POLITECNICO DI MILANO

Department of Electronics, Informatics and Bioengineering
M.Sc. course Engineering of Computing Systems

HyperSpark : A Framework for Scalable Execution of Computationally-Intensive Algorithms over Spark

Mentor:
Prof. Danilo Ardagna — Politecnico di Milano
Co-mentors:
Dr. Michele Ciavotta — Politecnico di Milano
Dr. Srđan Krstić — Politecnico di Milano

Master of Science Thesis of:
Nemanja Stolić
Matriculation ID 814842

Academic year 2015-2016

To my wonderful family

Contents

List of Figures

List of Tables

Abstract

Nowadays computing platforms are available to many companies and individuals wanting to process vast amounts of data sets. With the term *Big Data* we refer to the analysis of huge datasets, allowing the extraction of information of utmost importance for business purposes. The focus of Big Data community has turned to providing high performance over the cluster of computing resources that was previously used only for data-intensive computations. A newly developed technology, *Apache Spark*, with its characteristic property of executing computation and data-processing in main memory, unconsciously allowed users to execute computationally-intensive algorithms in a distributed fashion. For solution quality considerations, a common approach entails instantiating multiple parallel user algorithms, as long as the computational time overhead does not surpass the allowed time limit. Such a common technique should provide every user a way to build and run custom algorithms in parallel, at the same time ensuring fault tolerance and out of the box scalability.

In this work we describe *HyperSpark* - a general, extensible and portable framework for scalable execution of user-defined, computationally-intensive algorithms over the cluster of commodity hardware. Our goal is the utilisation of computing resources of the whole cluster in order to deliver high-performance (i.e., solution quality) in a limited amount of computational time. The parallel algorithm execution, fault-tolerance, data distribution, cooperation between algorithms and results aggregation are provided by the underlying technology and the framework in a very transparent way to the user. The ground case for the conducted experiments is Permutation Flow Shop Problem (PFSP), an extremely hard optimisation problem from scheduling theory that can not be solved optimally with conventional methods when limited time is provided. Together with the framework we provide a library for distributed solving of PFSP, that contains many efficient approximate algorithms, known as (meta-)heuristics. The preliminary tests show the encouraging results. Further improvements could be gained by taking into account an asynchronous communication between algorithms, which can greatly decrease the computational time overhead caused by the introduction of unnecessary synchronisation points.

Keywords: parallel and distributed meta-heuristics, optimisation framework, distributed computation, Permutation Flow Shop problem, combinatorial optimisation, scheduling, local search, Scala, Spark, Big Data.

Sommario

Al giorno d'oggi molte compagnie e singoli professionisti hanno a disposizione piattaforme di calcolo per il processamento di grandi quantità di dati. Con il termine *Big Data* si intende l'insieme di processi e strumenti che permettono l'analisi di basi di dati di grandissime dimensioni; l'obiettivo finale consiste nell'estrazione di informazioni di grande importanza (business, sicurezza, ecc.). L'interesse della comunità di Big Data si è concentrato negli ultimi anni nell'offrire agli utenti piattaforme non sono dedicate all'analisi di date ma che forniscano allo stesso tempo alte prestazioni di calcolo. Tra le nuovo proposte tecnologiche in quest'ambito, Apache Spark è una delle più promettanti; questo strumento permette agli utenti di usufruire un paradigma di programmazione più completo e di un set di strutture distribuite efficienti, tale da poter permettere la realizzazione e l'esecuzione di applicazioni distribuite sia per l'analisi di dati sia per il calcolo distribuito. Per quanto riguarda il calcolo distribuito, in questa tesi abbiamo utilizzato Spark per realizzare un framework in cui molti algoritmi di ottimizzazione (*CPU intensive*) possano essere eseguiti in parallelo e sincronizzati per poter affrontare con successo problemi di ottimizzazione complessi. Spark permette in maniera trasparente il deployment degli esecutori e la loro comunicazione, inoltre siccome per la particolare categoria di algoritmi considerati i dati da gestire sono di piccole dimensioni e possono essere distribuiti con il codice, Spark permette anche una scalabilità praticamente illimitata degli algorimi. L'altra faccia della medagglia è sicuramente l'altro costo, in termini di tempo, che è richiesto dal framework per poter garantire i suoi servizi.

In questa tesi, presentiamo *HyperSpark*, un framework generale ed estendibile che permette di realizzare ed eseguire algorimi paralleli ed estremamente scalabili su cluster di hardware non specializzato. Il framwork è stato pensato, in particolare, per permettere l'esecuzione parallela e la comunizazione all'interno di un'applicazione distribuita per l'ottimizzazione. L'obiettivo è quello di realizzare uno strumento flessibile ed indipendente dal harwdare a disposizione e dalla sua configurazione per realizzare applicazioni distribuite ad alte prestazioni. HyperSpark si basa su Spark per la (trasparente all'utente) distri-

buzione del codice eseguibile tra i nodi, per la relativa esecuzione, la distribuzione dei dati, la fault-tolerance, la cooperazione fra i nodi e l'aggregazione dei risultati. HyperSpark è stato quindi utilizzato per realizzare una libreria di algoritmi euristici e metaeuristici dedicati all'ottimizzazione del problema di Permutation Flow Shop (PFSP), un problema $\mathcal{NP}$-hard per il quale un approccio esatto non è possibile se non per casi molto piccoli. All'interno di tale libreria sono presenti alcuni dei più conosciuti algoritmi per il PFSP.

Una campagna preliminare di test è stata realizzata allo scopo di verificare la validità dell'approccio incarnato in HyperSpark sia in termini di overhead che come strumento per realizzare algoritmi competitivi rispetto allo stato dell'arte. I risultati di tali esperimenti sono incoraggianti sebbene il paradigma basato sulla sincronizzazione, imposto dalla attuale versione di Spark è evidente fonte di overhead. Recentemente però, sono apparse in letteratura delle soluzioni che permettono di implementare un livello di comunicazione asincrona tra gli esecutori in Spark. HyperSpark potrebbe in futuro adottare un approccio simile riducendo il punti di sincronizzazione al fine di ottenere un miglioramento globale delle prestazioni.

Parole chiave: meta-euristiche parallele, algoritmi distribuiti, Permutation Flow Shop, ottimizzazione combinatoria, programmazione, ricerche locali, Scala, Spark, Big Data.

CHAPTER 1

Introduction

The approach to problem solving has changed during the last 30 years, together with the evolution of software technologies. In fact, new emerging technologies have influenced greatly on the way the problems are addressed. For extremely hard optimisation problems more and more computing resources are introduced, together with the emerging need to have an overlay software management tool for running user algorithms and analysing the data.

Until 1980s, the exact algorithms were the most commonly used methods to solve the optimisation problems. Exact algorithms always solve an optimisation problem to optimality, but sometimes are not applicable because they do not scale with the size of the problem. Some exact algorithms perform an exhaustive search over the solution space (set of all possible solutions), and with a limited amount of computing resources, they may require weeks, months or years of computation [31]. That led to creation of heuristics - approximate algorithms that efficiently explore the solution space in order to find the near-optimal solution in a bounded time. Although they may solve one optimisation problem efficiently, when employed on another problem of the same type they would yield poor results. In order to solve this drawback, meta-heuristics (or meta-heuristic algorithms) appeared as a significant advance [21]; they are problem-agnostic algorithms that can be adapted to incorporate the problem-specific knowledge.

Although the use of meta-heuristics allows to significantly reduce the computational complexity of the solution retrieval process, they are still time consuming for many problems in diverse domains of application. This can be due to the complex objective functions or constraints associated with the problem or perhaps the large size of the solution space. Additionally, more and more complex and resource-intensive meta-heuristics are developed even for simple

problems in order to obtain high-fidelity solutions. The introduction of multi-core architectures paved the way for research in parallel meta-heuristics, which take advantage of parallelism to reduce the computation time and increase the solution quality. Increasing memory capacity or CPU power of a single machine became a trend that lasted for a couple of years. This practice created a new paradigm, High Performance Computing (HPC), that aimed to deliver higher performance than once could possibly get out of a typical desktop computer or workstation in order to solve large problems in mathematics, physics, genetics... As such, HPC was specifically convenient for CPU-intensive applications. However, this approach was quickly abandoned because the increase of computing power did not scale linearly with the cost and solution quality.

In 2006, Apache Hadoop [23] emerged as the first open source framework for large-scale parallel/distributed computation using a cluster of interconnected machines (or nodes). Using Hadoop is much more cost effective than the HPC solution because cluster nodes can be simple commodity hardware. Adding more computing nodes to the cluster does not only increase the quality of solutions [71], but it might also decrease the total computational time, sometimes enough to produce real time business decisions. Hadoop became a base for many commercial and scientific applications, and until 2014th it was the only fault-tolerant platform that supports large-scale data-intensive computations. Despite its success in the industry, for complex problems Hadoop is still inconvenient, because it uses many input/output disk operations during the computation. Apache Spark [3] alleviates this issue by providing support for in-memory computations and opens the possibility to execute both data-intensive and CPU-intensive applications, at the same time considering reliability, availability and fault-tolerance.

The evolution of CPU-intensive and data-intensive computations brought to a convergence in the same point - data-intensive computing platforms started being used for CPU-intensive computations and HPC systems started executing more parallel data-intensive tasks. This technological trend has fostered a new computing paradigm called *Big Calculations*, which tries to exploit the current data-intensive platforms (not only a single powerful machine) for high-performance decision making [6]. The goal of this new paradigm is to obtain fast and efficient results using the knowledge of time provided and computing resources at our disposal.

Problem statement

The ideas of Big Calculations paradigm motivated us to explore the potential of large-scale data-intensive computing platforms and their suitability for CPU-intensive applications. According to a recent survey on meta-heuristics frame-

works [45], most of them do not support the execution of parallel and distributed meta-heuristics. The study identified only the following parallel (and distributed) meta-heuristic frameworks: ECJ [16], ParadisEO [44], EvA2 [33], and MALLBA [1]. The main drawbacks of the above-mentioned frameworks are that (1) they assume deep knowledge of the underlying parallelisation technology, they are (2) not designed to be extensible in order to accommodate different applications. Furthermore, the engineering issues of (3) portability and (4) code reuse have not been properly addressed in any of the frameworks. Finally, none of the frameworks allow the user to (5) define particular parallelisation strategy nor (6) the particular aggregation of results of the parallel algorithm instances. This thesis intends to address these drawbacks, as stated with the following overall research goal:

> *"To study the feasibility and challenges of using a distributed computing platform for CPU-intensive computations and to develop a framework for parallel and distributed execution of meta-heuristic algorithms."*

Contributions

This thesis presents *HyperSpark*, a general, extensible and portable framework which enables scalable execution of CPU-intensive algorithms over a cluster of commodity hardware. HyperSpark is implemented in Scala and uses Spark to enable the execution of parallel meta-heuristics to solve custom-defined problems. The framework is designed following the well-known *convention over configuration* approach, therefore its setup is almost minimal. The underlying platform and the developed framework transparently manage the scaling of algorithms over the cluster, distribution of data, execution flow, cooperation between algorithms, aggregation of results. The users can develop their own data-distribution and aggregation strategy in order to solve the problem. Through a series of user-defined number of computing iterations the results are communicated between the algorithm instances and improved during the execution. The number of instantiated algorithms and execution time are also adjustable. The following lists the main features of HyperSpark:

- **Ease-of-use** - the framework is designed using Spark to handle distribution and parallelisation in a transparent way to the user;
- **Portability** - the framework is implemented in Scala that inherits Java portability;
- **Uniformity** - the framework provides an uniform representation of problems, algorithms and solutions;

- **Extensibility** - the framework is developed such that through Scala mechanisms like inheritance, mixins, implicits and late binding one can easily extend metaheuristic algorithms;
- **Reuse** - the object-oriented features of Scala facilitate the code-reuse;
- **Loose coupling** - the framework is designed to decouple particular metaheuristic algorithms from the parallelisation technology;
- **Flexibility** - the framework allows users to define arbitrary parallelization strategies, run different metaheuristic algorithms and define how different solutions can be aggregated;
- **Cooperation** - framework allows for synchronous communication among parallel instances of the algorithm and therefore supports a large class of cooperative parallel metaheuristic algorithms;

The rest of the thesis is organized as follows. Chapter 2 discusses the state of the art, giving an overview of the technologies dealing with large-scale data-intensive computations, and describing their evolution towards high-performing cluster systems. Next, Chapter 3 presents the fundamental concepts of problem solving, the developed HyperSpark framework and its comprising elements. Chapter 4 describes a library based on HyperSpark framework, developed to solve Permutation Flow Shop Problem (PFSP), a popular optimisation problem from scheduling theory. Then, in Chapter 5, we present the results of conducted experiments, where we carry out a thorough analysis of the optimal setup of the proposed framework. In the end, Chapter 6 wraps up this work and draws conclusions on the outcomes. Furthermore, it points out relevant issues that remain open and will be the focus of future work.

CHAPTER 2

State of the Art

This chapter is devoted to the exploration of the state of the art in the fields of interest for the thesis. The discussion in this chapter will follow two paths. The first path, covering Section 2.1 and Section 2.2, will present Big Data concept, its related challenges, data parallelism and large-scale data-intensive computations. The second path (Section 2.3 and Section 2.4), describes the MapReduce programming model, focusing on its benefits for large-scale data-intensive computing. Within this path, technologies relying on MapReduce model and their internal structures and paradigms are being explained. In the end, there is a convergence of both paths in (Section 2.5), which introduces a new paradigm, called Big Calculations, and it tries to encompass all the requirements posed by modern big data and high-performance computing communities.

2.1 Big Data

Let us start with a story how the data became big, before the phrase Big data started being used in everyday life. Around seventy years ago people tried to quantify the growth rate in the volume of data or what has popularly been known as the "information explosion" (a term first used in 1941, according to the Oxford English Dictionary). Later on, data sets kept increasing creating the problem of overcoming the capacities of main memory, local disk, and even remote disk. We call this problem the problem of big data. Big data is being generated by multiple sources around us at all times. Every digital process and social media exchange produces it. Big data is arriving from multiple sources at an alarming velocity, volume and variety. In general, Big data refers to the data space so big that it can not be processed so easily. Hence, the demand for

computing power and efficient exploration procedures has been increasing over the last half of the century.

Handling workloads of great diversity and enormous scale is necessary in almost all significant fields of today's society, due to the penetration of Information and Communication Technology (ICT) in our daily interactions with the world both at personal and community levels, encompassing business, commerce, education, manufacturing, and communication services. With the rapid development of processing and storage technologies, and with the success of the Internet, computing resources have become cheaper, more powerful and more universally available than ever before. In such a setting, dynamic systems are required to provide services and applications that are more competitive, more scalable, and more responsive with respect to classical systems. This technological trend has fostered a new computing paradigm called Cloud Computing, in which resources (e.g., CPU and storage) are provided as general utilities that can be leased and released by users through the Internet in an on demand fashion.

2.1.1 Big data challenges

Big data technologies are maturing to a point in which more organizations are prepared to pilot and adopt big data as a core component of the information management and analytics infrastructure. Big data, as a compendium of emerging disruptive tools and technologies, is positioned as the next great step in enabling integrated analytics in many common business scenarios. As big data wends its inextricable way into the enterprise, information technology (IT) practitioners and business sponsors alike will face a number of challenges that must be addressed before any big data program can be successful. Some of those challenges are:

- **The Choice and Configuration of the Computing Infrastructure** – Selecting the physical components for the computing infrastructure and their configuration based on a level of computing demands of desired application can be hard. The reason is that this choice requires some effort, higher technical expertise and later management of hardware failures. Therefore, many parties choose to rent the existing resources from big data centers. Big data applications are deployed over the Internet connection, to the "cloud" platform.

- **The Choice of the Data Management Technology** – There are many competing technologies, and within each technical area there are numerous rivals. Making the best choices while not introducing additional unknowns

and risk to big data adoption is one of the challenges that big data practitioners need to face in the setup phase.

- **The Lack of Big Data Experts** – The excitement around big data applications seems to imply that there is a broad community of experts available to help in implementation. However, this is not yet the case, and finding right people to solve the problem poses an additional challenge.
- **Importing Data into the Big Data Platform** – The scale and variety of data to be absorbed into a big data environment can overwhelm the unprepared data practitioner, making data accessibility and integration another challenge.
- **The Choice of Data Processing Procedures** - The quality of final application result(s) greatly depends on the procedures and algorithms for exploring and processing Big Data.
- **Synchronization Across the Data Sources** – As more data sets from diverse sources are incorporated into a computing platform, the probability of data transfer impact on a computation environment performance grows higher. Also, the amount of data that needs to be transferred influence the performance. Hence, after the hardware and software setup, data synchronization might represent the most concerning challenge.
- **Getting Useful Information out of the Big Data Platform** – Lastly, using big data for different purposes ranging from simple data post-processing to enabling high-performance analytics is obstructed if the information cannot be adequately provisioned back within the other components of the enterprise information architecture, making big data syndication our next challenge.

2.2 Data Parallelism

Computer system architectures which can support data parallel applications were promoted in the early 2000s for large-scale data processing requirements of data-intensive computing [50]. Earlier practice of using the full capacity of one powerful machine (also known as CPU intensive paradigm) was quickly abandoned since it was not scalable in terms of cost and performance. The underlying idea behind data parallelism is that if the data computation is applied independently and in parallel to each data item of a set of data, the degree of parallelism becomes scalable with the volume of data. The possibility to scale the computation over the data has an implication of scaling the performance

of our application, leading to a decrease of several orders of magnitude of total computing time required or to highly improved quality of solution obtained when the same computing time is provided. These implications are the drivers for developing data-parallel applications.

The key issues with developing applications using data-parallelism are the choice of the algorithm, the strategy for data decomposition, load balancing on processing nodes, message passing communications between nodes, and the overall accuracy of the results [55]. The development of a data parallel application can involve substantial programming complexity to define the problem in the context of available programming tools, and to address limitations of the target architecture. Any scientific field like astronomy, atmospheric science, medicine, genomics, biologic, biogeochemistry, or a research areas like social network analysis, recommender systems, prediction markets that use data exploration, information extraction, data aggregation and data analysis may derive significant performance benefits from data parallel implementations since parallel operations are their intrinsic property.

The US National Science Foundation (NSF) [42] funded a research program from 2009 through 2010. Areas of focus were:

- Approaches to parallel programming to address the parallel processing of data on data-intensive systems
- Programming abstractions including models, languages, and algorithms which allow a natural expression of parallel processing of data
- Design of data-intensive computing platforms to provide high levels of reliability, efficiency, availability, and scalability.
- Identifying applications that can exploit this computing paradigm and determining how it should evolve to support emerging data-intensive applications.

2.2.1 Data-intensive Computing

The broad availability of data coupled with a trend of decreased hardware prices and increased computing power has given us possibility to solve the problems that were previously impossible to solve or impractical to implement because of the time required to perform computing process. Data volume and complexity are increasing fast. Despite the technological advances and decreasing costs of computing and storage solutions, the increase in data volumes introduces a storage issue, but too much data also introduces a massive analysis issue. Hence, it requires more effort to process and store data. Therefore, big data practitioners have to think how to store, retrieve, explore, analyse, and communicate this pile of data.

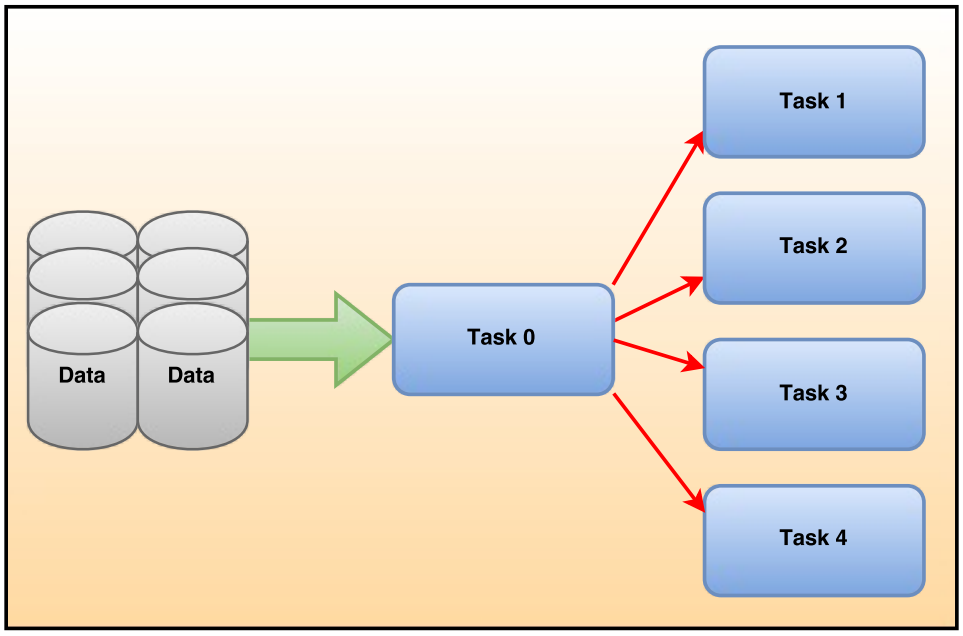

Figure 2.1 – Traditional Data-intensive Application Storage

These technical and social drivers have increased the urgent need to support computation on data of far larger scales than ever previously contemplated. Large-scale commercial data processing is required by the industry, and, therefore, the industry is investing in data-centers to support such a need. The data-centers are comprised of countless number of servers storing petabytes of data to support their business objectives and to provide services at Internet-scale. Such data centers are instances of data-intensive computing environments, the target of this request. For data-intensive computing, massive data is the dominant issue with emphasis placed on the data-intensive nature of the computation. Handling diverse hardware infrastructure issues, software setup, maintenance and many other aspects might cost a lot and require a lot of effort. Hence, many non-IT organizations will often turn to big data companies which are offering variety of services and accompanying pricing models deployed in the "Cloud".

2.2.2 From Traditional to Modern Approach to Data Management

The modern way of handling big data differs from traditional way in using the data locality principle.

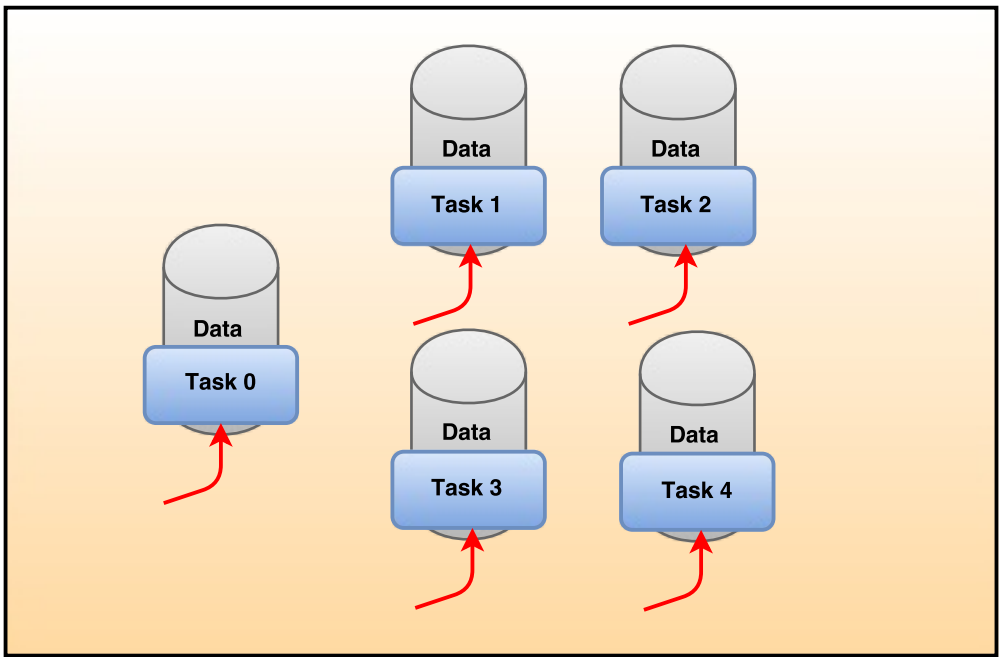

Figure 2.2 – Modern Data-intensive Application Storage

Traditionally parallel application (shown in Figure 2.1) required from a master parallel worker to read the input data from disk (green arrow), split the input data into chunks and send partial data to each of the other workers (red arrows). Later on, the workers that received the partial data would continue executing their computation in parallel. Upon completion, they communicate their results with each other, then continue the next iteration of the computation.

There is a serious issue of scalability present in Figure 2.1. The process of reading the input data (green arrow) is performed serially. Even if we can use some parallel File System (e.g. MPI-IO) to obtain, import and process the data in parallel, the data is separated from the tasks (computing resources) by some channel through which data can flow at some finite rate. While it is possible to increase the speed of this connection between the data and compute resources by introducing faster storage mediums or faster storage networking, the cost of doing it does not scale linearly.

The difference between data intensive computing and mainstream computing lies in a fact that previous one admits to large-scale parallelism over the data, and it is more suitable to environments in which the user performs operations via high-level programming primitives and the run-time system manages parallelism, data access and (optionally) fault tolerance. With the advent of

cheap and capacious storage devices the role that storage played in large-scale computing systems changed to support parallelism even more.

While traditional parallelism brings the data to the compute, modern approach does the opposite – it brings the compute to the data (Figure 2.2). The input data is not stored on a separate, high-capacity storage system. Rather, the data is already pre-divided and exists in the form of little pieces stored on computing nodes. The benefit of it is that there is no need to move any data since it is pre-divided and already exists on nodes capable of acting as computing elements. Only compute functions are sent to parallel workers which are running on the nodes where their respective pieces of the input data reside. Parallel workers perform their calculations and communicate results with each other, move data if necessary, then continue the next step of the calculation. Thus, the only time data needs to be moved (not necessarily) is when all of the parallel workers are communicating their results with each other. It might happen that parallel workers are completely independent and do not need to communicate with each other, in which case there is also present a saving of time spent on network bandwidth. Since the data already exists on the computing resources there is no more serial step of loading data from a storage device before being distributed to the computing resources. There is only one precondition for the computing elements to be able to do their calculations on these chunks of input data - both calculations and data must be completely independent from the input data on other computing elements. This is the main constraint for modern data-intensive applications: they are ideally suited for trivially parallel calculations on large quantities of data, but if each worker's calculations depend on data that resides on other nodes, side-effects and decrease in performance will occur.

Large-scale data-intensive computations often require high degree of fault tolerance, availability and reliability. Support of these requirements is dependent on a specific implementation of big data computing system. Examples of such systems are OpenMPI, Hadoop and Spark. OpenMPI started supporting fault tolerance from version 1.3 (year 2007), while Hadoop had it almost since the beginning of framework development, but it appeared later (version 0.21 supporting fault tolerance was released in 2010). As a new technology, initial version of Spark (year 2014) had all required computing nodes management demands supported. Data intensive applications might also often require real-time responsiveness and support heterogeneous data types. Uncertainty in the data might also be present. Scale impacts computing system's ability to retrieve computational results and to provide, if required, integrity and availability functionalities in various levels of uncertainty. A common practice is to include such levels in different pricing models of cloud services.

The Data intensive Computing paradigm seeks to increase our understanding of the capabilities and limitations of data-intensive computing. When a data-intensive computing platforms is used a couple of questions arise. Those are:

- How can large-scale parallelism be expressed in natural way for the user of computing platform?
- What are the programming abstractions (models, languages, algorithms, etcetera) needed to support fundamental requirements?
- What if the volume of data surpasses the capabilities of computing and storage devices? Is the platform still in consideration for usage?
- Is the computing platform suitable for the domain of problems that we want to solve?
- How to exploit multiple computing nodes for specific application?
- Does introducing parallelism reduces computation time with respect to acceptable ratio?
- What is the overhead of managing multiple computing instances during computation? Is it acceptable?
- Is the quality of results improved by providing the same computation time to multiple computing elements as when provided to single computing element?

These are fundamental questions that every scientist should ask him/herself before diving into the world of data-intensive implementations. They should all be treated with special care, otherwise the spent time on specific application might be pointless.

2.3 MapReduce and Hadoop

MapReduce is a general algorithm supporting the concepts of map and reduce functions, named after the first commercial implementation by Google [12], which allows for handling huge datasets in a distributed, fault-tolerant framework. The most widespread open source implementation of this programming paradigm is Apache Hadoop [23].

