HyperSpark: A Data-Intensive Programming Environment for Parallel Metaheuristics

Michele Ciavotta
DISCO
University of Milan-Bicocca
Milan, Italy
michele.ciavotta@unimib.it

Srđan Krstić
Dept. of Computer Science
ETH Zurich
Zurich, Switzerland
srdan.krstic@inf.ethz.ch

Damian A. Tamburri, Willem-Jan Van Den Heuvel
TU/e and Uni. Tilburg,
Jheronymus Academy of Data Science
s'Hertogenbosch, The Netherlands
d.a.tamburri@tue.nl, W.J.A.M.v.d.Heuvel@jads.nl

***Abstract*—Metaheuristics are search procedures used to solve complex, often intractable problems for which other approaches are unsuitable or unable to provide solutions in reasonable times. Although computing power has grown exponentially with the onset of Cloud Computing and Big Data platforms, the domain of metaheuristics has not yet taken full advantage of this new potential. In this paper, we address this gap by proposing HyperSpark, an optimization framework for the scalable execution of user-defined, computationally-intensive heuristics. We designed HyperSpark as a flexible tool meant to harness the benefits (e.g., scalability by design) and features (e.g., a simple programming model or ad-hoc infrastructure tuning) of state-of-the-art big data technology for the benefit of optimization methods. We elaborate on HyperSpark and assess its validity and generality on a library implementing several metaheuristics for the Permutation Flow-Shop Problem (PFSP). We observe that HyperSpark results are comparable with the best tools and solutions from the literature. We conclude that our proof-of-concept shows great potential for further research and practical use.**

***Index Terms*—Parallel Metaheuristics, Programming Model, Hyperheuristics, Optimization, Framework**

I. Introduction

The word *Metaheuristic* (from ancient Greek *meta* = “beyond, higher-level" and *heuriskein* = “to find") defines a class of search algorithms able to find near-optimal solutions for *hard* optimization problems by working on an abstract plane [41]. While ordinary heuristics are explicitly designed to efficiently tackle a specific problem, by exploiting a profound knowledge about it, metaheuristic algorithms implement a more general optimization schema, flexible and easily adaptable to multiple different problems, which usually entails a reduced design and implementation time. The main shortcoming of this class of methods is the relative inefficiency with respect to *ad hoc* solutions. For these reasons, metaheuristics are typically applied in scenarios where no satisfactory heuristic is known.

Research Context. Metaheuristic approaches, often hybridized with local search techniques [42] are popular in *Search-Based Software Engineering* (SBSE) [19], [22], e.g., in defect prediction [27], [28], automated testing [24] and more [21], [23]. However, to achieve further generality, streamline the creation of new metaheuristics, and organize the knowledge acquired over the years metaheuristic optimization frameworks (MOFs) have been proposed. A MOF is an abstraction that provides a diverse set of reusable components, and the necessary mechanisms to selectively alter them with user-defined functions to handle specific optimization problems. Parejo surveys [33] and compares systematically top existing MOFs. The major drawbacks are summarized as follows: (a) MOFs have minimal support for parallel and distributed execution; (b) MOFs lack support for the activity of hyperheuristics programming (i.e., the search-based selection and combination of multiple search methods into a single optimization solution [7]); and (c) MOFs are not designed according to known software engineering best practices.

Research Objective. In response to the above limitations, we take a software engineering perspective on the problem and propose *HyperSpark*, a MOF that features a transparently-distributed programming model to create *parallel-by-design* search methods. In fact, HyperSpark supports the design of parallel metaheuristics and their execution on a cluster of distributed and interconnected computational nodes by relying on the programming model of Apache Spark[1].The key contribution of this paper is the detailed outline and proof-of-concept evaluation of a programming environment for parallel search algorithms with particular reference to metaheuristics.

Research Solution Features. The main properties featured by HyperSpark are:

(1) *User-invisible parallelization* - as it handles code distribution and parallelization transparently; (2) *Configurability* - as it exposes configuration parameters to fine tune the execution and parallelization of optimization algorithms (e.g., number of execution nodes, number of used cores per node, etc); (3) *Flexibility* - as it exposes a programming model that allows users to, e.g., (a) define arbitrary parallelization strategies, (b) design and run hyperheuristics (combining different metaheuristics); (4) *Cooperation* - as it allows for synchronous communication among parallel instances of the algorithm and, therefore, supports a large class of cooperative parallel search algorithms, which are yet to be fully explored both in theory and practice [26]; (5) *Extensibility* - as it is developed in a object-oriented and functional programming language (*Scala*) with mechanisms (e.g., inheritance, traits, implicits, and late binding) that facilitate the adaptation of metaheuristic

[1]https://spark.apache.org/

algorithms to specific problems, fostering extensible design and code reuse; (6) *Portability* - as it can run on any JVM-enabled architecture;

Solution Evaluation. To assess HyperSpark's efficiency and generality in offering easy-to-use abstractions for create suitable optimization algorithms, we implemented a proof-of-concept library offering several methods for the widely-known Permutation Flow-Shop Problem (PFSP) [45]. PFSP reflects a class of scheduling problems "*[...] in which the flow control shall enable an appropriate sequencing for each job and for processing on a set of resources in compliance with given processing orders*" [12], [45]. In particular, the goal of this proof-of-concept was to demonstrate two properties: a) that it is possible for the user to implement a sequential search algorithm, that the framework then commits itself to distribute and synchronize the various replicas in a transparent way; b) That the technological and architectural choices underlying HyperSpark do not prevent the achievement of results comparable with the state-of-the-art. We observe that the non-optimized nor fine-tuned algorithms of the library achieve in experimenting with PFSP results that are comparable with the best solutions from literature, e.g., differing by around 7-14% in terms of time overheads to achieve an acceptable solution.

