*, in which each particle is connected to its nearest neighbors, usually two. In both topologies, the particles share their best position with their neighbors and the best of the neighborhood is used to update the velocity of the particles. As it is well known that the topology affects the performance of the algorithm, and that the optimal topology is problem dependent, other topologies, static and dynamic, and *models of influence* (using the nomenclature of [99]) have been proposed in the literature [100–103].

Algorithm 4: Pseudocode of the PSO algorithm.

```
1: initialize a swarm of N particles
2: while the termination condition is not met do
3:     for $i = 1 \to N$ do
4:         update the velocity of particle$_i$
5:         update the position of particle$_i$
6:         evaluate the fitness of particle$_i$
7:         if the fitness of particle$_i$ is better than its historical best then
8:             update the historical best of particle$_i$
9:         end if
10:    end for
11: end while
12: return the best particle
```

4.2. PSO variants

Over the years many modifications have been proposed to overcome PSO limitations and improve its performance, making it widely applicable to a great variety of optimization problems. For this work, a representative set of modifications were selected to provide a significant number of choices in all the main components of the algorithm. The *Strategy* approach (Section 3.2) was applied to implement the selected variants, which are summarized in Table 4 and their signatures shown in Table 2. The variants were selected, for the most part, based on some recent PSO reviews [104–109] and can be grouped in the following categories:

- *Particle initialization*, which comprises modifications in the initialization of the particles position and velocity. For the position, the *ElementGenerator* strategy already implemented in *BaseEvolutionaryMetahuristic* (Section 3.5) that, by default, generates random positions (*Elements* in the framework jargon) was reused. For the velocity, a strategy *VelocityInitialization* with two instances to initialize the velocity to zero or to a random value within the reduced search space range, were implemented.
- *Particle update*, which comprises modifications in the updating of the position and velocity of particles. For the position, a strategy *PositionUpdate* with an instance to apply the position update of the standard PSO 2007 [110] was implemented. For the velocity, a strategy *VelocityUpdate* with three instances was implemented: (i) the velocity update of the standard PSO 2007; (ii) the version with constriction coefficient [111]; and (iii) the condensed form of the previous. All the instances of the *VelocityUpdate* strategy were implemented to accept different models of influence (represented as $\vec{I}_i^t$ in the equations) to take into account the contribution of the neighborhood to the velocity update (Section 4.5). This category is completed with optional strategies, *MovementBound* and *VelocityClamping*, to keep the position and velocity of particles within given limits.
- *Control parameters*, which comprises proposals to initialize and adjust the control parameters of the algorithm and its components. A strategy *InertiaWeightAdjustment* with seven instances was implemented to dynamically adjusting the inertia weight parameter (w) used in the velocity update equation of the standard PSO, and two strategies, *VelocityLimitInitialization* and *VelocityLimitUpdate*, were implemented to optionally initialize and update the velocity limit when velocity clamping is activated.

Table 2
Signatures of the strategies used to implement the PSO variants.

Strategy	Signature
ElementGenerator	$() \longrightarrow Position$
VelocityInitialization	$() \longrightarrow Velocity$
PositionUpdate	$(Particle, Velocity) \longrightarrow Particle$
VelocityUpdate	$(Particle, Influence) \longrightarrow Velocity$ *Influence* = neighborhood influence
MovementBound	$(Movement, Bound, Bound) \longrightarrow Movement$ $Movement = (Position, Velocity)$
VelocityClamping	$(Velocity, Bound) \longrightarrow Velocity$
InertiaWeightAdjustment	$(Double, State) \longrightarrow Double$
VelocityLimitInitialization	$() \longrightarrow Velocity$
VelocityLimitUpdate	$(Velocity, State) \longrightarrow Velocity$

Table 3
Notation used in the article.

$\vec{x}_i^t$	position of the i-th particle at time t
$\vec{v}_i^t$	velocity of the i-th particle at time t
w	inertia weight
χ	constriction factor
c_1	cognitive coefficient
c_2	social coefficient
c_{max}	upper limit of coefficients sum
$\vec{r}_{1i}^{\,t}, \vec{r}_{2i}^{\,t}$	uniformly distributed random value vectors
$\vec{p}_i^t$	best historical position of the i-th particle at time t
$\vec{g}^t$	best solution at time t
$\vec{I}_i^t$	neighborhood influence on the i-th particle at time t
$\vec{x}_{min}, \vec{x}_{max}$	search space limits
$\vec{v}_{max}$	velocity limit
w_{min}, w_{max}	inertia weight limits
g	current generation
T	maximum elapsed time
G	maximum number of generations
f	fitness function
U	uniform distribution

4.3. The PSO template

One of the difficulties to be faced when implementing a template for PSO is that, due to the huge number of modifications introduced to the original proposal over the years, it would be unfeasible to have a template that could accommodate every proposed modification. But unlike SS, for which a five-method template was proposed in [114] that has served as the main reference for most implementations to date, including our germinal work [19] on the framework proposed in this paper, even there is no agreement on a common template for PSO. Different proposals can be found in the literature, e.g. the template used in [93] for the grammar-guided genetic programming (GGGP) of PSO algorithms; the template used in PSO-X [99], a component-based framework for the automatic generation of PSO algorithms; or the templates found in some of the most popular MOFs, like Paradiseo [5], ECJ [4], HeuristicLab [115] or JMetal (v6.0) [116].

The pseudo-code of the PSO template used in this paper is shown in algorithm 5. The strategies in Table 4, the fitness function and the termination condition are the inputs of the template. The strategies applied at each step are shown on the right side. When the strategy is used to parameterize the step, it is shown inside square brackets (i.e. the parameters initialized and updated in steps 1 and 12 of the algorithm depend on the concrete instance of the *VelocityUpdate* strategy, although the strategy is not applied in the step). It should be noted that, although it would be possible to implement both synchronous and asynchronous versions of the PSO algorithm with the template, a synchronous version was assumed in which all particles are updated at once, which has an effect on the implementation of models of influence (Section 4.5) and parallel models (Section 4.6).

Algorithm 5 Pseudo-code of the PSO template.

```
    Input strategies: EG (position generation); VI (velocity initialization);
    VU (velocity update); VC (velocity clamping); PU (position update);
    MB (movement bound); WA (inertia weight adjustment); VLI (velocity
    limit initialization); VLU (velocity limit update); FF (fitness function);
    TC (termination condition)
    Output: the best particle found
 1: initialize the algorithm parameters                      ▷ [VU], VLI
 2: initialize a swarm of N particles                        ▷ EG, VI, FF
 3: while ¬$TC$ do                                           ▷ TC
 4:     for $i = 1 \to N$ do
 5:         update the velocity of particle$_i$              ▷ VU, VC
 6:         update the position of particle$_i$              ▷ PU, MB
 7:         evaluate the fitness of particle$_i$             ▷ FF
 8:         if the fitness of particle$_i$ is better than its best so far then
 9:             update the historical best of particle$_i$
10:         end if
11:     end for
12:     update the algorithm parameters                      ▷ [VU], WA, VLU
13: end while
14: return the best particle
```

In the implementation of the template (Fig. 9 and listing 4), the abstractions provided by the framework were applied as explained in Section 3.2. Furthermore, the template reuses *BaseEvolutionaryMetaheuristic* (Section 3.5), so only the implementation of the evolution step (Fig. 3) is required, as all other components of the algorithm are already provided by the framework. The PSO evolution step is declared as a trait that extends the *BaseEvolutionaryMetaheuristic* evolution step and overrides the *evolution* method. The creation of the concrete instance of the evolution strategy defined in the configuration parameters is delegated to the companion object. Several instances have been implemented, both sequential and parallel, which are described in more detail in Section 4.6.

Similarly, the PSO algorithm is declared by extending the *BaseEvolutionaryMetaheuristic* algorithm and adding the dependencies on traits that are specific to PSO, which includes those required by the evolution step. These dependencies bring into the context of the algorithm the definitions of the strategies needed to manage the PSO variants described in Section 4.2. Since a detailed description of their implementations is not necessary for a general understanding of the proposal, it is left out of the scope of this article. Interested readers are referred to the source code in the companion repository [21].

The rest of the functionality required by the PSO template is implemented by overriding the hook methods defined in the *BaseEvolutionaryMetaheuristic* algorithm, which mainly includes:

1. Converting the initial *Population* created by the *BaseEvolutionaryMetaheuristic* population generation method into a *Swarm* and grouping or distributing it according to the configuration parameters.
2. Initializing and updating algorithm and strategy-related parameters in the properties store, as well as evolution metrics not included in the evolution state.
3. Detailed logging for debugging.

4.4. Particle and swarm

Some abstractions has been added to the framework, as part of the PSO template implementation, to model particles and swarms. Particles (Fig. 10) have been modeled as individuals (Fig. 1) with an identifier that is used to assemble neighborhoods, a velocity, and a historical best. The *Particle* class defines a *move* method with a parameter for the

Table 4
PSO variants implemented as strategies in this paper. The notation used is explained in Table 3.

Strategy	Instances	Description
ElementGenerator	Random	$\vec{x}_i^0 \sim U[\vec{x}_{min}, \vec{x}_{max}]$
VelocityInitialization	Zero	$\vec{v}_i^0 = \vec{0}$
	Bounded	$\vec{v}_i^0 \sim U[\vec{x}_{min}, \vec{x}_{max}] \times k$ for a given $k \in [0,1]$
PositionUpdate	Standard 2007	$\vec{x}_i^{t+1} = \vec{x}_i^t + \vec{v}_i^{t+1}$
VelocityUpdate	Standard 2007	$\vec{v}_i^{t+1} = w\vec{v}_i^t + c_1\vec{r}_{1i}^{\,t}(\vec{p}_i^t - \vec{x}_i^t) + c_2\vec{r}_{2i}^{\,t}(\vec{I}_i^t - \vec{x}_i^t)$ with $\vec{r}_{1i}^{\,t}, \vec{r}_{2i}^{\,t} \sim U[0,1]$
	ConstrictionFactor	$\vec{v}_i^{t+1} = \chi(\vec{v}_i^t + c_1\vec{r}_{1i}^{\,t}(\vec{p}_i^t - \vec{x}_i^t) + c_2\vec{r}_{2i}^{\,t}(\vec{I}_i^t - \vec{x}_i^t))$ with $\vec{r}_{1i}^{\,t}, \vec{r}_{2i}^{\,t} \sim U[0,1]$
	CondensedConstrictionFactor	$\vec{v}_i^{t+1} = \chi(\vec{v}_i^t + c_{max}(\vec{I}_i^t - \vec{x}_i^t))$
MovementBound	None	movement is not restricted (i.e. *let them fly*)
	Standard 2007	$(\vec{x}_i^{t+1}, \vec{v}_i^{t+1}) = \begin{cases} (\vec{x}_{min}, \vec{0}) \text{ if } \vec{x}_i^{t+1} < \vec{x}_{min} \\ (\vec{x}_{max}, \vec{0}) \text{ if } \vec{x}_i^{t+1} > \vec{x}_{max} \\ (\vec{x}_i^{t+1}, \vec{v}_i^{t+1}) \text{ otherwise} \end{cases}$
VelocityClamping	None	velocity is not limited
	Bounded	$\vec{v}_i^{t+1} = \begin{cases} -\vec{v}_{max} \text{ if } \vec{v}_i^{t+1} < -\vec{v}_{max} \\ \vec{v}_{max} \text{ if } \vec{v}_i^{t+1} > \vec{v}_{max} \\ \vec{v}_i^{t+1} \text{ otherwise} \end{cases}$
InertiaWeightAdjustment [112]	None	$w = c$ for a given c
	LinearDecreasingTime	$w^t = w_{min} + (w_{max} - w_{min}) \times (\frac{T-t}{T})$
	LinearDecreasingGenerations	$w^t = w_{min} + (w_{max} - w_{min}) \times (\frac{G-g}{G})$
	Random	$w^t \sim U[0.5, 1)$
	ChaoticTime	$w^t = w_{min} \times z + (w_{max} - w_{min}) \times (\frac{T-t}{T})$ with $z = 4.0 \times r \times (1-r), r \sim U[0,1]$
	ChaoticGenerations	$w^t = w_{min} \times z + (w_{max} - w_{min}) \times (\frac{G-g}{G})$ with $z = 4.0 \times r \times (1-r), r \sim U[0,1]$
	ChaoticRandom	$w^t = 0.5 \times (z + r)$ with $z = 4.0 \times r \times (1-r), r \sim U[0,1]$
VelocityLimitInitialization	None	$\vec{v}_{max}$ is not set
	Constant	$\vec{v}_{max} = c$ for a given c
	Factor	$\vec{v}_{max} = 0.5 \times (\vec{x}_{max} - \vec{x}_{min}) \times k$ for a given $k \in (0,1]$
VelocityLimitUpdate	None	$\vec{v}_{max}$ if set is kept constant
	LVDM [113]	$(\vec{w}^{\,t+1}, \vec{v}_{max}^{\,t+1}) = (\alpha\vec{w}^{\,t}, \beta\vec{v}_{max}^{\,t})$ if $f(\vec{g}^{\,t}) \geq f(\vec{g}^{\,t-h})$ for given h and $\alpha, \beta \in (0,1)$

