HyperSpark: A Software Engineering Approach to Parallel Metaheuristics

Michele Ciavotta[1], Srđan Krstić[2], Damian A. Tamburri[3]

[1] University of Applied Sciences of Southern Switzerland
Via Galleria 2, 6928 Manno, Switzerland
michele.ciavotta@supsi.ch

[2] ETH Zurich
Universitätstrasse 6, 8006 Zürich, Switzerland
srdan.krstic@inf.ethz.ch

[3] Politecnico di Milano
Piazza Leonardo da Vinci, 32, 20133 Milano, Italy
damianandrew.tamburri@polimi.it

Abstract

Metaheuristics are used to solve complex, untractable 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, the domain of metaheuristics has not yet taken full advantage of this new potential. In this paper we address this gap by proposing HyperSpark, a metaheuristic optimization framework for the scalable execution of user-defined, computationally-intensive metaheuristics. HyperSpark provides a way 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 metaheuristic computation. We elaborate on HyperSpark and evaluate its efficiency and generality on several different metaheuristics for the Permutation Flow-Shop Problem (PFSP). We observe that HyperSpark results are comparable with the best solutions of the literature and this shows clearly the great potential behind the propsed approach.

1 Introduction

The word *Metaheuristic* (from ancient Greek *meta* = "beyond, higher-level" and *heuriskein* = "to find") defines a class of algorithms able to find near-optimal solutions for *hard* optimization problems by working on an abstract level [29] . 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, suitable for a wider set of problems. Since metaheuristics do not rely on problem specifics and know-how, they are flexible and easily adaptable to many different problems, and subsequently require less time to design and implement. 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 wherefore no satisfactory heuristic is known. Literature shows plenty of solutions that seek a substantial speed up by trading generality for performance via hybridization with heuristics and local search based techniques. Alternatively, one can still improve performance without trading off generality by means of *parallelization*. Modern parallel and distributed computation clusters offer a powerful way to increase efficiency and effectiveness. Hence, *parallel* metaheuristics have been a focus of extensive research [3].

In the last three decades many different metaheuristics have been proposed; these can be broadly classified into trajectory-based and population-based metaheuristics [6], along with many orthogonal variations defined by specific solution encoding, neighborhood structure definition, operators, and more. To achieve further generality, streamline the creation of new metaheuristics, and organize the knowledge acquired over the years some metaheuristic optimization frameworks (MOFs) have emerged. A MOF is an abstraction that provides a diverse set of reusable components, and the basic mechanisms to selectively change them with user-written code, thus adapting them to specific optimization problems. In [19] the authors survey and systematically compare the existing MOFs. The major drawbacks they identified can

be summarized as follows: (a) MOFs have very limited support for parallel and distributed execution; (b) MOFs lack support for hyperheuristics; and (c) MOFs are not designed according to known software engineering best practices.

From the literature overview, one thing is clear: parallel metaheuristic research has reached a point of rather narrowed, perhaps even "siloed" view whereby the proposed approaches are designed to handle specific problems, using specific metaheuristics implemented in specific, often closed-source technology. This trend makes the reproducibility of existing approaches as well as the rigorous comparison of different solutions difficult if not impossible. Also, we noticed that in response, metaheuristics research community often reimplements existing solutions only to be able to extend them or compare them to novel approaches. We argue that this hinders future research in the field, even more as the number of proposed algorithms increases.