Structure of the paper. The reminder of this paper is structured as follows. Section II sets the work in context and briefly outlines the state-of-the-art. Section III introduces HyperSpark and outlines basic implementation details and examples. Section IV presents and discusses the results of an experimental evaluation of the framework. Specifically, the efficiency of HyperSpark in solving PFSP and its comparison with existing MOFs. Finally, Section V concludes the paper.

II. Background and State of The Art

The long-term goal of our intended contribution with HyperSpark is two-fold. On the one hand, state-of-the-art in search-based software engineering is rich with approaches, concepts, prototypes, and advances specifically designed to address some of the hardest problems in software engineering (e.g., testing, defect analysis). However, there is a sensible gap between operations research and Search-Based Software Engineering (SBSE) disciplines, e.g., in applying metaheuristics and hyperheuristics flexibly to software engineering problems and doing so *at scale*. With HyperSpark we aim to help to address this gap by providing a programming environment built on top of the flexible Apache Spark computing engine, making parallelization as seamless as possible and enabling the flexible use of metaheuristics in development and application.

On the other hand, as previously introduced, state-of-the-art in operations research is dense with *ad-hoc* solutions to parallelization of metaheuristics. HyperSpark offers a data-intensive perspective on the matter — it is our intention to offer a large-scale compute platform that helps to address the scalability issues of *ad-hoc* solutions for the benefit of both academics and practitioners in the field.

The framework presented in this paper is related and inspired by other works that can be roughly classified into two groups: parallel MOFs and Big Data Platforms to support metaheuristics. The first group corresponds to the current platforms that support parallel execution of metaheuristic algorithm: ECJ [1], ParadisEO [8], EvA2 [25], MALLBA [2], and jMetal [14]. However, none of the MOFs above exposes a programming model to support the design of hyperheuristics. Besides, jMetal does not support distributed execution, while the rest of the MOFs presented provide limited flexibility in parallel metaheuristic design. More specifically, they do not support a class of metaheuristics that performs the local search using parallel and distributed neighborhood exploration [33]. The second group consists of a large body of works that adopt different Big Data processing technologies to implement particular algorithms [3], [34], [44] or a particular class of algorithms [5], [16], [18].

A work that shares some analogies with our framework (as MOF over Spark) is presented in [34]. However, the authors extend the jMetal [14] framework only to accommodate the streaming feature of Apache Spark without fully exploiting its parallel and distributed execution capabilities.

Finally, from a multi-agent perspective, Multi-Agent MOFs such as MAGMA [29] or followup works by Lopes et al. [17] offer an orthogonal perspective wherefore metaheuristic agents offer specific implementations of multi-agent systems (MAS) [4] to solve optimization problems with agent-based metaheuristic algorithms.

III. HyperSpark: an Overview

This section illustrates our research solution. First, we focus on how HyperSpark pursuits that goal, elaborating on its architecture (see Sec. III-A). Second, we present the programming model and a typical workflow (Sec. III-B), the framework runtime model is presented in Sec. III-C and, lastly, Sec. III-D presents some usage scenarios.

A. HyperSpark Core Architecture

The primary goal of HyperSpark is to provide a programming environment that is general enough to accommodate various representations of *Problem* instances, solved employing different kinds of search *Algorithms* that produce sets of *Solutions* with some arbitrary encoding. We refine previous research [14] to distill a generic scheme.

The core architectural entities of HyperSpark are shown in white boxes on Figure 1. The architecture consists of a single Scala trait[2] and three classes that capture the main concepts enforced by HyperSpark. The *Algorithm* trait represents a generic optimization method: its purpose is solving a problem, represented as an instance of *Problem* class and producing a (set of) *Solution* object(s). The *Problem* is an abstract class defined to encode the solution space and objective function of a particular problem. Given a *Solution* object, a *Problem* object must provide a value of the objective function, typed V for that solution. We argue that this design is simple and flexible enough to accommodate any arbitrary combination

[2]Traits are specific concepts inherited from the Scala programming language similar, but more powerful than Java interfaces.

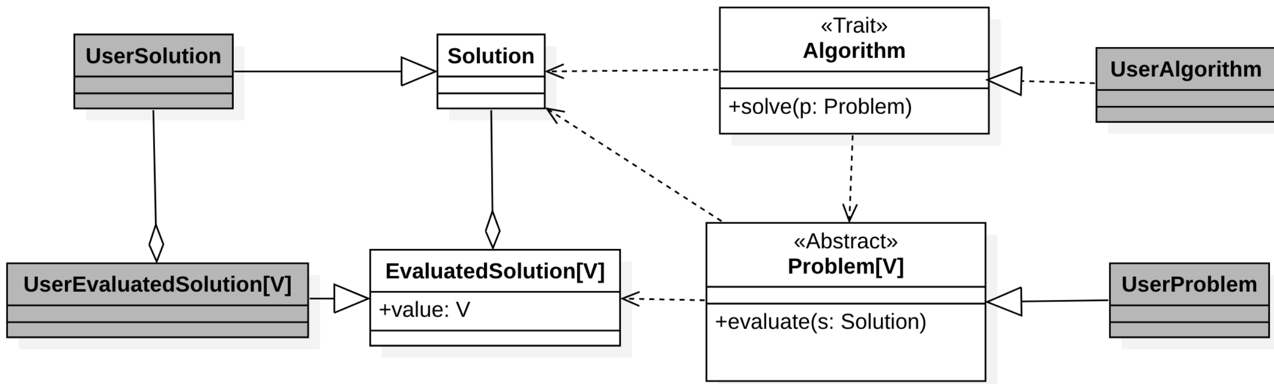

Fig. 1: Class diagram of HyperSpark base classes and traits.

of a problem representation, search algorithm, and solution encoding. Users can implement custom algorithms by mixing[3] in the *Algorithm* trait and overriding their *solve* method. Entities colored in grey on Figure 1 exemplify how users can extend HyperSpark elements. In order to introduce a new algorithm represented by the class *UserAlgorithm*, one needs to mix in the *Algorithm* trait. Similarly, user-defined problems and solutions need to extend the respective base classes with instances describing the extensions.