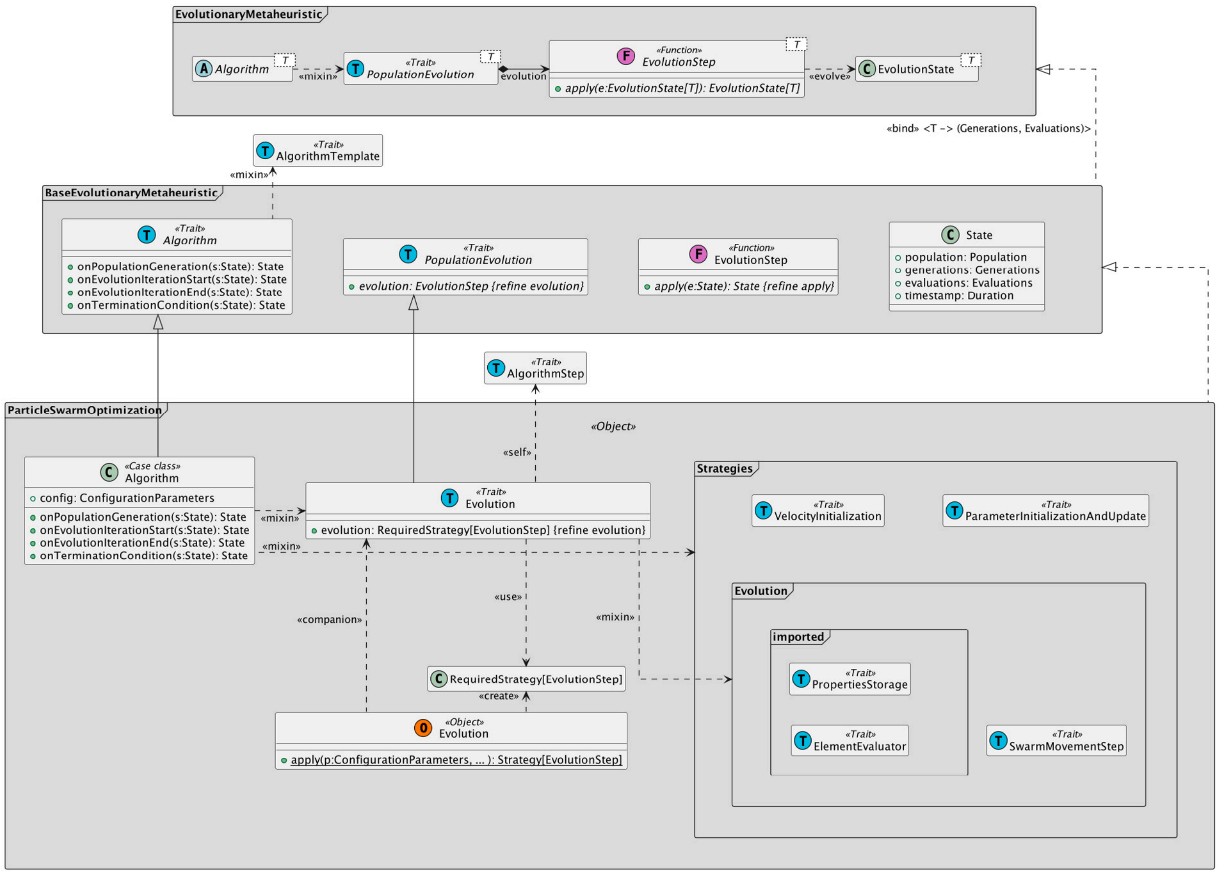

EvolutionaryMetaheuristic
Algorithm
T
«mixin»
«Trait»
PopulationEvolution
evolution
«Function»
EvolutionStep
apply(e:EvolutionState[T]): EvolutionState[T]
«evolve»
EvolutionState
«bind» <T -> (Generations, Evaluations)>
«Trait»
AlgorithmTemplate
«mixin»
BaseEvolutionaryMetaheuristic
«Trait»
Algorithm
onPopulationGeneration(s:State): State
onEvolutionIterationStart(s:State): State
onEvolutionIterationEnd(s:State): State
onTerminationCondition(s:State): State
«Trait»
PopulationEvolution
evolution: EvolutionStep {refine evolution}
«Function»
EvolutionStep
apply(e:State): State {refine apply}
State
population: Population
generations: Generations
evaluations: Evaluations
timestamp: Duration
«Trait»
AlgorithmStep
ParticleSwarmOptimization
«Object»
«self»
«Case class»
Algorithm
config: ConfigurationParameters
onPopulationGeneration(s:State): State
onEvolutionIterationStart(s:State): State
onEvolutionIterationEnd(s:State): State
onTerminationCondition(s:State): State
«mixin»
«mixin»
«Trait»
Evolution
evolution: RequiredStrategy[EvolutionStep] {refine evolution}
«use»
«companion»
«mixin»
RequiredStrategy[EvolutionStep]
«create»
«Object»
Evolution
apply(p:ConfigurationParameters, ...): Strategy[EvolutionStep]
Strategies
«Trait»
VelocityInitialization
«Trait»
ParameterInitializationAndUpdate
Evolution
imported
«Trait»
PropertiesStorage
«Trait»
ElementEvaluator
«Trait»
SwarmMovementStep

Fig. 9. Class diagram of the implementation of the PSO template.

Table 5
Topologies supported by the framework out of the box. The topologies in the second row are supported through JGrapT [117]. With few exceptions, the topologies are parameterizable, e.g. for the Ring topology, the generalized version with k neighbors is provided.

Complete (*aka* GBest, Star, Global, All-to-All), Directed Ring, Global (alias for Complete, optimized to be more efficient), KClusters, Pyramid, Random, Regular (with small-world shortcut [100]), Regular Random, Ring (*aka* LBest), Square (*aka* VonNewman)
Directed Scale Free, Generalized Petersen, G_{nm} Random, G_{np} Random Bipartite, Grid, HyperCube, Kleinberg's Small World, Linear, Planted Partition, Scale-Free Network, Star (one particle in the center connected to all the others), Wheel, Windmill

```
object ParticleSwarmOptimization {
 // aliases for types in BaseEvolutionaryMetaheuristic
 type State = BaseEvolutionaryMetaheuristic.State
 type EvolutionStep =
       BaseEvolutionaryMetaheuristic.EvolutionStep

 // the evolution step reuses the base evolution and
 // uses an strategy specific to PSO
 trait Evolution
 extends
  BaseEvolutionaryMetaheuristic.PopulationEvolution {
    self: AlgorithmStep
     with PropertiesStorage
     with ElementEvaluator
     // factory of functions for moving particles
     with SwarmMovementStep =>
   override
   def evolution:RequiredStrategy[EvolutionStep] =
     Evolution()
 }

 // the companion object extends a custom strategy
 // factory and overrides the factory method to
 // instantiate the evolution step
 object Evolution
 extends
  EvolutionStrategyFactory[EvolutionStep] {
   override
   def apply()(implicit config:ConfigurationParameters,
     ps:PropertiesStore,
     ff:FitnessFunction,
     fm:ParticleMovementFunctionFactory
   ):Strategy[EvolutionStep] = {...}
 }

 // the template extends the base algorithm with PSO
 // strategies and overrides the hook methods
 case class Algorithm(
   implicit override val config:ConfigurationParameters)
 extends
  BaseEvolutionaryMetaheuristic.Algorithm
   // strategies to initialize and update PSO parameters
   with ParameterInitializationAndUpdate
   with Evolution
   // strategies to initialize velocities
   with VelocityInitialization
   with SwarmMovementStep {
  // overridden hook methods of type EvolutionStep
  // i.e. a State is received as input and an updated
  // State is returned as output
  override
  def onPopulationGeneration:State = { State => ... }
  override
  def onEvolutionIterationStart:State = { State => ... }
  override
  def onEvolutionIterationEnd:State = { State => ... }
  override
  def onTerminationCondition:State = { State => ... }
 }
}
```

Listing 4: Implementation of the PSO template.

```
GBest {
  topology {
    self = false
    shape.name = Global
    neighborhood_influence.name = Best
  }
}

CustomRing {
  topology {
    self = true
    shape {
      name = Custom
      neighborhoods {
        0: [1, 2, 4],
        1: [2, 3, 0],
        2: [3, 4, 1],
        3: [4, 0, 2],
        4: [0, 1, 3]
      }
    }
    neighborhood_influence.name = wdFIPS
  }
}
```

Listing 5: Examples of configurations of social topologies supported by the framework.

concrete movement function to use and overrides some of the methods inherited from *Individual*. Furthermore, in the *Particle* companion object, the velocity initialization strategy to be used when creating particles can be specified via a parameter.

With regard to swarms, they have been modeled as populations extended with operations to add support to some specific swarm behaviors (Fig. 11) and to neighborhoods (Section 4.5). Like *Population*, swarms can be in one of two states, grouped – its initial state – or distributed, which are implemented by extending the corresponding population state (Section 3.4) with swarm operations, and overriding them differently in the subclasses. The implementation of swarm states and parallel models is explained in Section 4.6.

4.5. Topology and neighborhood influence

The PSO template includes a generic representation for social topologies. It assumes that particles have unique identifiers, and represents a social topology as a tuple $(self, shape, influence)$, where:

- *self*, is a logic value to indicate whether particles should be considered to be part of their neighborhoods.
- *shape*, is the *topology shape* that configures the swarm neighborhoods, represented as a DAG (*Directed Acyclic Graph*) of particle identifiers [101]. An edge $p_i \rightarrow p_j$ in the graph indicates that p_i is a neighbor of p_j and could share information with it. The topology shapes included out of the box are listed in Table 5. The framework also supports the configuration of custom topologies and exporting/importing topologies to/from GraphViz *.dot* files.
- *influence*, is a function $f : i \rightarrow \vec{I}_i$ that maps particle identifiers to the neighborhood influences used in velocity update equations (Table 4). The framework also provides abstractions to represent generic models of influence (Fig. 12) that are instantiated through the implementation of different strategies. To the best of our knowledge, this is a unique feature not available in other PSO templates [4,5,93,99,115,116].

An example of the configurations of two social topologies is shown in listing 5: (i) the *GBest* topology; and (ii) a custom *Ring* topology with a FIPS weighted by distance model of influence for a swarm of size N=5, in which the neighborhood of a particle p_i is defined as $Neighborhood(p_i) = \{p_i, p_{i+1\ mod\ N}, p_{i+2\ mod\ N}, p_{i-1\ mod\ N}\}$.

The instantiation of models of influence is supported by two operations defined in the *NeighborhoodOps* trait (Fig. 12):

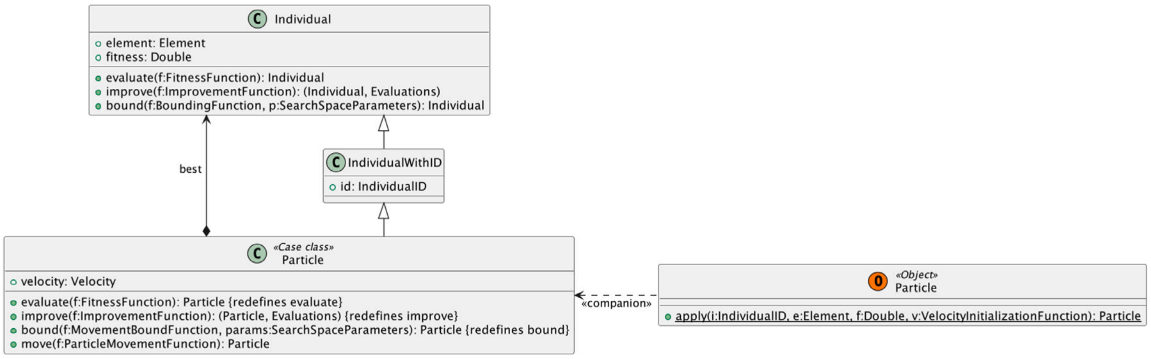

Individual
element: Element
fitness: Double
evaluate(f:FitnessFunction): Individual
improve(f:ImprovementFunction): (Individual, Evaluations)
bound(f:BoundingFunction, p:SearchSpaceParameters): Individual
best
IndividualWithID
id: IndividualID
«Case class» Particle
velocity: Velocity
evaluate(f:FitnessFunction): Particle {redefines evaluate}
improve(f:ImprovementFunction): (Particle, Evaluations) {redefines improve}
bound(f:MovementBoundFunction, params:SearchSpaceParameters): Particle {redefines bound}
move(f:ParticleMovementFunction): Particle
«companion»
«Object» Particle
apply(i:IndividualID, e:Element, f:Double, v:VelocityInitializationFunction): Particle

Fig. 10. Class diagram of the *Particle* class.

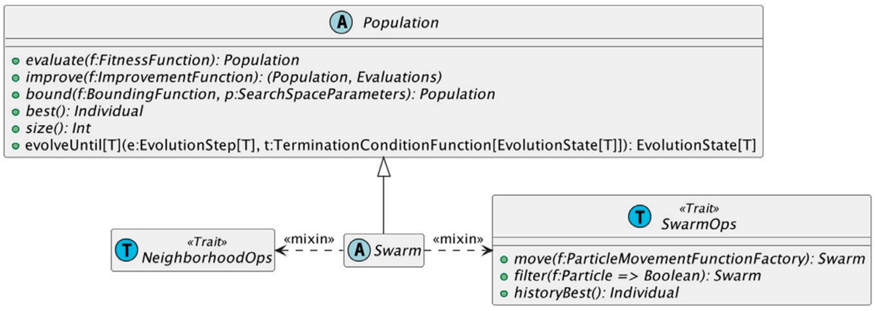

Population
evaluate(f:FitnessFunction): Population
improve(f:ImprovementFunction): (Population, Evaluations)
bound(f:BoundingFunction, p:SearchSpaceParameters): Population
best(): Individual
size(): Int
evolveUntil[T](e:EvolutionStep[T], t:TerminationConditionFunction[EvolutionState[T]]): EvolutionState[T]
«Trait» NeighborhoodOps
«mixin»
Swarm
«mixin»
«Trait» SwarmOps
move(f:ParticleMovementFunctionFactory): Swarm
filter(f:Particle => Boolean): Swarm
historyBest(): Individual

Fig. 11. Class diagram of the *Swarm* type.

1. *neighborhoodInfluence*, a generic function builder to instantiate *neighborhood influence functions*, i.e. the functions to get the neighborhood influence used in velocity update equations (Table 4). The concrete function to instantiate is configured through the following arguments: (i) *self* and *topology*, the components defined in the social topology configuration; (ii) *collect* and *reduce*, the strategies used to collect the contribution of each particle to the neighborhood, and to calculate the influence the neighborhood has on a particle from the contributions of the particle and its neighbors. Currently, the framework provides implementations of these strategies to support all the models described in [103], i.e. *best*, FIPS, wFIPS, wdFIPS, Self and wSelf.
2. *contribution*, a template abstract method called from the implementation of *neighborhoodInfluence* to collect the information shared by each particle with its neighborhood using the *collect* strategy passed as argument.