The root of Hadoop lies in Apache Lucene, the open source API for information retrieval, specifically in one of its sub-projects: the web search engine

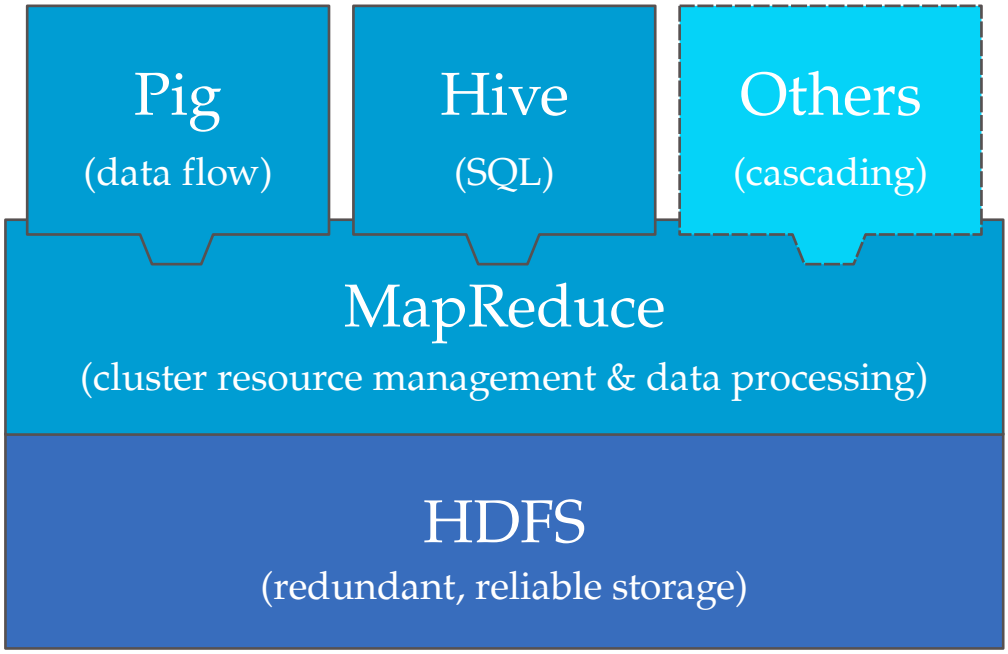

Figure 2.3 – Hadoop MapReduce v1 architecture

Apache Nutch, born in 2002. In 2003 Google made public the architecture of its distributed file system GFS and the next year also a paper explaining the MapReduce paradigm [12], providing in such a way two important technologies to overcome the scalability limits of the Nutch project. In 2006 Hadoop became an independent sub-project of Lucene and a top level Apache project in 2008.

The first version of Hadoop aimed just at providing a framework for the MapReduce programming model: every application that could be rewritten in terms of map and reduce tasks could take advantage of Hadoop distributed computation capabilities. As seen in Figure 2.3, Hadoop uses the Hadoop Distributed File System (HDFS) and its related service as a storage layer, while the MapReduce framework enables users to submit their own applications as MapReduce jobs. Furthermore, applications like Hive and Pig have been developed on top of Hadoop to hide the underlying MapReduce engine, providing the capability of using high level query languages to operate on the data.

2.3.1 MapReduce Applications

A MapReduce application, or job, comprises input data (typically stored in HDFS) and user code defining the logic of map and reduce operations. The job execution can be seen as split into a map phase, a shuffle phase, and a reduce

phase.

In the *map phase* input data splits are fetched by a map task each, making the number of map tasks data dependent. For performance reasons the intermediate map output data are saved in the local file system of the worker, since the replication would be time consuming and keep the network busy, whilst redundancy is not required for this kind of information.

In the *shuffle phase* key-value pairs from all the mappers are fetched in parallel by some dedicated threads at the reducers, then the obtained files are sorted. This phase begins as soon as the first map task finishes, making the first data available for the reducers. The number of reducers is not data dependent, but can be manually set. Also, the set of keys is hash-partitioned so that each key will be fetched by one and only one reduce task.

The *reduce phase* is the execution of the reduce logic on the intermediate data. The persistent output will be written in the output file system, typically HDFS.

It is important to notice that since reducers do not focus on a specific source to collect their key-value pairs, data locality has less importance here, while a map task is instead executed as close as possible to the chunk of data it was assigned to. Since the bandwidth usage optimization is a crucial element in the performance of a MapReduce job, it is convenient to limit the amount of data transferred from the map tasks to a single reducer. By means of a so called combiner function it is often possible to execute an aggregation of the output data of a map task on the physical node where it was executed, still keeping them in the form of a valid input for the subsequent reduce task.

2.3.2 HDFS

The Hadoop Distributed File System has its origin in the Nutch Distributed File System (NDFS), developed to specifically overcome the lack of a distributed file system to handle big sized files with a cluster of barebones commodity hardware. This kind of file system already had most of the features and concepts that we find in HDFS, except for the lack of a user permission system, absence of quotas, and a much shorter set of configurable settings.

In a Hadoop cluster we typically find a master node running the NameNode service for HDFS: it is responsible for maintaining a file system tree with the location and properties of files in the so called FsImage file and metadata changes into a transaction log, the EditLog. Both the FsImage and the EditLog are stored in the local OS file system. Another relevant process in the HDFS infrastructure is the DataNode, running on every node and physically storing the data, as can be seen in Figure 2.4. This last process is resident on every node

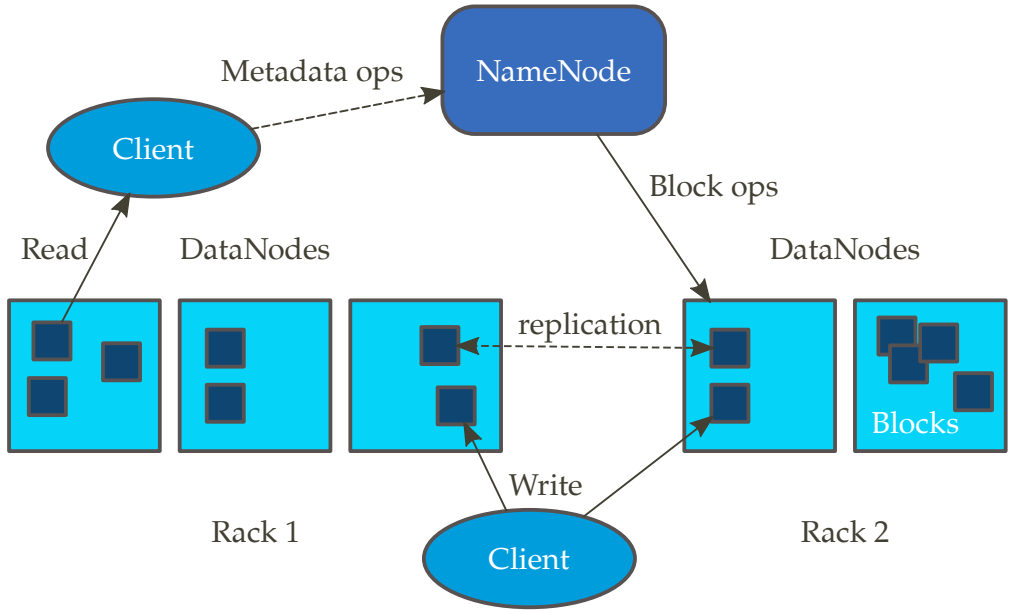

Figure 2.4 – HDFS architecture

hosting part of the HDFS file system and handles the requests coming from the NameNode about physical operations on the files.

HDFS enables the MapReduce framework to operate over huge-sized files (typically at tera or petabyte scale) by organizing them in blocks of fixed size and replicating them among different nodes. There is a default replication factor of three and, in order to provide data availability and reliability even upon different degrees of system failure, while reducing network bandwidth utilization, two replicas are, possibly, kept on different nodes of the same rack, and one replica on another rack. The replication of data takes place in a pipelined fashion: broadcasting the file from a unique source to the destination nodes would take up too much of the local network resources, so when the first DataNode starts receiving and writing the data from the client, it forwards them to the second DataNode, and so does the second, until all the DataNodes have their replica of the file. In such a way the client has to forward the data just once, spreading the workload farther. In case of node failures, or upon changes in the data, the DataNodes communicate among themselves, without the intervention of the central NameNode, to keep the file system in a coherent state, as shown in Figure 2.4.

HDFS has been specifically developed to store files of huge size and provide a high throughput, for this reason the data are expected not to change, maintaining a write-once-read-many access model. This is especially important if we

consider the relevant amount of network traffic that would be generated by a change in the data content, caused by the update of every replica of every block where the modification occurred.

Client applications cannot directly write a file in HDFS: first the data has to be written in a temporary file on the OS file system, once the file reaches the size of a HDFS data block, a request is sent to the NameNode, which will record the change in the file system structure and send back the location of the DataNode where the new data will be physically stored. Only upon the closure of the file the NameNode will commit the file creation operation to the persistent log. A user can interact with the file system at the NameNode with a pseudo-POSIX interface, hiding the underlying NameNodes and DataNodes structure.

In the first version of HDFS the NameNode was a single point of failure. In order to provide high availability two redundant NameNodes are instantiated on different machines of the same cluster. At any point in time one is in an active state (Active node), the other is kept in a hot standby state (Passive node) to provide an automated failover in case of necessity. The standby node can keep its state synchronized with the active one in two ways: either by communicating the changes through a shared common storage device, or through a group of separate daemons, called Journal Nodes (JNs), on which the Active node can communicate the applied changes and from which the Passive can read them.

In case of user errors or situations when a disaster recovery is needed, HDFS now provides snapshots of the file system. Snapshots are instantaneous and just record the block list and the file size of a sub-tree of the file system or the entire file system, so they do not replicate the actual data. Upon accidental deletion of a file, the related blocks are "protected" by the snapshot and just its metadata are deleted: in such a way, by restoring the snapshot it is possible to recover the previous state of the file system for that directory.

Another feature introduced with HDFS 2 is HDFS Federation. In the previous version of the file system a single NameNode and a single namespace were allowed, now this limit is overcome with multiple and independent (federated) NameNodes and namespaces. In this way the horizontal scalability of the storage is supported by an horizontal scalability of the namespace, in addition to isolation and a throughput improvement.

2.3.3 Hadoop YARN

Hadoop Yet Another Resource Negotiator (YARN) was designed to address the unsolved issues typical of the previous versions, while maintaining a backward compatibility with legacy MapReduce applications. Instead of just trying to rewrite the JobTracker, a radical change in the architecture has been made, with the main effects of obtaining a framework decoupled from the MapReduce

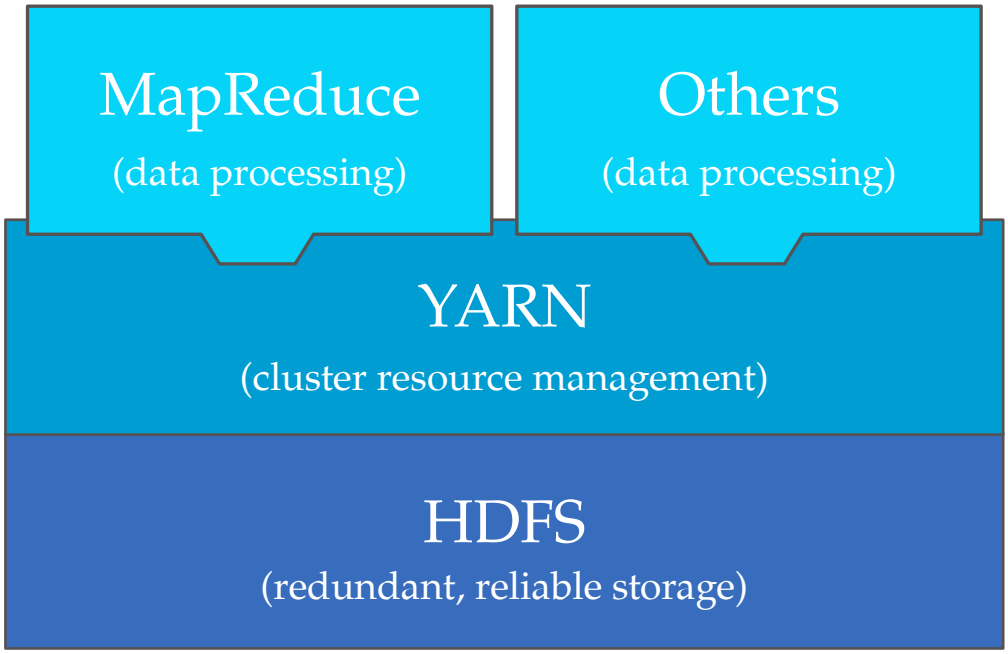

Figure 2.5 – Hadoop YARN architecture

paradigm and moving the management of the application layer away from the system daemons.

While initially Hadoop had the MapReduce engine as the only framework available to developers, now this layer is just one of the possible application running on YARN, as shown in Figure 2.5. For example, YARN allows the user to execute Distributed-Shell applications on multiple worker nodes in the cluster. The system is still taking advantage of HDFS, but as we leave the storage layer we face important changes in the Hadoop architecture.

Since applications are not necessarily MapReduce jobs, it becomes too simplistic to organize the resources of nodes in terms of fixed map and reduce slots: the cluster is now seen as a resource pool and requests are now satisfied by assigning containers, providing a multiple of fixed minimum amounts of memory, disk, network, and CPU resources to the running user code. In this way, applications can request and release resources according to their needs, gaining a high flexibility for the YARN resource model.

The JobTracker disappears and the resource management is now accomplished by a Resource Manager (RM), that still resides on the master node and takes care of assigning containers to the requesting applications. Since the optimality of scheduling policies is strictly dependent on the kind of applications that are going to be run on the system, the ResourceManager works with a plug-

gable scheduler, allowing users to implement their own. By default we find the Capacity scheduler with the chance to replace it with the Fair scheduler or the old FIFO scheduler. This last choice might have a sense only in very small cluster with a limited workload, since the lack of job priority awareness and other advanced features makes it unsuitable for large shared cluster.

The other feature of the JobTracker, the job scheduling and monitoring, is now accomplished by Application Masters (AMs), per-application framework instances operating decoupled from the central system, typically on a slave node. When an application has to start, the respective AM is the application-specific first component deployed in a dedicated container. It then negotiates more containers with the RM on behalf of the single instances of that application, coordinates their execution, and monitors their resource consumption interacting with the NodeManager. This change is not trivial, since the scalability is no more limited by the capacity of the JobTracker and the sole role of the ResourceManager at the master node is just to schedule the resources.

The NodeManager is a per-node process and can be seen as the evolution of the TaskTracker. It receives container requests from the AM, runs and possibly kills them, monitors the resources in use, manages logs and distributed caches, and periodically sends heartbeats to the RM about the health and resource utilization of its node.

2.3.4 From FIFO to Capacity and Fair Schedulers

Before the shared cluster era, every job was scheduled in a FIFO order, thus potentially taking advantage of all the cluster resources once running. In newer versions of Hadoop, where jobs were coming from different users, such a scheduler had clearly to be replaced.

Yahoo! proposed the Capacity scheduler [8]: the general principle behind it was to guarantee a minimum capacity to every organization submitting jobs to the cluster. A hierarchical structure of queues is exploited, where we find parent queues and leaf queues. The former can contain other parent queues or leaf queues, the latter actually contain jobs. The root queue represent the whole cluster and its total capacity is statically divided among the first subqueues according to the administrator's preferences. Each of these queues will then statically assign the obtained capacity among their children, and so on until the leaf queues are reached.

In order to achieve a better throughput, within a leaf queue jobs are scheduled in a FIFO order (according to the application submission time), with the chance to set upper and lower bounds to the capacity that users or AMs in a specific queue can obtain, preventing greedy or malevolent agents from occupying all the available resources. At any level, available resources are first provided to

those queues with the most underserved capacity, where the capacity of a parent queue is the aggregation of the sum of its children capacities, in a recursive fashion.

The Capacity scheduler tries to maximize cluster utilization by assigning also idle resources, even when this takes a user or a queue beyond the imposed limits. As of today the default behavior allows a queue to take up to the total cluster resources when they are idle, but a single user cannot take up more than the queue's configured capacity, even if a proper setting of the configuration properties can allow this. If a queue has obtained more resources than its limit and new users from other queues submit new applications, the scheduler will prioritize those users and queues until they meet their granted capacity requirements. The behaviour with respect to the extra capacity holders is defined by the administrator: newcomers may just wait for resources to be made available again, or a preemption mechanism can force extra resources to be released by the holders. Currently the Capacity scheduler lets application submit resource requests in terms of CPU and memory; since this is connected to the current limits of the YARN resource model, it is expected in the future to be able to specify also resources in terms of disk, GPU or bandwidth requirements.

The Fair scheduler is similar to the Capacity scheduler, but embraces a different philosophy: it will try to assign to different applications an equal amount of resources over time, in terms of memory and optionally also CPU resources. A single running application is entitled to possibly obtain the entire cluster and, as others arrive, free resources are allocated to grant fairness, again with the chance to force the migration with preemption. Small application can finish in reasonable time, without incurring in starvation due to longer ones using large amounts of computational resources. Weights can be set to balance the resource assignment in favour of specific applications. Similarly to the Capacity scheduler, applications can be organized into queues and minimum resources guarantees can be set on them, again trying to assign also the idle capacity to obtain high cluster utilization. Within the single queues FIFO scheduling is not the default choice (but still configurable, as well as multi-resource with Dominant Resource Fairness), instead a memory-based fair sharing is adopted.

2.4 Spark

In recent years we witnessed the rise and consolidation of Cloud systems as a new powerful ICT paradigm and, consequently, a remarkable increase in their applications in many fields. As a consequence, there are many frameworks being developed to provide simple and effective way to process Big Data. Apache Spark™ [3] appeared in May 2014 as an open source cluster computing frame-

work and quickly rose to the title of one of the most active Big Data projects, not only in Apache Software Foundation, but also in the open source community in general. Its main characteristic - in memory computation, and its promise “"Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk."” [3] motivated us to examine its suitability for Data/CPU intensive applications.
This section is organized as follows. Subsection 2.4.1 provides a high-level overview of Spark framework and its components. Subsection 2.4.2 and Subsection 2.4.3 describe the Spark environment initialisation and application architecture, respectively. Subsection 2.4.4 provides details on task and application scheduling within the cluster. Finally, in Subsection 2.4.5, Spark framework deployment-to-cluster possibilities are enumerated.

2.4.1 Overview

> In a very short time, Apache Spark has emerged as the next generation big data processing engine, and is being applied throughout the industry faster than ever. Spark improves over Hadoop MapReduce, which helped ignite the big data revolution, in several key dimensions: it is much faster, much easier to use due to its rich APIs, and it goes far beyond batch applications to support a variety of workloads, including interactive queries, streaming, machine learning, and graph processing.
>
> Ion Stoica, CEO of Databricks and Co-director, AMPlab, UC Berkeley

Apache Spark is a cluster computing platform designed to be fast and general purpose [24]. Its speed is based on an in-memory computation, but it is also more efficient than MapReduce for complex applications running on disk. Spark integrates popular MapReduce model to support many types of computations, such as interactive queries and stream processing. It avoids unnecessary disk I/O operations and uses main memory and caching in order to be more time efficient. These characteristics are crucial and particularly noticeable when big data sets are processed. Therefore, Spark reduces waiting time necessary for processing and provides a way to explore data interactively. Since data are pre-divided some partial results may be obtained at any time.

Spark unites several closely integrated components for managing workloads that previously had to be used as separate engines, batch applications, algorithms, distributed systems, interactive queries, streaming, etc. Unlike Hadoop MapReduce, Spark can combine SQL, streaming, and graph analytics within cloud analytics applications. A visualisation of Spark components stack is shown

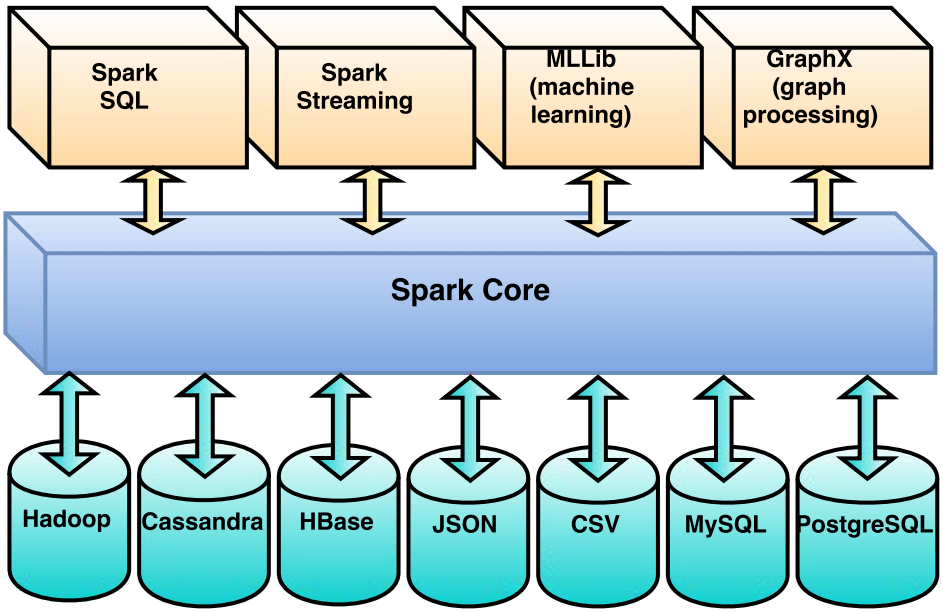

Figure 2.6 – Unified stack of Spark components

in Figure 2.6. Further, Spark tries to make framework usage as simple and more general as possible. Hence the management of such sub-elements is done completely by Spark, reducing the user's necessary expertise for their maintenance. Also, Spark provides APIs in different programming languages like Python, Java, Scala, R and SQL, making it highly accessible for different IT industry sectors. It also integrates closely with other Big Data tools [24]. It is specially focused on supporting popular cluster management engines like Hadoop YARN and Amazon EC2, but also supporting different data sources, including Hadoop data sources, Cassandra databases, JSON, MySQL etc.

Spark core is responsible for scheduling, distributing and monitoring the applications submitted by the end user [24]. Those applications might consist of several computational tasks that have to be processed by computing nodes of the cluster. Spark uses multiple higher-level components specialised for various workloads, such as SQL or machine learning. Those components are represented as libraries in a software project and can be combined in order to build powerful applications. For example, an application might classify real time data (e.g. social network news feed), using machine learning algorithms from MLLib library and streaming library at the same time.

2.4.2 Starting Spark Environment

Each Spark application needs to invoke a start hook of the Spark environment. That is achieved by creating a SparkContext object inside the user application (Figure 2.9), and invoking some function on it. Each Spark application is an instance of SparkContext, meaning that only one instance of SparkContext may be running during the lifetime of the application. Before initialising SparkContext Spark configuration should be set by specifying command line options when submitting a packaged application to bin/spark-submit script, or by setting spark configuration within the application. SparkContext constructor accepts SparkConf object where all Spark-related settings should be set before initialising SparkContext. In both cases, properties inside bin/spark-defaults.conf are overriden by user-specified options. The minimum requirements for a user are to set cluster (master) URL and application name.

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object MyApp {
def main(args: Array[String]) {
   val conf = new SparkConf()
   .setMaster("local")
   .setAppName("MyApp")
   val sc = new SparkContext(conf)
   //application code goes here
}
```

Figure 2.7 – A simple Spark application

2.4.3 Spark Application Architecture

Each Spark application represents a driver program that distributes computing operations over the cluster. The entry point of the driver is application's main function, and *SparkContext* object represents a connection to a computing cluster. The driver program is responsible for establishing a connection to the cluster (by creating SparkContext object), serialisation of tasks and data, their distribution to each of the parallel workers (Figure 2.8), and results collection and aggregation.

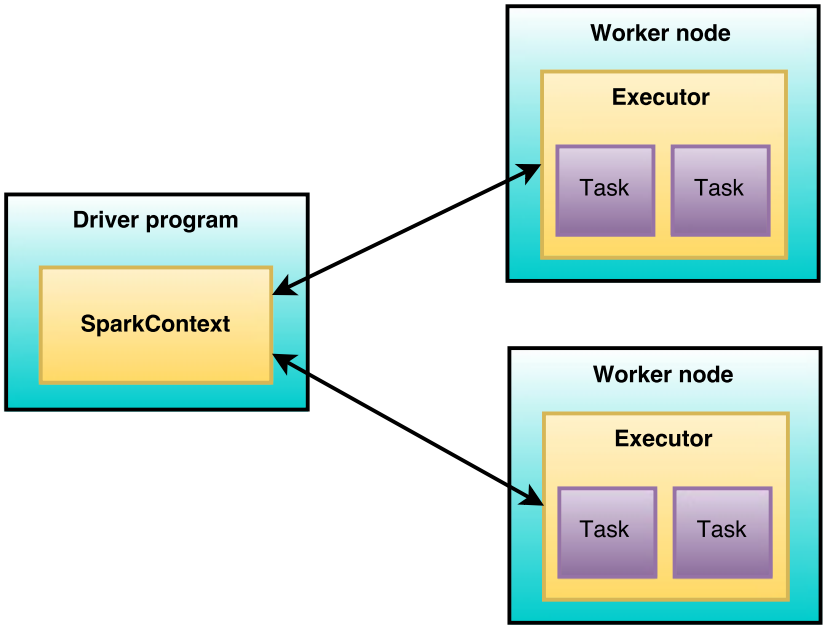

Figure 2.8 – Spark Application Components

2.4.3.1 Resilient Distributed Dataset (RDD)

Usually, instead of manually pre-dividing big data and putting parts on each of the phisical computing nodes, user instructs Driver program to do it for him/her. That is done by invoking SparkContext.*parallelize(...)* function over desired data collection:

```
val data = Array(1, 2, 3, 4, 5, 6)
val distData = sc.parallelize(data)
```

Figure 2.9 – Creation of distributed data set

There are two ways to specify parallelize function parameters using its signature:

```
def parallelize[T: ClassTag](
    seq: Seq[T],
    numSlices: Int = defaultParallelism
  ): RDD[T]
```

The first way is to specify data collection and the number of partitions of the provided data collection. The second parameter is very important, as it states how many distributable units of data are going to be created. For example, if we specified that number to be 3 in Figure 2.9 the output would result in three partition objects, e.g. (1,2), (3,4), (5,6). The second way of partitioning a data collection is by not specifying numSlices parameter and let Spark automatically find the optimal number of partitions considering the number of CPUs inside the cluster. Usually, that number will be 2-4 partitions per each CPU in the cluster.

What is important to note here is that the number of partitions has a one-to-one mapping with the number of parallel tasks that will be run during the computation. More about the tasks will follow later in the chapter.

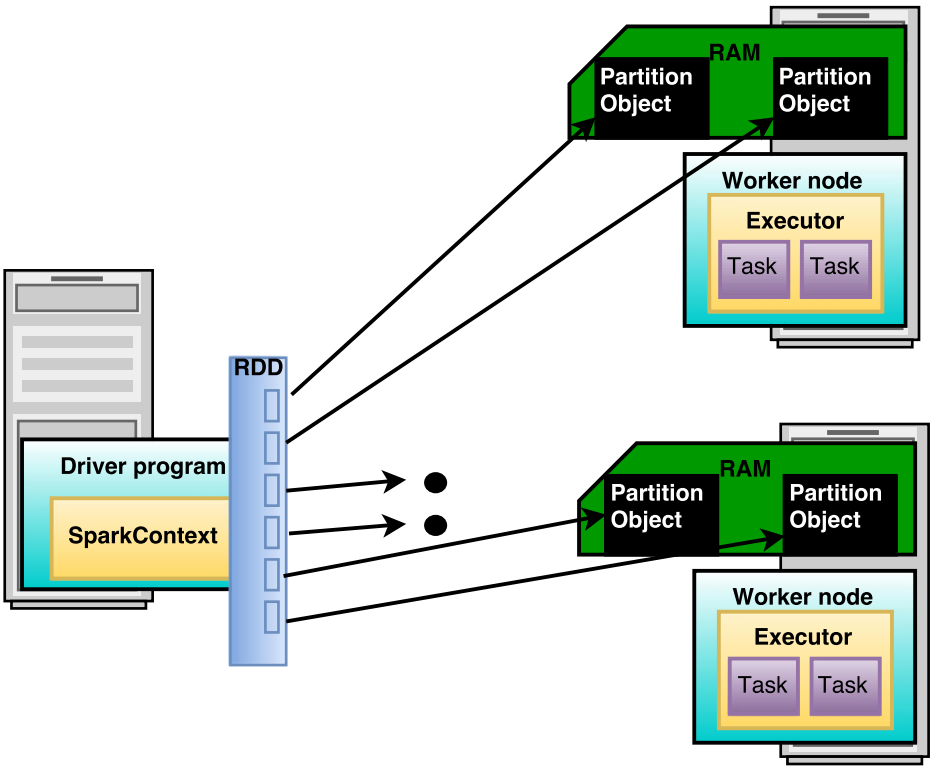

Figure 2.10 – Creation of Resilient Distributed Dataset

The output of parallelize function represent an immutable distributed dataset called Resilient Distributed Dataset (RDD), on which various parallel operations might be invoked: apply a transformation on each of the elements in collection, filter or count the elements, etc. There are even operations defined

between multiple RDDs, like union, join, cartesian product... In Spark, all parallel operations are expressed using creation, transformation and processing of RDDs. RDD may contain any basic class type, but also user-defined classes. The only condition is that all elements of one RDD need to be of the same type. RDD is actually an array containing the references to the Partition objects (Figure 2.10), which are created by the driver program when parallelize is invoked on SparkContext and are sent to worker nodes on which they are stored in main memory (RAM). During the execution, the driver program will send necessary parallel operations to each of the workers, and each worker will perform one parallel task for each partition. The Partial Objects will stay on the same Worker node as much as possible to avoid unnecessary sending of data over the network.

2.4.3.2 Basic RDD Operations

RDD operations are divided in two groups: transformations and actions. The difference between those two is that transformations are lazily evaluated when actions are called, not before. Further, transformations construct a new RDD from previous one, while actions compute a result using existing RDD and send it to the driver program or save it to some storage system (typically HDFS) [24]. What Spark does, when it receives driver instructions consisting of transformations and actions over RDD, is creating a Directed Acyclic Graph (DAG) using its internal class named DAGScheduler, which is also used for scheduling of parallel operations. An example of such a graph is shown in Figure 2.11.

Lazy evaluation has the goal of decreasing the number of MapReduce operations that need to be performed during the application life-cycle. Spark internally records meta-informations about data collection that needs to be loaded from file system and operations that need to be performed on RDDs. DAGScheduler will perform topological sort of requested transformations producing an execution sequence (a DAG), and once an action (e.g. count(), reduce(), take(), collect()) is encountered Spark driver program will ship the execution sequence and source code to worker nodes, which will lazily load the data (if not already in memory) and perform all transformations included in execution sequence.

There are two kinds of transformations:

- **Narrow transformations** - All partitions from a parent RDD will be included in a single child RDD. This kind of transformation does not require any network shuffling of the data. Examples are map, flatMap, filter, sample, etc.

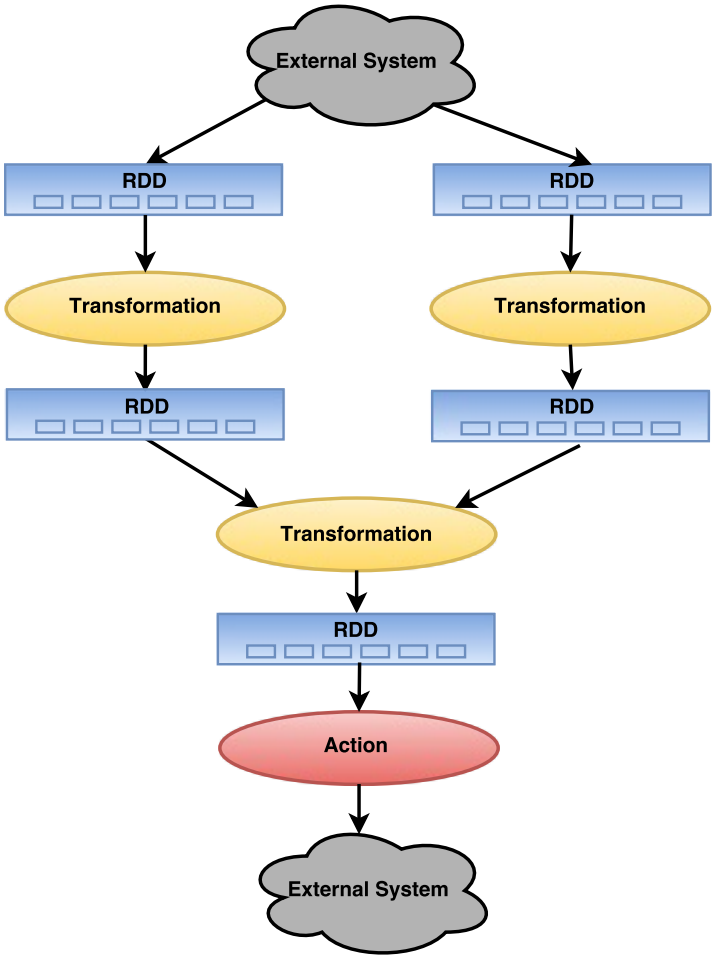

Figure 2.11 – Creation of Directed Acyclic Graph [51]

- **Wide transformations** - Partitions from one or multiple parent RDDs might finish in different child RDDs, depending on a value of the partition object (e.g. a key for key-value pairs). Since the value-based transformation might require from a partition object to go to different RDD slot than the one it was previously placed in, this kind of transformation might require sending partition objects to different machines over the network. Examples are reduceByKey, groupByKey, repartition, etc.

For more details about specific transformations the reader is advised to read [52].

2.4.4 Execution Scheduling Within an Application

In order to explain how the scheduling is performed, we need to provide an existing terminology adopted in Spark environment:

- **Job** - Each Spark application (SparkContext instance) might consist of several jobs, where each job represents an execution plan (a DAG) submitted by one application thread. Hence, a job corresponds to previously described concept of DAG. If multiple parallel jobs need to be started they must be submitted by multiple application threads. Starting several consecutive jobs is achieved by submitting multiple consecutive execution plans by the same thread.

- **Stage** - Starting from a terminal node in DAG (which is an action), Spark tracks back all transformation needed to execute the DAG and, in order to minimize data shuffling, it groups all subsequent narrow transformations in a stage (Figure 2.12). For example, many map(..) transformations can be grouped in one stage. Transformations inside a stage do not require any data shuffling since they are performed within the partition. Stages are classified as a "Map" or "Reduce" stages (concept adopted from traditional Hadoop MapReduce model). Compared to Hadoop's execution model, the execution model of Spark allows one job to contain more than two phases. DAGScheduler is responsible for creation of DAG, analysing it and passing a set of stages to TaskManager.

- **Task** - Each stage is executed as a series of parallel tasks. The term task is a synonym for basic RDD operation and therefore it represents a fundamental computing unit of Spark environment. A task is created for each partition in an RDD. Basically, when an RDD operation is executed - a set of parallel tasks, performing the same operation, run in parallel on each partition of RDD on their assigned executor (worker node process).

- **Master** - A computing node within the cluster on which a Driver program is executed.

- **Driver** - A program containing users instructions, usually placed on separate computing node inside the cluster.

- **Worker** - A computing node within the cluster on which multiple Executor processes may run.

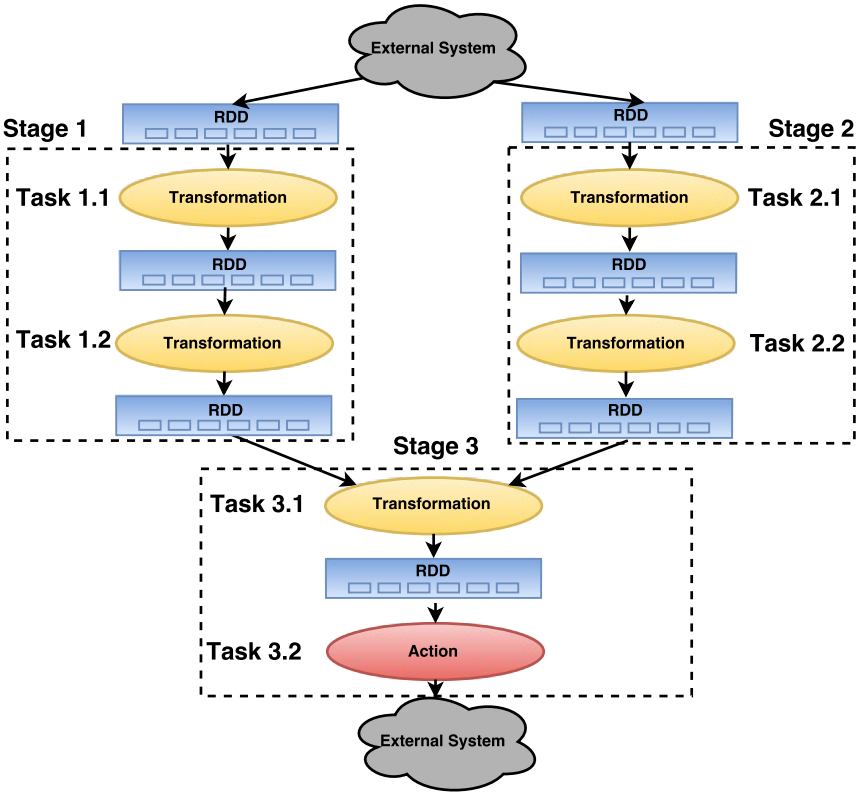

Figure 2.12 – Analysis of DAG Operations [51]

- **Executor** - A process executing computing tasks sent by the driver. When an executor is started it registers itself in driver's executor list. The Spark driver keeps a list of running executors and tries to assign them individual tasks based on data locality. The executor may cache RDD data and notify the driver program about it, in order to schedule the next tasks more efficiently. Inside Spark configuration submitted with user application there is a property which specifies how many cores of CPU will one executor occupy. Coupled with the property that allows a user to change the number of executors instantiated, it provides him/her an easy way to set the scale of computation in Spark environment. For instance, if executor occupies one core (default value in Yarn deployment mode[1]), and the user sets the number of executors to 48, the application will use 48 cores inside the cluster.

TaskScheduler receives a set of tasks submitted by DAGScheduler for each

[1]Deployment modes are described in Subsection 2.4.5.

stage, and acts as a controller over a set of active executors in the cluster. It is responsible for sending tasks to executors, running them using data locality in best way, retrying failed tasks and mitigate straggler tasks. It sends events back to the DAGScheduler. The order of scheduling the tasks is "FIFO" or "FAIR", which is read from the same property that is used to set scheduling of applications. Fair scheduler in Spark is modeled after Hadoop Fair scheduler, described in Subsection 2.3.4. When there are multiple jobs submitted, the parallel tasks of each job are assigned to computing resources in a "round robin" fashion, providing a roughly equal share of resources between the jobs.

A DAG defines a deterministic transformation steps and therefore fault recovery is very straightforward [51]. At the end of each stage worker node saves results in a local file. If some of the worker crashes during the execution of a stage, another worker node can pull files from crashed worker node and re-execute a stage. If the file is not accessible, a parent stage needs to be re-executed as well.

2.4.5 Execution Scheduling Across Applications

So far, we have talked about scheduling of jobs within one application. If there are multiple users using the cluster at the same time, cluster resources need to be managed by some higher-level software engine. That additional layer, known as cluster manager, is shown in Figure 2.13.

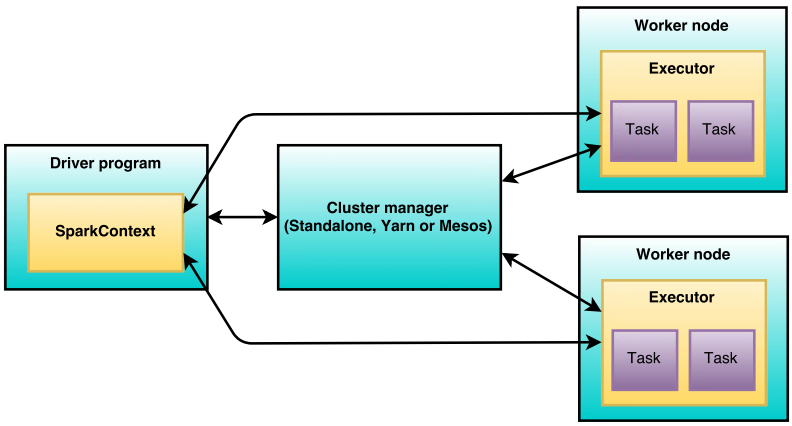

Figure 2.13 – The role of Cluster Manager in Spark Application Submission

When a user submits an application to the cluster manager, an application gets a set of executor Java Virtual Machines (JVMs) that are only running tasks and storing application data. How the cluster resources (executor JVMs) are

allocated and managed depends on a specific implementation of cluster manager. There are three types of cluster managers supported in Spark, shown in Figure 2.14. Note that Standalone deployment mode is a part of Spark core, but in Figure 2.14 it is presented as a separate component for easier explanation of concept.

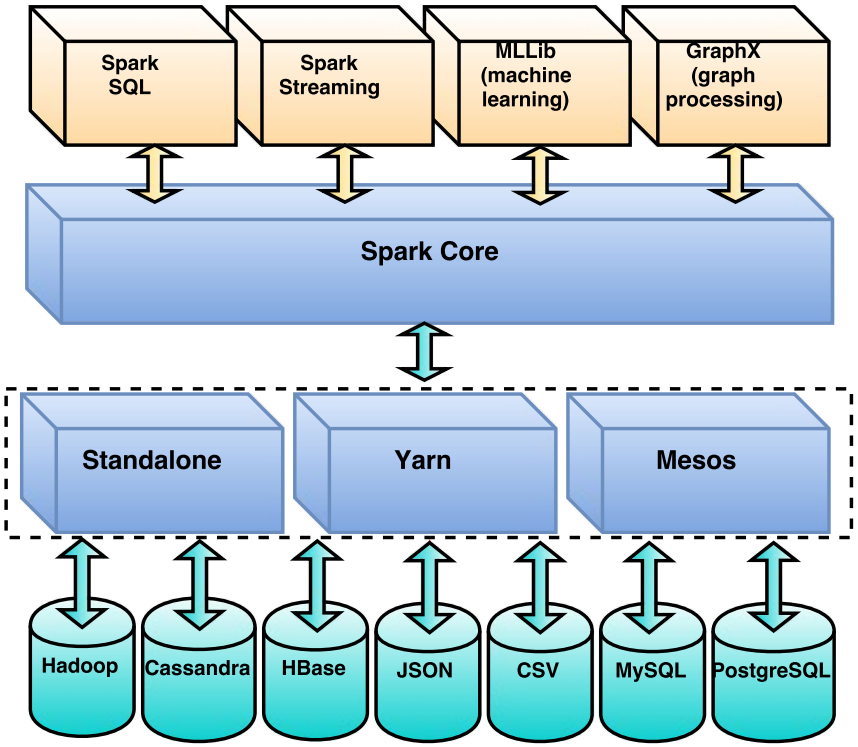

Figure 2.14 – The place of Cluster Manager in Spark Components Stack

Static partitioning of cluster resources is available in all deployment modes. When an application is submitted to the cluster manager, it gets all available resources that it requested, except in a case where that number surpasses the limit assigned. The application releases the resources upon finishing its execution.

- **Standalone Deployment Mode** - This is the simplest deployment mode available, and its current version (within Spark 1.5.1.) supports only FIFO scheduling of applications. Each application tries to statically allocate necessary resources, and if it succeeds, it is accepted for execution. In contrast to other two modes, here the resource management is done by SparkCore itself.

- **Yarn Deployment Mode** - provides Spark compatibility support for Hadoop clusters on which Yarn has already been installed. As a separate component, Yarn provides simplified deployment, provisioning, management and monitoring of resources. Thus, it requires minimal IT involment. Yarn also provides secure, multi-directional inter-communication between Virtual Machines and locality-aware access to data stored in HDFS. Spark Yarn deployment mode supports both FIFO and FAIR scheduling of applications. Yarn is for now the only Spark deployment mode which provides dynamic scaling of cluster resources (executor JVMs) based on the amount of workload, although there is work in progress to include it in other two modes. There are two subtypes of Yarn deployment mode: yarn-client - in which a client can query the application during its runtime from a local machine, and yarn-cluster - in which the application is submitted completely to the cluster and runs on the same computing node as the master node inside the cluster.
- **Mesos Deployment Mode** - Spark engine does not need to be the only framework installed on a cluster. Beside support for multiple Spark instances, Apache Mesos also supports Storm, MPI and many other products, which can be installed and running at the same time on a cluster.

There is also an option to debug the Spark application on a local machine. This is called "Local" deployment mode, but is not actually a deployment to a cluster. It serves only for testing the applications during the development process and interactive use.

2.5 Big Calculations - Moving Beyond Big Data

Earlier in the past, many typical scientific experiments posed a limit on a volume of data that could be processed to obtain a high-quality solution for a specific problem. In Subsection 2.2.1 we explained the fact that increasing memory capacity or CPU power of a single machine does not scale linearly with the cost and solution quality, since there exists a limit on a speed of data transfer through physical wiring.

> "High Performance Computing most generally refers to the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business."
>
> — G.Sravanthi, B.Grace and V.kamakshamma, A review of High Performance Computing, IOSR Journal of Computer Engineering

Although in the past there were many efforts to support and adapt to High Performance Computing (HPC) paradigm, CPU speed of new computers was not growing fast enough to follow the trend of increased size of data sets.

MapReduce programming model, proposed by Google and later developed by Yahoo as an open-source project under the name of Hadoop, was designed to support large-scale, parallel processing of big volumes of data using a cluster of commodity hardware, enabling users to scale their calculations up to the number of computing resources available in the cluster. The trick is that the data is already pre-divided and data and calculations on different computing nodes are independent of each other. If that pre-condition is met, it is experimentally proven that using more computing resources (physical machines) to increase the scale of computations tends to increase the quality of solution introducing minimal, in some cases insignificant, overhead for the cluster management.

In the beginning (Hadoop v1 release) Distributed File System, Resource Manager and MapReduce model were so tightly coupled (in contrast to HPC where the integration of components is loosely coupled) that the iterative algorithms, for example from machine learning field, could not benefit from that kind of platform [28], although Hadoop has shown convincing results in all other applications. With the appearance of YARN (Hadoop v2) as a separate layer in a big data stack, resource scheduling was decoupled from application scheduling. Dr. Geoffrey Fox and his colleagues from Indiana University pointed out the potential benefit of using HPC paradigm with existing large-scale data-intensive computing platforms [28], and that the machine learning field might be the one benefiting the most out of such an approach:

> "Most of these algorithms are familiar to HPC because they have linear algebra at their core... Solve large-scale optimization problems like learning networks comes from a different world than HPC but look like HPC because it's all very large parallel jobs with lots of optimizations for performance."
>
> Dr. Geoffrey Charles Fox, Professor of Informatics and Computing and Physics, Indiana University

As Cloud systems are being more and more exploited, a pile of MapReduce applications has emerged. Adding more computing nodes to the cluster does not only increase the quality of solutions, but it might also decrease total time for processing big data set, sometimes enough to produce real time business decisions. In the past HPC and data-intensive computing were two paradigms with support of different workloads. HPC favoured high-performance computing machines, while data-intensive computing preferred cheap storage and in-

formation retrieval. Their evolution brought to a convergence to the same point - data-intensive computing platforms started being used for CPU-intensive computations and HPC systems started executing more parallel data-intensive tasks. Victor Allis, CEO of supply chain information system provider Quintiq, has provided an interesting idea of exploiting current data-intensive platforms for high-performance decision making [6]. Big data volumes are unstoppably continuing to grow, and there is not much we can do about it. Actually, so much attention has been dedicated to management of data sets volumes that it dimmed our perspective of what we really want to achieve. What we should focus on is obtaining fast and efficient results, if possible in real-time. The goal of this new paradigm, called Big calculations, is optimizing data-intensive computations using the knowledge of time provided and computing resources at our disposal. Almost abandoned HPC paradigm has left the shadow in which MapReduce has put it to, and started to merge with it in many terms. On the first sight slightly different paradigm has started to give the scientists ideas how to blend "best of the both worlds" [25]. Instead of providing high performance of a single powerful machine, the aim of "blended world" is to manage workloads and provide high performance over the whole cluster of machines using higher-level abstractions, which is more challenging but still attainable goal. These higher-level abstractions have a tendency of adding a new layer above existing big data stack components like Distributed File System (DFS) and Resource Management, implementing their own version of job scheduling to support MapReduce model. Examples of such layers are MPI, Spark, Storm, etc. The main supported features of hybrid paradigms, e.g. Big Calculations, should be failure resistance, out-of-the-box scalability and data locality exploitation, but also a natural way to execute cpu-intensive calculations.

In our research we have encountered many open-source projects that are using hybrid approaches. The motivation of each research was the fact that cloud system potential was not fully exploited by its respective research area community. Some of the researches we analysed were:

- In [28] K-Means clustering algorithm was implemented using Hadoop, Mahout, MPI, Python scripts, Harp and Spark, and a comparison in terms of execution time and efficiency for the same amount of work was made. For the biggest data set Spark framework was superior over other candidates.

- Parallel implementation of Monte Carlo sampling algorithm using Scala [38] showed the need to implement CPU-intensive algorithm, such is Monte Carlo, using large-scale data-intensive computations platform.

- In [26], MapReduce programming model was used to overcome the limitations of Genetic Algorithm population size.
- MRPGA [29] is a MapReduce framework for automatic parallelization of Genetic Algorithm. MPRGA showed us the benefits of using MapReduce programming model for HPC applications, and also an urgent need for framework over MapReduce model for CPU-intensive applications.
- MapReduce implementation of Simulated Annealing algorithm [48].
- MapReduce implementation of Large Neighbourhood Search [35].
- MapReduce implementation of Ant Colony Optimization Approach [72].
- MapReduce implementation of Max Min Ant System [64].
- MapReduce implementation of Cucko Search [14].
- Apache Spark Machine Learning Library [59].
- A Library to Run Evolutionary Algorithms in the Cloud Using MapReduce [17].
- A Framework for Genetic Algorithms Based on Hadoop [18].
- EvoSpace-Interactive [20], an open source framework for the development of collaborative-interactive evolutionary algorithms.
- Although common data-intensive computing algorithms require independence between data and computing functions on different nodes, in some cases synchronization between nodes is necessary. For instance, in swarm environments and cooperative iterative algorithms. Usually implementation of such systems in MapReduce fashion is done using a sequence of supersteps, at which nodes exchange messages. Some implementation examples are Google Pregel [36] and Apache Spark Bagel [4].
- Parallel Metaheuristics: Recent advances and new trends [2]. Good analysis of parallel algorithms and communication between instances.

Cited projects and research papers (but also many more not listed) and the results which were presented gave us a clear idea about the need for a framework that combines high-performance computing algorithms with large-scale data-intensive computing infrastructures. That kind of framework should provide users:

- **Easy deployment to a computing cluster** - A framework should adapt to computing infrastructure, requiring from the user only a minimal setup of environment variables.
- **Straightforward implementation of computing algorithms** - The algorithms may be light-weight or cpu-intensive, depending on desires of end users. It is important that productivity of a developer is not influenced by setup distractions.
- **Setup of computation time limit** - Sometimes people need to make a business decision that can not wait for long time. In some cases it needs to be provided in real time.
- **Easy way to scale parallel instances of algorithms over the cluster** - Instantiate instances in a way transparent for a user, e.g. by providing a single parameter (an integer number) to the framework.
- **A way to distribute the initial parameters to parallel instances** - Initialisation of parallel instances might not be the same for all of the algorithms ran, since, in many cases, it might not give enough diversity in exploration of solutions search-space.
- **A way to aggregate the results of parallel instances** - Aggregation should be provided as an operator of a framework, that user can easily set or change. For example, to get the minimum/maximum of all parallel solutions.
- **A way for parallel instances to communicate their results during the computation execution** (if they require such a feature).
- **No limitations on data set size**
- **Out-of-the-box fault tolerance support** - The user might want to have failed-computations restarts supported, but (s)he might not be an expert in handling such situations.
- **High-performance** - Provide fast and efficient results. If possible, computations should be executed in-memory. A user should only care about the efficiency of his/her own algorithms. Framework should ensure that increasing the scale of parallel computations will yield better results, if the results were aggregated in the right way by the user.

We have analysed the requirements posed by HPC and big data communities, and implemented a framework, called HyperSpark, which tries to fulfil all the demands stated above. The framework is presented in the following chapter.

CHAPTER 3

HyperSpark : A Framework for Scalable Execution of Computationally-Intensive Algorithms over Spark

After a cautious and thorough analysis of requirements posed by High Performance Computing (HPC) and Data-intensive computing communities (listed at the end of Chapter 2), we developed a framework that addresses all of the requirements and provides all the features in an extensible and composable way. Initially the framework was conceived as a tool that combines approaches to distributed problem solving to effectively execute user-defined algorithms. The starting point was a concrete problem that came from a set of finite capacity scheduling problems [40]. Primary objective was to provide users a simple and natural way to implement and scale their algorithms over the cluster of commodity hardware. During the development process we learned how to extend and make more general software structure that supports solving various types of problems and provides the communication between the distributed algorithms. Framework development posed many challenges that were resolved with a number of design choices. This chapter motivates every such choice, showcases the implementation of the framework and presents some usage examples.

Chapter 3 is organized as follows. Section 3.1 introduces fundamental concepts necessary for problem solving using HyperSpark. Next, Section 3.2 presents the challenges of using distributed environment addressed by the current version of the developed framework. Section 3.3 discusses technologies considered

for the framework implementation and related design choices. Afterwards, Section 3.4 provides a high-level workflow of HyperSpark framework. Finally, Section 3.5 describes the internal classes and function signatures used to support the presented workflow.

3.1 Fundamental Concepts

This section is intended to explain and illustrate the building blocks for the framework used to model arbitrary, distributed, computationally-intensive applications that are easily written without the prior knowledge of the underlying, in most cases complex, distributed software environment. Therefore, in order to form a sound foundation for the development of HyperSpark framework we first need to define the basic concepts of distributed problem solving.

- **Problem** - We define a problem as a set of inputs needed to execute some particular (user-defined) algorithm. In mathematics every problem has to have a clear definition in terms of parameters and constraints. From object-oriented design perspective, *Problem* can be defined as a class which encapsulates all of the variables (parameters and constraints) necessary for its solving.

- **Solution** and **Evaluated Solution** - A solution to a specific problem might take different forms. When solving equations one needs to find the value of a single unknown variable, or values of multiple unknown variables. In scheduling optimization problems, like Travelling Salesman Problem (TSP) [54], a traveller has to visit a list of provided cities, usually optimizing the total distance travelled. In TSP the solution represents the optimal order of cities visited. Sometimes a user may be interested only in the order of cities, but sometimes one might also want to know what is the cost of such a schedule in terms of distance travelled, time, money or gasoline spent. Therefore, we have introduced two levels of solutions that can be obtained. *Solution* represents a descriptive solution, like the order of cities in the case of the TSP. *Evaluated Solution* is a *Solution* with an additional evaluation based on some criterion like distance travelled, time, money or gasoline spent.

- **Solution Space** - In optimisation problems a solution space, search space, candidate set or feasible region represents a set of all possible solutions that satisfy the problem's constraints. In TSP there is a constraint that every city must be visited, and it must be visited only once. Therefore, a

solution for the TSP is a permutation of cities, and its solution space is a set of all possible permutations.

- **Algorithm** - A procedure or methodological approach that solves some specific problem provided. In MapReduce paradigm, algorithms are usually simple stateless applications consisting of a sequence of map and reduce functions. In High Performance Computing, an algorithm is often a resource-intensive procedure that solves particularly hard problem. Such procedures may require to store their own state, independent from the parameters of the problem, like algorithm execution time limit, initialisation variables, and so on. Note that, in automatically provisioned distributed environment, an *Algorithm* is also considered as "data" that must be serialized and distributed to the computing nodes.

- **Seed** - Sometimes an algorithm may require a *Solution* as a starting point for executing the computation. The solution considered as a starting point is called a *seed*. A seed can be provided at the initialisation phase of an algorithm (a.k.a. an initial seed), or in later phases of its computation (e.g., for path guidance). An algorithm performs transformations on a seed, or based on a seed, to guide the execution towards a satisfying result. An example of a seed, in vehicle routing problems, is a position of the vehicle. Based on its initial position (a starting point), the routing system will commence the calculation of the path towards the desired end point. To avoid collision with other vehicles/objects, the routing system can suggest the next position to the vehicle during its movement.

A typical usage of a distributed environment for problem solving is to employ multiple parallel algorithm instances to solve the same, usually hard, problem. Big data does not always need to be represented in terms of physical files (e.g. textual records) placed on a disk. It can also refer to a more abstract concept of the solution space of a problem. The optimal solution(s) of some problem might belong to an abstract solution space with an encoding reaching terabytes of data. One idea is to use different algorithm initialisations in order to divide solution space among the parallel algorithm instances, such that each algorithm, during the computation, explores different solution subspaces. During the computation the algorithms may communicate their best results (bases for next algorithms' iterations) to check if they are exploring subspaces with good candidate solutions, i.e. that its computation will provide a satisfactory solution. If the solutions obtained in that subspace are not satisfying, the algorithm can take a better result provided by some of the other algorithms to continue its execution.

3.2 Challenges Addressed With the Current HyperSpark Version

Let us imagine the following situation: A user has a problem and a single-core algorithm that solves it. There are several reasons why the user should execute the algorithm in a distributed computing environment:

- **Time efficiency** - The user has to repeat an experiment (a single-core algorithm) N times and to take the best result obtained. Multiple, sequential executions of an algorithm on a personal computer might take a lot of time (N x experiment time), enough to consider renting a cloud cluster and learning basics of some simple framework for distributed execution. Distributed environment can turn a single-core algorithm into a distributed one and execute it on a cluster of computers more or less seamlessly by defining a couple of new operators and environment configuration properties. The total time of N repetitions would be reduced from N x experiment-time to 1 x experiment-time.
- **Computation execution in a distributed environment provides better results** - Execution of a single-core algorithm shows satisfying results, except in some runs of the algorithm. A small percentage of algorithm executions provided really bad results, the effect which would be easily removed by using aggregation of results of several independent executions.
- **One machine is not enough to process big data set using user's algorithm** - Even using a full capacity of a powerful multi-core machine is not enough to perform computation over a big data set in a reasonable time. Large data set can be split in several smaller data sets over which computations can later be performed in parallel on independent machines.
- **Advanced Analytical Ability** - Computation execution in a distributed environment provides greater analytical ability, since the number of parallel algorithm instances can be increased to a far larger scale, comparing to the case when a single powerful machine is used. Increasing the computing power directly raises the scale for analytical computations, providing the user more possibilities for the advanced analysis.

Even though distributed environment introduces many benefits, it also introduces some challenges that need to be addressed.

- **Hardware failures** - Due to its low cost, commodity hardware is highly available and therefore typically used to build computing clusters. How-

ever, they are more prone to hardware failures than some more expensive alternatives. Hardware failures need to be handled by using computation restarts, data replication and other fault-tolerant schemes. Handling failures is hard, and it requires a lot of time and technical knowledge. Usually the users want to have a fault-tolerant system, but without the need to be involved in a complex implementation. HyperSpark relies on the fault-tolerance implemented in Spark core, engineered by Amplab in Berkeley and maintained by a part of the open-source community. As such, it provides users seamless fault tolerance support, a simple programming model, and a decreased overall complexity.

- **Setup of the environment** - Software setup of hardware machines often requires high technical expertise, but also the time invested in a process. Complex environment setup is undesirable, and therefore many non-IT organizations often turn to big data companies which offer cluster computing environments as a service, i.e., the so called cluster as a service (CaaS). Affordable pricing models and pre-configured software environments (e.g. MapReduce paradigm support through MPI/Hadoop/Spark with underlying Resource Manager like YARN) make CaaS an attractive choice for overcoming this difficulty. CaaS requires only a minimal set of configuration properties to be provided by the user. On the other hand, there is always a possibility to manage underlying hardware infrastructure and its setup by the company itself, but this choice is typical only for IT organizations.

- **Splitting the data set** - Processing data locally is a key to efficient computation. In order to process the data in a distributed fashion, the data set must be pre-divided and distributed with user algorithms to the computing resources. It is desirable that the division of large data sets is performed automatically by the framework (through native support in Hadoop and Spark case, or software libraries like DataMPI [11] in MPI case), with a prerequisite for the user to specify an input format that is going to be used when splitting of the data set is performed.

- **Distribution of data and compute-functions** - One of the basic requirements of distributed environment is to have independent data and compute functions collocated on computing nodes, so the locality principle is used as much as possible. Placing all of the data and copying executables manually on computing nodes is undesirable and usually avoided. In the ideal case, the distribution of data and computation should be handled by the framework. In that case, the user does not need to directly program messages and topologies. HyperSpark uses an abstract distributed

data wrapper to merge data and algorithms in one data structure, which is later going to be serialized and distributed automatically by Spark, using the concept of RDD. More on this will be said in Subsection 3.4.1.

- **Aggregation of results** - In order to produce meaningful output the framework must provide means for collecting and combining the individual outputs from each distributed computation. Such a process is called aggregation and it is typically a problem-specific computation. Therefore, a framework needs to allow the user to specify a custom computation to be used as aggregation. For instance: +, -, ==, !=, >, < operators can be used within a compute function to accumulate, compare or filter the results.

- [**Cooperative computation**] - Although many distributed algorithms are independent and require only the basic data and compute distribution before running and later results aggregation, there are algorithms that use cooperation during the execution to make a decision, change a search procedure or search direction etc.. Therefore, a framework for distributed computation needs also to support cooperation of different algorithms during the execution. HyperSpark is designed to support two modes of algorithms execution, presented in Figure 3.1. For simplicity, the master is drawn twice, but it represents the same entity. The first case, on the left, executes the algorithms as independent parallel tasks, without the need to cooperate (to exchange messages/solutions) with other algorithms during the execution. Second case, presented on the right, requires cooperation between the algorithms during their execution. Algorithms are processing the data until they reach a synchronization point in which they (not necessarily) send messages towards other parallel instances. One such a phase is often called a *superstep* of execution. During the provided time for execution of algorithms, algorithms may perform several phases of execution. It can be deduced that a case on the left side of Figure 3.1 can be considered as one *superstep* of the cooperative case on the right. Therefore, the second model is more general and it is adopted in HyperSpark framework.

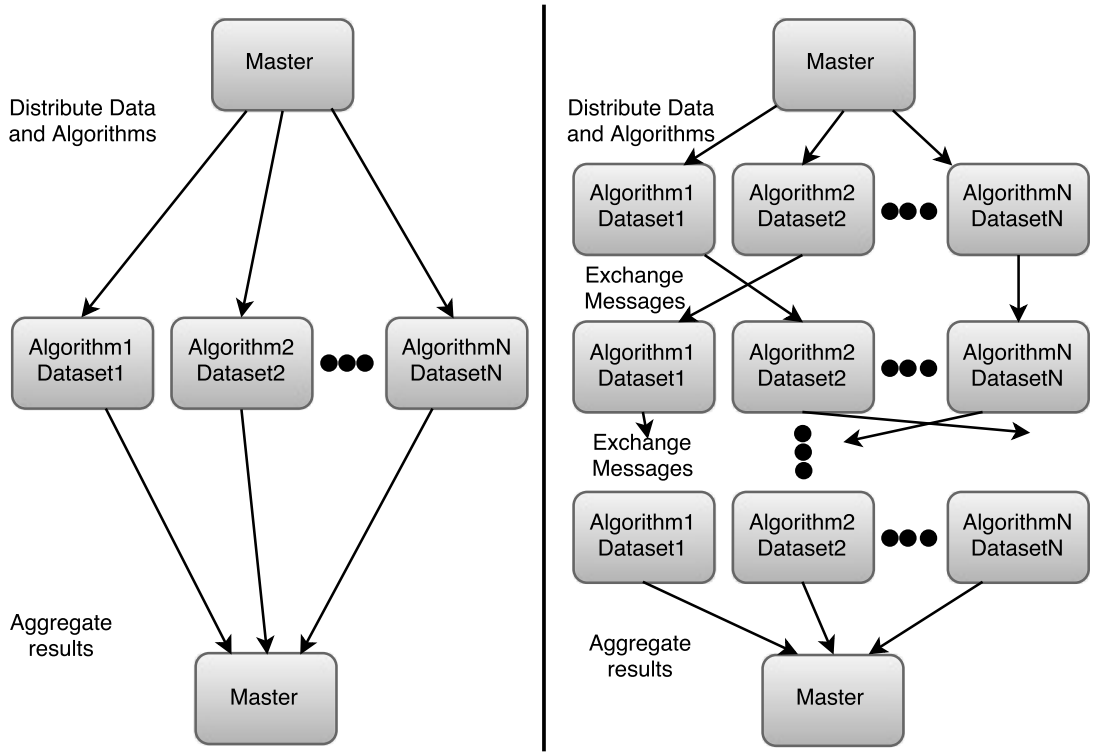

Figure 3.1 – Execution models: Without (on the left) and with (on the right) messages exchange

3.3 Technologies used

In the early planning phase the choice of technology to support Big Calculations paradigm had to be made. As we mentioned in Section 2.5, Big Calculations paradigm appeared as a need of IT society to use data-intensive computing platforms for CPU-intensive computations and also HPC systems for executing more parallel data-intensive tasks. During the planning phase, technologies considered were MPI / OpenMP, Hadoop and Spark. Some technologies were more suited for HPC appllications (MPI/OpenMP), and some for large-scale data-intensive computing (Hadoop, Spark). The main differences between those frameworks reflect in: **1)fault tolerance support**, 2)**data replication** and **3)computational speed**. These factors can not be observed separately since they influence each other. In following, we list the advantages and disadvantages of each technology and motivate our particular choice.

MPI/OpenMP efficiently exploits multi-core cluster architecture and shared memory multiprocessing [56], probably most effectively out of all three tech-

nologies. It is oriented towards High Performance Computing (HPC) applications, but it has a couple of significant disadvantages. First, it does not support fault tolerance, meaning that if a computing node fails the partial results will be lost indefinitely, leading to incorrect solutions, or not providing solutions at all. This is unacceptable, especially in the case where the cluster used consists of commodity hardware and each component has a relatively high probability of failing. The second disadvantage is a lack of support for data replication. Even if some computing nodes fail, the nodes containing replicated data will continue their execution and the aggregated results will still be accurate. Although there were successful attempts made by third-parties to support these two factors [68] [70], MPI/OpenMP core still does not provide that kind of support. Also, in MPI/OpenMP the computing nodes involved in data distribution (MPI_Bcast, MPI_Send primitives) and data aggregation (MPI_Recv) need to be stated as participants of a communication group, and to be addressed explicitly through MPI primitives every time when a communication is employed, making the development harder than usual.

Apache Hadoop and **Apache Spark** are both suitable for large-scale data-intensive computing, while for high-performance applications they are still in the research phase. It can be said that their performance depends on the specific HPC application. They provide remarkable support for fault tolerance and data replication, and everything is done in a transparent way for a user. The ease of use comes at some performance cost incurred by automatic parallel task scheduling, data replication distribution and aggregation, failure recovery... Hadoop is good for batch applications and offline data analysis since its interaction with data is constrained to use persistent storage. Spark, on the other hand, offers in-memory data processing that minimises the interaction with the persistent storage. Spark uses persistent storage during the initial reading of the data and if data volume is larger than the available memory of the cluster. In other words, Spark keeps a part the data set necessary for the current computation in the memory, while any data that could not fit is kept in the persistent storage. Somewhat similar to the virtual memory paging, Spark materializes data from the persistent storage when needed by the computation. The primary objective of HPC applications is to provide high performance, but they may not require handling extremely large amount of data. Given these advantages, it can be concluded that Spark is a better fit for executing HPC applications than Hadoop. As it is stated on its official website: Spark performs from 10 to 100 times better than Hadoop for specific applications. Another advantage is that data distribution and aggregation are done automatically by both frameworks, and can be partly manipulated by technology's internals.

After taking all three previously mentioned factors into considerations, we

decided to adopt Spark as a development technology. We were interested in a simple and fast programming model, which supports fault tolerance and data replication, and which automatically distributes calculation without requiring direct programming of communication messages and topologies. At the same time we wanted to investigate what is the exact performance overhead that these features will introduce. As the computation is done in memory, there is definitely a possibility for high performance execution, and the management overhead of parallel tasks is insignificant for long-running jobs.

Spark offers support for three programming languages: Python, Java and Scala. Therefore, we needed to choose one particular programming language to develop a Spark-based framework. Among the candidates, Scala dominated for several reasons:

- Spark is developed in Scala, and it's the main language used at the Berkeley AMPLab. To understand what happens internally in Spark during the execution, the user needs to understand Scala principles.
- Scala adopts object-oriented and functional paradigms which allows very efficient and concise implementation.
- Scala is a popular, new, emerging programming language with a large community support.
- Scala inherits Java portability due to the JVM-based runtime.
- Scala outperforms Java in execution time, conciseness and memory footprint [27].
- Scala outperforms Python, based on the comparison of the fastest benchmarks [47].

As Scala is proven as more efficient than the other two candidates, we decided to develop HyperSpark framework using a Scala/Spark technologies combination.

3.4 Architecture

Apache Spark is started and initialised by passing a SparkConf configuration object to SparkContext. In Spark transformations do not trigger any execution until an action is called. Similar model is adopted in HyperSpark architecture - there is a FrameworkConf object responsible for Framework setup. Framework execution is not started until a run function is called, with attached configuration object. The workflow of HyperSpark is presented in Figure 3.2.

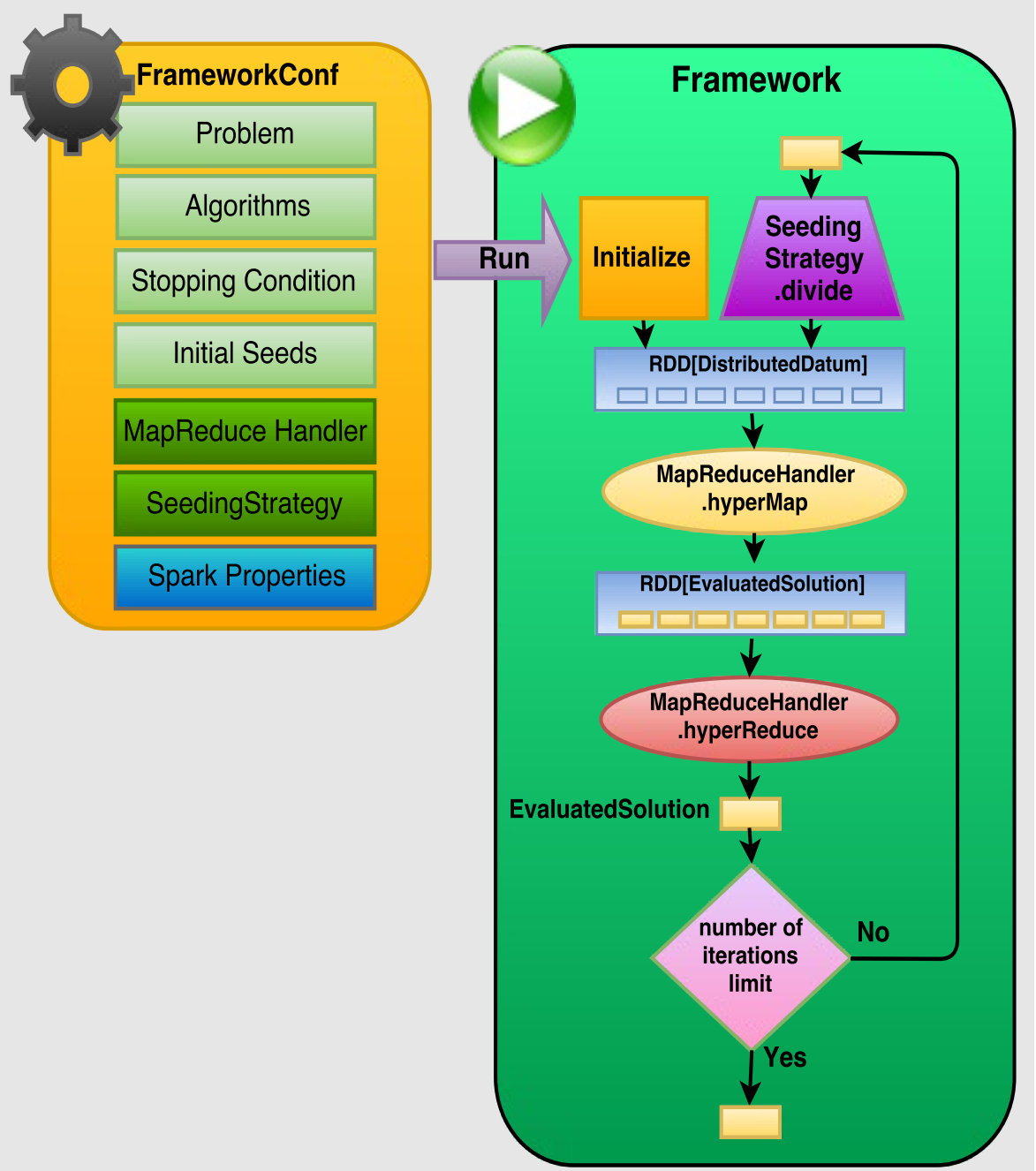

Figure 3.2 – Workflow of HyperSpark Framework

3.4.1 Framework

Framework is the main figure in this work. It represents a joint of problem solving ability and Spark's ability to execute any action in a distributed manner - it is a starter of distributed problem solving engine. In HyperSpark project the *Framework* is implemented as a Scala object, which is similar to the singleton design pattern in object-oriented design. It means that only one instance of the Framework class can be created.

There are two types of run methods exposed by the Framework, used to trigger the execution of parallel computation:

- ***run*** and
- ***multipleRuns***.

Since the limitation of Spark is to use only one instance of SparkContext during the runtime of application, the user should tell HyperSpark whether (s)he wants one run or multiple runs of the framework. When a *run* method is invoked Spark environment is initialized (SparkContext is created), and initialisation might last some time {details on the performance are given in Chapter 5. Upon the end of computation Spark environment is stopped (SparkContext is destroyed). If a user wants to run the same experiment several times (to obtain multiple results), (s)he might not like the fact that there is a long waiting time present between different runs, since it would require to re-initialize the Spark environment. That is why *multipleRuns* function is provided. When *multipleRuns* is called, Spark environment is started, multiple runs of an experiment conducted, and finally Spark evironment is terminated.