It is worth noticing that in the HyperSpark core architecture there is no reference to parallelization whatsoever. This is due to the particular policy that the framework enforces, which can be resumed with the expression, *"write locally, distribute painlessly"*. It means that the developer is encouraged to write plain single-threaded methods to tackle the considered problem, as it were to be executed on a local machine. The framework takes care of autonomously and transparently distributing the code, running it in parallel, and collecting results following user specifications.

B. Programming Model and Typical Workflow

As previously seen, the first step for the developer interested in creating a parallel and distributed algorithm is to provide a suitable implementation for the core elements of HyperSpark, that is providing the appropriate problem representation, solution encoding, and at least one (single-threaded) algorithm. Without loss of generality, we refer to the most straightforward case of one single algorithm to be parallelized; nonetheless, HyperSpark is able to seamlessly handle the cooperation of multiple different algorithms (as in a hyperheuristic).

The immediately following step consists in extending and instantiating a HyperSpark execution workflow. Figure 2 illustrates the base workflow executed by the framework. In a nutshell, HyperSpark iteratively splits the problem, distributes the algorithm code to the available computational nodes, executes it, aggregates the outcomes and uses them to feedback the process. In the following, we describe the internal details of each phase behind this process.

Starting from a problem (i.e., *solution space* and *objective function*), the framework can optionally split it into different sub-problems. This means, for example, that each parallel *instance* is assigned to a different region of the solution space to explore, or a different objective function to optimize. Splitting a problem, however, is a specific task that highly depends on the particular problem at hand as well as its representation; this is consequently left at the discretion of the user. Assuming that the problem is indeed split, for each (sub-)problem the user may specify an algorithm to solve it. There are no constraints on the type of algorithm that can be used as long as the user provides a meaningful way to aggregate the outcomes (see solution aggregation phase). The framework does not provide a default value for the algorithm selection - this is consequently a mandatory step. If the user specifies a single algorithm at this step, it will be executed in parallel. This use makes sense in scenarios whereby multiple algorithm instances are set for cooperative optimization.

In the next step of the workflow in Figure 2, HyperSpark allows the users to (optionally) specify a seeding strategy; it determines the way an initial solution is generated at each iteration of HyperSpark (also referred to as *stage*). Since this step is optional, HyperSpark does not provide initial defaults. Nonetheless, it must be pointed out that this phase is of paramount importance when it comes to design an effective parallel optimization as it defines a way to implement information distribution and collaboration among the instances of the algorithm. For example, at each iteration, the user can be interested in preserving the best solution and using it to generate good seeds for the next stage.

At this point, the framework distributes and runs the algorithm. To avoid high synchronization times the user is encouraged to implement algorithms that stops after the same amount of time as HyperSpark has to wait that all the algorithms complete their execution to collect the solutions generated and combine them suitably. Such a programming model allows the user to easily implement an aggregation function that combines solutions from different algorithms. In case the user

[3]Mixing in traits in Scala is analogous to implementing interfaces in Java. Yet, mixing is more powerful as, besides establishing the type hierarchy, it also allows the sub-type to inherit both trait's functionality and state.

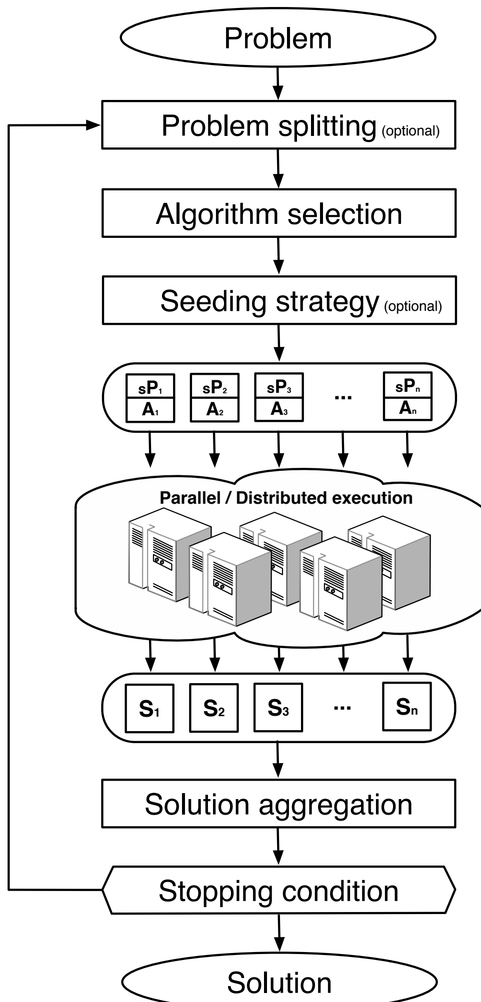

Fig. 2: Workflow of the HyperSpark.

is not interested in particular re-combination strategies, the framework provides by design a simple aggregation function that returns the solution featuring the minimum value of the objective function.