The *neighborhoodInfluence* function builder is called on every swarm move to obtain the neighborhood influence function to use in the updating of the velocity of particles. This will allow to add support for dynamic topologies in future releases, although only static topologies are supported for now. The *neighborhoodInfluence* implementation applies the *collect* and *reduce* strategies following the *collect-reduce* approach shown in algorithm 6. Currently, only the distribution of the *collect* strategy is supported in the parallel versions of the algorithm.

Note that in this approach, social topologies and swarms are treated as independent entities, only related by the particle identifiers. The main reasons to have adopted this decision are:

- It is not necessary to store information about the topology either in the particles or in the swarm itself, which is very convenient when the topology has to be changed (e.g. when using dynamic topologies or swarms of varying size), especially if the swarm is distributed.
- The topology can be stored in the properties store and accessed only by the strategies requiring it, instead of being passed through all the strategies as part of the evolution state (Section 3.3).

Algorithm 6 Pseudo-code of *neighborhoodInfluence*.

```
    Inputs: self; topology; collect(); reduce()
    Output: the neighborhood influence function
    ▷ the contribution of each particle is collected
 1: $contrib$ = contribution(collect)
 2: for each $id$ in topology do
        ▷ neighbors are the predecessors in the DAG
 3:     $neighbors$ = topology.predecessors($id$)
 4:     if self then
            ▷ the particle $id$ is included in the neighborhood
 5:         $neighborhood = id +\!\!+ neighbors$
 6:     else
 7:         $neighborhood = neighbors$
 8:     end if
        ▷ the influence the neighborhood has on the particle is calculated and stored in a map indexed by $id$
 9:     $influence[id]$ = reduce(contrib($id$), contrib($neighborhood$))
10: end for
    ▷ returns the function that maps particle $ids$ to neighborhood influences
11: return $id \Rightarrow influence[id]$
```

- Topologies are implemented in an loosely-coupled package that can be easily reused for other purposes (e.g. to model the migration topology of cooperative islands).
- Relations between swarms and topologies are not restricted to be only one-to-one.

4.6. Sequential and parallel PSO instances

To support the instantiation of different sequential and parallel PSO instances from the PSO template, the same general approach explained in Section 3.4 was applied to the PSO evolution step. Different instances are supported in its companion object through pattern matching

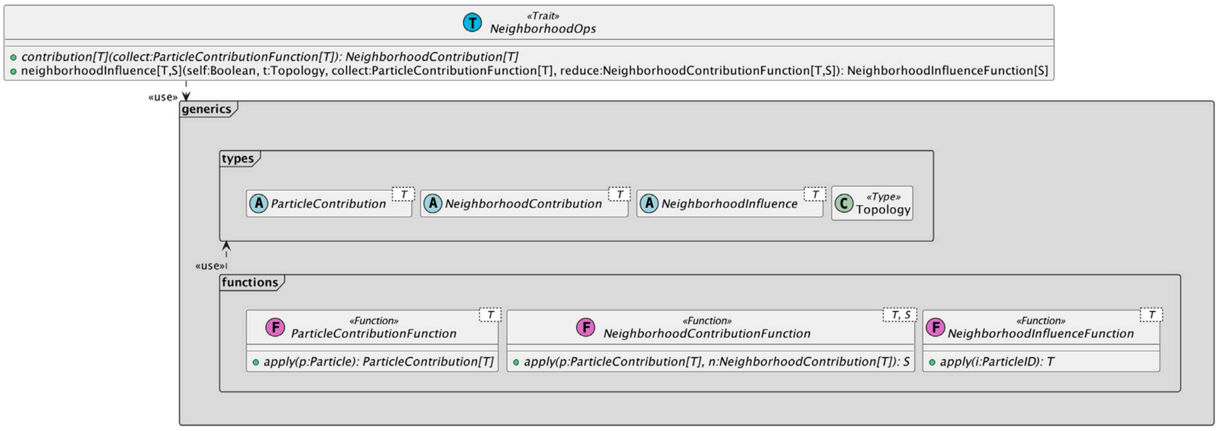

«Trait» NeighborhoodOps
contribution[T](collect:ParticleContributionFunction[T]): NeighborhoodContribution[T]
neighborhoodInfluence[T,S](self:Boolean, t:Topology, collect:ParticleContributionFunction[T], reduce:NeighborhoodContributionFunction[T,S]): NeighborhoodInfluenceFunction[S]
«use»
generics
types
ParticleContribution T
NeighborhoodContribution T
NeighborhoodInfluence T
«Type» Topology
«use»
functions
«Function» ParticleContributionFunction T
apply(p:Particle): ParticleContribution[T]
«Function» NeighborhoodContributionFunction T, S
apply(p:ParticleContribution[T], n:NeighborhoodContribution[T]): S
«Function» NeighborhoodInfluenceFunction T
apply(i:ParticleID): T

Fig. 12. Class diagram of the generic types and functions to represent models of influence.

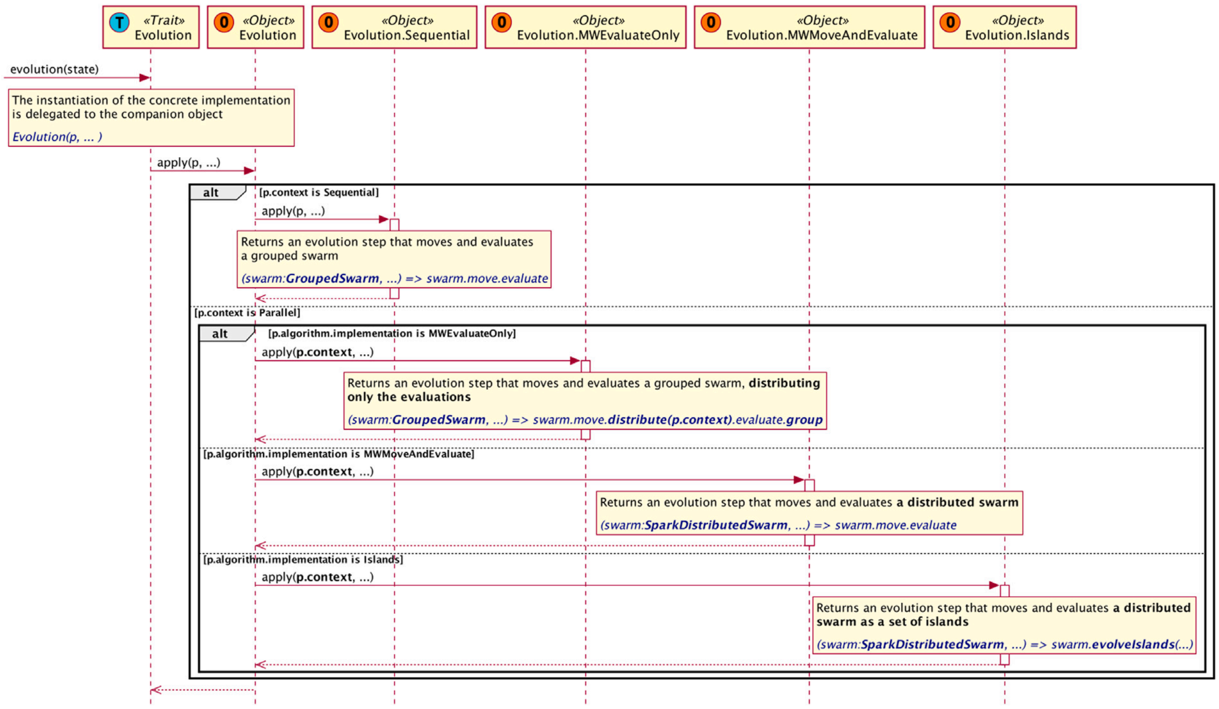
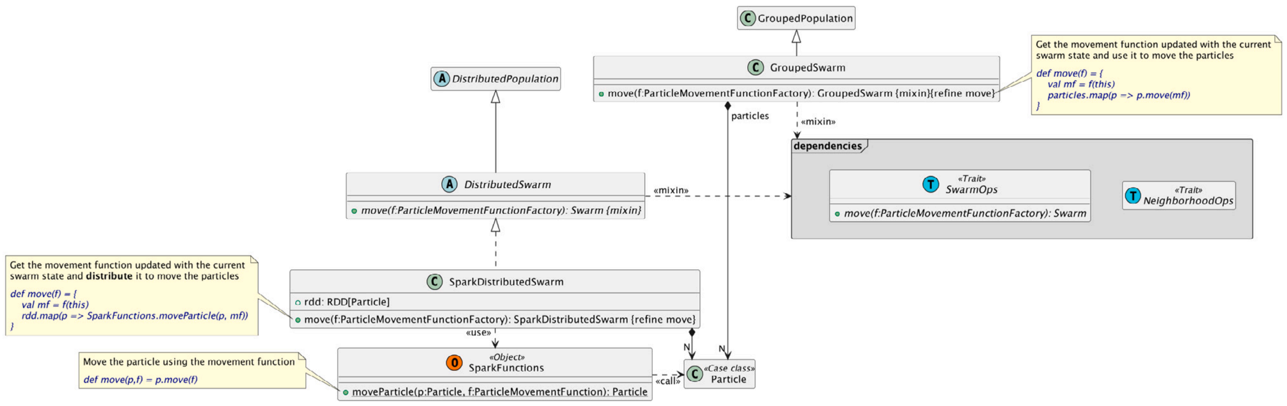

«Trait» Evolution
«Object» Evolution
«Object» Evolution.Sequential
«Object» Evolution.MWEvaluateOnly
«Object» Evolution.MWMoveAndEvaluate
«Object» Evolution.Islands
evolution(state)
The instantiation of the concrete implementation is delegated to the companion object
Evolution(p, ...)
apply(p, ...)
alt
[p.context is Sequential]
apply(p, ...)
Returns an evolution step that moves and evaluates a grouped swarm
(swarm:GroupedSwarm, ...) => swarm.move.evaluate
[p.context is Parallel]
alt
[p.algorithm.implementation is MWEvaluateOnly]
apply(p.context, ...)
Returns an evolution step that moves and evaluates a grouped swarm, distributing only the evaluations
(swarm:GroupedSwarm, ...) => swarm.move.distribute(p.context).evaluate.group
[p.algorithm.implementation is MWMoveAndEvaluate]
apply(p.context, ...)
Returns an evolution step that moves and evaluates a distributed swarm
(swarm:SparkDistributedSwarm, ...) => swarm.move.evaluate
[p.algorithm.implementation is Islands]
apply(p.context, ...)
Returns an evolution step that moves and evaluates a distributed swarm as a set of islands
(swarm:SparkDistributedSwarm, ...) => swarm.evolveIslands(...)

Fig. 13. Sequence diagram of the sequential and parallel implementations of the PSO evolution step.

on configuration parameters, like in the example of Fig. 7. Fig. 13 shows how this was applied to implement a synchronous sequential version and various parallel models: (i) two variants of the master–worker model, one that keeps the swarm distributed between iterations and distributes the updating and evaluation of particles, and the most common that distributes only the evaluations; and (ii) an island model.

As it can be seen, with the exception of the island model, all the implementations are very similar. They only differ on the swarm state they expect to match and whether they change the swarm state. This was achieved using with *Swarm* the same approach used with *Population* (Fig. 8), as it is shown in Fig. 14. The states of a swarm are represented by classes that extend their *Population* counterparts: (i) a grouped swarm that stores the particles using a Scala collection; and (2) a distributed swarm that uses a Spark RDD. Fig. 14 also shows an example of how the *move* operation defined in *SwarmOps* (Fig. 11) was implemented by delegating the movement to the particles, using the collection *map* method in the grouped swarm and the RDD *map* in the distributed swarm. Note that the actual function used to move the particles is obtained from a factory that uses the current state of the swarm to update on the fly the neighborhood influence used in the velocity update equations and the adaptive strategies involved in moving the swarm, e.g. the inertial weight adjustment strategies or LVDM (Table 4). Dynamic topologies will be supported in the same way in future releases.