```
def run(conf:FrameworkConf): EvaluatedSolution
def multipleRuns(conf:FrameworkConf, runs:Int): Array[EvaluatedSolution]
```

Figure 3.3 – Triggers for Framework Execution

To call one of the run methods declared in Figure 3.3, a user has to provide a framework configuration represented by a *FrameworkConf* object. When a user calls a run method of the framework, the *Framework* is initialised with parameters present in *FrameworkConf*. After that, the Framework enters the loop of execution presented in Figure 3.2.

DistributedDatum is an abstraction which contains an algorithm and the data necessary for its execution. In each loop iteration we form an RDD of *DistributedDatum*s to distribute the data and computation towards computing resources. We provide more details on the DistributedDatum data structure in Section 3.5 - Framework Internals. The loop presented in Figure 3.2 is named ***hyperLoop***, and one its iteration consists of the following phases:

- **hyperMap** is similar to map transformation used on Spark RDDs. It transforms the RDD of *DistributedDatum* objects into an RDD of objects representing evaluated solutions. Each evaluated solution is obtained as a result of running an *algorithm* from an appropriate *DistributedDatum*. Being a collection of Spark transformations itself, hyperMap phase is executed in a lazy fashion, i.e., algorithms are not actually executed at the moment of invocation. Instead, the execution plan ((Directed Acyclic Graph) is updated in the anticipation of a Spark action.

- **hyperReduce** - is similar to reduce action used on Spark RDDs. It is used as an aggregation operator for combining multiple evaluated solutions obtained from the parallel algorithm execution. How the reduction (aggregation) is performed depends on a user specific implementation, or default behaviour (minimisation) in the case it is not overriden. Every maximisation problem can be converted to a minimisation problem, and vice versa. We adopted minimisation as an equally general operator. *hyperMap* and *hyperReduce* form an execution plan, shown in Figure 3.4, which is lazily evaluated when *hyperReduce* method is invoked.

- **Solution distribution** - Once hyperReduce returns an *EvaluatedSolution*, whether it is going to be provided to the user or used to run another iteration (consisting of hyperMap and hyperReduce) depends on a run method used to start the Framework. If a *run* method is called, the solution is going to be provided to the user and the system will shut down the Spark environment. On the other hand, if multiple runs are requested, the Framework needs to distribute the aggregated *EvaluatedSolution* from current run execution to multiple parallel computing resources in order to start the execution of the next parallel run. Hence, it needs to create an RDD of *EvaluatedSolution*s based on one *EvaluatedSolution* provided. A user may want to use different strategies, as sending the same evaluated solution (a seed) to all of the parallel algorithm instances, or to use some transformation on the solution to obtain multiple different solutions - employing the parallel algorithms to explore different solution subspaces. Later on in this chapter we refer to solution distribution strategy under the name *SeedingStrategy*.

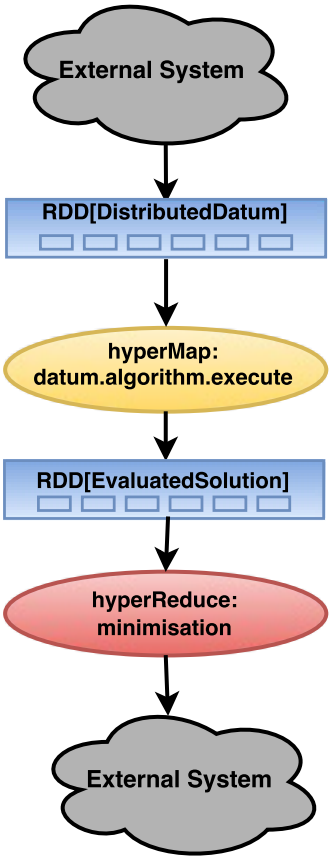

Figure 3.4 – Directed Acyclic Graph of *hyperMap*'s and *hyperReduce*'s Default Behaviour

By considering the flow of execution in *hyperLoop*, we might say that one hyperLoop iteration (consisting of *hyperMap* and *hyperReduce*) represents a *superstep* of the computation [1], and that a *SeedingStrategy* implements communication among parallel algorithms which enables cooperation.

3.4.2 Framework Configuration

Framework configuration presented in Figure 3.2 consists of several parameters that need to be defined before the Framework can be executed.

- **Problem** - A problem that needs to be solved by the framework. It has to have a clear definition in terms of parameters and constraints. In Hyper-

[1] superstep concept was introduced in Section 3.2, in the part *Distribution of data and compute-functions*

Spark a *Problem* is an abstract class which should encapsulate all of the variables - parameters and constraints, necessary for its solving. An abstract class enables extensibility for future usages. We provide a concrete implementation of one specific problem, and a set of algorithms that solve it. The details about Framework internals can be found in Section 3.5.

- **Algorithms** - An array of algorithms that are going to run in parallel to solve provided problem. The size of this array determines the level of parallelism to be set inside of HyperSpark framework (e.g., 4 algorithms map to 4 parallel computing instances). Algorithms do not necessarily need to be of the same type, meaning that one may choose different algorithms to solve the same problem and instruct the framework to take the best result accomplished.

- **Stopping Condition** - This is the stopping condition (e.g., execution time) that is going to be provided to each instance of algorithm, and it is the same for each parallel instance. Different algorithms' stopping condition introduce higher deviations of the result and large synchronization times, since aggregation cannot be performed until the last algorithm finishes its execution. We implement the stopping condition as an abstract class named *StoppingCondition* and provide a concrete implementation of time-based stopping condition, named *TimeExpired*.

- **Number of Iterations** - Setting this parameter is optional, and its default value is one. One iteration of HyperSpark framework corresponds to a parallel, independent execution of the encapsulated algorithms. When the algorithms require cooperation, the number of iterations needs to be set to a value greater than one. The communication (solution exchange) is performed through a series of synchronisation points defined by the number of computational *supersteps*, i.e., iterations. When the stopping condition is time-based the total algorithm execution time corresponds to the multiplication of the execution time for one iteration with the *Number of Iterations*.

- **Initial Seeds** - An array of optional initial solutions provided to each instance of algorithm at the start of parallel execution. Some algorithms may require an inital solution (starting point or a seed) to base the execution on. In combinatorial optimisation, the seed has a purpose of guiding the algorithm exploration towards a biased direction in a solution space. *Initial Seeds* parameter separates initial seeds from the seeds exchanged after the synchronisation (at the end of each superstep, when the communication between the algorithms is performed). The reason for intro-

ducing an additional parameter is that some algorithms may require an initial solution (a seed), but not cooperation afterwards. This feature is also convenient when the user wants to provide an initial solution to each algorithm, but later (s)he expects from the framework to automatically manage the solution exchange between algorithms (see SeedingStrategy below). If not required, the initial seeds can be omitted.

- **Spark Properties** - A simple associative array of (key,value) pairs passed to Spark environment. There are plenty of properties provided by Spark [60], but the basic ones that need to be set are cluster URL (also called master URL) and application name. A full list of Spark properties used by HyperSpark is given in the appendix.[link here]

- **MapReduceHandler** - As stated earlier, each iteration of the HyperSpark framework performs computation in two phases. Once *hyperReduce* is triggered, the framework performs *hyperMap* transformation over provided RDD of DistributedDatum and produces an RDD of *EvaluatedSolution*s. Afterwards, hyperReduce aggregates produced results. By default, in hyperMap phase the Framework starts the parallel execution of algorithms over a provided problem, and in hyperReduce phase it takes the EvaluatedSolution with minimal value. This behaviour can be overriden by setting a custom MapReduceHandler.

- **Seeding Strategy** - Once the result from one HyperSpark iteration is obtained, how this result (*EvaluatedSolution*) is distributed towards multiple parallel algorithm instances is determined by a Seeding Strategy set in *FrameworkConf*. Therefore, a *SeedingStrategy* generates multiple solutions based on a single solution provided. Being the most general strategy for seeding, sending the same solution to all of the parallel algorithm instances is adopted as the default behaviour of HyperSpark framework.

3.5 Framework Internals

Up to now, all high-level concepts necessary for understanding HyperSpark's workflow were explained in a very abstract way. Now, it should be easy to understand low-level details of implementation.

3.5.1 Problem

```
abstract class Problem() extends Serializable
{
  def evaluate(s: Solution): EvaluatedSolution
}
```

Figure 3.5 – Problem Class

Problem class (Figure 3.5) should contain all parameters and constraints necessary for its solving. The main method in this class is evaluate, which accepts a Solution, evaluates it and returns the EvaluatedSolution. We already mentioned that in scheduling problems the solution may have a value accompanied to it, e.g. the distance travelled next to the order of cities visited. The solution evaluation procedure of a particular problem does not depend on the algorithm that solves it. Rather, it is a problem-specific characteristic. Therefore, it is placed inside the *Problem* class.

3.5.2 Solution

As we discussed in Section 3.1, *Solution* a descriptive solution of a particular *Problem*. The descriptive representation is especially convenient in scheduling problems, where there is an ordered sequence present as a solution (e.g. the order of cities visited in TSP). *Solution* is an abstract class (Figure 3.6) which has a method for evaluation with a default implementation that calls the evaluate method from the Problem class in a Visitor-like fashion. An implementing class should only provide a class a concrete representation of the solution (e.g., citiesVisited = Array[Int] for the TSP).

```
abstract class Solution extends Serializable {

def evaluate(p:Problem):EvaluatedSolution = {
  p.evaluate(this)
}
override def toString = "abstract solution"
}
```

Figure 3.6 – Solution Class

3.5.3 EvaluatedSolution

EvaluatedSolution class (Figure 3.7) adds a value to the solution once it is evaluated. Because EvaluatedSolution already contains a value, it overrides evaluate function to return itself and prevent unnecessary computation over the solution.

```
abstract class EvaluatedSolution(
       val value: AnyVal,
       val solution: Solution)
extends Solution with Ordered[EvaluatedSolution] {

  override def evaluate(problem: Problem): EvaluatedSolution = this
  override def toString = {
    val solString = solution.toString()
    val str = "EvaluatedSolution(value:" + value + ", solution:" + solString
        + ")"
    str
}
}
```

Figure 3.7 – EvaluatedSolution Class

EvaluatedSolution also implements *Ordered* trait (interface) and overrides its compare function. This allows a user to use comparison functions like max, min, and so on. For instance:

List(evaluatedSolution1, evaluatedSolution2).min

Method min can be invoked on the list of evaluated solutions and it will use the custom compare function specified by the user.
Parameter *value* within EvaluatedSolution is of Scala type AnyVal, which is a base class for predefined value classes Byte, Short, Int, Long, Float, Double, Char, Boolean and Unit. Therefore, in a concrete implementation of EvaluatedSolution a user is free to use any of the mentioned subclasses.

3.5.4 Algorithm

Algorithm trait, presented in Figure 3.9, is a starting point for implementing custom user algorithms. Every algorithm that is going to be executed in parallel needs to extend it. Algorithm trait contains four *evaluate* function signatures (Figure 3.8) which are used to start the execution of algorithm in order to solve a specific problem.

```
  evaluate(p:Problem)
  evaluate(p:Problem, s:StoppingCondition)
  evaluate(p:Problem, seedSol:Option[Solution], stopCond:StoppingCondition)
  evaluate(p:Problem, seedSol:Option[Solution], stopCond:StoppingCondition,
      run:Int)
```

Figure 3.8 – Algorithm's Evaluate Function Signatures

```
trait Algorithm extends Serializable {
  protected var seed: Option[Solution] = None
  protected var runNo: Int = 1
  protected var random: Random = new Random(runNo)
  def evaluate(p:Problem): EvaluatedSolution
  def evaluate(p:Problem, stopCond:StoppingCondition): EvaluatedSolution
  def evaluate(p:Problem, seedSol:Option[Solution],
      stopCond:StoppingCondition): EvaluatedSolution = {
    seed = seedSol
    evaluate(p, stopCond)
  }
  def evaluate(p:Problem, seedSol:Option[Solution],
      stopCond:StoppingCondition, runNo:Int): EvaluatedSolution = {
    seed = seedSol
    this.runNo = runNo
    random = new Random(runNo)
    evaluate(p, stopCond)
  }
  def name = this.getClass.getSimpleName
}
```

Figure 3.9 – Algorithm Trait

The full source code of this trait is provided in Figure 3.9. The user is required to override first two *evaluate* methods, while the third and fourth are already implemented. *Algorithm* evaluates (solves) the *Problem* and returns *EvaluatedSolution*. Second signature sets limitation on algorithm's execution time

by introducing a stopping condition. Additional parameters in next signatures are an optional initial solution (a seed) and a parameter "runNo" used to set the random initialisation of Java's Random class. Algorithm class provides the Java Random object named *random* to the user who wants to implement an algorithm that has a random behaviour. If the user wants a finer control over the seeding of the random object the third and fourth method must be also overriden. An example of such situation follows.

A user executed an experiment and obtained the optimal solution, but there is no way to repeat the execution in the same way because the algorithm has a random behaviour. (S)he cannot prove that the new optimal result is found.

Java's Random class has a deterministic procedure for generating random numbers, and using the same parameter provided to it the random behaviour can be repeated.

3.5.5 StoppingCondition

The stopping condition, termination criterion or stop rule is needed to tell the algorithm when to stop its execution. Once started, an algorithm continues its execution until the stopping criterion is satisfied. Therefore, *StoppingCondition* presented in Figure 3.10 has only one method, *isSatisfied()*, which has to be implemented by the user.