Finally, the stopping condition of the entire workflow is an arbitrary predicate that determines when HyperSpark stops its execution - this stopping condition is checked after each iteration of the parallel execution (stage). This condition is an arbitrary predicate since it can depend on various aspects of the execution. For instance, one can just specify a fixed number of iterations, specify a timeout, or a more complex condition that depends on the solution, e.g., solution precision must be within a fixed threshold. Once the above workflow is fully instantiated for the metaheuristic at hand, HyperSpark performs transparent parallel and distributed execution of the selected algorithms.

C. Runtime Model

As previously stated, the HyperSpark architecture relies on the Apache Spark project and inherits its runtime model. Spark is a leading, efficient, general, open source Big Data processing engine. A Spark application runs on an independent

Listing 1: Example of HyperSpark workflow definitiom

```
1    val problem =
         PFSProblem.fromResources("inst_ta054.txt")
2    val conf = new FrameworkConf()
3                .setProblem(problem)
4                .setNAlgorithms(new GAAlgorithm(), 4)
5                .setStoppingCondition( new
                     TimeExpired(100))
6                .setDeployment("local", 4)
7    val solution = Framework.run(conf)
8    println(solution)
```

set of processes (supervised by agents called *executors* in Spark lingo) distributed on a cluster and coordinated by a primary process (called the *driver*). A driver program cannot decide on the node, core and memory allocation for executors, but rather delegates this to a cluster manager.

More specifically, once the driver program is executed, it connects to a cluster manager (either Mesos, YARN, or a standalone Spark cluster manager), which allocates resources. By communicating with the cluster manager, the driver program sets up executors to run singular algorithms, sends the code to execute (packaged into JAR files) to the executors and coordinates their execution. More details on the internal Spark runtime model are beyond the scope of this paper; the reader is, nonetheless, referred to Apache Spark homesite for more details.

D. Usage Scenarios

In this section, the use of HyperSpark is presented, and some insights are provided utilizing two illustrative examples. Suppose an implementation of the genetic algorithm from [36] for the PFSP (see Sec. IV for more details) is available and the used wish to run four instances of the algorithm in parallel. This scenario can be realized in HyperSpark as in Listing 1. The first line instantiates an object of the *PFSProblem* class, which is our custom representation of the PFSP. This is done through a factory method *fromResources* that reads the instance parameters from a file. Next, a HyperSpark configuration object called *FrameworkConf* is created that exposes the main API of HyperSpark. When developing the configuration object, we committed to the *convention over configuration* design paradigm. This means that all of the setter methods, excluding the ones defining the problem and the algorithms, are optional. If the user does not set a particular property, the framework uses a reasonable default value. The execution will not start until the run method is executed passing the

Listing 2: More complex example of HyperSpark workflow definitiom

```
1    val problem =
         PFSProblem.fromResources("inst_ta054.txt")
2    val conf = new FrameworkConf()
3              .setProblem(problem)
4              .setNAlgorithms( new HGAlgorithm(), 100)
5              .setSeedingStrategy( new
                   SlidingWindow(sqrt(problem.numOfJobs)))
6              .setStoppingCondition(new TimeExpired(100))
7              .setStages(5)
8              .setDeployment("spark", 20)
9              .setProperty(spark.executor.cores,5)
10             .setProperty(spark.executor.memory,8g)
11   val solution = Framework.run(conf)
12   println(solution)
```

configuration object, as shown in line 7. Lines 3–6 constitute a minimalist example of the use of the HyperSpark programming model, and they define a simple high-level execution workflow. In Line 3 we specify that all the algorithms solve the same *problem* instance. Line 4 states that the algorithm *GAAlgorithm* has to be executed four times in parallel with no cooperation and stop after 100 seconds of execution (line 5). The *setDeployment* method is used to specify the deployment mode and the number of parallel execution processes. In this case, the *local* mode is used that creates four processes on the local machine to execute the algorithms. Other modes include *spark* (Spark standalone cluster manager), *mesos* (i.e., using Mesos) and *yarn-client* or *yarn-cluster* for YARN.

The extended example reported in Listing 2 shows how to set up the execution of algorithms that cooperate synchronously. Specifically, line 4 specifies that 100 instances of the cooperative hybrid genetic algorithm from [48] implemented in the *HGAlgorithm* class are run in parallel on the same problem. The cooperation is facilitated by adopting an appropriate seeding strategy and setting a certain number of stages to a value greater than 1 (line 7). In particular, the seeding strategy provides a function that generates a seed solution for each instance at stage n altering the results of the solution aggregation phase at stage $n-1$. In the specific case reported here, the solution aggregation returns the minimal makespan solution found during a stage whereas the seeding strategy, named *SlidingWindow*, operates selecting w adjacent elements (window) of a base permutation and randomly permuting the others. Then, the window is shifted one position. The process is repeated until a new solution is generated for each instance of the algorithm.

In line 7 we set the number of stages in the workflow that we expect the framework to execute. Consequently, the framework executes each stage in roughly 20 seconds to account for the 100-second stopping condition. At the end of each stage, the outcomes of each instance are aggregated and reported to the next stage to allow for a suitable cooperation mechanism. Lastly, HyperSpark will deploy the executors using the Spark standalone cluster manager (line 8); assigning to each executor five cores (line 9) and 8 gigabytes of memory (line 10).

IV. Evaluation

The aim of this Section is to present the proof-of-concept experiments carried out to evaluate HyperSpark and discuss the achieved results.

A. Research Questions and Evaluation Scope

In the scope of our evaluation, we set out to understand the degree to which our research solution offers a valid programming environment alternative to *ad hoc* solutions specifically designed to address PFSP or to support specific approaches. Hence, we aim at operating a proof-of-concept [11], by evaluating the computational overhead (if any). Similarly, we aims at comparing HyperSpark processing results to the state-of-the-art.