With regard to the island model, a non-cooperative version with support for both homogeneous and heterogeneous islands has been implemented. The approach followed was to define an abstract operation *evolveIslands* in the trait *SwarmIslandOps* (Fig. 15) that is mixed in with *DistributedSwarm* and overridden by its subclasses. The operation *evolveIslands* regards subswarms as islands that are evolved independently from the current state, possibly using different evolution strategies, until a common termination condition is met. It has the following parameters: (i) the components of the global evolution state needed to initialize the local states from which islands will evolve;


C GroupedPopulation
C GroupedSwarm
move(f:ParticleMovementFunctionFactory): GroupedSwarm {mixin}{refine move}
A DistributedPopulation
A DistributedSwarm
move(f:ParticleMovementFunctionFactory): Swarm {mixin}
«mixin»
particles
«mixin»
dependencies
«Trait» SwarmOps
move(f:ParticleMovementFunctionFactory): Swarm
«Trait» NeighborhoodOps
C SparkDistributedSwarm
rdd: RDD[Particle]
move(f:ParticleMovementFunctionFactory): SparkDistributedSwarm {refine move}
«use»
N
N
«Object» SparkFunctions
moveParticle(p:Particle, f:ParticleMovementFunction): Particle
«call»
«Case class» Particle

Get the movement function updated with the current swarm state and use it to move the particles

```
def move(f) = {
    val mf = f(this)
    particles.map(p => p.move(mf))
}
```

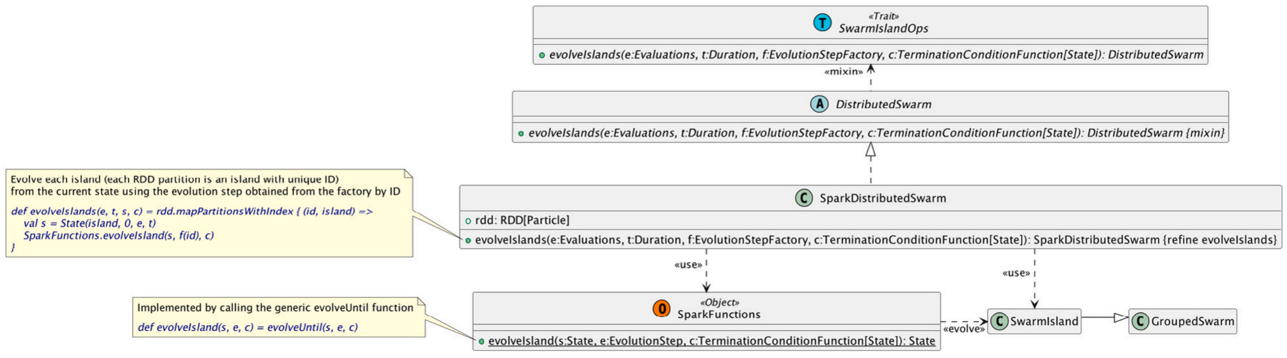
Get the movement function updated with the current swarm state and **distribute** it to move the particles

```
def move(f) = {
    val mf = f(this)
    rdd.map(p => SparkFunctions.moveParticle(p, mf))
}
```

Move the particle using the movement function

```
def move(p,f) = p.move(f)
```

Fig. 14. Class diagram of the implementation of the *move* operation in the grouped and distributed states of a *Swarm*.


«Trait» SwarmIslandOps
evolveIslands(e:Evaluations, t:Duration, f:EvolutionStepFactory, c:TerminationConditionFunction[State]): DistributedSwarm
«mixin»
A DistributedSwarm
evolveIslands(e:Evaluations, t:Duration, f:EvolutionStepFactory, c:TerminationConditionFunction[State]): DistributedSwarm {mixin}
C SparkDistributedSwarm
rdd: RDD[Particle]
evolveIslands(e:Evaluations, t:Duration, f:EvolutionStepFactory, c:TerminationConditionFunction[State]): SparkDistributedSwarm {refine evolveIslands}
«use»
«use»
«Object» SparkFunctions
evolveIsland(s:State, e:EvolutionStep, c:TerminationConditionFunction[State]): State
«evolve»
C SwarmIsland
C GroupedSwarm

Evolve each island (each RDD partition is an island with unique ID) from the current state using the evolution step obtained from the factory by ID

```
def evolveIslands(e, t, s, c) = rdd.mapPartitionsWithIndex { (id, island) =>
    val s = State(island, 0, e, t)
    SparkFunctions.evolveIsland(s, f(id), c)
}
```

Implemented by calling the generic evolveUntil function

```
def evolveIsland(s, e, c) = evolveUntil(s, e, c)
```

Fig. 15. Class diagram of the implementation of the *evolveIslands* operation.

(ii) a factory to get the concrete evolution step of each island by ID; and (iii) a termination condition, common to all islands. Note that the dependencies on the concrete types of the members of the global evolution state, prevent *evolveIslands* from being generally applicable. It is still a very early implementation that will serve as a basis for the implementation of a general version, with support also for cooperation between islands, which is planned for a future release of the framework.

Fig. 15 also shows how *evolveIslands* was overridden in *SparkDistributedSwarm*. Basically, each RDD partition is regarded as an island with unique ID. For each island, the generic *evolveUntil* function (Fig. 2) is used to evolve an initial state, which contains the island itself and the values passed from the global state, using the evolution step that corresponds to the island ID until the termination condition is fulfilled. Finally, the evolved island is returned as output and the number of evaluations of the fitness function updated in the global evolution state – not shown in the figure –.

An example of how PSO instances with homogeneous and heterogeneous islands would be defined in a configuration file is shown in listing 6. The configuration has three parameters: (i) the maximum number of iterations to evolve the islands locally before returning the control to the global evolution; (ii) a collection of island configurations and their amounts; and (iii) a boolean parameter to configure if the amounts should be regarded as quantities or percentages. The actual number of islands of each configuration is calculated by the framework depending on the number of subswarms, i.e. partitions, available at runtime. In the example, all the subswarms will use the same configuration in the homogeneous instance, whereas in the heterogeneous one half of the subswarms will use one configuration and one half the other. Note that a sequential PSO instance is also configured in the example of listing 6 using exactly the same configuration, i.e. topology and strategies, as for the islands. In fact, the evolution steps used in the islands are instantiated using the same *Sequential* constructor as for the sequential instance (Fig. 13). The same approach will be used to support using other metaheuristics as island evolution steps in future releases.

5. Experimental evaluation

The results of the experimental evaluation are presented in this section. The experiments are divided into two groups: a first group focused on validating the approach, and a second group focused on profiling the parallel implementations to remove inefficiencies and improve their performance. Due to the space limitations, only a summary of the results is reported here. Configuration files, logs and detailed analysis of the results are available in a companion repository [118].

5.1. Validation of the approach

In order to validate the genericity and correctness of the approach, the same experiments as in [103] were reproduced using the sequential version of the PSO algorithm implemented in the framework. Only experiments with symmetrical initialization were considered, and a MacBook Pro M1 Pro with 10-core CPU and 16 GB RAM was used to carry out the experiments. The same methodology described in that paper was followed to evaluate the performance, iterations to criteria and proportion reaching criteria of six models of neighborhood influence (i.e. Best, FIPS, wFIPS, wdFIPS, Self, and wSelf) combined with five topologies (i.e. Square, Ring, 4-clusters, Pyramid and All), all of them except 4-clusters are used in two different configurations, with and without including the target particle in the neighborhood. The results of the experiment were obtained by combining the standardized individual results of five benchmark functions (i.e. Sphere,

```
# define some topologies (details omitted)
topologies {
  0: topology { ... },
  1: topology { ... }
}
# define some strategies (details omitted)
strategies {
  0: strategies { ... },
  1: strategies { ... }
}
# define some configurations reusing previous definitions
configs {
  0: ${topologies.0} ${strategies.0},
  1: ${topologies.1} ${strategies.1},
}
# a sequential PSO
algorithm {
  name = ParticleSwarm
  implementation.name = Sequential
  implementation.conf { ${configs.0} }
}
# a PSO with homogeneous islands
algorithm {
  name = ParticleSwarm
  implementation {
    name = Islands
    local_iterations = 10
    conf.as_percentage = true
    conf.islands = [
      { amount = 100, conf = ${configs.0} }
    ]
  }
}
# a PSO with heterogeneous islands
algorithm {
  name = ParticleSwarm
  implementation {
    name = Islands
    local_iterations = 10
    conf.as_percentage = true
    conf.islands = [
      { amount = 50, conf = ${configs.0} }
      { amount = 50, conf = ${configs.1} }
    ]
  }
}
```

Listing 6: Example of the configuration of PSO instances. Only the parts that are relevant for the example are shown.

Rastrigin, Griewank – in two sizes –, Rosenbrock and Schaffer f6), each of which was run 40 times for each combination. When available, the functions in our own Scala implementation of the Large Scale Global Optimization (LGSO) test suite [98], which is integrated in the framework, were used. For the rest, *ad hoc* Scala implementations were added to the framework. It is worth noting that all the experiments were described and executed from a single configuration written in the declarative format provided by the framework, which demonstrates its expressiveness.

The first series of experiments were performed using the same parameters for the test functions reported in [103], which are reproduced in Table 6. For the PSO parameters, since the values used are not reported in [103], we have used the values shown in Table 7. Although all the experiments run successfully, demonstrating the genericity of the proposal, the results were quite different to those reported in [103] for all the combinations evaluated.

The standardized results for the performance, i.e. best function result after 1,000 iterations, are shown in Table 8. In the table, the names of the configurations without the target particle are prefixed with an "U", e.g. USquare, URing. The fully connected topologies, All and UAll, got the best or near best results in all models except for the Best model, for which they got the worst result. The Best model obtained results slightly below the mean for the rest of topologies. The FIPS variants obtained results far above the mean in most combinations, with the exception of the fully connected topologies. The Self variants, where

Table 6
Test functions used in the validation experiments and their parameters. In the Search Space and VTR (*Value To Reach*) columns, values on the left are from the first series of experiments and values on the right from the second.

Function	Dimension	Search Space		VTR	
Sphere	30	±100	±5.12	0.01	0.05
Rastrigin	30	±5.12	±5.12	100	0.05
Griewank	10, 30	±600	±600	0.05	0.05
Rosenbrock	30	±30	±10	100	0.05
Schaffer f6	2	±100	±100	0.00001	0.05

the previous best of the particle is weighted by half, show the most homogeneous behavior, obtaining results well below the mean in all the topologies evaluated.

The proportion of experiments reaching criteria, i.e. the proportion of runs that found the VTR (*Value To Reach*) within 10,000 iterations, is shown in Table 9. In general, the ratios of success are well below those reported in [103]. The best results were obtained by the wdFIPS model with a 99.6% ratio for the fully connected topologies, and the other FIPS variants with ratios higher than 91% for the UAll topology. But the ratios of the FIPS variants with the other topologies are lower than 29%. Except for the fully connected topologies, the Self variants performed best in all other cases, with ratios above 80% for the URing topology. The worst results were obtained by the Best model, being in all cases below 3.8%.

In view of the differences in results, and given that the values of the PSO parameters are not reported in [103] or whether any strategies were used to improve the performance of the algorithm, we decided to conduct a second series of experiments, this time using the test function parameters from [119] which are shown in Table 6, and the PSO parameters and strategies shown in Table 7. The results are shown in Tables 10 and 11. Reducing the search space of some functions, randomly initializing velocities, and limiting the velocities and positions of the particles, improved the performance of the Best and FIPS variants compared to the first series of experiments, except for the fully connected topologies which are slightly worse in almost all cases. The Self variants show the opposite behavior, performing much worse than in the first series of experiments in all the topologies. Similar behavior is observed for the ratio of success, where Best improved in all cases and the FIPS variants in all but the fully connected topologies. The Self variants obtained worse ratios, falling by about 30%–40% in almost all cases.

Although the results of the second series of experiments show more similarities with those of [103], they are worse in general. To further validate our approach and confirm that the differences might be due to differences in the PSO parameters and not to bugs in our code or to an incorrect analysis of the experimental data, we performed a third set of experiments to compare the results of our proposal with those of the PSO implementation of ECJ, a well-known and mature MOF.

The first series of experiments was repeated with the combinations supported by ECJ, i.e. the Best model of influence with the Ring – with degrees 1 and 4 –, Random – with degree 4 – and All topologies, all of them in two configurations, with and without the target particle in the neighborhood, and the same benchmark functions, except Schaffer f6. Table 12 shows the comparison of the performance results using a two-sample Kolmogorov–Smirnov test. The null hypothesis is rejected at significance level $\alpha = 0.043$ in only one of the 40 combinations tested. The ratio of success results were compared in the same way, with the null hypothesis being rejected at the same level of significance in only two combinations. The similarity between the results of our approach and ECJ can also be seen in Table 13, which shows the results of the standardized performance and ratio of success for the topologies evaluated.

The validation by comparison with ECJ was completed with a second series of experiments using the Standard 2007 velocity update

Table 7
Parameters and strategies used in the validation experiments. The Constriction Factor variant of the Velocity Update equation was used with the parameters proposed in [111]. In the framework, the expanded form of the equation is used for the Self and wSelf models of influence and the condensed form for the rest.

	Experiment 1	Experiment 2
Velocity Update	Constriction Factor ($\chi = 0.72984$, $c_1 = c_2 = 2.05$)	
	Condensed Constriction Factor ($\chi = 0.72984$, $c_{max} = 4.1$)	
Velocity Initialization	Zero	Bounded (k: 0.5)
Velocity Clamping	None	Bounded
Velocity Limit Initialization	None	Factor (k: 0.05)
Movement Bound	None	Standard 2007

Table 8
Standardized performance of the first series of experiments. Negative values are below the mean, the more negative the better, while positive values are above. In bold are the best results for each model of neighborhood influence.

	Square	Ring	4-Clusters	Pyramid	All	USquare	URing	UPyramid	UAll
Best	−0.057	−0.110	−0.017	−0.049	1.057	−0.087	**−0.149**	−0.148	0.415
FIPS	0.683	0.971	0.050	−0.130	**−0.657**	0.955	1.017	0.479	−0.637
wFIPS	0.808	0.988	0.297	0.112	**−0.674**	0.976	1.021	0.620	−0.666
wdFIPS	0.948	0.965	0.755	0.478	−0.719	0.951	0.962	0.534	**−0.723**
Self	−0.637	−0.610	−0.611	**−0.676**	−0.656	−0.609	−0.555	−0.623	−0.663
wSelf	−0.638	−0.590	−0.639	−0.619	−0.643	−0.594	−0.569	−0.610	**−0.675**

Table 9
Proportion of the first series of experiments reaching the VTR within 10,000 iterations. In bold are the best results for each model of neighborhood influence.

	Square	Ring	4-Clusters	Pyramid	All	USquare	URing	UPyramid	UAll
Best	0.017	0.021	0.029	0.008	0.008	**0.038**	0.029	0.008	0.017
FIPS	0.167	0.167	0.217	0.283	0.663	0.167	0.167	0.167	**0.917**
wFIPS	0.167	0.167	0.167	0.167	0.625	0.167	0.167	0.167	**0.917**
wdFIPS	0.167	0.167	0.167	0.167	**0.996**	0.167	0.167	0.167	**0.996**
Self	0.417	0.546	0.479	0.475	0.375	0.571	**0.854**	0.446	0.383
wSelf	0.300	0.458	0.379	0.400	0.296	0.633	**0.804**	0.388	0.354

Table 10
Standardized performance of the second series of experiments.