In this paper we address these drawbacks by proposing *HyperSpark*, a framework that supports design of parallel metaheuristics and their execution on a cluster of distributed and interconnected computational nodes. HyperSpark is implemented in Scala and harnesses the features of Apache Spark, the reference technology for Big Data processing. Moreover, the following design principles were borrowed from the field of Software Engineering to design our research solution: (1) *Ease-of-use* - the framework handles distribution and parallelization transparently; (2) *Configurability* - the framework exposes configuration parameters to fine tune the execution and parallelization (e.g., number of execution nodes, number of used cores per node, etc); (3) *Flexibility* - the framework exposes a new programming model that allows users to define arbitrary parallelization strategies, run different metaheuristics at once and define how they can be combined; (4) *Cooperation* - framework allows for synchronous communication among parallel instances of the algorithm and therefore supports a large class of cooperative parallel metaheuristic algorithms; (5) *Extensibility* - the framework is developed in a object-oriented and functional programming language (*Scala*) with mechanisms (e.g., inheritance, traits, implicits and late binding) that facilitate the adoption of generic metaheuristic algorithms to specific problems, extensible design and code reuse. (6) *Portability* - the framework inherits the portability of Apache Spark and can run on any JVM-enabled architecture. Features (1) and (2) contribute towards parallel and distributed computing support for metaheuristics, while (3) and (4) provide abstractions for designing hyperheuristics using existing algorithms. Finally, features (5) and (6) contribute towards adopting good software engineering practices in metaheuristics design.

The reminder of this paper is structured as follows. Section 2 introduces HyperSpark while Section 3 outlines basic implementation details and examples. In Section 4 we experiment with the widely known *permutation flow-shop problem* (PFSP) [7] for preliminary evaluation purposes. We discuss results and compare HyperSpark with existing MOFs in Section 5. Finally, Section 6 concludes the paper.

2 HyperSpark Overview

This section outlines the features of our research solution structured according to a base architecture (see Section 2.1), its intended programming model (see Section 2.2), as well as the runtime model (see Sec. 2.3) that underpins the execution of HyperSpark programs. More specifically, we first elaborate the base architecture that allows the creation of arbitrary parallel algorithms. We also provide guidelines on how to extend the base architecture with the custom user-defined code. Following on, we introduce the flexible HyperSpark programming model and elaborate on the properties and features that allow users to write high-level parallel metaheuristic algorithms in a simple and straightforward fashion. Moreover, we provide an intuition on how Apache Spark maps a high-level parallel metaheuristic algorithm specified using the HyperSpark programming model to an independent set of processes in order to effectively parallelize the execution.

2.1 Base Architecture: Problem – Algorithm – Solution

The primary goal of our base architecture is to provide a programming model general enough to accommodate various representations of *Problem* instances solved by means of different kind of metaheuristic

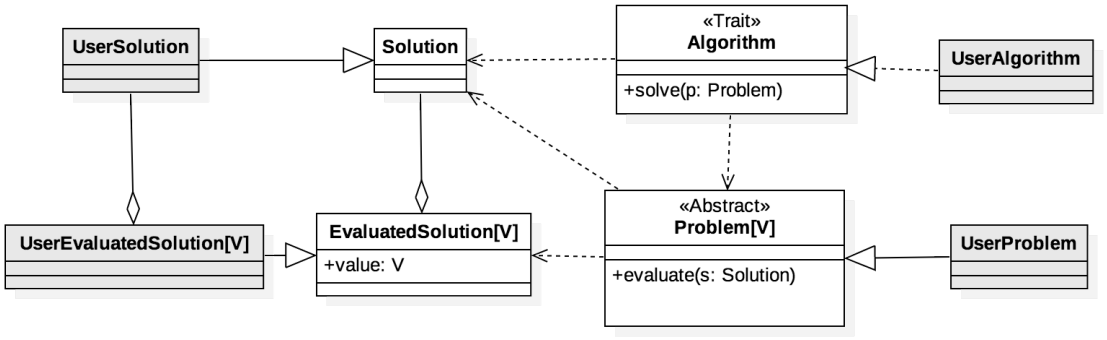

Figure 1 – Class diagram of HyperSpark base classes and traits.

Algorithms that produce sets of *Solutions* with some arbitrary encoding. We draw inspirations from many previous attempts [16] to design this generic scheme, however, we choose to simplify and omit details in order to provide higher flexibility.