In this evaluation scope two research questions are addressed, namely:

TABLE I: List of algorithms implemented in HyperSpark-PFSP library

Algorithm	Authors	Year	Ref.	Name
NEH	Nawaz, Enscore and Ham	*1983*	[30]	*NEH*
Iterated Greedy	Ruiz and Stützle	*2007*	[37]	*IG*
Genetic Algorithms	Reeves	*1995*	[36]	*GA*
Hybrid Genetic Algorithms	Zheng and Wang	*2003*	[48]	*HG*
Simulated Annealing	Osman and Potts's adaptation for PFSP	*1989*	[32]	*SA*
Improved S-Annealing	Xu and Oja	*1990*	[46]	*ISA*
Taboo Search	Taillard	*1996*	[39]	*TS*
Taboo Search + Backjump	Novicki and Smutnicki	*1996*	[31]	*TSAB*
Ant Colony Optimization	Dorigo and Stützle	*2010*	[13]	*ACO*
Max Min Ant System	Stützle	*1997*	[38]	*MMAS*
mMMAS	Rajendran and Ziegler	*2004*	[35]	*MMMAS*
PACO	Rajendran and Ziegler	*2004*	[35]	*PACO*

TABLE II: Time overhead analysis - highest and lowest values for time percentages highlighted in bold.

Instance ID	# CPU	time (s)	ω_P (s)	ω_I (s)	ω_T (s)	$\sum\omega$ (%)
ta001	1	15.0	2.8	5.8	0.6	**61.3**
ta001	8	16.2	2.9	7.2	0.2	63.6
ta001	16	20.6	4.7	7.6	0.6	62.6
ta001	24	25.9	6.5	9.4	0.4	63.1
ta001	32	30.9	7.8	11.8	0.6	**65.2**
ta001	40	36.9	10.3	12.8	0.6	64.1
ta111	1	341.5	18.1	5.2	0.2	**6.9**
ta111	8	346.4	20.9	4.4	0.2	7.4
ta111	16	367.9	36.4	4.8	0.4	11.3
ta111	24	371.7	32.6	6.2	0.4	10.5
ta111	32	382.3	37.3	7.2	0.6	11.8
ta111	40	405.3	46.3	11.8	0.8	**14.5**

- (1) *is the *time-overhead* introduced by HyperSpark acceptable in the context of parallel cooperative optimization?*
- (2) *are the algorithms implemented using HyperSpark competitive with respect to the state-of-the-art in terms of *solution quality*?*

B. Evaluation Target and Materials

In response to the previous two research questions, we carried out two different experiments, both involving the Permutation Flow Shop Problem (PFSP), a well-known $\mathcal{NP}$-Complete optimization problem [20] arising from the field of machine scheduling. This problem comes with the well-known Taillard's benchmark [40], which consists of 120 instances providing different processing times, number of jobs (ranging from 20 to 500), and number of machines (from 5 to 20). Each job/number of machines combination features 10 instances. Also, another reason behind the selection of PFSP as a case study is the fact that, for many problem instances, the optimal solution is known and freely available.

Simply put, the PFSP can be defined as a set J of n jobs that need to be processed on a set M of m machines. Every job has to go through all the machines in the same predetermined order. Without loss of generality, we can reorder the machines in such a way that each job has to visit them in order, from machine 1 to machine m. Each job j is associated with a fixed, non-negative, and known in advance processing time p_{ij} for each machine i. Furthermore, at any

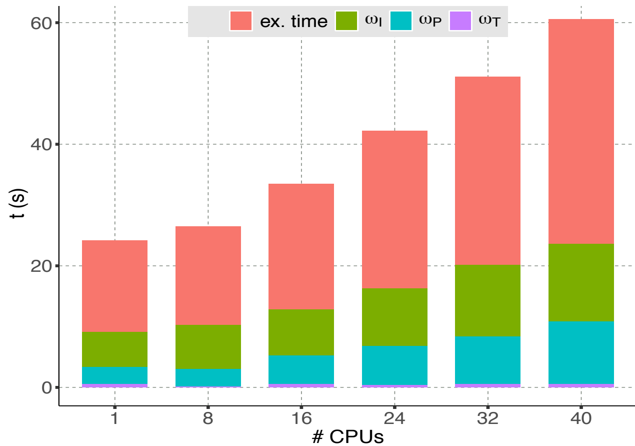

Fig. 3: Execution time vs overheads - a 20-job instance

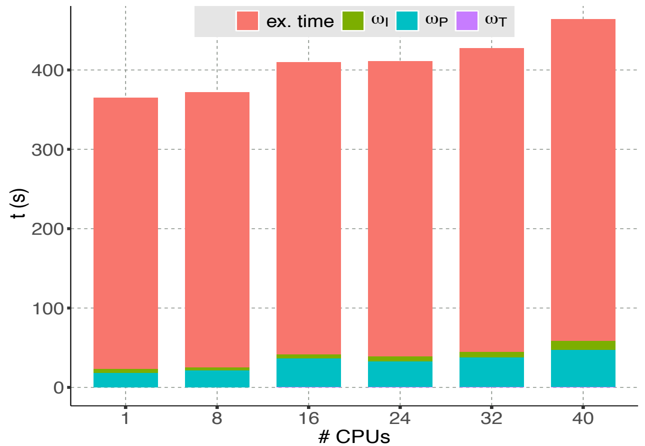

Fig. 4: Execution time vs overheads - a 500-job instance

point in time, each machine can process at most one job, and each job can be processed by (at most) one machine. As a consequence, each machine of the line processes the same sequence of jobs. This problem aims to find a particular sequence that optimizes a specific performance metric. Research on the PFSP introduced several distinct optimization criteria. The most commonly studied objective (also the one used in this work) is the minimization of the maximum completion time (i.e., *Makespan*):

$$C_{max} = \max_{j=1}^{n}\{C_{mj}\};$$

where C_{mj} is the completion time of job j on machine m. The Makespan minimization is directly related with the maximization of machine utilization and reduction of the work-in-progress. For the purposes of the evaluation, we implemented a library of algorithms (listed in Table I) published in the literature for the considered case study problem. This library is open-source, integral part of the contributions conveyed in this paper, and shipped directly within our distribution of HyperSpark. All the algorithms are single-threaded and no specific speed up is implemented.