	Square	Ring	4-Clusters	Pyramid	All	USquare	URing	UPyramid	UAll
Best	−0.302	−0.341	−0.196	−0.302	1.586	−0.328	**−0.372**	−0.321	0.517
FIPS	−0.196	0.184	−0.172	−0.221	−0.499	−0.210	0.014	−0.232	**−0.628**
wFIPS	−0.156	0.234	−0.094	−0.184	−0.474	−0.256	0.089	−0.247	**−0.585**
wdFIPS	−0.205	0.003	−0.144	−0.214	**−0.746**	−0.246	0.018	−0.258	−0.711
Self	0.430	0.193	0.635	0.581	0.550	**−0.355**	−0.241	−0.114	0.357
wSelf	0.740	0.327	1.219	0.857	0.769	**−0.276**	−0.186	−0.013	0.720

Table 11
Proportion of the second series of experiments reaching the VTR within 10,000 iterations.

	Square	Ring	4-Clusters	Pyramid	All	USquare	URing	UPyramid	UAll
Best	0.229	0.275	0.179	0.208	0.167	0.258	**0.304**	0.242	0.167
FIPS	0.333	0.333	0.333	0.333	0.496	0.333	0.333	0.333	**0.663**
wFIPS	0.333	0.333	0.333	0.333	0.413	0.333	0.333	0.333	**0.663**
wdFIPS	0.333	0.333	0.333	0.333	**0.667**	0.333	0.333	0.333	0.663
Self	0.254	0.300	0.246	0.233	0.192	0.383	**0.663**	0.296	0.208
wSelf	0.208	0.263	0.179	0.196	0.179	0.313	**0.642**	0.238	0.200

Table 12
Statistics and p-values of the two-sample Kolmogorov–Smirnov test used to compare the performance results with those of ECJ. In bold, the configurations where the null hypotheses is rejected at significance level $\alpha = 0.043$.

		Ring	Ring_4	All	Rand_4	URing	URing_4	UAll	URand_4
Sphere	*D-stat*	0.10	0.18	0.23	0.13	0.15	0.15	0.13	0.10
	p-value	0.983	0.531	0.231	0.893	0.724	0.724	0.893	0.983
Rastrigin	*D-stat*	0.20	0.18	0.18	0.15	0.18	0.13	**0.33**	0.15
	p-value	0.361	0.531	0.531	0.724	0.531	0.893	**0.022**	0.724
Griewank10	*D-stat*	0.10	0.15	0.15	0.25	0.15	0.20	0.18	0.13
	p-value	0.983	0.724	0.724	0.139	0.724	0.361	0.531	0.893
Griewank30	*D-stat*	0.10	0.30	0.18	0.25	0.25	0.20	0.25	0.10
	p-value	0.983	0.043	0.531	0.139	0.139	0.361	0.139	0.983
Rosenbrock	*D-stat*	0.18	0.20	0.15	0.30	0.20	0.30	0.15	0.15
	p-value	0.531	0.361	0.723	0.043	0.361	0.043	0.724	0.724

Table 13
Standardized performance and ratio of success of our approach (PSO[T]) and ECJ for the topologies evaluated. When available, results from [103] are also provided (Mendes row). Note that these are the only ones that include Schaffer f6.

	Ring	Ring$_4$	All	Rand$_4$	URing	URing$_4$	UAll	URand$_4$
PSO[T]	0.043	−0.449	−0.424	−0.208	0.529	0.171	−0.025	0.452
ECJ	−0.002	−0.316	−0.481	−0.276	0.479	0.086	−0.023	0.444
Mendes	−0.307		−0.317		−0.316		−0.330	
PSO[T]	0.045	0.015	0.04	0.0	0.02	0.02	0.015	0.0
ECJ	0.04	0.05	0.03	0.005	0.01	0.015	0.005	0.0
Mendes	0.913		0.754		0.908		0.754	

Table 14
Configuration of the PSO instances used in the performance experiments. For the strategies not listed *None* was used.

Strategy	Configuration
Velocity Initialization	Zero
Velocity Update	Standard 2007
Velocity Limit Initialization	Constant
Velocity Limit Update	LVDM
Termination condition	number of generations **or** objective value **or** swarm stagnation

equation and the ECJ default PSO parameters (i.e. $w = 0.7$, $c_1 = c_2 = 0.4$). The null hypothesis was rejected in 3 of the 80 combinations tested at significance level $\alpha = 0.043$, showing once again the similarity of the results. It is worth noting that thanks to the experiments performed in this comparison, several bugs were detected in the PSO implementation of ECJ [120] and a pull request with the fix was contributed to the ECJ project.

5.2. Performance of the parallel implementations

A second group of experiments was conducted focusing on profiling the parallel implementations to detect and eliminate inefficiencies and improve their performance. The main goal of these experiments was to minimize the number of Spark jobs launched in each iteration of the algorithm, thus avoiding the overhead of unnecessary jobs. The PSO instances used in the experiments were configured with the strategies shown in Table 14. Only information relevant to the discussion is shown in the table. The configuration was selected to test the worst-case scenario, i.e. the one requiring the launch of the largest number of Spark jobs per iteration.

The first series of experiments were focused on analyzing the runtime behavior of each of the three parallel implementations provided by the framework (Section 4.6), and were carried out in the same computer of Section 5.1, using a Spark cluster deployed on the single-node standalone Kubernetes cluster included as part of Docker Desktop. The cluster was configured with 6 CPUs, 8 GB RAM, and 1 GB of swap space, and the official Apache Spark image (`apache/spark:latest`, v3.4.1) pulled from Docker Hub, and Docker Desktop v4.21.1 (engine: 24.0.2) with Kubernetes v1.27.2 were used.

The logical flow of an iteration, obtained from the analysis of the runtime behavior, is shown in Figs. 16 to 18 for each of the parallel implementations. Spark jobs are shown as dotted boxes in the figures. The simplest case is the Master–Worker implementation in Fig. 16, which requires launching only one Spark job per iteration to distribute the particles using a Spark RDD, evaluate them via a *map* transformation, and *collect* them again. In this version of the Master–Worker model, the rest of the iteration logic is executed in the *driver* (i.e. the main process in Spark terminology) and does not require launching any other Spark job.

An alternative implementation of the Master–Worker parallel model, which keeps the swarm distributed between iterations and distributes the execution of part of the iteration logic to Spark *executors* (i.e. worker processes in Spark terminology), is shown in Fig. 17. In this variant, seven Spark jobs are launched per iteration:

- When an objective value is configured in the termination condition, a job is launched by a *min* action to get the best solution found in the last iteration and check if the objective value was reached.
- To log the state of the evolution at the default logging level, two jobs are launched by *min* actions to get the best solution found in the last iteration and the historical best solution found so far. The number of jobs launched could vary depending on the logging level configured.
- To gather the particle contribution to the neighborhood influence (Section 4.5), a job is launched by a *collect* action.
- To update the state and parameters at the end of the iteration, three jobs are launched by *min* and *count* actions. They basically get the best solution and swarm size, and update the number of evaluations of the objective function performed and the information stored by the LDVM strategy and the stagnation detection condition. The number of jobs launched depends on the strategies and conditions configured. Note that the move and evaluation of the swarm is performed by two *map* transformations in the first job launched.

The logical flow of the island-based parallel implementation is shown in Fig. 18. In this implementation, every partition of the swarm RDD represents an island with its own configuration. With all the islands configured with the configuration in Table 14, the number of jobs launched per iteration is $6+N$, being N the number of islands. The main differences with the Master–Worker implementation in Fig. 17 are the following:

- The evolution of islands for a configured number of iterations, which includes updating the neighborhood influence and moving and evaluating the island, is executed as part of the same job launched by a *min* action.
- A job per island launched by a *min* action is required to update the information stored by the LDVM strategy. The total number of jobs launched depends on the configuration of each island.

As it can be seen, most of the jobs are launched to access swarm properties that remain unchanged until the iteration ends and the swarm is updated, like the best solution or the swarm size. Although Spark RDDs are immutable, and executing actions like *min* or *count* several times will yield the same result each time, Spark does not implement an optimization and launches a job on each action call. In the parallel implementations of Figs. 17 and 18, the number oj jobs was reduced to two per iteration by implementing two optimizations in the framework:

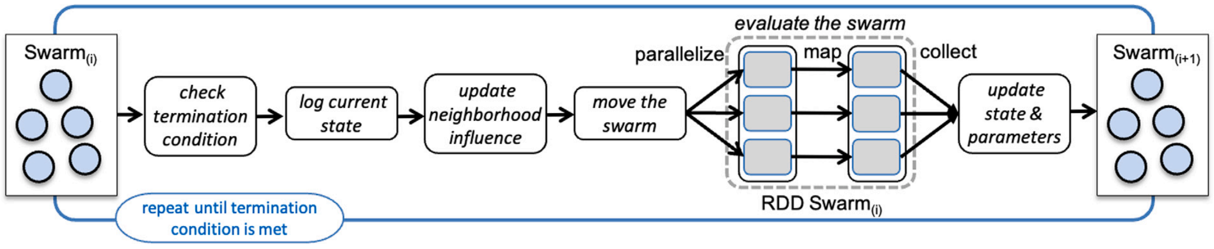

Swarm(i)
check termination condition
log current state
update neighborhood influence
move the swarm
parallelize
evaluate the swarm
map
collect
RDD Swarm(i)
update state & parameters
Swarm(i+1)
repeat until termination condition is met

Fig. 16. Logical flow of an iteration of the Master–Worker implementation that distributes only the evaluation of particles.

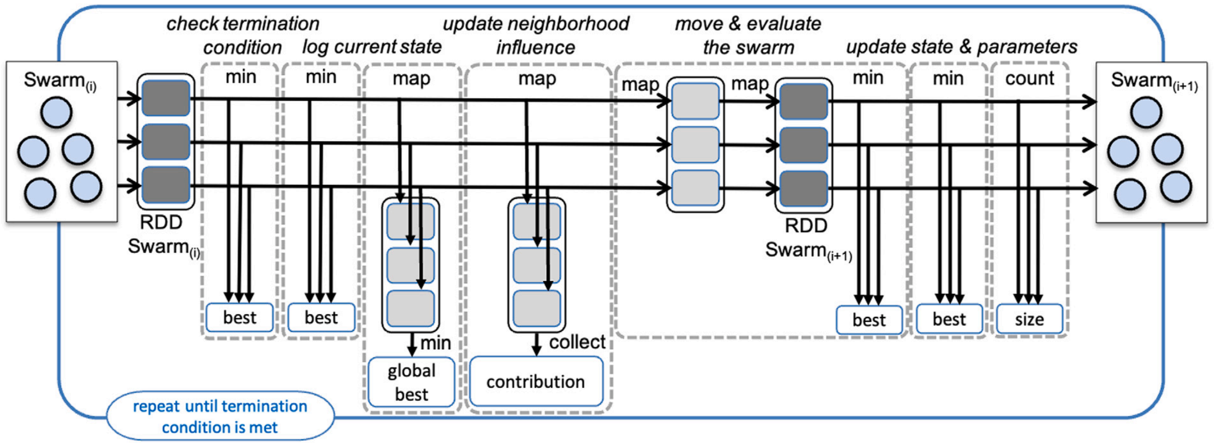

Swarm(i)
RDD Swarm(i)
check termination condition
min
best
log current state
min
best
update neighborhood influence
map
min
global best
map
collect
contribution
move & evaluate the swarm
map
map
RDD Swarm(i+1)
update state & parameters
min
best
min
best
count
size
Swarm(i+1)
repeat until termination condition is met

Fig. 17. Logical flow of an iteration of the Master–Worker implementation that keeps the swarm distributed between iterations and executes part of the iteration logic on Spark *executors*.

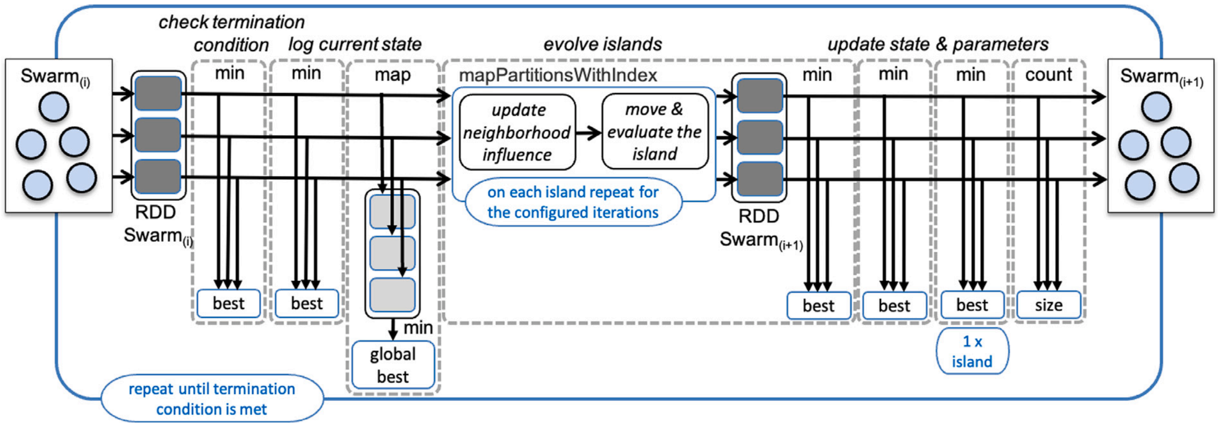
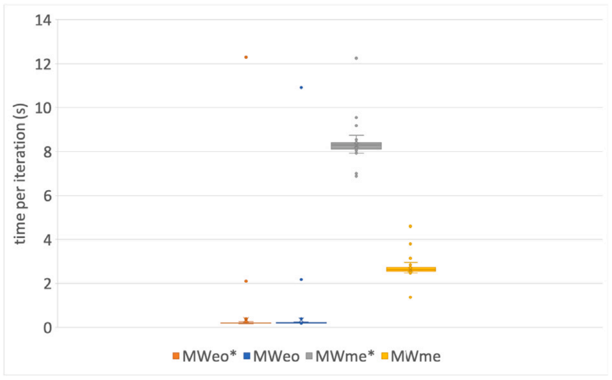

Swarm(i)
RDD Swarm(i)
check termination condition
min
best
log current state
min
best
map
min
global best
evolve islands
mapPartitionsWithIndex
update neighborhood influence
move & evaluate the island
on each island repeat for the configured iterations
RDD Swarm(i+1)
update state & parameters
min
best
min
best
min
best
1 x island
count
size
Swarm(i+1)
repeat until termination condition is met

Fig. 18. Logical flow of an iteration of the island-based implementation.