```
abstract class StoppingCondition extends Serializable {
  def isSatisfied(): Boolean
  def isNotSatisfied(): Boolean = { ! isSatisfied() }
}
```

Figure 3.10 – Stopping Condition class

Execution time, number of iterations and solution fitness are only a few examples of stopping conditions that are used the most in practice. For the purposes of the use case presented in Chapter 4, we developed a stopping condition *TimeExpired* which is used to measure algorithm's execution time. Its implementation is presented in Figure 3.11. The method *getThreadTime()* returns the thread time - a CPU usage time of the calling thread. *initialiseLimit()* adds the time limit provided to the constructor and the thread time and saves the result in the variable *internalLimit*. The overridden method *isSatisfied()*, when invoked, checks if the current thread time has reached the internal limit saved.

```
class TimeExpired(timeLimitMillis: Double) extends StoppingCondition {
  private var internalLimit: Double = 9999999999999999.0

  private def getThreadTime(): Long = {
    val threadTimeNanos = ManagementFactory.getThreadMXBean()
    .getThreadCpuTime(Thread.currentThread().getId());
    val threadTimeMillis = threadTimeNanos / 1000000
    threadTimeMillis
  }
  def initialiseLimit() = {
    val threadTimeMillis = getThreadTime()
    val expireTimeMillis = threadTimeMillis + timeLimitMillis
    internalLimit = expireTimeMillis
    this
  }
  override def isSatisfied(): Boolean = {
    val threadTimeMillis = getThreadTime()
    if (threadTimeMillis > internalLimit) true
    else false
  }
def getLimit(): Double = { timeLimitMillis }
}
```

Figure 3.11 – TimeExpired class

3.5.6 DistributedDatum

As we mentioned in Subsection 3.4.2, the user needs to set the problem, the algorithms array, the stopping condition and the array of seeds in the framework configuration object before running the framework. When (s)he runs the *Framework*, it will take all objects present in *FrameworkConf* and initialise itself. Since the elements that are going to be used for parallel execution within an Executor need to be serialised in order to be sent over the network, there is a need to introduce some kind of abstraction that supports serialisation. Abstraction introduced is named "**DistributedDatum**" and its form is presented in Figure 3.12. DistributedDatum is a serialisable, container class used to store all the necessary data needed to run a single algorithm on a distributed node. In framework initialisation phase, for every algorithm supplied by the user to framework configuration one distributed datum is created. Together with the algorithm, a distributed datum wraps an id, an optional initial solution (a seed), and a stopping condition for algorithm's execution.

Why do we wrap an algorithm, a seed, and a stopping condition into a *DistributedDatum* and then operate with an RDD of DistributedDatums? In Spark

```
class DistributedDatum(ind: Int,
            alg: Algorithm,
            seedOption: Option[Solution],
            stopCond: StoppingCondition) extends Serializable {
  def id = ind
  def algorithm = alg
  def seed = seedOption
  def stoppingCondition = stopCond
}
object DistributedDatum {
  def apply(id: Int, algorithm:Algorithm, seed:Option[Solution],
      stopCond:StoppingCondition)= {
    new DistributedDatum(id, algorithm, seed, stopCond)
  }
}
```

Figure 3.12 – DistributedDatum - Distributed Data Abstraction

a common practice is to make an RDD of simple types (Int, Double, etc.) and then invoke transformations and actions on the RDD. By using classes inside an RDD a few extraordinary benefits are introduced. Using standard MapReduce approach (Hadoop or Spark) a compute function (an algorithm) sent to the computing resources (Executors) is the same for all parallel tasks running inside the computing resources. Also, the compute function is stateless. Object-oriented paradigm allows us to extend Algorithm trait and implement several different algorithms that solve the same problem. *DistributedDatum* will accept any algorithm which extends the *Algorithm* trait. An algorithm can keep its state within the encapsulated variables. When *run* or *multipleRuns* method of the framework is invoked, the application master tries to *parallelize* the provided array of *DistributedDatum*s. All fields of DistributedDatum abstraction are serialisable, which allows the master node to serialise every DistributedDatum and create an RDD of DistributedDatums. Finally, aplication master sends DDs to the slave computing resources (Executors) in the cluster. The introduction of *DistributedDatum* wrapper class enables placing of various, stateful compute functions (algorithms) on different computing resources which are going to perform computation in parallel, and try to solve the problem provided by the master node.

Creating an array of *DistributedDatum*s from an array of Algorithms, an array of seeds and a stopping condition is done inside the *DistributedDataset* Scala object (Figure 3.13). DistributedDataset takes previously mentioned parameters and creates an array of *DistributedDatum*s. Spark needs to parallelise a collection of data elements of the same type, and *DistributedDataset* creates such a

```
object DistributedDataset {
  def apply(numOfNodes:Int, algorithms:Array[Algorithm],
      seeds:Array[Option[Solution]], stopCond:StoppingCondition) = {
    var array: Array[DistributedDatum] = Array()
    for(i <- 0 until numOfNodes) {
      val datum = DistributedDatum(i, algorithms(i), seeds(i), stopCond)
      array :+= datum
    }
    array
  }
}
```

Figure 3.13 – Distributed Dataset

collection (*Array[DistributedDatum]*) using parameters provided. An example is shown in Figure 3.14.

```
1  val algorithms = conf.getAlgorithms()
2  val seeds = conf.getInitialSeeds()
3  val stopCond = conf.getStoppingCondition()
4  //algorithms array size will map to the number of parallel tasks
5  val numOfTasks = algorithms.size
6  //...
7  val dataset = DistributedDataset(numOfTasks, algorithms, seeds, stopCond)
8  //...
9  val rdd = sc.parallelize(dataset, numOfTasks).cache
```

Figure 3.14 – Partitioning of Data Collection

As it is stressed in Spark section, second parameter provided to *parallelize* function (line 9 from Figure 3.14) is very important, since it instructs Spark environment how many partitions to produce, and therefore, how many parallel tasks to run. Here, for each instance of DistributedDatum, one parallel task is started, and each datum maps one-to-one with algorithm instances. Hence, one parallel task will execute one algorithm.

3.5.7 MapReduceHandler

MapReduceHandler (Figure 3.15) instructs what should be done when a parallel task is executed over each DistributedDatum in RDD. By default, *hyperMap* instructs an algorithm to solve a problem and return EvaluatedSolution, whereas *hyperReduce* uses minimisation operator in the result aggregation phase to return minimal *EvaluatedSolution*. The user interested in a different behaviour

must extend the MapReduceHandler class and override *hyperReduce* method. Overriding *hyperMap* is forbidden, because it would completely change the purpose of the framework.

```
class MapReduceHandler {
  final def hyperMap(problem: Problem, d: DistributedDatum, runNo: Int):
      EvaluatedSolution = {
    d.algorithm.evaluate(problem, d.seed, d.iterationTimeLimit, runNo)
  }
  def hyperReduce(sol1: EvaluatedSolution, sol2: EvaluatedSolution):
      EvaluatedSolution = {
    List(sol1, sol2).min
  }
}
```

Figure 3.15 – MapReduceHandler

MapReduceHandler is used inside Framework's hyperloop function. From Figure 3.16 it can be seen that hyperMap is called inside a transformation "*map*" over an rdd, and hyperReduce is called inside an action "*reduce*" over the same rdd. CPU-intensive part, algorithm execution, is performed inside map action, when reduce action is triggered. Directed Acyclic Graph (Section 2.4), automatically constructed by Spark, based on this sequence is very simple, and it is the same as the one presented earlier in Section 3.4, when a high-level overview of the Framework has been described.

```
  rdd
  .map(datum => mrHandler.hyperMap(problem, datum, runNo))
  .reduce((sol1, sol2) => mrHandler.hyperReduce(sol1, sol2))
```

Figure 3.16 – One Iteration of Framework's Loop

3.5.8 SeedingStrategy

SeedingStrategy (Figure 3.17) manages the distribution of one solution to all of the parallel tasks. It is a trait that custom implementations of data distribution need to extend. Divide function tells the framework how to distribute an optional seed to N parallel tasks. UsesTheSeed is a simple query function that tells if the seed is used for cooperation purposes or it is discarded.

```
trait SeedingStrategy extends Serializable {
  def divide(seed: Option[Solution], N: Int): Array[Option[Solution]]
  def usesTheSeed(): Boolean
}
```

Figure 3.17 – Seeding Strategy trait

There are a couple of seeding strategies already implemented shown in Figure 3.18.

```
class NoStrategy extends SeedingStrategy {
  override def divide(seed: Option[Solution], N: Int):
      Array[Option[Solution]] = {
    Array.fill(N)(None)
  }
  override def usesTheSeed(): Boolean = false
}
class SameSeeds extends SeedingStrategy {
  override def divide(seed: Option[Solution], N: Int):
      Array[Option[Solution]] = {
    Array.fill(N)(seed)
  }
  override def usesTheSeed(): Boolean = true
}
```

Figure 3.18 – Seeding Strategy Implementations

NoStrategy sends "None" seeds to all of parallel instances, and it means that initial solution (a seed) is not necessary for algorithm execution. SameSeeds sends the same seed to all of the parallel instances, meaning that all parallel instances are initialized with the same initial solution.

The way HyperSpark framework employs a *SeedingStrategy* is presented in Figure 3.19. Framework uses the seedingStrategy obtained from *FrameworkConf* configuration object to generate new seeds, and then it distributes them to parallel algorithms. If FIFO policy of scheduling parallel tasks is set (and it is by default), parallel algorithms will receive seeds in the order they were generated. Note that, if, by some chance, the user's implemented seeding strategy returned an array of seeds that is smaller than the number of algorithms, a runtime exception will interrupt the computation.

```
def updateRDD(rdd: RDD[DistributedDatum], seed: EvaluatedSolution):
    RDD[DistributedDatum] = {
  val numOfTasks = getConf().getAlgorithms().size
  val seeds = seedingStrategy.divide(Some(seed), numOfTasks)
  if(seeds.size < numOfTasks)
    throw new RuntimeException("Seeding strategy did not produce the correct
        number of seeds.")
  val updatedRDD = rdd.map(d =>
  DistributedDatum(d.id, d.algorithm, seeds(d.id), d.stoppingCondition))
  updatedRDD
}
```

Figure 3.19 – UpdateRDD before the start of subsequent iteration

hyperReduce and SeedingStrategy are powerful operators of HyperSpark framework, providing users a simple, effective and flexible way to distribute and aggregate data for their specific algorithms. There is no limit to creativity of an individual user when it comes to problem solving. By defining a single-threaded algorithm and a couple of new framework's operators, as hyperReduce and SeedingStrategy, a user can instruct HyperSpark to turn single-threaded algorithm into a distributed one. The algorithm can be light-weight or cpu-intensive, and can be different on each computing resource. This approach is completely new - up to our knowledge there is not anything like it in big data research literature. Also, in Appendix B (Users Manual) it will be shown how simple it is to make one application using HyperSpark, almost with no setup of Spark environment parameters.

3.5.9 Framework Execution

The source code of Framework's **run** method is presented in Figure 3.20. The Framework takes the configuration object provided and creates SparkContext, an RDD of DistributedDatums, and starts the *hyperLoop* to perform the defined number of computing iterations (supersteps).

```
def run(conf: FrameworkConf): EvaluatedSolution = {
  setConf(conf)
  //problem specific settings
  val problem = conf.getProblem()
  val algorithms = conf.getAlgorithms()
  val numOfTasks = algorithms.size
  val seeds = conf.getInitialSeeds()
  val stopCond = conf.getStoppingCondition()
  val iterations = conf.getNumberOfIterations()
  val dataset = DistributedDataset(numOfTasks, algorithms, seeds, stopCond)
  //spark specific settings
  val sparkConf = new SparkConf().setAll(conf.getProperties())
  if(notStarted){//allow only one instance of SparkContext to run
    sparkContext = Some(new SparkContext(sparkConf))
    notStarted = false
  }
  val sc = getSparkContext()
  val rdd = sc.parallelize(dataset, numOfTasks).cache
  mrHandler = conf.getMapReduceHandler()
  seedingStrategy = conf.getSeedingStrategy()
  //run the hyperLoop
  val solution = hyperLoop(problem, rdd, iterations, 1)
  solution
}
```

Figure 3.20 – Framework's Run Trigger Source Code

```
def multipleRuns(conf: FrameworkConf, runs: Int): Array[EvaluatedSolution] = {
  //up to this point
  //implementation is the same as in run method

  //run the hyperLoop
  var solutions: Array[EvaluatedSolution] = Array()
  for(runNo <- 1 to runs) {
    val solution = hyperLoop(problem, rdd, iterations, runNo)
    solutions :+= solution
  }
  solutions
}
```

Figure 3.21 – Framework's Multiple Runs Trigger Source Code

The only difference between *run* method and *multipleRuns* method is the way the *hyperLoop* is employed. Multiple runs method is presented in Figure 3.21. For clarity, the source code that was the same as in *run* trigger was

marked as omitted. The difference between two is only in the number of *hyperLoop* runs performed in the period between the initialisation of Spark environment and its termination.

3.5.10 hyperLoop

hyperLoop manages the flow of computation within HyperSpark framework. Its implementation is presented in Figure 3.22. Inside hyperLoop there is a recursive function "iterloop" present, which applies iteration of the framework, compares a result of iteration to the best result of all previous iterations, and updates RDD if there are subsequent iterations to be executed, otherwise it returns the best result found.

```
def hyperLoop(problem: Problem, rdd: RDD[DistributedDatum], maxIter: Int,
    runNo: Int):EvaluatedSolution = {

  var bestSolution: EvaluatedSolution = null

  def iterloop(rdd: RDD[DistributedDatum], iteration: Int): EvaluatedSolution
      = {
    //apply framework iteration to obtain the result
    val bestIterSolution = rdd
    .map(datum => mrHandler.hyperMap(problem, datum, runNo))
    .reduce((sol1, sol2) => mrHandler.hyperReduce(sol1, sol2))
    //compare the result with best results obtained
    //during previous iterations
    if(iteration == 1) bestSolution = bestIterSolution
    else bestSolution = mrHandler.hyperReduce(bestIterSolution, bestSolution)

    if(iteration == maxIter)//if it is last iteration
      bestSolution       //return best result found
    else {            //otherwise, update the rdd
      val updatedRDD = updateRDD(rdd, bestSolution)
      iterloop(updatedRDD, iteration+1)
    }
  }
  iterloop(rdd, 1)
}
```

Figure 3.22 – Framework's Loop Mechanism

The loop is initialized with an RDD of DistributedDatums and an iteration counter equal to one, and it runs until it reaches the maximum number of iterations allowed (maxIter), set through configuration object. It was already mentioned that the default value of "Number of Iterations" parameter is equal

to one, if not changed by the user. Iteration number equal to one means that the algorithm is distributed and executed in parallel without any form of cooperation.

Therefore, the difference between execution of parallel independent algorithms and cooperative algorithms (algorithms that require communication) is determined by the number of iterations performed by the framework. One might consider one iteration of HyperSpark framework as a *superstep* of computation. If the number of iterations is equal to one, after one iteration *hyperLoop* will provide a solution. If it is more than one, at the end of each superstep *hyperloop* will use previously provided RDD and create a new RDD with updated seed for next iteration. The way a seed is distributed depends on a way in which the divide function of a SeedingStrategy is implemented. Hence, SeedingStrategy can be used to create a virtual topology among the algorithms. It implicitly determines the way of communication between previous iteration algorithms and next iteration algorithms. If, for example, only third algorithm out of four needs to receive the message, SeedingStrategy might produce an array of optional seeds in this way: Array(None, None, Some(seed), None).

3.5.11 FrameworkConf - Framework Configuration class

For the end of internals description, we provide a list (Figure 3.23) of function definitions for problem-specific configuration of HyperSpark framework and a list (Figure 3.24) of function definitions for Spark-specific configuration.

FrameworkConf encapsulates an array of algorithms, an array of seeds, and other parameters like number of iterations, stopping condition, etc.. Get and set function names are mostly self-explainable. There are few which names might be ambiguous, and, just in case, they will be explained.

- ***setNAlgorithms*** populates the array of algorithms with N instances of the algorithm provided in the function signature.
- ***setNInitialSeeds*** populates the array of seeds with N instances of the seed provided in the function signature.
- ***setNDefaultSeeds*** populates the array of seeds with N instances of the *None* seed. Passing a None seed to an algorithm means that the algorithm does not require it. In the end, the seed, or initial solution, is optional.

```
def setProblem(p: Problem)
def getProblem()

def setAlgorithms(algorithms: Array[Algorithm])
def setNAlgorithms(algorithm: Algorithm, N: Int)
def getAlgorithms()
def clearAlgorithms()
def setStoppingCondition(stopCond: StoppingCondition)
def getStoppingCondition()

def setNumberOfIterations(n: Int)
def getNumberOfIterations()

def setInitialSeeds(seeds: Array[Option[Solution]])
def setNInitialSeeds(seedOption: Option[EvaluatedSolution], N: Int)
def setNDefaultInitialSeeds(N: Int)
def getInitialSeeds()
def clearSeeds()

def setSeedingStrategy(strategy: SeedingStrategy)
def getSeedingStrategy()

def setMapReduceHandler(h: MapReduceHandler)
def getMapReduceHandler()
```

Figure 3.23 – Framework Configuration - Problem-specific Properties

```
def setProperty(key: String, value: String)
def getProperties()
//for a full spark properties reference visit
//http://spark.apache.org/docs/latest/configuration.html

def setAppName(name: String)

def setSparkMaster(url: String)
def getSparkMaster()
//... a complete list is present in the appendix
```

Figure 3.24 – Framework Configuration - Spark-specific Properties

Spark properties passed to FrameworkConf in the format of (key,value) pairs are taken by the Framework when a run method is invoked, and forwarded to SparkConf instance. This specific line from run method (presented in Figure 3.20) does the whole job:

```
val sparkConf = new SparkConf().setAll(conf.getProperties())
```

Figure 3.25 – Framework Configuration - Passing Spark-specific Properties

Function signature ***setProperty(key: String, value: String)*** is the main function for setting any Spark property. All other functions presented in Figure 3.24 are just the auxiliary functions that keep the development of HyperSpark applications as simple as possible. For example, ***setSparkMaster*** (Figure 3.26) does the following:

```
def setSparkMaster(url: String) = {
  setProperty("spark.master", url)
  this
}
```

Figure 3.26 – Framework Configuration - Setting Master URL

From Figure 3.26 it can be seen that *setSparkMaster* function (and any other function that deals with Spark properties) internally uses *setProperty* function. *FrameworkConf* keeps a list of (key,value) pairs set by the user in its private variable. When *setProperty* is invoked, the private list of properties is updated and the instance of *this* (FrameworkConf) object is returned - on which the function setProperty was invoked. This approach is also adopted in implementation of *SparkConf* inside the Spark environment, and it enables an easy chaining of setter functions on a configuration object, e.g.

conf.setSparkMaster("local").setAppName("mySimpleApp").

Other spark-specific properties used by the framework, like the ones responsible for setting the cluster deployment mode, the number of executors and so on, are explained in Appendix A. User manual is provided in Appendix B.

CHAPTER 4

Case Study : Permutation Flow Shop Problem

In this chapter we propose and discuss the use of HyperSpark to solve the Permutation Flow Shop Problem (PFSP). PFSP is a hard optimisation problem [32] and therefore there are no efficient exact methods to solve it. Most of the research efforts were focused on the optimisation of single-threaded heuristic and meta-heuristic algorithms for PFSP. HyperSpark framework, presented in Chapter 3 aims at being a general, distributed tool, well suited for execution of long-running, CPU-intensive algorithms. Therefore, we decided to use HyperSpark as a base for examining the performance of distributed computation when applied to the PFSP.

This chapter is organized as follows. We start with the problem statement in Section 4.1. Next, Section 4.2 shows the computational complexity of PFSP problem. Further, Section 4.3 presents an overview of the most representative algorithms and enumerates the implemented ones. In the end, Section 4.4 describes the concrete implementations developed in order to solve the PFSP problem.

4.1 Problem Statement and Design Assumptions

In Permutation Flow Shop Problem (PFSP) scheduling there are m machines and n jobs, and the imposed assumption is that each job $\mathbf{J_i}$ **(i=1, 2, ..., n)** has to be processed on m machines $\mathbf{M_j}$ **(j = 1, ..., m)**, following the same order in all machines. Also, every job has to be processed on the machines in the same order (1, ..., m) [32]. Parameters given for this problem are the number of jobs **n**, the number of machines **m**, and the processing times of job j_i on machine M_j is $\mathbf{p_{ij}}$. To solve PFSP, we have to consider the following constraints:

- All jobs are ready for processing at time zero.
- The machines are continuously available from time zero onwards.
- At any time, each machine can process at most one job and each job can be processed on at most one machine.
- Once the processing of a job on a machine has started, it must be completed without interruption.
- Only permutation schedules are allowed (i.e. all jobs have the same ordering sequence on all machines).

Statement 1: Each job has to be processed by all machines and in the same scheduled order (permutation schedule). This constraint is common in real-life production environments. For example, consider a factory that produces fruit yoghurts. Engineers found a special recipe (a schedule) that improves the quality fo the final product. Each production batch (job) has to be produced on each machine by following the same addition order of ingredients (schedule) in order to get the desired quality. In general however, there are specific production constraints as the lack of interstage handling systems that impose the sequential production.

The final goal is to find a permutation of jobs that minimises (or maximise) one or more objectives. For PFSP there are many objectives proposed in literature: makespan, total flow-time, maximum tardiness, total tardiness, etc. As far as PFSP is concerned, the optimisation of one objective is far more common than the optimisation of several goals at the same time. We chose the *makespan* because it is by far the most common and it has an important meaning in industrial setting. As a matter of fact, the minimisation of this criterion is directly related with the maximisation of machines utilisation and reduction of the work-in-progress. In a nutshell, in PFSP processing starts from the first job (in a permutation schedule) on a first machine, and it ends with the last job on the last machine. Makespan is the completion time of last job on last machine. An example of a problem consisting of 5 jobs and 4 machines, with jobs permutation schedule (1, 2, 3, 4, 5) is given in Table 4.1.

Table 4.1
Execution time for each job (shown in columns) on each machine (shown in rows)

Times\Jobs	J1	J2	J3	J4	J5
p_{1J_i}	5	5	3	6	3
p_{2J_i}	4	4	2	4	4
p_{3J_i}	4	4	3	4	1
p_{4J_i}	3	6	3	2	5

Statement 2: Every job has to be processed on all machines in the same order (1, . . . , m). This means that machine 2 cannot start the processing of job until machine 1 finishes with it. Similarly, machine 3 cannot start processing of job until its processing on machine 2 is finished, and so on. Processing of particular job has different duration on every machine. In Table 4.1, processing times for the permutation schedule {J1, J2, J3, J4, J5} are shown. $\mathbf{p_{j,J_i}}$ (j = 1, ..., m; i = 1, 2, ..., n) denotes the time necessary to process i-th job in the schedule, J_i, on machine j. For instance, the processing time of the 2nd job in the schedule (J2) on machine 4, $p_{4,2}$, is equal to 6.

A graphical representation of this problem is shown in Figure 4.1 [46]. *Statement 1* and *Statement 2* are responsible for creating the dependency grid constituted of nodes and arrows. Processing starts from the job in the first position of a schedule (Job 1) on the first machine, and it ends with the completion of the last job (Job 5) on the last machine.

Definition 4.1.1. *Let $\pi = (\pi(1), \pi(2), ..., \pi(n))$ be a permutation of jobs and Π be the set of all permutations. Each permutation $\pi \in \Pi$ defines a processing order of jobs on each machine. We can define the PFSP as the problem of finding a permutation $\pi^* \in \Pi$ such that:*

$$C_{max}(\pi^*) = \min_{\pi} C_{max}(\pi) \tag{4.1}$$

i.e., the permutation that minimises the makespan.

The makespan $C_{max}(\pi)$ can be calculated as follows:

$$C_{k\pi(j)}(\pi) = max\ \{\ C_{k\pi(j-1)}\ ,\ C_{k-1\pi(j)}\ \} + p_{k\pi(j)} \tag{4.2}$$

under the following conditions:

(1) $\pi(0) = 0,$
(2) $C_{k0} = 0, \quad k = 1, 2, ..., m,$
(3) $C_{0j} = 0, \quad j = 1, 2, ..., n.$
(4) $C_{max}(\pi) = C_{m\pi(n)}(\pi)$

In simple words, the goal is to find a job schedule, i.e. a permutation of jobs, that minimizes the maximum completion time of all the jobs in the schedule. The completion time can be efficiently calculated using the recursive formula shown in Equation 4.2. It is not the only way but the formula presented above fits naturally into Spark/Scala paradigm.

Equation 4.2 calculates the completion time as a maximum completion time of the previous job on the same machine $C_{k\pi(j-1)}$ and of the same job on the previous machine $C_{k-1\pi(j)}$, plus processing time of the considered job $p_{k\pi(j)}$.

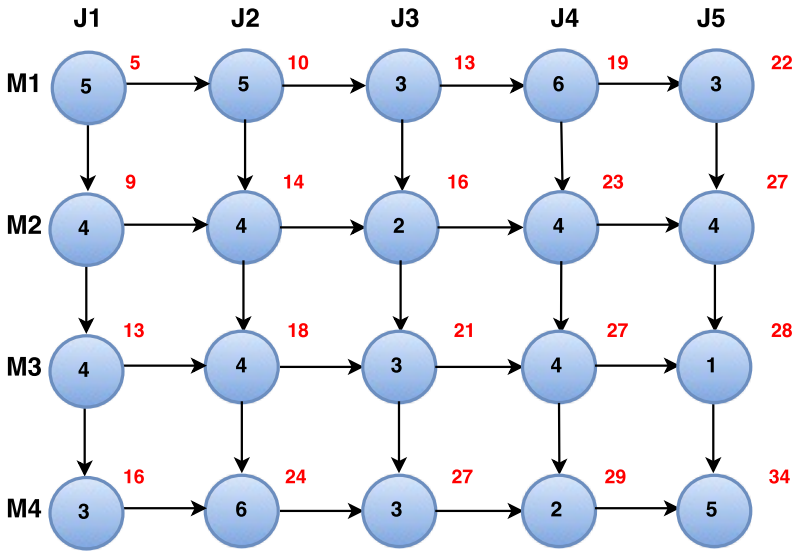

Figure 4.1 – Graphical representation of a PFSP example

In Figure 4.1, the execution times for each job on each machine (marked in red) is calculated using the Equation 4.2. Note that, conditions Equation 4.2 (1-4) are used to produce the boundary values.

Condition 4 allow to calculate the makespan value from the execution times matrix; it is the value placed in the bottom right corner of the matrix, and it corresponds to n-th job and m-th machine. Hence, makespan represents the completion time of the last job on last machine. In the example from Figure 4.1 it is equal to 34.

Figure 4.2 [46] presents the Gantt chart of the same example, which probably better depicts the problem of permutation flow shop scheduling. The gaps between two jobs on the same machine represent its idle times (when machine is not processing anything). The goal is to find the permutation of jobs that minimises the makespan, i.e., the completion time of the last job of the schedule and on the last machine. Minimisation of gaps decreases the makespan value, and therefore, maximises the machine utilisation.

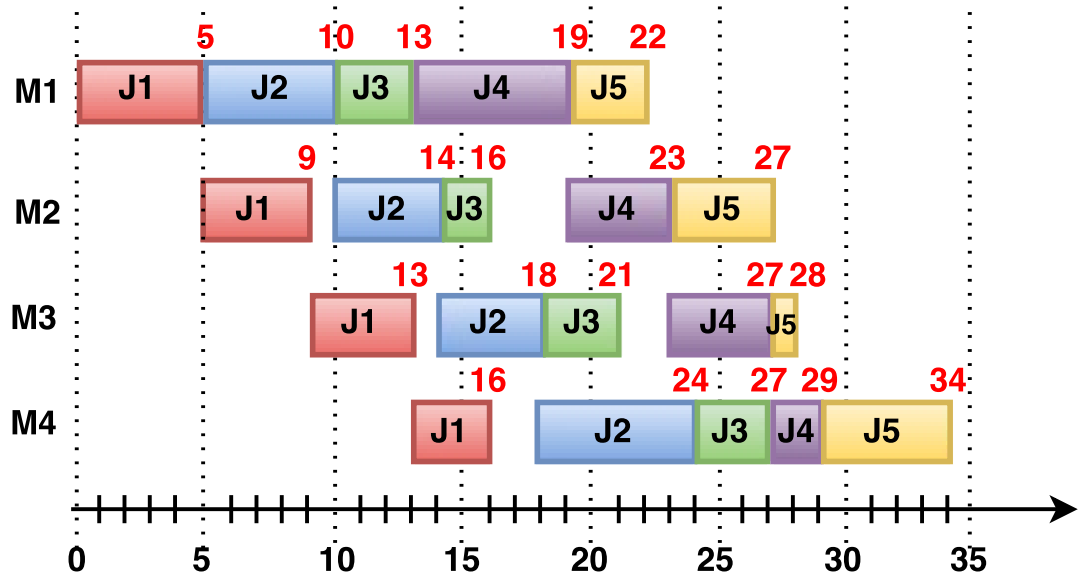

Figure 4.2 – Gant Chart of a PFSP example

4.2 Computational Complexity

Let n be the size of the set of jobs to be schedulated, $n!$ are the possible permutations to evaluate for a Permutation Flow Shop Problem consisting of n jobs. That makes PFSP a combinatorial optimisation problem with a very large solution space; furthermore it belongs to a class of $\mathcal{NP}$-Complete ($\mathcal{NPC}$) problems as demonstrated in [65]. NPC means that the time necessary to solve it is very likely to be Non-Polinomial (unless $\mathcal{P} = \mathcal{NP}$), despite the fact that every given solution (a permutation in this case) can be verified in polynomial time [10].

As a consequence, the solution space grows very fast with the problem size (the number of jobs), and the problem soon becomes soon intractable. In other words, the time necessary for examining all possible job permutations rises exponentially with the number of jobs. Up to date there are no efficient exact algorithms that yield an optimal solution for large-size Permutation Flow Shop Problem (PFSP) problem in a reasonable time.

4.3 Common approaches

Based on the approach to problem solving, the methods presented in literature can be classified in the following groups:

1. **Exact algorithms** - An exact algorithm is a solution techniques aiming at solving the problem guaranteeing the optimality of the solution found [19]. For some problems exact algorithms are not suitable because they perform an often too large (but optimised) search over the solution space using a limited amount of computing resources, and therefore may require weeks, months or years of computation [31]. For the $\mathcal{NP}$-Hard problems like PFSP, where the solution space grows super-exponentially with the problem size, exact algorithms are applicable only to small-sized problems. A good representative of this group is Branch and Bound algorithm [9], which uses speculation to prioritise the evaluation of solution subspaces that will possibly return the optimised solution.

2. **Heuristics** - Exact algorithms are meant to solve a problem to optimality, however, for the resolution of $\mathcal{NP}$-Hard problems, they may require an exponential time. Because of that, many approximate algorithms were developed to efficiently explore the solution space and localise a feasible and satisfactory solution to a problem. Such techniques are often referred to as *heuristics*. Heuristics use some previous knowledge on the problem to improve the search procedure and terminate in a reasonable and useful time. They bring a tool to balance between solution quality and affordable time and cost. Examples of heuristics are Polynomial-Time Approximation Schemes (PTAS), Greedy algorithms, Construction heuristics, rule-based heuristics, etc. *Construction* algorithms start from an empty initial permutation, and iteratively add solution components based on some criterion (e.g. minimize makespan) until a complete solution is obtained. A common application of construction algorithms is to find an initial solution that is going to be a starting point for the advanced local-search procedures.

3. "**Meta-heuristics** are higher-level stochastic optimization methods designed to find, generate or select a lower-level heuristic or a method that will provide a sufficiently good solution to an optimization problem" [5]. They are capable to sub-optimally solve the whole class of previously unseen problem instances. Techniques which constitute meta-heuristic algorithms range from simple local search procedures to a complex learning processes. *Local Search* algorithms start from a complete initial solution and, by small modifications (also called moves) on it, explore the solutions that are in the "neighbourhood" of the current solution. Hence, they explore solutions space around a current solution. The initial solution is commonly generated randomly or using an heuristic approach. Some of the most popular meta-heuristics are Hill Climbing [58], Iterated Greedy [57],

Ant Colony Optimization (MMAS, mMMAS, PACO) [13][61][49], Evolutionary Computation [53], Simulated Annealing [43], Tabu Search [62] [41], Particle Swarm Optimization [30], Cuckoo Search [74].

4. **Hyper-heuristics** - Compared to meta-heuristics, which are customised for solving one class of problems, hyper-heuristics are able to handle a wide range of problem domains [7]. They are even more general systems than meta-heuristics, trying to automate the process of selecting, combining, or adapting several simpler (meta-)heuristics to solve a problem.

The literature dealing with PFSP is extremely vast. With the standardisation of Objected-Oriented software design, meta-heuristics became easy to adapt to the specific problem to solve. That led to creation of techniques, platforms and languages that are jointly referred to as Meta-heuristics Optimisation Frameworks (MOFs) [45]. Following the extendible and easy-to-use meta-heuristics tools, and due to the increased computing power and the decreased price of computing resources, parallel and distributed versions of popular algorithms appeared in the research community. The focus of MOFs has turned to the utilisation of computing environment resources for the sake of the optimisation techniques. The property by which the MOFs are distinguished the most is whether they use the cooperation or not between the parallel algorithm instances in order to find a solution. This property divided the existing meta-heuristics into two groups: non-cooperative and cooperative. **Non- cooperative meta-heuristics** concentrate on the exploration of neighbourhood around a unique point in the solution space at a given time. Each algorithm instance performs an independent computation, and at the end of the process, the best solution found is returned as a final solution. **Cooperative meta-heuristics** rely on a parallel exploration of the solution space, where each parallel algorithm cooperates with the others by information exchange in order to select new promising potential solutions [15]. The information exchange can be performed synchronously or asynchronously, and the topology over which the communication takes place can assume fully-connected star-like shape, a ring shape, etc.. One of the most popular topologies is the Island Model [69], where, at each synchronisation point, or in the particular time instant when an event happened, a certain percentage of the island population (a group of solutions) is being sent to other islands.

According to a recent survey on meta-heuristics frameworks [45] (33 frameworks were analysed), in almost all the MOFs there is a significant lack of support for hyper-heuristics, and parallel and distributed computing capabilities. Also, many of them are really complex and require a lot of technical expertise to be used. That is the reason why HyperSpark framework presented in Chap-

ter 3 was developed in the first place. Many of the previously referenced metaheuristics in this section have been implemented for the Permutation Flow Shop Problem (PFSP) as a library of HyperSpark discussed in Subsection 4.4.6.

4.4 HyperSpark-PFSP Library

This section presents the software implementations created to tackle the Permutation Flow Shop Problem (PFSP) using HyperSpark framework. It is organized as follows. First, Subsection 4.4.1 describes the representation of permutation schedule. Afterwards, subsections 4.4.2, 4.4.3 and 4.4.4 deal with the object-oriented implementations of *Solution* and *EvaluatedSolution* in the context of PFSP. Next, Subsection 4.4.5 describes the *Problem* class. Further, in Subsection 4.4.6 we list the PFSP algorithms implemented in HyperSpark. Subsection 4.4.7 and Subsection 4.4.8 demonstrate the implementation of a few algorithms (among those available in the library) that showed the best results during the testing phase. In Subsection 4.4.9 some more advanced techniques developed for solution space splitting are explained. Finally, in Subsection 4.4.10, a HyperSpark application that uses the presented components is described.

4.4.1 Permutation

Before the description of *Solution* and *EvaluatedSolution* classes proposed in the context of PFSPs, we introduce one Scala type abbreviation. In Scala, type abbreviations provide clarity of the source code, and have the purpose of giving meaning to a specific object type in the context of usage. Therefore, we used one abbreviation to express the Permutation type.