C. Experimental Setup Features

For the experiments we exploited a workbench cluster consisting of 10 virtual machines, each having 8 CPU cores running at 2.4 GHz and 15 GB of RAM at their disposal for a total of 80 CPU cores and 150 GB. Furthermore, in all the experiments the algorithms are stopped as soon as a timeout of $\mu_s = n \cdot m/2 \cdot 60$ milliseconds is reached (n and m are, respectively, the number of jobs and number of machines for a certain instance of the problem. The execution time has been measured in milliseconds of actual CPU compute time as in the best practices from literature [43]; in this way, more time is assigned to larger (in terms of number of jobs and machines) and harder-to-solve problems.

The Apache Spark environment has been installed in *Standalone* mode, meaning that it manages directly both application scheduling and provisioning of hardware resources. This mode of operation has proven to greatly hamper the performance [47]. Consequently, we argue that the evaluation of our research solution is harnessing a *baseline and improvement-free* performance that HyperSpark can to deliver. Conversely, in future work, we plan to experiment with HyperSpark assessing possible performance boosts using specific resource managers.

D. Experiment 1 – Framework overhead estimation

The aim here is to evaluate the overhead introduced by the framework due to context creation (initialization, denoted ω_I), data and code distribution to the nodes and synchronization (evaluated with parallelization, ω_P) and context termination (ω_T) with respect to the cluster and instance sizes. Therefore, we set up the following experiment. We increased the number of cores available linearly, that is, we considered configurations with 1, 8, 16, 32, 40 cores respectively. Subsequently, five Taillard's instances of size 20, 50, 100, 200 and 500 jobs were randomly chosen; replicating approaches from the literature [43], each one of them was solved five times using one algorithm from our library (namely IG) without cooperation (number of stages equal to 1). Notice that such a choice does not reduce the generality of the experiment since the algorithm and cooperation are factors that do not influence the overhead.

Table II reports an excerpt from the data harvested in the experiment. We can observe that the overhead due to parallelism and synchronization ω_P depends on either the cluster or instance *size*, going from 3 to 47 seconds. Conversely, initialization depends on the cluster size only; however, the growth is quite limited, remaining in the range 5-13 seconds. The termination overhead, instead, is constant and less than 1 second. Figure 3 is a barchart representing variation of the overheats when the number of CPUs allocated is increased for a 20-job/5-machine case. Conversely, by increasing the problem size (bottom half of Tab. II), we noticed that the impact of the total overhead in percentage ($\sum \omega$), which showed quite high values (up to 70%) on small instances (in the range of the tens of jobs), accounts only for 7-14% for

TABLE III: Average RPD from best known solution for methods (IG, HG) parallelized with SS, SPSW and SPFW strategies

		RPD (%)					
		HG	**IG**	**HG**	**IG**	**HG**	**IG**
$\|J\|$	$\|M\|$	SS	SS	$SPSW$	$SPSW$	$SPFW$	$SPFW$
50	5	0.18	0.10	0.15	0.06	0.15	0.06
	10	2.24	1.74	2.27	1.74	2.28	1.75
	20	3.42	2.87	3.50	2.62	3.52	2.67
100	5	0.19	0.10	0.21	0.16	0.21	0.16
	10	1.32	1.11	1.38	1.50	1.38	1.49
	20	3.98	3.58	4.17	3.96	4.17	4.02
200	10	0.87	1.05	0.92	1.03	0.90	1.03
	20	3.65	3.76	3.73	3.87	3.76	3.87
Avg.		1.98	1.79	2.04	1.87	2.05	1.88

the instance with 500 jobs. Such different impact is outlined clearly in Figure 4.

Summary for RQ1. All previous observations lead to the conclusion that the proposed solution shows an overhead on small problems but can be, instead, considered suitable to large (over one hundred jobs) optimization problems, that is, those that would benefit the most from HyperSpark parallelism.

E. Experiment 2 – Solution Quality Evaluation

To evaluate the capability to achieve state-of-the-art results, we set up an experimental evaluation on a cluster with 20 cores. All 120 available problem instances have been solved 10 times using two algorithms, namely IG and HG, with three different seeding strategies and the average relative percentage deviation (RPD) [10] with respect to the best known solution has been calculated. We selected the IG and HG among other algorithms implemented in the HyperSpark library as they resulted the most performing ones in preliminary tests. The three different seeding strategies we consider were the “SameSeeds" (SS), “SeedPlusSlidingWindow" (SPSW), and “SeedPlusFixedWindow" (SPFW).

In Table III we report the aggregate results of this second experiment. We observe that all the considered algorithms return similar results, which are on average about 2% worst than the best known solution for the PFSP. This is a considerable outcome, considering the early stage of the HyperSpark prototype and the fact that the considered algorithms have been implemented, as previously outlined, without any particular framework settings optimization [9], [15].