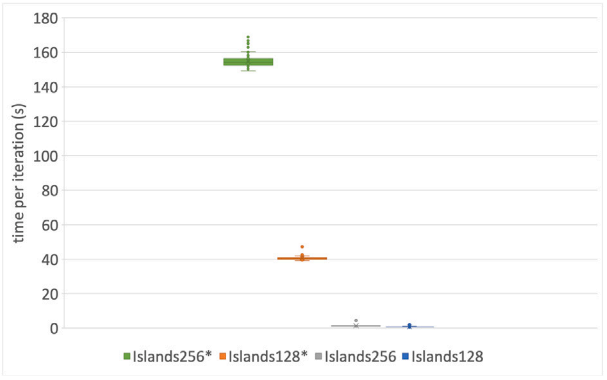
- A RDD wrapper with caching was implemented to store the distributed swarms. The first time an action is called, a job is launched and the result is stored and returned on subsequent calls without launching additional jobs.
- Some properties like, for example, the historical best solution, which can be calculated from other properties, are updated and stored in the properties storage (Section 3.3) at the end of each iteration, instead of being obtained by calling an action.

To evaluate the impact of these optimizations, a second series of experiments was conducted to compare the time per iteration of the optimized and non-optimized versions of the parallel implementations. The experiments were run on Spark clusters (Spark v3.2.4, Hadoop with YARN v2.10.2) dynamically deployed with BDEv [121] (v3.8) on a partition of our compute cluster *Pluton* [122], which groups 15 nodes powered by two octa-core Intel Xeon E5-2660 CPUs with 64 GB of RAM and connected by a 1GbE network.

The parameters and objective function shown in Table 15 were used in the experiments. They were chosen to evaluate the impact of optimization on LSGO problems while keeping the computation-to-overhead ratio of Spark jobs as low as possible. The reason is that our optimization is focused on reducing the number of jobs per iteration, so increasing the computation performed by Spark jobs will increase the execution time by the same ratio in both optimized and non-optimized versions, which is of no value for their comparison. Using the lowest computation-to-overhead ratio allows us to estimate an upper bound on the expected improvement. All parallel versions were run using all available vCores (threads) on eight worker nodes. In addition, the island-based versions were also run on four worker nodes to get a first insight into the scalability of the optimized version. The population is divided into a number of partitions equal to the total number of vCores available, so that each Spark job schedules as many tasks as there are vCores available. Furthermore, the partitions are configured



(a) Master-Worker



(b) Islands

Fig. 19. Distribution of times per iteration of the optimized and non-optimized versions of the parallel implementations. Non-optimized versions are marked with an asterisk. Outliers correspond to the initial iterations that, in Spark, are the most time consuming [18].

Table 15
Parameters used to evaluate the time per iteration of the parallel implementations.

Parameter	Master–Worker	Islands	
Worker nodes (total vCores)	8(256)	4(128)	8(256)
RDD partitions/tasks per job	256	128	256
Population (particles per partition)	256(1)	640(5)	1280(5)
Island local iterations	n/a	1	1
Objective function	Sphere (dim=1000, search space=±100, VTR=0.01)		
Number of generations	21 (initial plus 20 iterations)		

Table 16
Average (avg) and standard deviation (std) of the times per iteration of the non-optimized and optimized versions of the parallel implementations, and percentage of improvement of the optimized versions. MWeo is the Master–Worker variant of Fig. 16 (eo=evaluation only), and MWme is the Master–Worker variant of Fig. 17 (me=move&evaluation).

Implementation	#partitions	avg(std) non-opt.	avg(std) opt.	%improvement
MWeo	256	0.2057(0.040)	0.2048(0.027)	0.43%
MWme	256	8.2909(0.235)	2.6319(0.105)	68.26%
Islands	128	40.5012(0.712)	0.7746(0.064)	98.09%
	256	154.3621(2.654)	1.3656(0.103)	99.12%

to have one particle per partition in the Master–Worker versions and five particles per partition in the island-based versions for the reason explained above.

For each configuration, five independent runs of 20 iterations each were performed, for a total of 100 iterations per configuration. Table 16 and Fig. 19 show the average, standard deviation, and box plots of the distribution of times per iteration of the optimized and non-optimized parallel implementations, as well as the percentage of improvement of the optimized versions. The results in Table 16 show that for the Master–Worker implementations, the optimized *MWme* version has an average improvement of about 68% over the non-optimized one because the number of jobs per iteration has been reduced from seven to two, while the optimized *MWeo* version obtains similar results to the non-optimized one because the number of jobs per iteration has not been reduced despite the optimizations. As for the island-based implementation, the optimized version outperforms the non-optimized one by more than 98% in the two configurations tested. The reason for such good results is that the number of jobs per iteration was reduced from 134 and 262 to two, respectively. Furthermore, changing the number of jobs per iteration from a linear dependence on the number of islands to a constant also improves the scalability of the optimized version, as shown in Fig. 19(b), which is of paramount importance for LSGO problems. Doubling the number of islands increases the average time per iteration by a factor of 3.8 in the non-optimized version, but only by a factor of 1.7 in the optimized one. The reason is that while the number of jobs per iteration is almost doubled in the non-optimized version, it remains constant and equal to two in the optimized one, and the time per iteration only increases due to the overhead of scheduling more tasks per job.

To conclude this section, we would like to comment on the implications of RDD caching when implementing parallel stochastic algorithms, such as population-based metaheuristics, in Spark. RDD is the basic abstraction used in Spark to represent partitioned, read-only, fault-tolerant collections of records that are distributed across multiple machines. RDDs are created from data in stable storage or by transforming other RDDs (e.g., *map*, *filter* or *join*). These transformations are pipelined to form a *lineage* that is computed lazily when an action is called to obtain a result (e.g., *count*, *min* or *collect*). Computing RDDs from their lineages provides fault tolerance in case any RDD partition is lost, but when lineages start to be huge and the same RDDs are recomputed multiple times, as in iterative algorithms, performance can degrade significantly. To avoid recomputing lineages again and again, RDDs can be reused by caching them in memory. But special care must be taken with lineage computation and caching when dealing with stochastic algorithms. Repeatedly recomputing a transformation that uses randomness is not deterministic and can lead to incorrect results. In the parallel implementations of Figs. 17 and 18, determinism between iterations is guaranteed by caching the swarm RDD (shown as dark shaded rectangles in the figures) after the swarm has been evaluated at each iteration. Subsequent action calls will reuse the cached RDD without recomputing its lineage. Even with this approach, the lineage could still be recomputed in the event of an *executor* failure if the cached RDDs are lost. To overcome this problem, it is planned to tolerate *executor* failures in future versions of the framework through replication and checkpointing of RDDs.

6. Conclusions

The current status of a framework to support the development of distributed population-based metaheuristics and their application to the global optimization of large-scale problems in Spark clusters is presented in this paper. The framework provides a reduced set of abstractions to represent the general structure of population-based metaheuristics as templates from which different variants of algorithms can be instantiated through implementation of strategies. Strategies can be reused between metaheuristics, enforcing code reusability.

To validate the approach, a template for Particle Swarm Optimization (PSO) is implemented as a proof of concept applying the general abstractions provided by the framework. The template includes strategies to instantiate different variants of the PSO algorithm that can be run on Spark clusters selecting one of the distributed execution models provided, and a generic representation for social topologies, with a long list of configurable topologies supported out of the box and also support for custom topologies. To the best of the authors' knowledge there is no other implementation of a PSO template that supports all these features together.

An initial experimental evaluation of the framework was conducted using two groups of experiments: a first group focused on validating the genericity and correctness of the approach by comparison of the results with other proposals, and a second group focused on profiling the distributed execution models implemented to remove inefficiencies and improve their performance.

Due to the early stage of development, the framework has some limitations, for example: (i) the set of abstractions defined focuses on canonical distributed population-based metaheuristics, approaches such as hybridization, cooperation, or decomposition are not yet supported; (ii) only single continuous LSGO problems are addressed, and other types of large-scale problems (e.g. multi-objective, combinatorial) are not yet supported; (iii) the number of strategies implemented in the PSO template is still small, and only static topologies and non-cooperative heterogeneous islands are currently supported.

As future work, we plan to evaluate the parallel performance of the distributed execution models implemented and extend the framework by adding support for cooperative islands, dynamic topologies and other population-based metaheuristic templates.

CRediT authorship contribution statement

Xoán C. Pardo: Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Patricia González:** Conceptualization, Formal analysis, Methodology, Project administration, Supervision, Writing – review & editing. **Julio R. Banga:** Conceptualization, Funding acquisition, Methodology, Resources, Writing – review & editing. **Ramón Doallo:** Conceptualization, Funding acquisition, Project administration, Resources, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The code and data used for this research are available in public repositories referenced in the article.

Acknowledgments

This work was supported by MCIN/AEI/10.13039/501100011033 [grant numbers PID2019-104184RB-I00, PID2022-136435NB-I00 and PID2020-117271RB-C22 (BIODYNAMICS)], PID2022 also funded by "ERDF A way of making Europe", EU. Also supported by Xunta de Galicia [grant number ED431C 2021/30]. Funding for open access charge: Universidade da Coruña/CISUG.

References

[1] K. Hussain, M.N. Mohd Salleh, S. Cheng, Y. Shi, Metaheuristic research: a comprehensive survey, Artif. Intell. Rev. 52 (4) (2019) 2191–2233, http://dx.doi.org/10.1007/s10462-017-9605-z.

[2] Y.-J. Gong, W.-N. Chen, Z.-H. Zhan, J. Zhang, Y. Li, Q. Zhang, J.-J. Li, Distributed evolutionary algorithms and their models: A survey of the state-of-the-art, Appl. Soft Comput. 34 (2015) 286–300, http://dx.doi.org/10.1016/j.asoc.2015.04.061.

[3] A. Elyasaf, M. Sipper, Software review: the heuristiclab framework, Genetic Program. Evolvable Mach. 15 (2) (2014) 215–218, http://dx.doi.org/10.1007/s10710-014-9214-4.

[4] E.O. Scott, S. Luke, Ecj at 20: Toward a general metaheuristics toolkit, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 1391–1398, http://dx.doi.org/10.1145/3319619.3326865.

[5] J. Dreo, A. Liefooghe, S. Verel, M. Schoenauer, J.J. Merelo, A. Quemy, B. Bouvier, J. Gmys, Paradiseo: From a modular framework for evolutionary computation to the automated design of metaheuristics: 22 years of paradiseo, in: Proceedings of the Genetic and Evolutionary Computation Conference Companion, Association for Computing Machinery, New York, NY, USA, 2021, pp. 1522–1530, http://dx.doi.org/10.1145/3449726.3463276.

[6] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, C. Gagné, DEAP: Evolutionary algorithms made easy, J. Mach. Learn. Res. 13 (2012) 2171–2175.

[7] A.J. Nebro, J.J. Durillo, M. Vergne, Redesigning the jmetal multi-objective optimization framework, in: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO Companion '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 1093–1100, http://dx.doi.org/10.1145/2739482.2768462.

[8] J. Parejo, A. Ruiz-Cortés, S. Lozano, P. Fernandez, Metaheuristic optimization frameworks: a survey and benchmarking, Soft Comput. 16 (3) (2012) 527–561, http://dx.doi.org/10.1007/s00500-011-0754-8.

[9] M.A. Lopes Silva, S.R. de Souza, M.J. Freitas Souza, M.F. de França Filho, Hybrid metaheuristics and multi-agent systems for solving optimization problems: A review of frameworks and a comparative analysis, Appl. Soft Comput. 71 (2018) 433–459, http://dx.doi.org/10.1016/j.asoc.2018.06.050.

[10] A. Ramírez, R. Barbudo, J.R. Romero, An experimental comparison of metaheuristic frameworks for multi-objective optimization, Expert Syst. 40 (4) (2023) e12672, http://dx.doi.org/10.1111/exsy.12672.

[11] M. Khalid, M.M. Yousaf, A comparative analysis of big data frameworks: An adoption perspective, Appl. Sci. 11 (22) http://dx.doi.org/10.3390/app112211033.

[12] J. Swan, S. Adriaensen, A.E. Brownlee, K. Hammond, C.G. Johnson, A. Kheiri, F. Krawiec, J. Merelo, L.L. Minku, E. Özcan, G.L. Pappa, P. García-Sánchez, K. Sörensen, S. Voß, M. Wagner, D.R. White, Metaheuristics in the large, European J. Oper. Res. 297 (2) (2022) 393–406, http://dx.doi.org/10.1016/j.ejor.2021.05.042.

[13] M. Zaharia, R.S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M.J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache spark: A unified engine for big data processing, Commun. ACM 59 (11) (2016) 56–65, http://dx.doi.org/10.1145/2934664.

[14] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113, http://dx.doi.org/10.1145/1327452.1327492.

[15] D. Teijeiro, X.C. Pardo, P. González, J.R. Banga, R. Doallo, Implementing parallel differential evolution on spark, in: G. Squillero, P. Burelli (Eds.), Applications of Evolutionary Computation, Springer International Publishing, Cham, 2016, pp. 75–90.

[16] D. Teijeiro, X. Pardo, P. González, J. Banga, R. Doallo, Towards cloud-based parallel metaheuristics: A case study in computational biology with differential evolution and spark, Int. J. High Perform. Comput. Appl. 32, http://dx.doi.org/10.1177/1094342016679011.