The core architectural entities of HyperSpark are showed colored white in Figure 1. The architecture is simple and consists of a single Scala trait[1] and three classes that capture the main concepts enforced by HyperSpark. The *Algorithm* trait represents a generic algorithm: 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 of a problem representation, metaheuristic algorithm, and solution encoding. Users are able to implement custom algorithms by mixing[2] in the *Algorithm* trait and overriding their *solve* method. Entities colored grey in Figure 1 exemplify how users can extend framework's functionality. In order to introduce a new algorithm represented by the the class *UserAlgorithm* one needs to mix in the *Algorithm* trait. Similarly, user-defined problems and solutions need to extend their respective base classes.

In HyperSpark core 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"*. This 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.

2.2 Programming Model

As previously seen, the first step for the developer interested in creating a parallel and distributed algorithm is to provide suitable implementation for the core elements of HyperSpark, that is providing the appropriate problem representation, solution encoding and at least one (non parallel) algorithm. Without lack of generality we refer to the simplest case of one single algorithm; nonetheless, HyperSpark is able to handle seamlessly the cooperation of multiple different algorithms. This mode of operation is especially useful when implementing hyperheuristics.

The following step consists in extending and instantiating a HyperSpark execution workflow. Figure 2 illustrates the base workflow provided 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 (solution space and objective functions), users can optionally split it into different sub-problems. This means, for example, that the user can parallelize the algorithm and assign to each parallel *instance* a different region of the solution space to explore, or a different objective function

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

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

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.

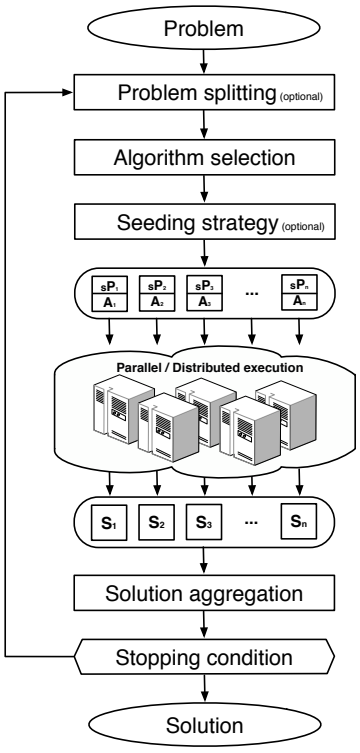

Figure 2 – Workflow of the HyperSpark.

Assuming that the problem is indeed split, for each (sub-) problem the user may specify an algorithm to solve it. There are no constrains 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 - this strategy 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. By way of example, at each iteration the user can be interested in preserve the best solution and use 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 algorithm complete their execution to collect the solutions generated and combine suitably. Our programming model allows the user to easily implement an aggregation function that combines solutions from different algorithms. In case the user is not interested in particular re-combination strategies, the framework provides by design a simple aggregation function that returns the solution with the minimal value obtained from the evaluation 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 simply 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 parallel and distributed execution of the selected algorithms.

2.3 Runtime Model

As previously stated, the HyperSpark architecture relies on 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 set of processes (called *executors* in Spark lingo) distributed on a cluster and coordinated by a main 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. 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 acquires executors to run individual algorithms, sends 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 and further details are available on the Apache Spark homesite[3].

[3] `http://spark.apache.org/`

3 Implementation and Examples

In this section we exemplify the use of HyperSpark and provide some insights into its implementation. Suppose we implemented a genetic algorithm from [22] for the PFSP and we wish to run 4 instances of the algorithm in parallel. This scenario can be realized in HyperSpark as follows:

```
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)
```

The first line instantiates an object of the *PFSProblem* class, which is our custom representation PFSP. This is done by means of a factory method *fromResources* that reads the parameters of the problem from a file. Next, we create a HyperSpark configuration object called *FrameworkConf* that exposes the main API of our framework. 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 will use a reasonable default value. The execution will not start until we execute the run method passing the configuration object, as shown in line 7. Lines 3–6 constitute a minimalistic example of the use of our programming model and they define a simple high-level execution workflow. In line 3 we specify that all the algorithms will solve the same problem instance. Line 4 specifies that we will execute one algorithm implemented by the *solve* method of class *GAAlgorithm* four times in parallel with no cooperation. We set the framework to stop after 100 seconds of execution in line 5. The *setDeployment* method is used to specify the modality of deployment and the number of parallel execution processes. In this case we use the `local` modality that creates 4 processes on the local machine to execute the algorithms. Other modalities include `spark` for using Spark standalone cluster manager, `mesos` for using Mesos [13] cluster manager and `yarn-client` or `yarn-cluster` for using YARN [31] cluster manager with the driver program running either on the client or on the cluster respectively.