```
package it.polimi.hyperh.types

object Types {
  type Permutation = Array[Int]

  def Permutation(x: Int, xs: Int*): Permutation = {
    // the omitted code will create a new array
    //place integer elements in it
    //and return the array back
  }
}
```

Figure 4.3 – Permutation Type

The jobs involved in the permutation schedule can be safely mapped (one-to-one) onto the sequence of integer numbers. Every possible permutation of integer numbers represents a valid (feasible) solution to PFSP. In HyperSpark-PFSP library, *Permutation* is just an abbreviation for an array of integers (the jobs). For example, from Figure 4.4 it can be seen that *Permutation(5,2,3,1,4)* is equivalent to *Array(5,2,3,1,4)*. But, by writing *Permutation* we make the code much easier to understand and maintain.

```
  //instead of
  val permutation1: Array[Int] = Array(5,2,3,1,4)
  //we can now write
  val permutation2: Permutation = Permutation(5,2,3,1,4)
```

Figure 4.4 – Scala Type Abbreviation - Usage Example

4.4.2 PfsSolution

PfsSolution class extends *Solution* abstract class described in Subsection 3.5.2. The purpose of this class is to encapsulate the solution representation, that is, a permutation of integers in this case. Its full source code is presented in Figure 4.5.

```
class PfsSolution(val permutation:Permutation) extends Solution {

  def asString() = "Array(" + permutation.mkString(", ")+")"
  override def toString = {
     val permString = asString()
     val str = "PfsSolution(permutation:" + permString+")"
     str
  }
  def toList = permutation.toList
}
```

Figure 4.5 – PfsSolution class

PfsSolution encapsulates a *permutation* variable, which contains a permutation of jobs. Therefore, this class represents a description of the solution structure for a PFSP problem. To make the library extensible, and also, to consider different objective functions, *PfsSolution* does not contain any information about the criterion to optimise.

4.4.3 PfsEvaluatedSolution

PfsEvaluatedSolution class (Figure 4.6) is a concrete implementation of *EvaluatedSolution* abstract class described in Subsection 3.5.3. It encapsulates both the *solution* (permutation of jobs) and the *value* associated to it. In the context of our case study, *value* represents the makespan, but, in general, it can be any criterion like the tardiness, the flow time, etc. When *problem.evaluate(pfsSolution)* is invoked the makespan of a *solution* is calculated and *PfsEvaluatedSolution*, containing both the schedule and the makespan value, is returned as a result. Makespan calculation is presented in Subsection 4.4.5.

```
class PfsEvaluatedSolution(override val value: Int,
                           override val solution: PfsSolution)
  extends EvaluatedSolution(value, solution)
{
  //Alternative constructor
  def this(value: Int, permutation: Permutation) = this(value,
      PfsSolution(permutation))
  override def toString = {
  val permString = solution.asString()
  val str = "PfsEvaluatedSolution(value:" + value + ", solution:" +
      permString + ")"
  str
  }
  def compare(that: EvaluatedSolution) = this.value -
      that.asInstanceOf[PfsEvaluatedSolution].value
  def compare(that: PfsEvaluatedSolution) = this.value - that.value
  def permutation = solution.permutation
}
```

Figure 4.6 – PfsEvaluatedSolution Class

4.4.4 NaivePfsEvaluatedSolution

Some algorithms may require a complete initial solution to start the execution. At each iteration they compare the makespans of newly evaluated permutation and best permutation found during the execution. At their initialisation phase the algorithms need initial/best *EvaluatedSolution*. *NaivePfsEvalutedSolution* object produces *PfsEvaluatedSolution* instances with the makespan value equal to the maximum number in Scala Int range (999999999), and copies the jobs sequence (1,..., numOfJobs) as a permutation.

```
object NaivePfsEvaluatedSolution {
  def apply(problem: PfsProblem) = new PfsEvaluatedSolution(999999999,
      problem.jobs)
}
```

Figure 4.7 – PfsEvaluatedSolution Factory

4.4.5 PfsProblem

To avoid the detailed explanation of Scala terminology, in Figure 4.8 we have presented only the skeleton of *PfsProblem* class. PfsProblem constructor accepts the number of jobs *numOfJobs*, the number of machines *numOfMachines*, and a matrix of processing times named *jobTimesMatrix*.

TABLE 4.2
PROCESSING TIMES

Times\Jobs	J1	J2	J3	J4	J5
p_{1,J_i}	5	5	3	6	3
p_{2,J_i}	4	4	2	4	4
p_{3,J_i}	4	4	3	4	1
p_{4,J_i}	3	6	3	2	5

The most important function signatures are:

- ***jobsInitialTimes*** - A function that calculates the completion times for all the jobs, as they were the first in the permutation schedule. Once the problem is instantiated, the aggregated sum of processing times is pre-calculated for each job (line 9 in Figure 4.8). E.g., for job J5 from Table 4.2 the aggregated sum of Array(3, 4, 1, 5) is Array(3, 7, 8, 13). These arrays are saved as columns inside *initEndTimesMatrix* variable (see Table 4.3). Later on, when a new permutation is evaluated, the whole completion times matrix needs to be calculated. The first job from new permutation schedule tells us which column to take from *initEndTimesMatrix* and place it as a first column in completion time matrix *table* (line 18 from Figure 4.8). This optimisation might not look that significant, but it saves an important amount of time in solution evaluation, especially with big problem instances.

Table 4.3
Completion time calculation for the first job in the schedule

Times\Jobs	J1	J2	J3	J4	J5
ct_{1,J_i}	5	5	3	6	3
ct_{2,J_i}	9	9	5	10	7
ct_{3,J_i}	13	13	8	14	8
ct_{4,J_i}	16	18	11	16	13

- ***evaluatePartialSolution*** - calculates the completion times matrix for all jobs present in the schedule on each machine (based on the recursive formula, Equation 4.2, presented in Section 4.1), finds the makespan (the completion time of last job on last machine) and encapsulates it within the returned *PfsEvaluatedSolution*. For instance, consider the *Permutation(5,1,3)*. The first column of the completion time matrix *table* is taken from pre-calculated *initEndTimesMatrix* (line 18 from Figure 4.8, J5 column from Table 4.3). Later on, the subsequent columns are calculated by applying Equation 4.2 to the extracted processing times for subsequent jobs in the schedule (lines 20-25 from Figure 4.8, J1, J3 from Table 4.4).

Table 4.4
Processing times extraction for a partial solution

Times\Jobs	J5	J1	J3
p_{1,J_i}	3	5	3
p_{2,J_i}	4	4	2
p_{3,J_i}	1	4	3
p_{4,J_i}	5	3	3

Table 4.5
Completion times matrix for a partial solution

Times\Jobs	J5	J1	J3
p_{1,J_i}	5	10	13
p_{2,J_i}	9	14	15
p_{3,J_i}	13	18	21
p_{4,J_i}	16	21	24

- ***evaluate*** - Basically, evaluating the whole solution is a special case of partial solution evaluation. When partial solution is employed, *evaluatePartialSolution* continues the calculation up to the last job present in permutation. Since the full solution (a complete permutation) is provided, the former will calculate the whole completion time matrix and provide the makespan of respective permutation. The returned object will be *PfsEvaluatedSolution* containing makespan and jobs permutation.

- **getExecutionTime** - returns the default execution time calculated based on the problem size (the number of jobs and machines). The formula is: *problem.numOfMachines*(problem.numOfJobs/2.0)*60* milliseconds, as in Vallada and Ruiz's experiments [67].

```
1  class PfsProblem(
2     val numOfJobs: Int,
3     val numOfMachines: Int,
4     val jobTimesMatrix: Array[Array[Int]] // processing times
5  ) extends Problem {
6
7  //calculate aggregated sums for all the jobs
8  def jobsInitialTimes(): Array[Array[Int]] = {...}
9  val initEndTimesMatrix = jobsInitialTimes()
10
11 def evaluatePartialSolution(jobsPermutation: Permutation):
       PfsEvaluatedSolution = {
12    val numOfPartJobs = jobsPermutation.length
13    val numOfMachines = jobTimesMatrix.size
14    //table represents completion time matrix
15    val table = Array.ofDim[Int](numOfMachines, numOfPartJobs)
16    //first column is taken from initEndTimesMatrix
17    for (mInd <- 0 until numOfMachines)
18       table(mInd)(0) = initEndTimesMatrix(mInd)(jobsPermutation(0) - 1)
19    //calculate the rest of completion time matrix
20    for (jInd <- 1 until numOfPartJobs; mInd <- 0 until numOfMachines) {
21       if (mInd > 0)
22          table(mInd)(jInd) = Math.max(table(mInd-1)(jInd), table(mInd)(jInd-1))
              + jobTimesMatrix(mInd)(jobsPermutation(jInd)-1)
23       else
24          table(mInd)(jInd) = table(mInd)(jInd-1) +
              jobTimesMatrix(mInd)(jobsPermutation(jInd)-1)
25    }
26    val makespan = table(table.size-1).max
27    new PfsEvaluatedSolution(makespan, jobsPermutation)
28 }
29 def evaluate(s: Solution): EvaluatedSolution = {
30    val solution = s.asInstanceOf[PfsSolution]
31    evaluatePartialSolution(solution.permutation.toList)
32 }
33
34 def getExecutionTime(): Double = {
35    numOfMachines * (numOfJobs / 2.0) * 60 //time limit
36 }
37 }
```

Figure 4.8 – PfsProblem class

4.4.6 Algorithms Implemented and Imposed Constraints

Algorithms implemented for the purposes of this thesis are listed below:

Table 4.6
Algorithms implemented

Name	Type	Authors	Year	Ref.	Class
NEH	Construction	Nawaz, Enscore and Ham	1983	[39]	*NEHAlgorithm*
Iterated Greedy	Local Search	Ruiz and Stützle	2007	[57]	*IGAlgorithm*
Genetic Algorithm	Local Search	Reeves	1995	[53]	*GAAlgorithm*
Hybrid Genetic Algorithm	Local Search	Ruiz and Stützle	2007	[75]	*HGAAlgorithm*
Simulated Annealing	Local Search	Osman and Potts's adaption for PFSP	1989	[43]	*SAAlgorithm*
Improved Simulated Annealing	Local Search	Xu and Oja	1990	[73]	*ISAAlgorithm*
Taboo Search	Local Search	Taillard	1989	[62]	*TSAlgorithm*
Taboo Search with backjump tracking	Local Search	Novicki and Smutnicki	1994	[41]	*TSABAlgorithm*
Ant Colony Optimization	Local Search	Dorigo and Stützle	2009	[13]	*ACOAlgorithm*
Max Min Ant System	Local Search	Stützle	1997	[61]	*MMASAlgorithm*
mMMAS	Local Search	Rajendran and Ziegler	2002	[49]	*MMMASAlgorithm*
PACO	Local Search	Rajendran and Ziegler	2002	[49]	*PACOAlgorithm*

All of the algorithms listed are single-thread algorithms[1], and they have been implemented as such. However, they are used as the basic elements by HyperSpark framework in order to create a parallel metaheuristic. This is true given that the user provides also a seeding/messaging mechanism (*SeedingStrategy*) and a reduce/selection mechanism (*MapReduceHandler*) too.

HyperSpark-PFSP library currently implements 12 methods. NEH is the only construction algorithm implemented, since it is dominant in its category, and therefore many of the algorithms listed use NEH to build an initial solution. Even though we have implemented many of them, in the experimental

[1] exception is maybe Hybrid GA, which can easily made multi-threaded

campaign presented in Chapter 5 we considered only two algorithms, that is, those that showed the best performance during preliminary tests. For the sake of completeness the selected methods are described in detail in the next sections.

4.4.7 Iterated Greedy Algorithm

Iterated Greedy (IG) algorithm [57] belongs to a group of local search algorithms, meaning that it uses an initial solution as a base for finding a better neighbouring solution in the solution space. IG iteratively improves the current solution by performing a local search procedure on it. Local search procedure could range from swapping two elements at random positions to a complex learning process in which a more convenient (job, position) pair accumulates the reward during the time.

The source code of IG algorithm is presented in Figure 4.9. *IGAlgorithm* extends the *Algorithm* trait, which demands the implementation of *evaluate(p:Problem)* and *evaluate(p:Problem, stopCond:StoppingCondition)* functions. The first evaluate function (lines 10-15 from Figure 4.9), when the stopping condition is not provided by the user, suggests to the algorithm to use the default stopping condition - execution time limit described in Subsection 4.4.6. At the start of the second evaluate function time limit is initialised (line 18 from Figure 4.9), and later the iterative loop is defined (lines 21-41) and started (line 44).

In the *loop*, before the start of any iteration, it is checked if the stopping condition is not satisfied (the time limit is not reached) (line 22 from Figure 4.9), and if that is not the case the best solution found is returned (line 42). The algorithm starts with an initial solution (line 27) and improves it by local search in its neighbourhood. Variable *currentSolution* is initialised using NEH construction algorithm [39] if a seed is not provided (lines 4-9). Later, in the *destruction* phase (line 33) it splits the current solution into two lists - by removing d jobs from the current solution. E.g. it removes 2 jobs out of 20 present in current solution, and makes a list of 18 jobs and a list of 2 jobs. In the *construction* phase (line 34) the removed jobs are reinserted again by considering different positions for insertion. A completely constructed solution is evaluated and improved by a local search (lines 35-37). Current solution is updated and the process is repeated as long as the stopping condition is not satisfied. For details on destruction, construction and localSearch please refer to the original work [57].

```
class IGAlgorithm(val d:Int,val T:Double, seedOption: Option[PfsSolution])
    extends Algorithm {
seed = seedOption

def initialSolution(p: Problem): EvaluatedSolution = {
   seed match {
      case Some(seed) => seed.evaluate(p).asInstanceOf[PfsEvaluatedSolution]
      case None => new
          NEHAlgorithm().evaluate(p).asInstanceOf[PfsEvaluatedSolution]
   }
}
override def evaluate(problem:Problem):EvaluatedSolution = {
   val p = problem.asInstanceOf[PfsProblem]
   val timeLimit = p.getExecutionTime()
   val stopCond = new TimeExpired(timeLimit)
   evaluate(p, stopCond)
}
override def evaluate(problem:Problem, stopCond:StoppingCondition):
    EvaluatedSolution = {
   val p = problem.asInstanceOf[PfsProblem]
   val stop = stopCond.asInstanceOf[TimeExpired].initialiseLimit()
   val dummySol = NaivePfsEvaluatedSolution(p)

   def loop(currentSol:PfsEvaluatedSolution, bestSol:PfsEvaluatedSolution,
       iter:Int): PfsEvaluatedSolution = {
      if(stop.isNotSatisfied()) {
         var currentSolution = currentSol
         var bestSolution = bestSol
         //INITIALISATION PHASE
         if(iter == 1){
            currentSolution = initialSolution(p)
            //IMPROVE IT BY LOCAL SEARCH
            currentSolution = localSearch(currentSolution.permutation,p)
            bestSolution = currentSolution
         }
         //IN EACH ITERATION DO
         val pair = destruction(currentSolution.permutation, d)
         val bestPermutation = construction(pair._1, pair._2,p)
         bestSolution = p.evaluate(PfsSolution(bestPermutation))
         .asInstanceOf[PfsEvaluatedSolution]
         val improvedSolution = localSearch(bestPermutation,p)
         //solution update omitted for clarity
         //RECURSIVE CALL
         loop(currentSolution, bestSolution, iter+1)
      }
      else bestSol
   }
   loop(dummySol, dummySol, 1)
}
```

Figure 4.9 – Iterated Greedy Algorithm class

4.4.8 Hybrid Gentic Algorithm

Hybrid Genetic (HG) algorithm [75] is a population-based meta-heuristic optimization algorithm, whose sub-procedures mirror the processes in the genetic evolution. *HGAlgorithm* initialises the population of N-1 random permutations and one sub-optimal solution (a seed or NEH solution) (line 14 in Figure 4.11). In the while loop, the following process is repeated until the stopping condition is satisfied.

First, the population is divided into four parts (lines 22-27), and on each part a different crossover operator is applied (lines 28-32). The use of multi-variant crossover operators contributes to the diversity in the set of obtained solutions. Afterwards, the population is updated, and the solution with minimum makespan is taken as *bestSolution* (line 35). Later on, the population is improved by *metropolis sampling* and the global best solution is saved (lines 37-38). Metropolis sampling, commonly used in Simulated Annealing type of algorithms, replaced the *mutation* phase of this genetic algorithm. It allows the algorithm to escape from local optima, by accepting a slightly worse solution over the best one during the search procedure (see Figure 4.10). This approach sometimes favours the exploration of the solution space over the constant improvement, and has shown good results in practice. For additional details on crossover and metropolis please refer to the original work [75].

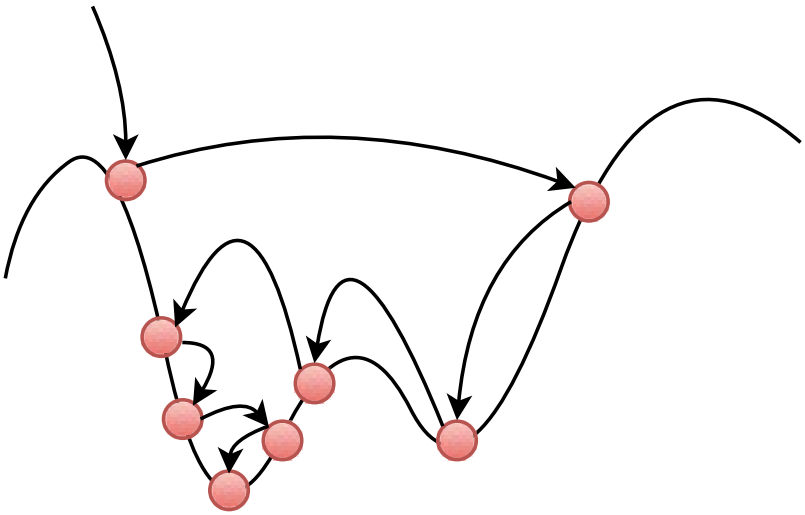

Figure 4.10 – Metropolis sampling

```
class HGAAlgorithm(
  p:PfsProblem, popSize:Int, prob:Double, coolingRate:Double,
      seedOption:Option[PfsSolution]) extends GAAlgorithm(popSize,
      seedOption) {
  var temperatureUB:Double = 2000.0 //dummy initialisation
  seed = seedOption
  override def evaluate(problem: Problem): EvaluatedSolution = {
    val p = problem.asInstanceOf[PfsProblem]
    val timeLimit = p.getExecutionTime()
    val stopCond = new TimeExpired(timeLimit)
    evaluate(p, stopCond)
  }
  override def evaluate(problem:Problem, stopCond:StoppingCondition):
      EvaluatedSolution = {
  val p = problem.asInstanceOf[PfsProblem]
  //INITIALIZE POPULATION
  var population = initSeedPlusRandom(p, popSize)
  var bestSolution = population.minBy(_.value)
  var worstSolution = population.maxBy(_.value)
  val delta = worstSolution.value - bestSolution.value
  temperatureUB = -delta / scala.math.log(prob)
  val stop = stopCond.asInstanceOf[TimeExpired].initialiseLimit()

  while (stop.isNotSatisfied()) {
    //DIVIDE POPULATION IN 4 SUBPOPULATION
    val subPopSize = popSize / 4
    var subpop1 = population.take(subPopSize)
    var subpop2 = population.drop(subPopSize).take(subPopSize)
    var subpop3 = population.drop(2*subPopSize).take(subPopSize)
    var subpop4 = population.drop(3*subPopSize).take(subPopSize)
    //CROSSOVER
    subpop1 = crossover(p, subpop1, bestSolution, crossoverLOX, stop)
    subpop2 = crossover(p, subpop2, bestSolution, crossoverPMX, stop)
    subpop3 = crossover(p, subpop3, bestSolution, crossoverC1, stop)
    subpop4 = crossover(p, subpop4, bestSolution, crossoverNABEL, stop)
    //UPDATE POPULATION
    population = subpop1 ++ subpop2 ++ subpop3 ++ subpop4
    bestSolution = List(population.minBy(_.value),
        bestSolution).minBy(_.value)
    //METROPOLIS MUTATION
    population = metropolis(p, population, stop)
    bestSolution = List(population.minBy(_.value),
        bestSolution).minBy(_.value)
  }
  bestSolution
}
```

Figure 4.11 – Hybrid Genetic Algorithm class

4.4.9 Seeding Strategies

HyperSpark provides a simple strategy called *SameSeeds* (sends the same solution to all parallel algorithm instances), presented in Subsection 3.5.8. However, as demonstrated in experimental Section 5.5, the impact of this component on the overall performance showed by the algorithms is quite important. For this reason we devised and implemented four permutation-specific seeding strategies.

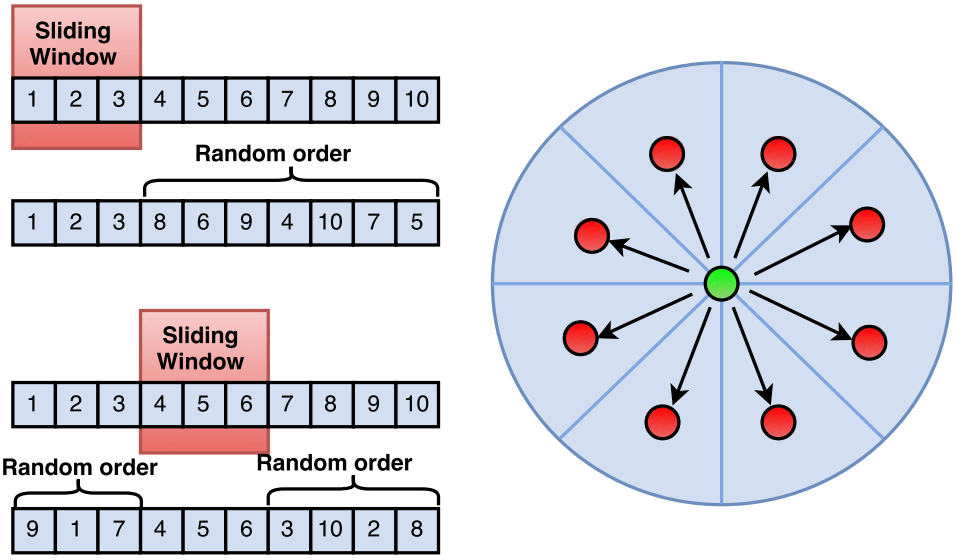

Figure 4.12 – SlidingWindow Seeding Strategy

SlidingWindow strategy (Figure 4.12) has the goal to explore the solution space in more effective way, by providing different initial solutions to different parallel algorithms. Assuming that the initial solution is a permutation, *SlidingWindow* slides the window of provided size on the solution at hand, keeps the elements contained in a window, and randomly permutes the elements outside the sliding window. The window starts from first element and slides one position to the right until it reaches the end of solution, if possible generating as many new solutions as there are parallel algorithms. If the solution is a green circle in Figure 4.12, by keeping a part of initial solution (which is considered as best solution of previous framework iteration) in newly generated solutions (red circles), new solutions will hopefully stay in the neighbourhood of the initial solution. How distant the new solutions will be from the initial one in the (abstract) solution space depends on a chosen window size, or to express it differently, it depends on a size of the reused sequence of elements from initial

solution. For instance, consider the *Permutation(1,2,3,4,5,6,7,8,9,10)* from Figure 4.12 as the initial solution. By applying the *SlidingWindow* of size 8 we will obtain three permutations, e.g.: (1,2,3,4,5,6,7,8,10,9), (10,2,3,4,5,6,7,8,9,1) and (2,1,3,4,5,6,7,8,9,10), that are very close to the initial solution in the solution space. As the window size decreases the new solutions become less biassed to the initial one, implying the longer distance in the solution space.

SeedPlusSlidingWindow strategy is similar to SlidingWindow in a way it generates new solutions. It differs in that it keeps the old solution (seed) for next iterations. If there are N parallel algorithms, it will generate N-1 solutions by using sliding window, and one (old solution) will be prepended to solutions generated.

FixedWindow strategy is similar to SlidingWindow in a way it generates new solutions, but it differs in window placement procedure. In SlidingWindow strategy the window is moved from left to right, but here the window position is randomly generated for each newly produced seed. After the window has been fixed, the seeds are produced in the same way as in SlidingWindow: the jobs within the window are kept in the same order, while the jobs outside of it are randomly permuted. The difference between *SlidingWindow* and *FixedWindow* strategies is expressed when there is a limit for the number of generated solutions. If the limit has a low value (e.g., the limit is 10 solutions, the number of solutions possible to generate is 100), *SlidingWindow* strategy will generate the solutions using only the first part of the solution at hand. With *FixedWindows* the sampling over the initial solution should be fairer.

SeedPlusFixedWindow strategy, besides applying FixedWindow, also keeps the provided solution (seed) for the next framework iteration.

4.4.10 HyperSpark Application

For the end of this chapter, and in order to have a clear idea of how the mentioned concepts can be used in practice, we present a simple HyperSpark application in Figure 4.13. As it can be seen, the application starts the execution of twenty IG algorithms in parallel using the HyperSpark framework. Each of them will have a fixed amount of time set for the execution, and a fixed number of iterations. Each framework iteration will consume $totalTime/numOfIterations$ of time for the execution. The best solution in each framework iteration will be distributed using SeedPlusFixedWindow strategy, having window size equal to the square root of the number of jobs.

Although IGAlgorithm presented in Figure 4.9 is a CPU-intensive single-thread procedure and the solution space for the problem inst_ta094 of 200 jobs contains 200! = 7.88 x 10^{374} solutions, HyperSpark returned a makespan that is only 0.23 percents higher than the best solution ever found.