[17] D. Teijeiro, X. Pardo, D.R. Penas, P. González, J. Banga, R. Doallo, A cloud-based enhanced differential evolution algorithm for parameter estimation problems in computational systems biology, Cluster Comput. 20 (2017) 1–14, http://dx.doi.org/10.1007/s10586-017-0860-1.

[18] D. Teijeiro, X. Pardo, D.R. Penas, P. González, J. Banga, R. Doallo, Evaluation of parallel differential evolution implementations on MapReduce and spark, 2017, http://dx.doi.org/10.1007/978-3-319-58943-5_32.

[19] X.C. Pardo, P. Argüeso-Alejandro, P. González, J.R. Banga, R. Doallo, Spark implementation of the enhanced scatter search metaheuristic: Methodology and assessment, Swarm Evol. Comput. 59 (2020) 100748, http://dx.doi.org/10.1016/j.swevo.2020.100748.

[20] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M.J. Sax, S. Schelter, M. Höger, K. Tzoumas, D. Warneke, The stratosphere platform for big data analytics, VLDB J. 23 (6) (2014) 939–964, http://dx.doi.org/10.1007/s00778-014-0357-y.

[21] X.C. Pardo, Udc-gac/spark-eclib, 2023, http://dx.doi.org/10.5281/zenodo.8431049.

[22] S. Mahdavi, M.E. Shiri, S. Rahnamayan, Metaheuristics in large-scale global continues optimization: A survey, Inform. Sci. 295 (2015) 407–428, http://dx.doi.org/10.1016/j.ins.2014.10.042.

[23] M. Gendreau, J. Potvin, Handbook of Metaheuristics, in: International Series in Operations Research & Management Science, Springer International Publishing, 2019.

[24] M. Abdel-Basset, L. Abdel-Fatah, A.K. Sangaiah, Chapter 10 - metaheuristic algorithms: A comprehensive review, in: A.K. Sangaiah, M. Sheng, Z. Zhang (Eds.), Computational Intelligence for Multimedia Big Data on the Cloud with Engineering Applications, Intelligent Data-Centric Systems, Academic Press, 2018, pp. 185–231, http://dx.doi.org/10.1016/B978-0-12-813314-9.00010-4.

[25] T. Dokeroglu, E. Sevinc, T. Kucukyilmaz, A. Cosar, A survey on new generation metaheuristic algorithms, Comput. Ind. Eng. 137 (2019) 106040, http://dx.doi.org/10.1016/j.cie.2019.106040.

[26] J. Del Ser, E. Osaba, D. Molina, X.-S. Yang, S. Salcedo-Sanz, D. Camacho, S. Das, P.N. Suganthan, C.A. Coello Coello, F. Herrera, Bio-inspired computation: Where we stand and what's next, Swarm Evol. Comput. 48 (2019) 220–250, http://dx.doi.org/10.1016/j.swevo.2019.04.008.

[27] C.L.C. Villalón, T. Stützle, M. Dorigo, Designing new metaheuristics: manual versus automatic approaches, Intell. Comput. 2 (2023) 0048, http://dx.doi.org/10.34133/icomputing.0048.

[28] E. Alba, G. Luque, S. Nesmachnow, Parallel metaheuristics: recent advances and new trends, Int. Trans. Oper. Res. 20 (1) (2013) 1–48.

[29] T. Crainic, Parallel Metaheuristics and Cooperative Search, Springer International Publishing, Cham, 2019, pp. 419–451, http://dx.doi.org/10.1007/978-3-319-91086-4_13.

[30] G. Schryen, Parallel computational optimization in operations research: A new integrative framework, literature review and research directions, European J. Oper. Res. 287 (1) (2020) 1–18, http://dx.doi.org/10.1016/j.ejor.2019.11.033.

[31] W.-N. Chen, F.-F. Wei, T.-F. Zhao, K.C. Tan, J. Zhang, A survey on distributed evolutionary computation, 2023, arXiv:2304.05811.

[32] T. Harada, E. Alba, Parallel genetic algorithms: A useful survey, ACM Comput. Surv. 53 (4) http://dx.doi.org/10.1145/3400031.

[33] S. Lalwani, H. Sharma, S.C. Satapathy, K. Deep, J.C. Bansal, A survey on parallel particle swarm optimization algorithms, Arab. J. Sci. Eng. 44 (4) (2019) 2899–2923, http://dx.doi.org/10.1007/s13369-018-03713-6.

[34] A.W. McNabb, C.K. Monson, K.D. Seppi, Parallel pso using mapreduce, in: 2007 IEEE Congress on Evolutionary Computation, 2007, pp. 7–14, http://dx.doi.org/10.1109/CEC.2007.4424448.

[35] Y. Wang, Y. Li, Z. Chen, Y. Xue, Cooperative particle swarm optimization using mapreduce, Soft Comput. 21 (22) (2017) 6593–6603, http://dx.doi.org/10.1007/s00500-016-2390-9.

[36] G. Zhang, Y. Li, Z. Chen, Y. Wang, L. Jiao, Y. Xue, A novel distributed quantum-behaved particle swarm optimization, J. Optim. 2017 (2017) 4685923, http://dx.doi.org/10.1155/2017/4685923.

[37] I. Aljarah, S.A. Ludwig, Parallel particle swarm optimization clustering algorithm based on mapreduce methodology, in: 2012 Fourth World Congress on Nature and Biologically Inspired Computing, NaBIC, 2012, pp. 104–111, http://dx.doi.org/10.1109/NaBIC.2012.6402247.

[38] A.P. Chunne, U. Chandrasekhar, C. Malhotra, Real time clustering of tweets using adaptive pso technique and mapreduce, in: 2015 Global Conference on Communication Technologies, GCCT, 2015, pp. 452–457.

[39] L. Cui, Parallel Pso in Spark (Master's thesis), Faculty of Science and Technology. University of Stavanger, 2014.

[40] Y. Ma, P. an Zhong, B. Xu, F. Zhu, Q. Lu, H. Wang, Spark-based parallel dynamic programming and particle swarm optimization via cloud computing for a large-scale reservoir system, J. Hydrol. 598 (2021) 126444, http://dx.doi.org/10.1016/j.jhydrol.2021.126444.

[41] B. Cao, W. Li, J. Zhao, S. Yang, X. Kang, Y. Ling, Z. Lv, Spark-based parallel cooperative co-evolution particle swarm optimization algorithm, in: 2016 IEEE International Conference on Web Services, ICWS, 2016, pp. 570–577, http://dx.doi.org/10.1109/ICWS.2016.79.

[42] Z. Zhang, W. Wang, N. Gao, Y. Zhao, Spark-based distributed quantum-behaved particle swarm optimization algorithm, in: Y. Luo (Ed.), Cooperative Design, Visualization, and Engineering, Springer International Publishing, Cham, 2018, pp. 295–298.

[43] Z. Zhang, W. Wang, G. Pan, A distributed quantum-behaved particle swarm optimization using opposition-based learning on spark for large-scale optimization problem, Mathematics 8 (11) http://dx.doi.org/10.3390/math8111860.

[44] Q. Duan, L. Sun, Y. Shi, Spark clustering computing platform based parallel particle swarm optimizers for computationally expensive global optimization, in: A. Auger, C.M. Fonseca, N. Lourenço, P. Machado, L. Paquete, D. Whitley (Eds.), Parallel Problem Solving from Nature – PPSN XV, Springer International Publishing, Cham, 2018, pp. 424–435.

[45] D. Fan, J. Lee, A hybrid mechanism of particle swarm optimization and differential evolution algorithms based on spark, KSII Trans. Internet Inf. Syst. 13 (12) (2019) 5972–5989.

[46] K. Wu, Y. Zhu, Q. Li, G. Han, Algorithm and implementation of distributed esn using spark framework and parallel pso, Appl. Sci. 7 (4) http://dx.doi.org/10.3390/app7040353.

[47] W. Zhang, Y. Huang, Using big data computing framework and parallelized pso algorithm to construct the reservoir dispatching rule optimization, Soft Comput. 24 (11) (2020) 8113–8124, http://dx.doi.org/10.1007/s00500-019-04188-9.

[48] K. Govindarajan, D. Boulanger, V.S. Kumar, Kinshuk, Parallel particle swarm optimization (ppso) clustering for learning analytics, in: 2015 IEEE International Conference on Big Data (Big Data), 2015, pp. 1461–1465, http://dx.doi.org/10.1109/BigData.2015.7363907.

[49] A. Verma, X. Llora, D.E. Goldberg, R.H. Campbell, Scaling genetic algorithms using MapReduce, in: 2009 Ninth International Conference on Intelligent Systems Design and Applications, IEEE, 2009, pp. 13–18.

[50] C. Jin, C. Vecchiola, R. Buyya, MRPGA: an extension of MapReduce for parallelizing genetic algorithms, in: IEEE Fourth International Conference on EScience, EScience'08, IEEE, 2008, pp. 214–221.

[51] D. Huang, J. Lin, Scaling populations of a genetic algorithm for job shop scheduling problems using mapreduce, in: 2010 IEEE Second International Conference on Cloud Computing Technology and Science, 2010, pp. 780–785.

[52] F. Ferrucci, P. Salza, F. Sarro, Using hadoop MapReduce for parallel genetic algorithms: A comparison of the global, grid and island models, Evolut. Comput. 26 (4) (2018) 535–567, http://dx.doi.org/10.1162/evco_a_00213.

[53] C. Paduraru, M.-C. Melemciuc, A. Stefanescu, A distributed implementation using apache spark of a genetic algorithm applied to test data generation, 2017, pp. 1857–1863, http://dx.doi.org/10.1145/3067695.3084219.

[54] C. Hu, G. Ren, C. Liu, M. Li, W. Jie, A spark-based genetic algorithm for sensor placement in large scale drinking water distribution systems, Cluster Comput. 20 (2) (2017) 1089–1099, http://dx.doi.org/10.1007/s10586-017-0838-z.

[55] R.-Z. Qi, Z.-J. Wang, S.-Y. Li, A parallel genetic algorithm based on spark for pairwise test suite generation, J. Comput. Sci. Tech. 31 (2) (2016) 417–427, http://dx.doi.org/10.1007/s11390-016-1635-5.

[56] H.-C. Lu, F. Hwang, Y.-H. Huang, Parallel and distributed architecture of genetic algorithm on apache hadoop and spark, Appl. Soft Comput. 95 (2020) 106497, http://dx.doi.org/10.1016/j.asoc.2020.106497.

[57] L. Alterkawi, M. Migliavacca, Parallelism and partitioning in large-scale gas using spark, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 736–744, http://dx.doi.org/10.1145/3321707.3321775.

[58] C. Zhou, Fast parallelization of differential evolution algorithm using MapReduce, in: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, ACM, 2010, pp. 1113–1114.

[59] H. Peng, X. Tan, C. Deng, S. Peng, Sparkcude: a spark-based differential evolution for large-scale global optimisation, Int. J. High Perform. Syst. Architect. 7 (4) (2017) 211–222, http://dx.doi.org/10.1504/IJHPSA.2017.092390.

[60] Z. He, H. Peng, J. Chen, C. Deng, Z. Wu, A spark-based differential evolution with grouping topology model for large-scale global optimization, Cluster Comput. 24 (1) (2021) 515–535, http://dx.doi.org/10.1007/s10586-020-03124-z.

[61] Y. Khalil, M. Alshayeji, I. Ahmad, Distributed whale optimization algorithm based on mapreduce, Concurr. Comput.: Pract. Exper. 31 (1) (2019) e4872, http://dx.doi.org/10.1002/cpe.4872.

[62] M. Alshayeji, B. Behbehani, I. Ahmad, Spark-based parallel processing whale optimization algorithm, Concurr. Comput.: Pract. Exper. 34 (4) (2022) e6607, http://dx.doi.org/10.1002/cpe.6607.

[63] M. AlJame, I. Ahmad, M. Alfailakawi, Apache spark implementation of whale optimization algorithm, Cluster Comput. 23 (3) (2020) 2021–2034, http://dx.doi.org/10.1007/s10586-020-03162-7.

[64] H. Chen, Z. Hu, L. Han, Q. Hou, Z. Ye, J. Yuan, J. Zeng, A spark-based distributed whale optimization algorithm for feature selection, in: 2019 10th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, Vol. 1, IDAACS, 2019, pp. 70–74, http://dx.doi.org/10.1109/IDAACS.2019.8924334.

[65] B. Wu, G. Wu, M. Yang, A mapreduce based ant colony optimization approach to combinatorial optimization problems, in: 2012 8th International Conference on Natural Computation, 2012, pp. 728–732.

[66] C.-W. Tsai, S.-J. Liu, Y.-C. Wang, A parallel metaheuristic data clustering framework for cloud, J. Parallel Distrib. Comput. 116 (2018) 39–49, http://dx.doi.org/10.1016/j.jpdc.2017.10.020, towards the Internet of Data: Applications, Opportunities and Future Challenges..

[67] X. Xu, Z. Ji, F. Yuan, X. Liu, A novel parallel approach of cuckoo search using mapreduce, in: Proceedings of the 2014 International Conference on Computer, Communications and Information Technology, Atlantis Press, 2014/01, pp. 114–117, http://dx.doi.org/10.2991/ccit-14.2014.31.

[68] L.T. Koczy, J. García, F. Altimiras, A. Peña, G. Astorga, O. Peredo, A binary cuckoo search big data algorithm applied to large-scale crew scheduling problems, Complexity 2018 (2018) 8395193, http://dx.doi.org/10.1155/2018/8395193.

[69] N. Al-Madi, I. Aljarah, S.A. Ludwig, Parallel glowworm swarm optimization clustering algorithm based on mapreduce, in: 2014 IEEE Symposium on Swarm Intelligence, 2014, pp. 1–8, http://dx.doi.org/10.1109/SIS.2014.7011794.