Notice that we specify the number of parallel algorithm instances (line 4) separately from the number of executors (line 6). In this particular example each algorithm is be executed on a single executor by design. However, we may have specified a larger number of executors, with some of them remaining idle during the execution. Conversely, if we have specified a smaller number of executors, some instances would have been executed on the same executor, at the discretion of Spark internal execution management mechanisms. Moreover, depending on the number of cores assigned to the executor such instances of the algorithm would be executed either sequentially (in the case we set a number of instances greater than the number of cores) or in parallel (in any other case). The number of cores assigned to each executor is specified within the Spark configuration.

In the next example we show the execution of algorithms that cooperate synchronously, as well as exemplify the fine-tuning of Spark-specific configurations. The following code runs 100 instances of the cooperative hybrid genetic algorithm from [35] implemented in the *HGAlgorithm* class in parallel on the same problem as in the previous example. The cooperation is facilitated by adopting an appropriate seeding strategy and setting a number of stages greater that 1 (line 7). On the one hand, the seeding strategy provides a function that generates a seed solution for each instance at stage n altering the results of the

```
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).toInt))
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)
```

TABLE 1
LIST OF ALGORITHMS IMPLEMENTED IN HYPERSPARK-PFSP LIBRARY

Algorithm	Authors	Year	Ref.	Name
NEH	Nawaz, Enscore and Ham	1983	[15]	*NEH*
Iterated Greedy	Ruiz and Stützle	2007	[23]	*IG*
Genetic Algorithm	Reeves	1995	[22]	*GA*
Hybrid Genetic Algorithm	Zheng and Wang	2003	[35]	*HG*
Simulated Annealing	Osman and Potts's addaption for PFSP	1989	[18]	*SA*
Improved Simulated Annealing	Xu and Oja	1990	[33]	*ISA*
Taboo Search	Taillard	1990	[27]	*TS*
Taboo Search with backjump tracking	Novicki and Smutnicki	1996	[17]	*TSAB*
Ant Colony Optimization	Dorigo and Stützle	2010	[9]	*ACO*
Max Min Ant System	Stützle	1997	[26]	*MMAS*
mMMAS	Rajendran and Ziegler	2004	[21]	*MMMAS*
PACO	Rajendran and Ziegler	2004	[21]	*PACO*

solution aggregation phase at stage $n-1$. In this particular case the solution aggregation return the minimal makespan solution found during a stage whereas the seeding strategy, named `SlidingWindow`, operates selecting w contiguous 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 generate for each instance of the algorithm.

In line 7 we determine the number of stages in the workflow 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, we deploy the executors using the Spark standalone cluster manger (line 8); we modify the number of cores (line 9) and amount of memory in gigabytes (line 10) that we request from the cluster manager to allocate for each executor.

4 Evaluation

The aim of this Section is to present the experiments carried out to evaluate HyperSpark and discuss the achieved results. In the scope of our evaluation, we set out to address two research questions, namely, (1) *is the overhead introduced by HyperSpark acceptable in the context of parallel cooperative optimization?* And (2) *are the algorithms implemented using HyperSpark competitive with respect to the state-of-the-art?*