To complete the comparison with the state-of-the-art, we experimented, on the same pool of instances, involving the well-known MOF ParadisEO [8]. Both in HyperSpark and in ParadisEO we implemented a simple genetic algorithm without speed up (in order to normalize the results), and all the instances of the problem have been solved 5 times. Finally, the RPD has been calculated with respect to the best known solutions. The main highlights for this evaluation are briefly outlined below:

- **Worst results**: For both platforms the highest RPD was achieved for instance 111, which features 500 jobs and 20 machines, 3.8% for ParadisEO and 4.31% for HyperSpark.
- **Best results**: ParadisEO has been able to find the optimal solution for the 24% of the runs (144) while HyperSpark found it 16.17% of the time (97 runs).
- **Statistical test for the difference**: Although the average values of the RPD are close, the Wilcoxon signed rank test [6] for the differences between ParadisEO and HyperSpark (using as paired samples the average RPD for each problem instance, 120 pairs in total) leads us to conclude that the ParadisEO outperforms HyperSpark ($p < 10^{-4}$).

Summary for RQ2. In conclusion, we can state that the parallelization-synchronization approach implemented in HyperSpark is promising and can lead, even in the prototypical, non-optimized form outlined and experimented in this article, to results that are comparable to state-of-the-art *ad-hoc* solutions.

V. Conclusion

Research in parallelization of metaheuristics is an important field aiming at tackle hard problems often using sub-optimal algorithms. Similarly, the uses of such metaheuristics are manifold, most prominently around search-based approaches in the filed of software engineering.

The state of the art provides several previous attempts to reconcile diversity and complexity of parallelization approaches within metaheuristic optimization frameworks (MOFs). These solutions lack: (a) support for parallel and distributed execution; (b) support for design and execution of hyperheuristics, as well as (c) software engineering best practices in their design. We remark that flexibly supporting aspects (a) - (c) with parallel and distributed frameworks is necessary to support the design and experimentation of metaheuristics in the era of Big Data, e.g., to experiment with search-based software engineering solutions over problems currently open or still undergoing experimentation.

In this paper we outline, evaluate, and discuss *HyperSpark*, a programming environment for execution of parallel metaheuristics implemented on top of the *Apache Spark* framework. We aimed at providing support for modern parallel metaheuristics following sound software engineering principles like, ease-of-use, configurability, flexibility, cooperation, extensibility, and portability. In this paper we offered an experimental proof-of-concept evaluation to validate the approach. We conclude that, despite some limitations mainly due to its prototypical nature, HyperSpark has shown potential, above all when dealing with large problem instances.

As future work, we plan to provide better framework support for problem splitting, as well as experimenting further with hyperheuristic algorithms and their impact in HyperSpark. Moreover, we are looking into harnessing Scala and Java

interoperability to integrate out framework with more mature MOFs such as jMetalḞinally, we are planning to investigate HyperSpark extensions that facilitate asynchronous communication for better cooperative optimization.

REFERENCES

[1] ECJ - A Java-based Evolutionary Computation Research System. http://cs.gmu.edu/~eclab/projects/ecj/. Last accessed: 30-01-2017.

[2] E. Alba, G. Luque, J. Garcia-Nieto, G. Ordonez, and G. Leguizamon. MALLBA a software library to design efficient optimisation algorithms. *Int. J. Innov. Comput. Appl.*, 1(1):74–85, 2007.

[3] Wu B., Wu G., and M. Yang. A MapReduce based Ant Colony Optimization approach to combinatorial optimization problems. In *Proc. of ICNC 2012*, pages 728–732, 2012.

[4] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE*. Wiley Series in Agent Technology. Wiley, 2 2007.

[5] M. Bhattacharya, R. Islam, and J. Abawajy. Evolutionary optimization: A Big Data perspective. *Journal of Network and Computer Applications*, 59:416 – 426, 2016.

[6] A. Di Bucchianico. Combinatorics, computer algebra and the wilcoxon-mann-whitney test. *Journal of Statistical Planning and Inference*, 79(2):349–364, July 1999.

[7] Edmund Burke, Emma Hart, Graham Kendall, JimNewall, Peter Ross, and Sonia Schulenburg. Hyper-Heuristics: An Emerging Direction In Modern Search Technology, 2003.

[8] S. Cahon, N. Melab, and E.-G. Talbi. ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. *Journal of Heuristics*, 10(3):357–380, 2004.

[9] Tatsuhiro Chiba and Tamiya Onodera. Workload characterization and optimization of tpc-h queries on apache spark. In *ISPASS*, pages 112–121. IEEE Computer Society, 2016.

[10] Corinna Cortes, Spencer Greenberg, and Mehryar Mohri. Relative deviation learning bounds and generalization with unbounded loss functions. *CoRR*, abs/1310.5796, 2013.

[11] Robin De Croon, Joris Klerkx, and Erik Duval. Design and evaluation of an interactive proof-of-concept dashboard for general practitioners. In Prabhakaran Balakrishnan, Jaideep Srivatsava, Wai-Tat Fu, Sanda M. Harabagiu, and Fei Wang, editors, *ICHI*, pages 150–159, 2015.

[12] Ebru Demirkol, Sanjay Mehta, and Reha Uzsoy. Benchmarks for shop scheduling problems. *European Journal of Operational Research*, 109(1):137–141, 1998.

[13] M. Dorigo and T. Stützle. Ant Colony Optimization: Overview and Recent Advances. In *Handbook of Metaheuristics*, volume 146 of *Inter. Ser. in Operations Research and Management Science*, pages 227–263. Springer, 2010.

[14] J. J. Durillo and A. J. Nebro. jMetal: A Java Framework for Multi-objective Optimization. *Adv. Eng. Softw.*, 42(10):760–771, 2011.