[70] G. Miryala, S.A. Ludwig, Comparing spark with mapreduce: Glowworm swarm optimization applied to multimodal functions, Int. J. Swarm. Intell. Res. 9 (3) (2018) 1–22, http://dx.doi.org/10.4018/IJSIR.2018070101.

[71] T. Ashish, S. Kapil, B. Manju, Parallel bat algorithm-based clustering using mapreduce, in: G.M. Perez, K.K. Mishra, S. Tiwari, M.C. Trivedi (Eds.), Networking Communication and Data Knowledge Engineering, 2018, pp. 73–82.

[72] H. Chen, P. Chang, Z. Hu, H. Fu, L. Yan, A spark-based ant lion algorithm for parameters optimization of random forest in credit classification, in: 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference, ITNEC, 2019, pp. 992–996, http://dx.doi.org/10.1109/ITNEC.2019.8729397.

[73] M. Alfailakawi, M. Aljame, I. Ahmad, Parallel and distributed implementation of sine cosine algorithm on apache spark platform, IEEE Access 9 (2021) 77188–77202, http://dx.doi.org/10.1109/ACCESS.2021.3082026.

[74] C. Li, T. Wen, H. Dong, Q. Wu, Z. Zhang, Implementation of parallel multi-objective artificial bee colony algorithm based on spark platform, in: 11th International Conference on Computer Science & Education, ICCSE 2016, Nagoya, Japan, August (2016) 23-25, IEEE, 2016, pp. 592–597, http://dx.doi.org/10.1109/ICCSE.2016.7581647.

[75] T. Wen, H. Liu, L. Lin, B. Wang, J. Hou, C. Huang, T. Pan, Y. Du, Multiswarm artificial bee colony algorithm based on spark cloud computing platform for medical image registration, Comput. Methods Programs Biomed. 192 (2020) 105432, http://dx.doi.org/10.1016/j.cmpb.2020.105432.

[76] Chun-Wei Tsai, Heng-Ci Chang, Kai-Cheng Hu, Ming-Chao Chiang, Parallel coral reef algorithm for solving jsp on spark, in: 2016 IEEE International Conference on Systems, Man, and Cybernetics, SMC, 2016, pp. 001872–001877.

[77] W. Lee, Y. Hsiao, W. Hwang, Designing a parallel evolutionary algorithm for inferring gene networks on the cloud computing environment, BMC Syst. Biol. 8 (1) (2014) 5.

[78] D. Gaifang, F. Xueliang, L. Honghui, X. Pengfei, Cooperative ant colony-genetic algorithm based on spark, Comput. Electr. Eng. 60 (2017) 66–75, http://dx.doi.org/10.1016/j.compeleceng.2016.09.035.

[79] M. Moslah, M.A.B. HajKacem, N. Essoussi, Spark-Based Design of Clustering using Particle Swarm Optimization, Springer International Publishing, Cham, 2019, pp. 91–113, http://dx.doi.org/10.1007/978-3-319-97864-2_5.

[80] M. Sherar, F. Zulkernine, Particle swarm optimization for large-scale clustering on apache spark, in: 2017 IEEE Symposium Series on Computational Intelligence, SSCI, 2017, pp. 1–8, http://dx.doi.org/10.1109/SSCI.2017.8285208.

[81] J. Yuan, An anomaly data mining method for mass sensor networks using improved pso algorithm based on spark parallel framework, J. Grid Comput. 18 (2) (2020) 251–261, http://dx.doi.org/10.1007/s10723-020-09505-3.

[82] K. Tadist, F. Mrabti, N.S. Nikolov, A. Zahi, S. Najah, Sdpso: Spark distributed pso-based approach for feature selection and cancer disease prognosis, J. Big Data 8 (1) (2021) 19, http://dx.doi.org/10.1186/s40537-021-00409-x.

[83] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, I. Stoica, 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, USENIX Association, USA, 2012, p. 2.

[84] J. Al-Sawwa, S.A. Ludwig, Parallel particle swarm optimization classification algorithm variant implemented with apache spark, Concurr. Comput.: Pract. Exper. 32 (2) (2020) e5451, http://dx.doi.org/10.1002/cpe.5451.

[85] C. Deng, X. Tan, X. Dong, Y. Tan, A parallel version of differential evolution based on resilient distributed datasets model, in: M. Gong, P. Linqiang, S. Tao, K. Tang, X. Zhang (Eds.), Bio-Inspired Computing – Theories and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 84–93.

[86] F. Chávez, F.F. de Vega, D. Lanza, C. Benavides, J. Villegas, L. Trujillo, G. Olague, G. Román, Deploying massive runs of evolutionary algorithms with ecj and hadoop: Reducing interest points required for face recognition, Int. J. High Perform. Comput. Appl. 32 (5) (2018) 706–720, http://dx.doi.org/10.1177/1094342016678302.

[87] C. Barba-González, J. García-Nieto, A.J. Nebro, J.A. Cordero, J.J. Durillo, I. Navas-Delgado, J.F. Aldana-Montes, jmetalsp: A framework for dynamic multi-objective big data optimization, Appl. Soft Comput. 69 (2018) 737–748, http://dx.doi.org/10.1016/j.asoc.2017.05.004.

[88] C. Barba-González, A.J. Nebro, A. Benítez-Hidalgo, J. García-Nieto, J.F. Aldana-Montes, On the design of a framework integrating an optimization engine with streaming technologies, Future Gener. Comput. Syst. 107 (2020) 538–550, http://dx.doi.org/10.1016/j.future.2020.02.020.

[89] C. Barba-Gonzaléz, J. García-Nieto, A.J. Nebro, J.F. Aldana-Montes, Multi-objective big data optimization with jmetal and spark, in: H. Trautmann, G. Rudolph, K. Klamroth, O. Schütze, M. Wiecek, Y. Jin, C. Grimme (Eds.), Evolutionary Multi-Criterion Optimization, Springer International Publishing, Cham, 2017, pp. 16–30.

[90] J. Kennedy, R. Eberhart, Particle swarm optimization, in: Proceedings of ICNN'95-International Conference on Neural Networks, Vol. 4, IEEE, 1995, pp. 1942–1948.

[91] M. Ciavotta, S. Krstić, W.-J. Tamburri, Hyperspark: A data-intensive programming environment for parallel metaheuristics, in: 2019 IEEE International Congress on Big Data (BigDataCongress), 2019, pp. 85–92, http://dx.doi.org/10.1109/BigDataCongress.2019.00024.

[92] K. Krawiec, C. Simons, J. Swan, J.R. Woodward, Metaheuristic Design Patterns: New Perspectives for Larger-Scale Search Architectures, IGI Global, 2018, pp. 1–36.

[93] P.B.C. Miranda, R.B.C. Prudêncio, A novel context-free grammar for the generation of pso algorithms, Nat. Comput. 19 (3) (2020) 495–513, http://dx.doi.org/10.1007/s11047-018-9679-9.

[94] A. REYES-Amaro, E. Monfroy, F. Richoux, POSL: A Parallel-Oriented Metaheuristic-Based Solver Language, Springer International Publishing, Cham, 2018, pp. 91–107, http://dx.doi.org/10.1007/978-3-319-58253-5_6.

[95] J.M. Cruz-Duarte, J.C. Ortiz-Bayliss, I. Amaya, Y. Shi, H. Terashima-Marín, N. Pillay, Towards a generalised metaheuristic model for continuous optimisation problems, Mathematics 8 (11) http://dx.doi.org/10.3390/math8112046.

[96] M. Odersky, L. Spoon, B. Venners, Programming in Scala, Artima, 2008.

[97] J. Swan, S. Adriaensen, M. Bishr, E.K. Burke, J.A. Clark, P. De Causmaecker, J. Durillo, K. Hammond, E. Hart, C.G. Johnson, et al., A research agenda for metaheuristic standardization, in: MIC 2015: The XI Metaheuristics International Conference, 2015, pp. 1–3.

[98] X.C. Pardo, Scala implementation of the LSGO2013 benchmark, 2022, http://dx.doi.org/10.5281/zenodo.6504231.

[99] C.L. Camacho-Villalón, M. Dorigo, T. Stützle, Pso-x: A component-based framework for the automatic design of particle swarm optimization algorithms, IEEE Trans. Evol. Comput. 26 (3) (2022) 402–416, http://dx.doi.org/10.1109/TEVC.2021.3102863.

[100] Q. Liu, W. Wei, H. Yuan, Z.-H. Zhan, Y. Li, Topology selection for particle swarm optimization, Inform. Sci. 363 (2016) 154–173, http://dx.doi.org/10.1016/j.ins.2016.04.050.

[101] A. McNabb, M. Gardner, K. Seppi, An Exploration of Topologies and Communication in Large Particle Swarms, vol. 869, Faculty Publications, 2009, http://dx.doi.org/10.1109/CEC.2009.4983015.

[102] N. Lynn, M.Z. Ali, P.N. Suganthan, Population topologies for particle swarm optimization and differential evolution, Swarm Evol. Comput. 39 (2018) 24–35, http://dx.doi.org/10.1016/j.swevo.2017.11.002.

[103] R. Mendes, J. Kennedy, J. Neves, The fully informed particle swarm: simpler, maybe better, IEEE Trans. Evol. Comput. 8 (3) (2004) 204–210, http://dx.doi.org/10.1109/TEVC.2004.826074.

[104] S. Wang, Y. Zhang, S. Wang, G. Ji, A comprehensive survey on particle swarm optimization algorithm and its applications, Math. Probl. Eng. 2015 (2015) 931256, http://dx.doi.org/10.1155/2015/931256.

[105] M.R. Bonyadi, Z. Michalewicz, Particle swarm optimization for single objective continuous space problems: A review, Evolut. Comput. 25 (1) (2017) 1–54, http://dx.doi.org/10.1162/EVCO_r_00180.

[106] S. Sengupta, S. Basak, R.A. Peters, Particle swarm optimization: A survey of historical and recent developments with hybridization perspectives, Mach. Learn. Knowl. Extract. 1 (1) (2019) 157–191, http://dx.doi.org/10.3390/make1010010.

[107] D. Freitas, L.G. Lopes, F. Morgado-Dias, Particle swarm optimisation: A historical review up to the current developments, Entropy 22 (3) 10.3390/e22030362.

[108] E.H. Houssein, A.G. Gad, K. Hussain, P.N. Suganthan, Major advances in particle swarm optimization: Theory, analysis, and application, Swarm Evol. Comput. 63 (2021) 100868, http://dx.doi.org/10.1016/j.swevo.2021.100868.

[109] D. Wang, D. Tan, L. Liu, Particle swarm optimization algorithm: an overview, Soft Comput. 22 (2) (2018) 387–408, http://dx.doi.org/10.1007/s00500-016-2474-6.

[110] M. Clerc, Beyond standard particle swarm optimisation, IJSIR 1 (2010) 46–61, http://dx.doi.org/10.4018/jsir.2010100103.

[111] M. Clerc, J. Kennedy, The particle swarm - explosion, stability, and convergence in a multidimensional complex space, IEEE Trans. Evol. Comput. 6 (1) (2002) 58–73, http://dx.doi.org/10.1109/4235.985692.

[112] J.C. Bansal, P.K. Singh, M. Saraswat, A. Verma, S.S. Jadon, A. Abraham, Inertia weight strategies in particle swarm optimization, in: 2011 Third World Congress on Nature and Biologically Inspired Computing, 2011, pp. 633–640, http://dx.doi.org/10.1109/NaBIC.2011.6089659.

[113] P.C. Fourie, A.A. Groenwold, The particle swarm optimization algorithm in size and shape optimization, Struct. Multidiscip. Optim. 23 (4) (2002) 259–267, http://dx.doi.org/10.1007/s00158-002-0188-0.

[114] F. Glover, A template for scatter search and path relinking, 1997, pp. 1–51, http://dx.doi.org/10.1007/BFb0026589.

[115] S. Wagner, G. Kronberger, A. Beham, M. Kommenda, A. Scheibenpflug, E. Pitzer, S. Vonolfen, M. Kofler, S. Winkler, V. Dorfer, M. Affenzeller, Advanced methods and applications in computational intelligence, 6 of topics in intelligent engineering and informatics, in: Ch. Architecture and Design of the HeuristicLab Optimization Environment, Springer, 2014, pp. 197–261.

[116] D. Doblas, A.J. Nebro, M. López-Ibáñez, J. García-Nieto, C.A. Coello Coello, Automatic design of multi-objective particle swarm optimizers, in: M. Dorigo, H. Hamann, M. López-Ibáñez, J. García-Nieto, A. Engelbrecht, C. Pinciroli, V. Strobel, C. Camacho-Villalón (Eds.), Swarm Intelligence, Springer International Publishing, Cham, 2022, pp. 28–40.

[117] D. Michail, J. Kinable, B. Naveh, J.V. Sichi, Jgrapht: a java library for graph data structures and algorithms, ACM Trans. Math. Software 46 (2) http://dx.doi.org/10.1145/3381449.

[118] X.C. Pardo, Experimental data from the initial validation of spark-eclib, a new framework for distributed metaheuristics in spark, 2023, http://dx.doi.org/10.5281/zenodo.8369329.

[119] Virtual library of simulation experiments: Test functions and datasets, 2023, https://www.sfu.ca/ssurjano/optimization.html. (Accessed: 1 June 2023).

[120] X.C. Pardo, Ecj issue 82: bugs in the pso implementation, 2023, https://github.com/GMUEClab/ecj/issues/82. (Accessed: 1 June 2023).

[121] J. Veiga, J. Enes, R.R. Expósito, J. Touriño, Bdev3.0: Energy efficiency and microarchitectural characterization of big data processing frameworks, Future Gener. Comput. Syst. 86 (2018) 565–581, http://dx.doi.org/10.1016/j.future.2018.04.030.

[122] Pluton computing cluster website, 2023, https://pluton.dec.udc.es. (Accessed: 9 December 2023).