In response to these questions, we carried out two different experiments, both rotating around the application of HyperSpark in addressing the Permutation Flow Shop Problem (PFSP) problem, a well-known $\mathcal{NP}$-Complete optimization problem. This problem is consistent with our analysis since it comes with the well-known Taillard's benchmark [28], 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 case study is the fact that, for each problem instance, the optimal (or the best) makespan is known and freely available. In layman's terms, 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 is associated with a fixed, non-negative, and known in advance processing time for each machine. This is denoted p_{ij}, for each $j \in J$ and $i \in M$. Furthermore, at any 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. The aim of this problem is to find a particular sequence that optimize a certain 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. Makespan minimization is directly related with the maximization of machine utilization and reduction of the work-in-progress. Following the well known Graham classification scheme, PFSP is $F/prmu/C_{max}$. For the purposes of the evaluation we implemented a library of algorithms (listed in Table 1) published in 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[4]. All the algorithms are single-threaded and coded in Scala without any speed up, sharing/reusing as much as code as possible, i.e., following the software engineering principles of modularization and code-reuse. Furthermore, in both experiments the algorithms are stopped after $n \cdot m/2 \cdot 60$ milliseconds of CPU time, as in [30]. In this way, more time is assigned to larger and harder-to-solve instances.

Finally, 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 in total available for the computations. The Apache Spark environment (version 1.5) was installed in "Standalone" mode, meaning that it manages both application scheduling and provisioning of hardware resources on its own, without any separate optimization or resource manager. This mode of operation has proven to greatly hamper the performance [24, 34] and actually allowed us to evaluate our research solution harnessing the baseline and improvement-free performance that HyperSpark is able to deliver. Conversely, in future work we plan to experiment with HyperSpark assessing possible performance boosts using specific resource managers for Apache Spark.

4.1 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 (parallelization, ω_P) and context termination (ω_T) with respect to the cluster and instance size. Therefore, we set up the following experiment. We increased the number of cores available as much as possible linearly. More precisely we consider configurations with 1, 8, 16, 32, 40 cores respectively. Then, five Taillard's instances of size 20, 50, 100, 200 and 500 jobs have been randomly chosen; each one of them is solved 5 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 have no influence on the overhead.

Table 2 reports an excerpt from the data harvested in the experiment. Based on them, we can observe that the overhead due to parallelism and synchronization ω_P depends on either the cluster and instance size going from 3 to 47 seconds. Initialization hinges on the cluster size only; however, the growth is quite limited, remaining in the range 5-13 seconds. Termination overhead, instead is constant and less than 1 second. Moreover, by increasing the problem size, we noticed that the impact of the total overhead in percentage ($\sum \omega$), which shows quite high values (up to 70%) on small instances, accounts only for 7-14% for the instance with 500 jobs. All previous observations lead to the conclusion that the proposed solution, which shows an excessive overhead on small problems, can be, instead, considered suitable to large optimization problems, that is, those that would benefit the most from HyperSpark parallelism.

4.2 Experiment 2 – Solution Quality Evaluation

To evaluate the capability to achieve state-of-the-art results we set up an experimental campaign on a cluster with 20 cores. All 120 available 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) with respect to the best known solution is calculated. We select IG and HG among other algorithms implemented in the library as they resulted the most performing ones in preliminary tests. Three different seeding strategies have been considered; since their description is out of the scope of this paper we report only their acronyms: SS, SPSW and SPFW. The interested reader is referred to [25] for further details. In Table 3 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 know solutions for the PFSP. This is a very good outcome considering the early stage of our HyperSpark prototype an the fact that the considered algorithms have been implemented, as previously outlined, without any particular optimization. In conclusion, we can state that the parallelization-synchronization

[4] `https://github.com/deib-polimi/hyperspark`

TABLE 2
TIME OVERHEAD ANALYSIS.

Inst.	# 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	11.5	0.81	14.5

TABLE 3
AVERAGE RPD FROM BEST KNOWN SOLUTION FOR METHODS (IG, HG) PARALLELIZED WITH SS, SPSW AND SPFW STRATEGIES

		RPD (%)					
$\|J\|$	$\|M\|$	**HG** SS	**IG** SS	**HG** $SPSW$	**IG** $SPSW$	**HG** $SPFW$	**IG** $SPFW$
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

approach implemented in HyperSpark is promising and can lead, even in the prototypical form outlined in this article, to results which are comparable to the state-of-the-art.