[15] Raul Estrada and Isaac Ruiz. *Big data SMACK – A Guide to Apache Spark, Mesos, Akka, Cassandra, and Kafka*. Apress, Berkeley, CA,; New York, 2016.

[16] P. Fazenda, J. McDermott, and U. O'Reilly. A Library to Run Evolutionary Algorithms in the Cloud Using MapReduce. In *Proc. of EvoApplications'12*, pages 416–425, 2012.

[17] Filipe Costa Fernandes, SÃl'rgio Ricardo de Souza, Maria AmÃl'lia Lopes Silva, Henrique Elias Borges, and Fabio Fernandes Ribeiro. A multiagent architecture for solving combinatorial optimization problems through metaheuristics. In *SMC*, pages 3071–3076. IEEE, 2009.

[18] F. Ferrucci, M. T. Kechadi, P. Salza, and F. Sarro. A Framework for Genetic Algorithms Based on Hadoop. *CoRR*, abs/1312.0086, 2013.

[19] Gordon Fraser and Jerffeson Teixeira de Souza. Guest editorial: Search-based software engineering. *Empirical Software Engineering*, 19(5):1421–1422, 2014.

[20] M. R. Garey, D. S. Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Math. Oper. Res.*, 1(2):117–129, 1976.

[21] Walter J. Gutjahr and Mark Harman. Search-based software engineering. *Computers & OR*, 35(10):3049–3051, 2008.

[22] Mark Harman and Francisco Chicano. Search based software engineering (sbse). *Journal of Systems and Software*, 103:266, 2015.

[23] Mark Harman and S. Afshin Mansouri. Search based software engineering: Introduction to the special issue of the ieee transactions on software engineering. *IEEE Trans. Software Eng.*, 36(6):737–741, 2010.

[24] Abdul Salam Kalaji. *Search-based software engineering : a search-based approach for testing from extended finite state machine (EFSM) models.* PhD thesis, Brunel University London, UK, 2010.

[25] M. Kronfeld, H. Planatscher, and A. Zell. The EvA2 Optimization Framework. In *Proc. of the 4th International Conference on Learning and Intelligent Optimization*, LION'10, pages 247–250. Springer, 2010.

[26] Young Choon Lee, Javid Taheri, and Albert Y. Zomaya. A parallel metaheuristic framework based on harmony search for scheduling in distributed computing systems. *Int. J. Found. Comput. Sci.*, 23(2):445–464, 2012.

[27] Thilo Mende. *On the evaluation of defect prediction models*. PhD thesis, Uni Bremen, 2011.

[28] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ay?e Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[29] M. Milano and A. Roli. Magma: A multiagent architecture for metaheuristics. *Trans. Sys. Man Cyber. Part B*, 34(2):925–941, April 2004.

[30] M. Nawaz, E. E. Enscore, and I. Ham. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11(1):91 – 95, 1983.

[31] E. Nowicki and C. Smutnicki. A fast tabu search algorithm for the permutation flow-shop problem. *European Journal of Operational Research*, 91(1):160–175, 1996.

[32] I. Osman and C. Potts. Simulated Annealing for Permutation Flow-Shop Scheduling. *Omega*, 17(6):551–557, 1989.

[33] J. Parejo, A. Ruiz-Cortes, S. Lozano, and P. Fernandez. Metaheuristic Optimization Frameworks: A Survey and Benchmarking. *Soft Comput.*, 16(3):527–561, 2012.

[34] A. Radenski. Distributed Simulated Annealing with Mapreduce. In *Proc. of the European Conference on Applications of Evolutionary Computation*, EvoApplications'12, pages 466–476, 2012.

[35] C. Rajendran and H. Ziegler. Ant-colony algorithms for permutation flowshop scheduling to minimize makespan/total flowtime of jobs. *European Journal of Operational Research*, 155(2):426 – 438, 2004.

[36] C. R. Reeves. A Genetic Algorithm for Flowshop Sequencing. *Comput. Oper. Res.*, 22:5–13, 1995.

[37] R. Ruiz and T. Stützle. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 177(3):2033 – 2049, 2007.

[38] T. Stützle. An Ant Approach to the Flow Shop Problem. In *Proc. of the 6th European congress on Intelligent Techniques and Soft Computing (EUFIT98)*, pages 1560–1564. Springer, 1997.

[39] E. Taillard. Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research*, 47(1):65 – 74, 1990.

[40] Taillard's Instances for Flow Shop Scheduling. http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/ordonnancement.html.

[41] E. Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.

[42] Tommaso Urli. Hybrid meta-heuristics for combinatorial optimization. *Constraints*, 20(4):473, 2015.

[43] E. Vallada and R. Ruiz. Cooperative metaheuristics for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 193(2):365 – 376, 2009.

[44] A. Verma, X. Llorà, D. E. Goldberg, and R. H. Campbell. Scaling Genetic Algorithms Using MapReduce. In *Proc. of ISDA '09*, pages 13–18, 2009.

[45] Frank Werner. On the combinatorial structure of the permutation flow shop problem. *ZOR - Meth. & Mod. of OR*, 35(4):273–289, 1991.

[46] L. Xu and E. Oja. Improved Simulated Annealing, Boltzmann Machine, and Attributed Graph Matching. In *Proc. of the EURASIP Workshop 1990 on Neural Networks*, pages 151–160, 1990.

[47] 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, and I. Stoica. Apache Spark: A unified engine for big data processing. *Comm. of the ACM*, 59(11):56–65, 2016.

[48] D.-Z. Zheng and L. Wang. An Effective Hybrid Heuristic for Flow Shop Scheduling. *The International Journal of Advanced Manufacturing Technology*, 21(1):38–44, 2003.