```
object SimpleApp {
  def main(args: Array[String]) {
    val problem = PfsProblem.fromResources("inst_ta094.txt")
    val algorithm = new IGAlgorithm()
    val numOfAlgorithms = 20
    val totalTime = problem.getExecutionTime()
    val numOfIterations = 10
    val iterTimeLimit = totalTime / numOfIterations
    val stopCond = new TimeExpired(iterTimeLimit)
    val windowSize: Int = scala.math.sqrt(problem.numOfJobs).toInt
    val strategy = new SeedPlusFixedWindow(windowSize)

    val conf = new FrameworkConf()
    .setSparkMaster("spark://master:7077")
    .setProblem(problem)
    .setNAlgorithms(algorithm, numOfAlgorithms)
    .setNDefaultInitialSeeds(numOfAlgorithms)
    .setNumberOfIterations(numOfIterations)
    .setStoppingCondition(stopCond)
    .setSeedingStrategy(strategy)

    val solution = Framework.run(conf)
    println(solution)
  }
}
```

Figure 4.13 – HyperSpark Application

CHAPTER 5

Experimental Results

In this chapter we present and examine some experimental results obtained from an experimental campaign in which we extensively tested the major solutions developed in Chapter 4 (namely HyperSpark-PFSP library), which rely on the HyperSpark framework discussed in Chapter 3. Until now, we did not specify any measurements of incurred time overhead nor attained solution quality. Therefore, in this chapter we will pose reasonable questions about performance and provide the answers supported by sound experimental results. The main research questions are:

1. How much time overhead does HyperSpark introduce? Is it acceptable in the context of parallel and cooperative optimization?
2. Are the algorithms implemented using HyperSpark competitive with respect to state-of-the-art?

These questions are mirrored in the structure for this chapter. Each question will be treated as a separate experiment, that is, a section in the context of this document. First of all, in Section 5.1 we briefly explain the experimental environment on which the experiments were conducted. Then, in Section 5.2, the general experimental conditions imposed are presented. Section 5.3 presents the benchmarks used to analyse the cost and the efficiency of HyperSpark framework. Section 5.4 describes the first experiment, which represents a preliminary analysis aiming at measuring the overhead introduced by HyperSpark under different conditions. In Section 5.5 we execute two algorithms with three seeding strategies on the set of Taillard instances in order to analyse the impact of different set ups on the quality of solutions. Finally, in Section 5.6 we employ the best hardware configuration, algorithm and seed-

ing strategy to try to find the best solution ever found for some of the PFSP instances.

5.1 Experimental environment

The experimental environment consists of 10 virtual machines, each having 8 CPU cores running at 2.4 GHz and 15 GB of RAM at their disposal. Hence, there are 80 CPU cores and 150 GB in total available for the computations. On top of this physical infrastructure, Spark environment (version 1.5) is installed in Standalone mode, meaning that it manages both application scheduling and provisioning of hardware resources on its own. Having Spark installed allowed us to control the exact number of cores and the amount of RAM used for the purposes of specific experiments.

5.2 Experimental conditions

In order to have a fair comparison between the algorithms and to support cooperation between them in the most efficient way we imposed three constraints on the algorithm execution:

- **the same solution evaluation procedure**, i.e., makespan calculation. All solutions are evaluated using *problem.evaluate(s:Solution)*, a function call that implements makespan calculation described in Section 4.1.
- **stopping condition** - for all parallel algorithm instances the same stopping condition is set. It is the total time provided for algorithm execution, and it is equal to *problem.numOfMachines*(problem.numOfJobs/2.0)*60* milliseconds, as in Vallada and Ruiz's experiments [67]. If the algorithm requires cooperation, the time provided for each superstep is equal to: total time divided by the number of supersteps (framework iterations). Execution time is measured using *TimeExpired* class, described in Subsection 3.5.5.
- **initialisation** - if there is an intial solution (a seed) provided to the algorithm by the user or by the framework, the algorithm will use it. Otherwise, the algorithm will use its own custom initialisation. For example, NEH constructed solution. This feature is especially convenient for cooperative algorithms, which exchange the seeds after each superstep.
- **number of runs** - Since we are considering randomised algorithms each experiment must be repeated a suitable number of times in order to have sound results. For each benchmark the experiment is repeated 5 or 10

times, depending on the estimated variability in the output. The final result is recorded as an average over all repetitions.

5.3 Benchmarks

The benchmarks used to demonstrate results are the well-known Taillard instances [63] for Flow Shop scheduling. There are 120 problem instances, ranging from 20 to 500 jobs. All instances and their respective best found solutions have been considered for the experiments with HyperSpark-PFSP library.

Table 5.1
Benchmarks - 120 PFSP instances

instance	jobs	machines	execution time (s)
inst_ta{001-010}	20	5	3.0
inst_ta{011-020}	20	10	6.0
inst_ta{021-030}	20	20	12.0
inst_ta{031-040}	50	5	7.5
inst_ta{041-050}	50	10	15.0
inst_ta{051-060}	50	20	30.0
inst_ta{061-070}	100	5	15.0
inst_ta{071-080}	100	10	30.0
inst_ta{081-090}	100	20	60.0
inst_ta{091-100}	200	10	60.0
inst_ta{101-110}	200	20	120.0
inst_ta{110-120}	500	20	300.0

5.4 Experiment 1 - Overhead estimation

Initialisation of Spark Context, compute function and data distribution to the nodes, collection and aggregation of results, and the termination of Spark Context accounts for some time overhead. In general, we wanted to know what is the computation time overhead for running the simplest HyperSpark application in Spark environment. By the simplest HyperSpark application, we mean that the application executes independently its *algorithm* instances without any kind of cooperation between them. In HyperSpark terminology, that means that the number of stages used for the computation is equal to one.

We chose one PFSP instance of size 20, 50, 100, 200 and 500 jobs, and for each of them we gradually increased the number of cores, that is, the number

of *algorithms* used from the set {1, 8, 16, 24, 32, 40}. For each instance, and for each number of cores the execution was repeated 5 times. Each time the Spark Context was initialised, the application was executed, and the context was terminated. To be more formal, we define below the input and output considered in this experiment.

5.4.1 Input parameters

The input parameters are: number of stages, number of CPU cores, algorithm execution time, algorithm and seeding strategy. The description of each input parameter follows.

- **Number of stages** - fixed to 1. As we mentioned, we want to test the most simplest case, when there is no cooperation between algorithms. Cooperation, in fact, would introduce times that are cannot be easily classified as overhead.

- **Number of CPU cores** - Chosen from the set {1, 8, 16, 24, 32, 40}. In this experiment the number of CPU cores is increased almost linearly. Having more algorithm instances demands more memory space and a higher network bandwidth for the distribution and aggregation of data. Therefore, it is necessary to know up to which point one can increase the number of cores. Using a big number of cores may introduce a time overhead which is not acceptable by project's specifications.

- **Algorithm execution time** - This time is provided to algorithm for its execution, and it does not depend on Spark environment. In the case of PFSP instances, it is calculated based on a number of jobs and a number of machines, as mentioned in Section 5.2. Since this parameter directly depends on the PFSP instance, in the table below we will provide the properties of used instances.

Table 5.2
Experiment 1 - Problem instances

instance	jobs	machines	execution time (s)
inst_ta001	20	5	3.0
inst_ta031	50	5	7.5
inst_ta061	100	5	15.0
inst_ta091	200	10	60.0
inst_ta111	500	20	300.0

- **Algorithm** - In this experiment, any algorithm implemented in HyperSpark-PFSP library can be arbitrarily. We selected the state-of-the-art algorithm *IGAlgorithm*.
- **Seeding Strategy** - In this experiment, any seeding strategy can be arbitrarily chosen as it does not affect the framework overhead. Since the number of stages is fixed to 1, the value of this parameter does not play a role in the output generation. The default value, *SameSeeds*, is used.

5.4.2 Output

The collected outputs are: environment initialisation time, stage_0 time and environment closing time. The description of each output follows.

- **Initialisation time** - Time spent from the moment in which SparkContext has been started to the moment in which the job (i.e., the application) is ready to start its computation. During this time, executors are created and the communication layer is established.
- **Stage_0 time** - Since there is only one stage in this experiment, the total execution time provided for the algorithm execution will be completely used by the first stage - *stage_0*. This output, besides the algorithm'sxecution time, includes the time necessary for creating memory managers, adding RDD partitions to memory, starting parallel computing tasks and aggregating the results.
- **Overhead** - Computation time overhead incurred by Spark environment. This value is obtained by deducting the algorithm execution time from stage_0 time.
- **Closing time** - The time necessary for releasing of allocated memory and terminating SparkContext.

5.4.3 Results

In Table 5.3 there is a summary of collected outputs. As aforementioned, for each instance and for each number of cores, the execution is repeated 5 times. For clarity, the averaged values are presented .

A couple of interesting remarks can be concluded based on the results from Table 5.3:

- By increasing the problem size, i.e. changing to an instance with a higher number of jobs, the overhead percentage decreases to an acceptable: 5-10 % (31.9/330.24 seconds on average) for the instance with 500 jobs. From this, we can conclude that HyperSpark is particularly suitable for the optimisation of large problems.

- Increasing the number of cores increases the time overhead, up to 30% for the smallest instance, and up to 8% for the biggest instance tested. This is an expected result. The higher the level of parallelism the higher the setup and synchronisation overhead.

- Increasing the number of cores increases the initialisation time, from 5 to 13 seconds approximatively on the considered environment, no matter what the problem size is. This time can be considered acceptable for problems that requires several minutes to be solved.

- Closing time is constant in the interval of 0-1 second. However, as for the setup time this time is attributable to Spark framework rather than to HyperSpark.

As the problem size increases, the total overhead percentage decreases. Therefore, to benefit from a distributed environment and support higher solution diversity we must make a trade-off between the number of cores used and the time overhead percentage. Typically, distributed environments are employed when there is a hard problem. In our case that refers to the solution space size, which is directly determined by the problem instance size. Therefore, when choosing the suitable number of cores we will consider only the results obtained for 200 and 500 jobs (inst_ta091 and inst_ta111). The acceptable overhead percentage lies in the range which corresponds to a range between 16 and 24 cores, and, therefore, for our future experiments, we will fix that number to 20.

Table 5.3
Experiment 1 - Time overhead

instance	cores	stage_0 (s)	overhead (s)	ovr (%)	init (s)	close (s)
inst_ta001	1	5.80	2.8	46.69	5.80	0.60
inst_ta001	8	5.91	2.91	49.00	7.20	0.20
inst_ta001	16	7.68	4.68	60.79	7.60	0.60
inst_ta001	24	9.54	6.54	68.50	9.40	0.40
inst_ta001	32	10.76	7.76	71.94	11.80	0.60
inst_ta001	40	13.26	10.26	77.33	12.80	0.60
inst_ta031	1	10.13	2.63	25.75	8.20	0.40
inst_ta031	8	10.56	3.06	28.91	7.60	0.20
inst_ta031	16	12.38	4.88	39.43	7.60	0.40
inst_ta031	24	14.29	6.79	47.47	10.80	0.40
inst_ta031	32	15.84	8.34	52.66	12.00	0.20
inst_ta031	40	18.26	10.76	58.89	13.60	0.20
inst_ta061	1	17.37	2.37	13.66	6.40	0.00
inst_ta061	8	18.22	3.22	17.63	7.60	0.20
inst_ta061	16	20.14	5.14	25.35	9.00	0.20
inst_ta061	24	22.21	7.21	32.38	9.20	0.60
inst_ta061	32	23.47	8.47	36.08	12.60	0.40
inst_ta061	40	25.89	10.89	42.02	12.80	1.00
inst_ta091	1	63.70	3.70	5.77	6.60	0.00
inst_ta091	8	66.61	6.61	9.91	4.60	0.40
inst_ta091	16	69.46	9.46	13.62	4.40	0.20
inst_ta091	24	70.16	10.16	14.47	7.60	0.20
inst_ta091	32	73.28	13.28	18.12	7.40	0.40
inst_ta091	40	76.97	16.97	22.04	8.00	0.80
inst_ta111	1	318.05	18.05	5.68	5.20	0.20
inst_ta111	8	320.89	20.89	6.51	4.40	0.20
inst_ta111	16	326.35	36.35	8.07	4.80	0.40
inst_ta111	24	332.56	32.56	9.79	6.20	0.40
inst_ta111	32	337.27	37.27	11.05	7.20	0.60
inst_ta111	40	346.33	46.33	13.38	11.80	0.80

5.5 Experiment 2 - Solution quality analysis

Now that there is a precise measurement of time overhead, and the number of cores is set to 20 (as prescribed in Experiment 1) we focus on examining the solution quality. In order to obtain the best performance, we will include the co-

operation between algorithms in this experiment. The number of HyperSpark iterations (stages) will be fixed to 10. For each of the 80 Taillard's problem instances (ranging from 50 to 200 jobs) 20 algorithms of the same type will run 10 times with a pre-determined seeding strategy. Two different algorithms and three different strategies will be employed, and at the end of the test the best (algorithm, seeding strategy) combination will be chosen. During the experiment, we also measured the distribution of computation time between stages. Some interesting remarks will be presented together with the results.

5.5.1 Input parameters

The considered input parameters are essentially the same of Experiment 1; the rangvaluesome of them have been restricted, thoughis changeddescription of input parameters follows:

- **Number of stages** - fixed to 10. There is a cooperation between algorithms, and it is determined by a seeding strategy.
- **Number of CPU cores** - Fixed to 20, as prescribed in Experiment 1. The PFSP instances used for the experiments are of a large size. Thus, we consider the time overhead of 10 % (corresponds to 20 cores) as an acceptable cost.
- **Algorithm execution time** - As mentioned in Experiment 1, this parameter directly depends on the PFSP instance. The execution times of test data are given in the table Table 5.4. Notice that, this time is divided by the number of iterations (stages). Therefore, the computation time provided to a HyperSpark iteration, in the current experiment, will be the execution time given in Table 5.4 divided by 10.
- **Algorithm** - Chosen from a set {*IGAlgorithm*, *HGAlgorithm*}. Iterated Greedy algorithm and Hybrid Genetic algorithm, described in Chapter 4, are the two best performing algorithms out of 10 implemented. The selection was made based on the results they had demonstrated during the Framework development phase.
- **Seeding Strategy** - Chosen from a set {*SameSeeds*, *SeedPlusSlidingWindow*, *SeedPlusFixedWindow*}. Here, we want to test two extremes: the most naive strategy and the advanced ones. The aim is to measure the impact of a greater diversity on the quality of final solutions. We decided, however, to preserve the best solution found during the iterations, and that is why we chose *SeedPlusSlidingWindow* and *SeedPlusFixedWindow*. For additional details, the reader can refer Subsection 4.4.9.

Table 5.4
Experiment 2 - Problem instances

instance	jobs	machines	execution time (s)
inst_ta{031-040}	50	5	7.5
inst_ta{041-050}	50	10	15.0
inst_ta{051-060}	50	20	30.0
inst_ta{061-070}	100	5	15.0
inst_ta{071-080}	100	10	30.0
inst_ta{081-090}	100	20	60.0
inst_ta{091-100}	200	10	60.0
inst_ta{101-110}	200	20	120.0

5.5.2 Output

The output values considered in this experiments are: environment initialisation time, stage_0 time, stage_0 overhead, stages_1to9 time, stages_1to9 overhead, environment closing time, solution found.

We aimed at measuring the average, per stage, computation time. While doing so, we noticed that stage_0 incurs much higher overhead than the succeeding stages (1 to 9). It is worth noticing that we are considering the different compute and overhead time measures for the first and the remaining stages. A short description of the output parameters follows.

- **Stage_0 time** and **Stages_1to9 time** - Time spent to compute the first and succeeding nine HyperSpark iterations, respectively.
- **Stage_0 overhead** and **Stages_1to9 overhead** - Time overhead incurred during the computation of the first and succeeding nine iterations, respectively.
- **Total computation time** - Derived as a sum of stage_0 time and stages_1to9 time.
- **Total time overhead** - Derived as a difference between total computation time and algorithm execution time.
- **Solution found** - Solution obtained by using HyperSpark framework.
- **Relative Percentage Deviation (RPD)** - Relative deviation of a solution found with respect to the best known solution, expressed in percents. This parameter serves to tell us how close the solution found is to the

best known solution[1]. If it is negative - new best solution is found. It is calculated as follows:

$$RPD = \frac{Solution^{found} - Solution^{best}}{Solution^{best}} * 100 \tag{5.1}$$

5.5.3 Analysis of experimental results

Since the input parameters are the same for both overhead estimation and solution quality measurement, there was no need to execute two separate experiments. The respective analyses are presented in the following subsections.

5.5.3.1 Overhead estimation

In Table 5.5 we provide an analysis of **time consumption by stages**. In the previous experiment we saw how much time is necessary for initialisation and termination of SparkContext, how much time is spent and how big is the time overhead for a single iteration (stage) of computation. Since there were 10 stages in this experiment, the distribution of computation time overhead for individual stages was unknown beforehand. The results from Table 5.5, particularly *stage 0 time* and *stages 1to9 time* columns, show that, for small instances, stage 0 consumes almost as much time as other nine stages all together. In general, the time overhead of stage 0 (*stage 0 ovr* column) is 3 to 30 times higher than average, per-stage time overhead for stages 1 to 9 (*stages 1to9 avg ovr* column). The inspection of logs showed that during the first stage there is an additional memory management activity: an RDD of *DistributedDatum*s is added to the main memory of each computing node, which, all together with computation, incurred an overhead from 6 to 11 seconds. During the succeeding stages memory management relied on the simple modifications of already existing memory contents. From *stage 0 ovr* and *stages 1to9 avg ovr* we can see that the difference in time overhead per stage becomes smaller with the increase of problem size. These results are supported by the *compute ovr* column, which is calculated as follows:

$$compute\_ovr = \frac{stage\_0\_ovr + stages\_1to9\_ovr}{stage\_0\_time + stages\_1to9\_time} * 100 \tag{5.2}$$

Equation 5.2 is used to calculate the computation time overhead of 10 stages with respect to the total time spent for their computation, without the initialisation and closing time included. From *compute ovr* column it can be seen that

[1]Best known solution in the literature used for comparison is the solution obtained from Eric Taillard's website [63] (last update: 22 of May 2015).

the time overhead percentage decreases with the problem size. The initialisation and termination of Spark Context are independent of computations and problem instance size, and therefore they are omitted from the equation.

Also, the grand average of initialisation and closing time is presented in the last row of Table 5.5. Calculating the average of other columns would be meaningless, since they directly depend on the problem size, which varies along the table.

Table 5.5
Experiment 2 - Average Stage Time Overhead

size	exec. time	init time	stage 0 time	stage 0 ovr	stages 1to9 time	stages 1to9 ovr	stages 1to9 avg ovr	close time	compute ovr (%)
50 x 5	7.5	9.64	6.73	5.98	8.94	2.19	0.24	0.48	51.19
50 x 10	15	9.84	7.50	6.00	15.87	2.37	0.26	0.49	35.25
50 x 20	30	6.78	11.41	8.41	29.52	2.52	0.28	0.65	26.55
100 x 5	15	9.06	8.05	6.55	17.04	3.54	0.39	0.48	39.62
100 x 10	30	6.70	11.35	8.35	30.96	3.96	0.44	0.40	28.91
100 x 20	60	6.43	14.63	8.63	59.52	5.52	0.61	0.54	18.91
200 x 10	60	6.24	16.37	10.37	72.56	18.56	2.06	0.62	30.19
200 x 20	120	6.42	22.63	10.63	139.09	31.09	3.45	0.72	23.75
average		7.64						0.55	

5.5.3.2 Solution quality analysis

- **size** - A class of PFSP problem instances in the format "*jobs* x *machines*".
- **"Algorithm _ SeedingStrategy"** columns - The name of the column suggests which algorithm and which seeding strategy were set as input parameters for this experiment. For clarity, a few abbreviations are introduced.
  Algorithm abbreviations: **HG** - Hybrid Genetic, **IG** - Iterated Greedy.
  Seeding Strategy abbreviations: **SS** - SameSeeds, **SPSW** - SeedPlusSlidingWindow, **SPFW** - SeedPlusFixedWindow.

For each PFSP instance the execution is repeated ten times and the RPD value is calculated, using Equation 5.3. When we have an RPD value for each repetition, ten in our case, we compute the average for those ten repetitions, obtaining the average RPD for each instance. Then we calculate the average RPD for all the instances of the same problem size, that is, instances with equal number of jobs and machines. Derived results are presented in Table 5.6. At its bottom there is a grand average - an average over all RPDs. Grand average tells us which one

out of six (algorithm, seeding strategy) performs better than the others. The smaller the number - the higher solution quality is achieved.

As a result, on average, Iterated Greedy algorithm performs better than Hybrid Genetic algorithm, even though for instances with 200 jobs, HG algorithm is always slightly better. Regarding the seeding strategy, the winner is SameSeeds - sending the best solution towards all distributed algorithms.

Table 5.6
Experiment 2 - Average RPD

size	HG_SS	IG_SS	HG_SPSW	IG_SPSW	HG_SPFW	IG_SPFW
50 x 5	0.18	0.10	0.15	0.06	0.15	0.06
50 x 10	2.24	1.74	2.27	1.74	2.28	1.75
50 x 20	3.42	2.87	3.50	2.62	3.52	2.67
100 x 5	0.19	0.10	0.21	0.16	0.21	0.16
100 x 10	1.32	1.11	1.38	1.50	1.38	1.49
100 x 20	3.98	3.58	4.17	3.96	4.17	4.02
200 x 10	0.87	1.05	0.92	1.03	0.90	1.03
200 x 20	3.65	3.76	3.73	3.87	3.76	3.87
average	1.98	1.79	2.04	1.87	2.05	1.88

Regarding the solution quality, having the solution that is only **1.79% higher than the best known solution** can be considered a good result owing to the early age of the tool. These results provide a proof that the developed framework is a working tool that can be efficiently used for user-specific needs. In our case, the framework was used for solving the PFSP problems, but it is not limited only to them.

Obtaining better results than the ones presented in Table 5.6 requires additional calibration and more sophisticated topologies, like the asynchronous island model used to generate the results published in Vallada's paper about cooperative metaheuristics [67]. For now, we are far away from Vallada's results [2], but the tool is still young. The framework can be competitive since there are many places for the improvement. In our experiments the computation is performed inside a Java Virtual Machine on each computing node, while in the mentioned paper it is computed on a bare metal. At the moment, there are efforts to bring Apache Spark closer to the underlying hardware - through a project named Tungsten [66]. We expect that this change will increase the

[2] Although in Vallada's experiments the best solution used in RPD calculation is the best solution obtained in N independent runs, not the best known solution in the literature. Therefore, it is a relative measure.

overall performance of HyperSpark framework. Also, Scala programming language, being written in Java, is much slower than *Delphi* programming language (which has similar performance to C++) used in Vallada's experiment. The future optimisations of JVM will decrease the computational time overhead of HyperSpark.

5.6 Experiment 3 - Finding New Best Solutions

This experiment aims to find better solutions than the ones already recorded in the technical literature. Out of 120 PFSP instances, we chose 54 for which the optimal solution has not be found yet. As suggested by the results in Experiment 2, *Iterated Greedy* algorithm and *SameSeeds* seeding strategy will be fixed throughout the test. Also, in Experiment 2, the execution time provided was pretty low (e.g., 2 minutes for the largest-size problem), especially when it had to be divided by the number of computing iterations. In that kind of a setup the algorithms did not have enough time to converge towards the best known solutions. Therefore, in this experiment, the execution time will be extended by multiplying the old time with 20. Unless stated differently, the input and output parameters will be the same as in Experiment 2.

5.6.1 Input parameters

The input parameters are the same as in Experiment 2, but the value of some of them is changed. A short description of input parameters follows:

- **Number of stages** - fixed to 10. There is a cooperation between algorithms, and it is determined by a seeding strategy.
- **Number of CPU cores** - Fixed to 20, as prescribed in Experiment 1.
- **Algorithm execution time** - Here, the execution time is calculated as follows: *problem.numOfMachines*(problem.numOfJobs/2.0)*60*20 milliseconds*. The execution times of test data are given in the table Table 5.7. As in Experiment 2, this time is divided by the number of iterations (stages).
- **Algorithm** - Fixed to *IGAlgorithm*, as prescribed in Experiment 2.
- **Seeding Strategy** - Fixed to *SameSeeds*, as prescribed in Experiment 2.

Table 5.7
Experiment 3 - Problem instances

instance	jobs	machines	execution time
inst_ta{007}	20	5	00 h 01 m 00 s
inst_ta{041-043,047-050}	50	10	00 h 05 m 00 s
inst_ta{051-054,056-060}	50	20	00 h 10 m 00 s
inst_ta{077-079}	100	10	00 h 10 m 00 s
inst_ta{081,083-090}	100	20	00 h 20 m 00 s
inst_ta{091-094,096-098,100}	200	10	00 h 20 m 00 s
inst_ta{101-103,107-110}	200	20	00 h 40 m 00 s
inst_ta{111-120}	500	20	01 h 40 m 00 s

5.6.2 Output

Here, we are only interested in the obtained solution, not in the time measurement. The output parameters are:

- **Solution found** - Solution obtained by using HyperSpark framework.
- **Relative Percentage Deviation (RPD)** - Relative deviation of a solution found with respect to the best known solution, expressed in percents. This parameter serves to tell us how close the solution found is to the best known solution. If it is negative - new best solution is found.

$$RPD = \frac{Solution^{found} - Solution^{best}}{Solution^{best}} * 100 \tag{5.3}$$

5.6.3 Results

In Table 5.8 there is a full list of collected outputs.

Table 5.8
Experiment 3 - Obtained solutions

instance	**found**	**best**	**RPD**
inst_ta007	1239	1234	0.41
inst_ta041	3025	2991	1.14
inst_ta042	2894	2867	0.94
inst_ta043	2869	2839	1.06
inst_ta047	3108	3093	0.48

inst_ta048	3042	3037	0.16
inst_ta049	2905	2897	0.28
inst_ta050	3078	3065	0.42
inst_ta051	3898	3850	1.25
inst_ta053	3710	3640	1.92
inst_ta054	3764	3723	1.10
inst_ta056	3702	3681	0.57
inst_ta057	3768	3704	1.73
inst_ta058	3762	3691	1.92
inst_ta059	3792	3743	1.31
inst_ta060	3788	3756	0.85
inst_ta077	5602	5595	0.13
inst_ta078	5650	5617	0.59
inst_ta079	5891	5871	0.34
inst_ta081	6323	6202	1.95
inst_ta083	6374	6271	1.64
inst_ta084	6366	6269	1.55
inst_ta085	6411	6314	1.54
inst_ta086	6460	6364	1.51
inst_ta087	6378	6268	1.75
inst_ta088	6507	6401	1.66
inst_ta089	6386	6275	1.77
inst_ta090	6534	6434	1.55
inst_ta091	10885	10862	0.21
inst_ta093	11017	10922	0.87
inst_ta094	10893	10889	0.04
inst_ta096	10375	10329	0.45
inst_ta097	10860	10854	0.06
inst_ta098	10753	10730	0.21
inst_ta100	10727	10675	0.49
inst_ta101	11368	11195	1.55
inst_ta102	11413	11203	1.87
inst_ta103	11539	11281	2.29
inst_ta107	11545	11360	1.63
inst_ta108	11568	11334	2.06
inst_ta109	11407	11192	1.92
inst_ta110	11521	11168	3.16
inst_ta111	26472	25931	2.09

inst_ta112	26977	26520	1.72
inst_ta113	26724	26371	1.34
inst_ta114	26731	26456	1.04
inst_ta115	26564	26334	0.87
inst_ta116	26843	26477	1.38
inst_ta117	26619	26389	0.87
inst_ta118	26914	26560	1.33
inst_ta119	26395	26005	1.50
inst_ta120	26770	26457	1.18
average			1.19

Unfortunately, there is no occurrence of negative RPD in Table 5.8 - the best known solution was not found for any of the tested PFSP instances. However, compared to Experiment 2, the increased execution time enhanced the quality of solutions. Having a solution that is 1.19 % far from best known solution is still a very good result, especially for $\mathcal{NP}$-Complete problems like PFSP [65]. An improvement could be achieved by saving the state of an algorithm in between the iterations, since each algorithm is destroyed and re-instantiated during each synchronisation point. This has the effect of potential exploration of already visited areas of the solution space. If an algorithm is able to remember the searched areas it can use the provided execution time more effectively. Previously mentioned asynchronous model may be a key to solving this issue, since the algorithm state is preserved during its run-time.

CHAPTER 6

Conclusions and Future Work

Throughout this thesis we investigated the challenges of using distributed computing platforms. After exploring the technical aspects related to Big Data processing and existing technologies, we analysed the advantages and drawbacks, and chose a particular technology which fits the best in High Performance Computing paradigm. The goal of this approach is to deliver high-performance over the whole structure of computing cluster. Once the underlying technology was determined, we focused on the development of framework that could be exploited for distributed solving of hard optimisation problems. The framework was named *HyperSpark*, because of the possibility to manage the scalable execution of meta-heuristic algorithms. We have seen that *hyperReduce* and *SeedingStrategy* are powerful operators of HyperSpark framework, providing users a simple, effective and flexible way to distribute and aggregate data for their specific algorithms. Algorithms injected in the framework might be computationally-intensive or light-weight, making it attractive choice for variety of needs. Also, the synchronous communication between algorithms has been enabled through a series of computing iterations. In Appendix B (Users Manual) it is shown how simple it is to make one application using HyperSpark, almost with no setup of Spark environment parameters.

The proposed framework was validated through benchmarks designed for a specific optimisation problem from scheduling theory - Permutation Flow Shop Problem (PFSP), for which we also developed a software library. Once the computational time was analysed, the number of parallel algorithm instances for which the percentage of time overhead is acceptable was set. Indeed, there is a high overhead for small problem instances, but for large-sized problems it becomes less significant. It has been concluded that the framework is especially suitable for executing long-running jobs. Later, we have analysed the quality of

obtained solutions. We employed two different algorithms and three aggregation strategies and presented the best performing configuration of the framework. The results were compared with best known solutions in the technical literature. Small deviations from best known solutions provide the evidence to support the future use of proposed framework as a scalable and accurate distributed tool for the PFSP.

Building upon the outcomes of this work, it is possible to investigate further open issues and relevant research questions. An aspect that might strongly affect performance of distributed algorithm execution in Spark clusters is asynchronous communication. If the algorithms were to communicate best results only when they found a better solution, the network time required to transfer data between nodes would not add to the job completion time. However, when it comes to asynchronous communication Spark lacks the transparent support. The synchronous dataflow of Spark forced us to instantiate algorithms in every new computing iterations, with updated best known solution. Certain algorithms may require more time to converge to satisfying solution, and therefore the time limit we set for one computing iteration was possibly not enough to express their real potential. With asynchronous communication the algorithms would not need to be destroyed and instantiated again. The algorithm would simply, when convenient, query the list of received solutions and continue its execution. Establishing the asynchronous communication in Spark is still possible by creating a new communication layer over the local network [22], using *Akka Actor* Scala library [34]. However, that would require the work on its own.

Similarly, there is still a place for the improvement of diversity in the exploration of solution space by different algorithm instances. The more diverse the algorithms are on different computing resources, the higher is the probability to improve the current solution. This behavior can be modeled by instantiating different types of algorithms on the computing resources or by adding different parameter values in initialisation phase of the algorithms, thus guiding the algorithm exploration to a different direction in the solution space.

In the end, we should be aware that the overall performance of the framework depends on the chosen inner-components and calculation procedures. For instance, TSAB algorithm uses an optimised solution evaluation procedure that takes less time than the simple procedure we have implemented. That could be the reason why in our preliminary tests TSAB did not perform as in the published paper [41]. Regarding the components selection, the overall performance of the cluster can greatly benefit from the choice of particular algorithms, cooperation techniques and solution space exploration strategies. At last, the goal of Big Calculations paradigm is building scalable, resource-sharing, high-performance systems. With *HyperSpark*, we hope that goal has been achieved.

APPENDIX A

HyperSpark Project Details

Section A.1 describes the overall structure of the HyperSpark project.
Section A.2 provides a full list of FrameworkConf methods that are used to set Spark-specific properties.

A.1 Packages Organization

Packages of HyperSpark project and containing classes are depicted in Figure A.1. Some of the packages contain use-case specific classes. Their description can be found in Chapter 4.

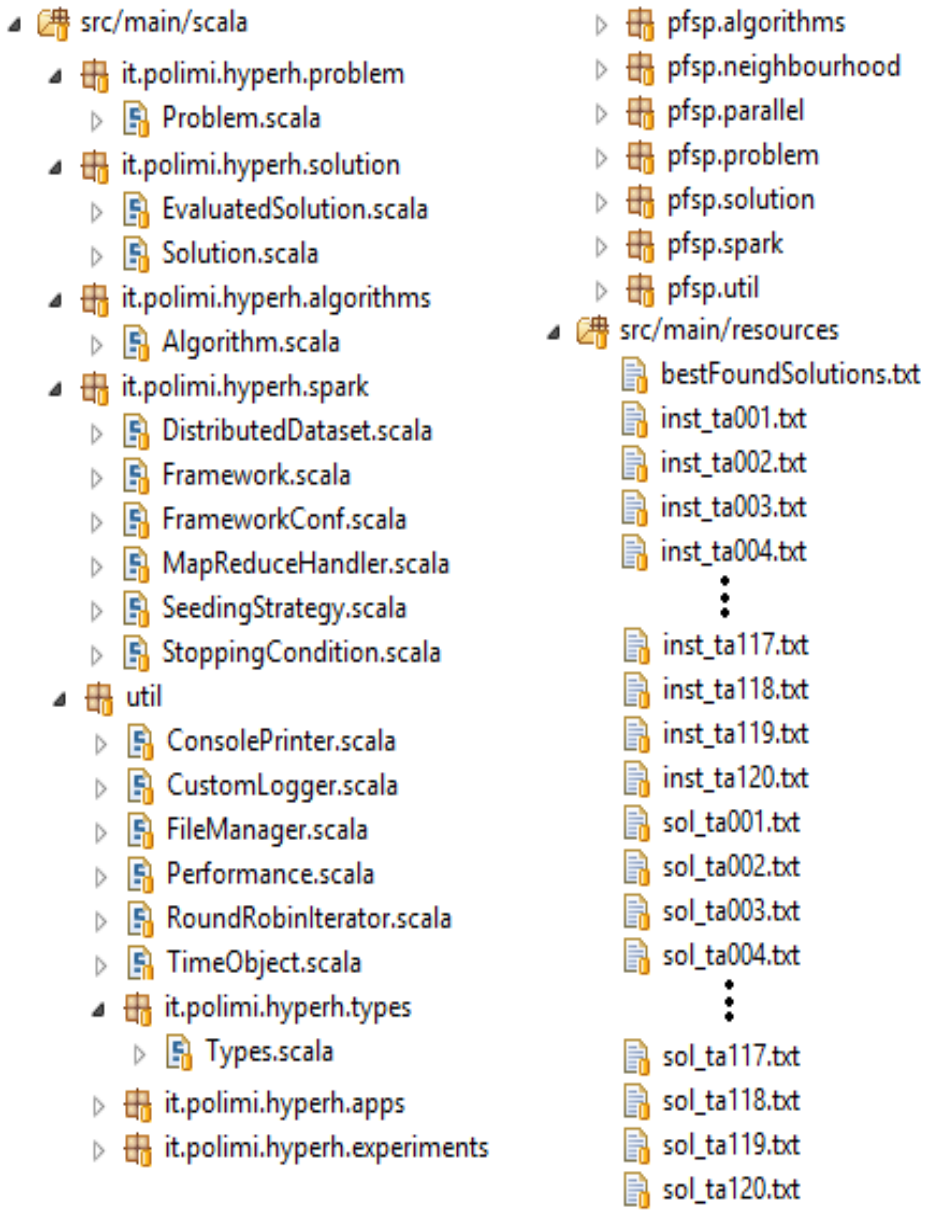

Figure A.1 – HyperSpark Project - Packages Organization

A brief description of each package follows.

1. ***it.polimi.hyperh.problem*** - Contains the Problem class used to define a problem that needs to be solved by the framework. It is an abstract class that needs to be extended in order to define a user-specific problem.
2. ***it.polimi.hyperh.solution*** - Contains Solution and EvaluatedSolution classes. These concepts were described in Section 3.1, and their implementation will be shown in Framework internals section (Section 3.5).
3. ***it.polimi.hyperh.algorithms*** - Contains the trait *Algorithm*. Each specific algorithm needs to extend the trait Algorithm in order to be transformed into a distributed one by the framework.
4. ***it.polimi.hyperh.spark*** - This the main package in the project, containing the core of HyperSpark framework. It consists of DistributedDataset file, which encapsulates DistributedDatum class and an auxiliary Scala object DistributedDataset. Already mentioned MapReduceHandler, SeedingStrategy, StoppingCondition and FrameworkConf concepts are placed in their respective classes in this package. Framework, belonging to the same package, is a Scala object.
5. ***util*** - Contains various utility classes (FileManager, CustomLogger, ConsolePrinter) that support input/output operations, encapsulation and printing of time (TimeObject), solution quality measurement (Performance), and so on.
6. ***it.polimi.hyperh.types*** - Types.scala file contains abbreviations/aliases for Scala types.
7. ***it.polimi.hyperh.experiments*** - Contains the experiment we conducted for the purpose of this thesis. Experiments will be presented in Chapter 5.
8. ***it.polimi.hyperh.apps*** - Consists of the applications developed as examples of HyperSpark Framework usage.
9. ***pfsp.**** - Packages with a prefix pfsp contain a concrete implementation of a problem, its solution representation, algorithms that solve it, specific seeding strategies and so on. Corresponding classes are presented in Chapter 4.
10. ***resources*** - Contains use-case specific (Taillard's) problem instances and their respective best-found solutions.

The first five packages comprise the core of HyperSpark Framework. Packages 6-10 are the auxiliary packages, used to provide examples, or functionalities necessary for presenting the use case of Chapter 4.

A.2 Spark Properties used by the Framework