5 Related work

The framework presented in this paper is related and inspired by other works that can be roughly classified into two groups: parallel MOFs and the use of BigData for metaheuristics.

The first group corresponds to the current state-of-the-art MOFs that support parallel execution: ECJ [1], ParadisEO [8], EvA2 [14], MALLBA [2], and jMetal [10]. However, none of the MOFs above exposes a programming model to support the design of hyperheuristics. In addition, JMetal does not support distributed execution, while the rest of the MOFs provide limited flexibility in parallel metaheuristic design. More specifically, they do not support a class of metaheuristics that performs local search using parallel and distributed neighbourhood exploration [19]. The second group consists of a large body of work the details of which are beyond the scope of this paper. They use different BigData technologies to implement particular algorithms [4, 20, 32] or a particular class of algorithms [5, 11, 12].

A work that shares some analogies with our framework (as MOF over Spark) is presented in [20]. However, the authors extend the jMetal [10] framework only to accommodate the streaming feature of Apache Spark without fully exploiting the parallel and distributed execution capabilities of Spark.

6 Conclusions

Research in optimization is an important field aiming at tackle hard problems often using sub-optimal algorithm as metaheuristics. There were many attempts to reconcile diversity and complexity of different approaches within many metaheuristic optimization frameworks (MOFs). However, 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. In this paper we outline, evaluate, and discuss *HyperSpark*, a framework 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. We realized a preliminary experimental evaluation to validate the approach. We can safely state that, despite some limitations mainly due to its prototype nature, HyperSpark has shown a great potential 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 HyperSpark with more mature MOFs such as jMetal[5] [10]. Finally, we are planning to investigate HyperSpark extensions that facilitate asynchronous communication for better cooperative optimization.

Acknowledgement

This work has been partially supported by the Swiss NSF under the grant agreement no. 407540_167162 (BigData); EC grant

[5] `http://jmetal.github.io/jMetal/`

no. 644869 (EU H2020), DICE. The authors wish to thank N. Stolić and C. Martínez for their efforts in implementing and evaluating HyperSpark.

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] E. Alba, G. Luque, and S. Nesmachnow. Parallel metaheuristics: recent advances and new trends. *International Transactions in Operational Research*, 20(1):1–48, 2013.

[4] 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.

[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] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3):268–308, 2003.

[7] W. Bozejko. Solving the flow shop problem by parallel programming. *J. Par. Dist. Comp.*, 69(5): 470–481, 2009.

[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] 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.

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

[11] 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.

[12] F. Ferrucci, M. T. Kechadi, P. Salza, and F. Sarro. A Framework for Genetic Algorithms Based on Hadoop. *CoRR*, abs/1312.0086, 2013. URL `http://arxiv.org/abs/1312.0086`.

[13] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proc. of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308. USENIX, 2011.

[14] 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.

[15] 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.

[16] A. J. Nebro, J. J. Durillo, and M. Vergne. Redesigning the jMetal Multi-Objective Optimization Framework. In *Proc. of GECCO Companion '15*, pages 1093–1100, 2015.

[17] 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.

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

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

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

[21] 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.

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

[23] 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.

[24] S. Salehian and Y Yan. Comparison of Spark Resource Managers and Distributed File Systems. In *BDCloud-SocialCom-SustainCom*, pages 567–572. IEEE, 2016.

[25] N. Stolić. HyperSpark: a framework for scalable execution of computationally intensive algorithms over Spark. Master's thesis, Politecnico di Milano, April 2016.

[26] 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.

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

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

[29] E. Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009. ISBN 0470278587, 9780470278581.

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

[31] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proc. of SOCC*, pages 1–16, 2013.

[32] 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.

[33] 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.

[34] 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. *Communications of the ACM*, 59(11):56–65, 2016.

[35] 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. ISSN 0268-3768.