```
def setProperty(key: String, value: String)
def getProperties()
//for a full spark properties reference visit
//http://spark.apache.org/docs/latest/configuration.html

def setAppName(name: String)

def setSparkMaster(url: String)
def getSparkMaster()

def setDeploymentLocalNoParallelism()
def setDeploymentLocalMaxCores()
def setDeploymentLocalNumExecutors(numExecutors: Int)
def setDeploymentSpark(host: String, port: Int)
def setDeploymentSpark(host: String)
def setDeploymentMesos(host: String, port: Int)
def setDeploymentMesos(host: String)
def setDeploymentYarnClient()
def setDeploymentYarnCluster()

def setNumberOfExecutors(N: Int)
def setNumberOfResultingRDDPartitions(N: Int)
```

Figure A.2 – Framework Configuration - Spark-specific Properties

The function ***setNumberOfExecutors*** sets the parameter "*spark.executor.instances*" to the number *numExecutors* specified in the function signature. This parameter is very important since it is responsible for scaling the execution of algorithms. Therefore, when a user sets an array of Algorithms in FrameworkConf, setNumberOfExecutors is internally called to change the scale of computation - number of executors assigned for the application is passed to the Spark environment and set before the execution of application is started.

Note: We noticed that sometimes the Spark environment does not accept to set the number of executors to the desired number, especially when the number of executors is greater than the number of CPU cores present in the cluster. Also, in the case when the number of cores in the cluster is less than the virtual/simulated number of cores, e.g. simulate the execution of two virtual (logical) cores on one phisical core. This is called Hyper-Threading [37]. In those cases Spark sets the number of executors to 2 by default. The solution to this problem is to pass the number of executors to spark-submit script directly

through command line arguments, when the pre-packed jar archive of his/her application is submitted.

Setter functions for **changing the deployment modes** are basically internally changing the spark master URL, and their meaning was mostly explained in Subsection 2.4.5 when there was a talk about the deployment modes. Please refer to the mentioned section.

"**Local deployment mode**" has a purpose of enabling a user to run the Spark application on a local PC. Although this is not an actual cluster deployment mode, it is very useful for checking the correct execution of application before submitting it to the real cluster. ***setDeploymentLocal**** function signatures change the deployment to local and the number of executors instances. For example:

- ***setDeploymentLocalNoParallelism*** means that the number of executors is set to one, and therefore there is no parallelism in the execution. Only one instance of the algorithm provided to the framework will start its execution.
- ***setDeploymentLocalNumExecutors*** - sets local debugging mode and the number of executors to the number specified in a function call.
- ***setDeploymentLocalMaxCores*** - sets local debugging mode and the number of executors to the maximum number of cores present in a local PC. Therefore, this option uses all computing resources (cores) of one PC for the computation.

setNumberOfResultingRDDPartitions function changes the parameter "*spark.default.parallelism*" which determines the number of partitions of an RDD when that number is not provided by the user, i.e., the default value of the second parameter in *parallelize* function. Just in case, when user sets the number of algorithms in FrameworkConf, besides the number of executors, we also set the default number of RDD partitions to the algorithms array size.

APPENDIX B

User Manual

This appendix will briefly explain how to use HyperSpark Framework for problem solving using any user-defined algorithm. For now, we have implemented one problem type that is considered in our case study (Chapter 4), but in a near future HyperSpark will be enriched with more diverse problems and algorithms.

The structure of Appendix B is as follows.

Section B.1 explains the prerequisites for using HyperSpark framework.

Section B.2 provides guidelines for creating HyperSpark applications.

Section B.3 presents a few examples of HyperSpark applications that exemplify the provided guidelines.

B.1 Prerequisites

Since HyperSpark framework is a Scala-based project, it requires Java Runtime Environment (JRE) version 1.6 or later, and Scala 2.10 or 2.11 binaries installed on a local PC.

The framework comes pre-packed inside a jar archive. The jar is light-weight, meaning that it relies on the dependencies installed in the environment on which the Framework is ran on. The cluster on which we tested HyperSpark had Scala v.2.10 and Spark v.1.3 installed. Even though it was tested on Spark v.1.3, recommended version of Spark installation is 1.4.1 or higher, because of the bugs present in Spark v.1.3 that were fixed in the later versions. As of Scala concerns, versions newer than v.2.10 keep a backward-compatibility with it.

B.2 Guidelines

Usage is quite strait-forward:

1. Import HyperSpark Framework jar archive inside an existing Scala project.
2. Extend the *Solution* class and encapsulate solution representation within variables.
3. Extend the *EvaluatedSolution* and override *value* and *solution* variable types. For reference, take a look at *pfsp.PfsEvaluatedSolution*.
4. Extend the *Problem* class and implement the following method:
   evaluate(s: Solution): EvaluatedSolution
5. Optionally, create a custom stopping condition for the algorithms by extending the *StoppingCondition* class, or use later the existing one *TimeExpired*.
6. Extend the *Algorithm* trait and implement the following evaluate methods inside your custom algorithm class:
   evaluate(p:Problem): EvaluatedSolution
   evaluate(p:Problem, stopCond:StoppingCondition): EvaluatedSolution
7. Write your own application that uses HyperSpark framework.
8. Instantiate a new problem.
9. Create an instance of a specific algorithm, implemented in step 6.
10. Create a FrameworkConf object.

11. In FrameworkConf set a problem, algorithms array, initial solutions (seeds) array and a stopping condition for the algorithms. Optionally, change the default number of iterations, MapReduceHandler, or a SeedingStrategy.

12. Invoke *Framework.run(conf: FrameworkConf)* to obtain one *EvaluatedSolution*, or *Framework.multipleRuns(conf:FrameworkConf, runs:Int)* to obtain multiple *EvaluatedSolution*s.

13. If run locally - run the application as a regular Scala Application.

14. If run on a cluster - Package your application to a jar that contains all the transitive dependencies, in order to avoid classpath problems in the distributed environment. Use spark-submit script to submit a jar to the cluster, e.g.:
spark-submit - -class AppClassPath JarName.jar AppParameters

B.3 Examples

it.polimi.hyperh.apps contains numerous examples of existing HyperSpark applications. We will show two simple applications: one used on a local PC and one used on a real cluster.

The application presented in Figure B.1 shows how one can convert a single-thread algorithm to a parallel one executed in a distributed setting, without the need to set up the parameters of Spark environment. In this example we used the existing *IGAlgorithm*, which implements two methods mentioned in step 6 of the guidelines. The problem that we want to solve is a permutation flow shop problem (PfsProblem), and it is loaded from a textual file "inst_ta001.txt" in resources package. The problem can be also instantiated by using "*new CustomProblem(...)*". Therefore, we instantiate a probem and a single-threaded algorithm (in this case an IGAlgorithm). We want to have 4 parallel algorithm instances running at the same time in our cluster, so we save that number in a variable called *numOfAlgorithms*. Next, we create a framework configuration object initialised using the builder pattern that sets number of executors, problem, algorithms, initial seeds and stopping condition, respectively. For each algorithm instance one parallel task will be created, and when the tasks are scheduled they will be sent to free Executors. By default, each Spark Executor is occupying one CPU core. Therefore, each algorithm instance will be executed on one physical core. Since we did not specify the number of iterations (which is one) the algorithms will be executed in one iteration (superstep) of the framework.

```
package it.polimi.hyperh.apps

import it.polimi.hyperh.problem.Problem
import it.polimi.hyperh.spark.Framework
import it.polimi.hyperh.spark.FrameworkConf
import it.polimi.hyperh.spark.TimeExpired
import pfsp.problem.PfsProblem
import pfsp.algorithms.IGAlgorithm

object LocalApp {
def main(args: Array[String]) {
val problem = PfsProblem.fromResources("inst_ta001.txt")
val algorithm = new IGAlgorithm()
val numOfAlgorithms = 4
val iterTimeLimit = 3000 //milliseconds
val stopCond = new TimeExpired(iterTimeLimit)

val conf = new FrameworkConf()
.setDeploymentLocalNumExecutors(numOfAlgorithms)
.setProblem(problem)
.setNAlgorithms(algorithm, numOfAlgorithms)
.setNDefaultInitialSeeds(numOfAlgorithms)
.setStoppingCondition(stopCond)

val solution = Framework.run(conf)
println(solution)
}
}
```

Figure B.1 – Framework Usage - Example1

In the second example, Figure B.2, we show how the same application is executed on a real cluster. Here, the only thing that is different is the deployment mode set inside the configuration object. It is changed from local (debugging mode) to Yarn Cluster mode (*setDeploymentYarnCluster*), meaning that the application will be submitted to a cluster that has Spark installed and running on top of Yarn Resource Manager. The application needs to be packed in a jar archive, which is going to be uploaded in the file system of a cluster and submitted using the following command:

spark-submit - -class it.polimi.hyperh.apps.LocalApp hyperSpark.jar

```
package it.polimi.hyperh.apps

import it.polimi.hyperh.problem.Problem
import it.polimi.hyperh.spark.Framework
import it.polimi.hyperh.spark.FrameworkConf
import it.polimi.hyperh.spark.TimeExpired
import pfsp.problem.PfsProblem
import pfsp.algorithms.IGAlgorithm

object LocalApp {
def main(args: Array[String]) {
val problem = PfsProblem.fromResources("inst_ta001.txt")
val algorithm = new IGAlgorithm()
val numOfAlgorithms = 4
val iterTimeLimit = 3000 //milliseconds
val stopCond = new TimeExpired(iterTimeLimit)

val conf = new FrameworkConf()
.setDeploymentYarnCluster()
.setProblem(problem)
.setNAlgorithms(algorithm, numOfAlgorithms)
.setNDefaultInitialSeeds(numOfAlgorithms)
.setStoppingCondition(stopCond)

val solution = Framework.run(conf)
println(solution)
}
}
```

Figure B.2 – Framework Usage - Example2

Acronyms

AM Application Master. 22

API application programming interface. 16

CPU central processing unit. 6, 7, 10, 21, 23

DAG Directed Acyclic Graph. vii, 29, 30, 32, 52

FIFO First In, First Out. 22, 23

GFS Google File System. 17

GPU graphics processing unit. 23

HDFS Hadoop Distributed File System. vii, 17–21, 29

HPC High Performance Computing. 6, 35, 37, 41, 48, 109

ICT Information and Communication Technology. 10, 23

JN Journal Node. 20

JVM Java Virtual Machine. 33, 104, 105

MOF Meta-heuristics Optimisation Framework. 77

NDFS Nutch Distributed File System. 18

NSF National Science Foundation. 12

OS operating system. 18, 20

PFSP Permutation Flow Shop Problem. ix, 1, 8, 71–73, 75, 76, 78, 95, 100, 104, 105, 108–110

POSIX Portable Operating System Interface. 20

RDD Resilient Distributed Dataset. iv, vii, 27–29

RM Resource Manager. 21, 22

RPD Relative Percentage Deviation. 101, 103, 106, 108

TSP Travelling Salesman Problem. 56

YARN Yet Another Resource Negotiator. vii, 20, 21, 23

Bibliography

[1] Enrique Alba, Gabriel Luque, Jose Garcia-Nieto, Guillermo Ordonez, and Guillermo Leguizamon. Mallba a software library to design efficient optimisation algorithms. *Int. J. Innov. Comput. Appl.*, 1(1):74–85, April 2007.

[2] Enrique Alba, Gabriel Luque, and Sergio Nesmachnow. Parallel metaheuristics: recent advances and new trends. *International Transactions in Operational Research*, 20(1):1–48, 2013.

[3] Apache Spark. `http://spark.apache.org/`. Accessed: 2015-10-30.

[4] Spark Programming Guide - Bagel. `https://spark.apache.org/docs/0.7.2/bagel-programming-guide.html`. Accessed: 2015-11-05.

[5] Leonora Bianchi, Marco Dorigo, Luca Maria Gambardella, and Walter J. Gutjahr. A Survey on Metaheuristics for Stochastic Combinatorial Optimization. *Natural Computing*, 8(2):239–287, June 2009.

[6] Gartner Supply Chain Executive Conference - To advance Big Data conversation to Big Calculations. `http://www.quintiq.com/news-2014/quintiq-to-advance-big-data-conversation-to-big-calculations-at-gartner.html`. Accessed: 2015-11-04.

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

[8] Hadoop MapReduce next generation - Capacity Scheduler. `http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html`. Accessed: 2015-03-12.

[9] Jens Clausen. Branch and bound algorithms-principles and examples. *Parallel Computing in Optimization*, pages 239–267, 1997.

[10] Stephen A. Cook. The Complexity of Theorem-proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[11] DataMPI library for Big Data support in MPI. `http://datampi.org/`. Accessed: 2015-11-15.

[12] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating Systems Design and Implementation*, pages 137–149, 2004.

[13] Marco Dorigo and Thomas Stützle. Ant Colony Optimization: Overview and Recent Advances. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research and Management Science*, pages 227–263. Springer US, 2010.

[14] Xin Du, Youcong Ni, Zhiqiang Yao, Ruliang Xiao, and Datong Xie. High Performance Parallel Evolutionary Algorithm Model Based on MapReduce Framework. *Int. J. Comput. Appl. Technol.*, 46(3):290–295, March 2013.

[15] Werner Dubitzky and Francisco Azuaje. *Artificial intelligence methods and tools for systems biology*. Springer, 2004.

[16] ECJ - A Java-based Evolutionary Computation Research System. `http://cs.gmu.edu/~eclab/projects/ecj/`. Accessed: 2016-03-11.

[17] Pedro Fazenda, James McDermott, and Una-May O'Reilly. A library to run evolutionary algorithms in the cloud using mapreduce. In *Proceedings of the 2012T European Conference on Applications of Evolutionary Computation*, EvoApplications'12, pages 416–425, Berlin, Heidelberg, 2012. Springer-Verlag.

[18] Filomena Ferrucci, Pasquale Salza, M-Tahar Kechadi, and Federica Sarro. A Parallel Genetic Algorithms Framework Based on Hadoop MapReduce. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1664–1667, New York, NY, USA, 2015. ACM.

[19] Fedor V. Fomin and Petteri Kaski. Exact Exponential Algorithms. *Commun. ACM*, 56(3):80–88, March 2013.

[20] Mario García-Valdez, Leonardo Trujillo, Francisco Fernández Vega, Juan Julián Merelo Guervós, and Gustavo Olague. *EvoSpace-Interactive: A Framework to Develop Distributed Collaborative-Interactive Evolutionary Algorithms for Artistic Design*, pages 121–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[21] Fred Glover. Future Paths for Integer Programming and Links to Artificial Intelligence. *Comput. Oper. Res.*, 13(5):533–549, May 1986.

[22] Gonzalez, Jesus. Asynchronous Complex Analytics in a Distributed Dataflow Architecture. http://blog.acolyer.org/2015/11/26/asip/. Accessed: 2016-03-13.

[23] Apache Hadoop. https://hadoop.apache.org. Accessed: 2015-03-31.

[24] M. Hamstra, H. Karau, M. Zaharia, A. Konwinski, and P. Wendell. *Learning Spark: Lightning-Fast Big Data Analytics*. O'Reilly Media, Incorporated, 2015.

[25] HPC and Big Data - A "Best of Both Worlds" Approach. http://www.hpcwire.com/2014/03/31/hpc-big-data-best-worlds-approach/. Accessed: 2015-11-05.

[26] Di-Wei Huang and Jimmy Lin. Scaling Populations of a Genetic Algorithm for Job Shop Scheduling Problems Using MapReduce. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 780–785, Washington, DC, USA, 2010. IEEE Computer Society.

[27] Robert Hundt. Loop Recognition in C++/Java/Go/Scala. In *Proceedings of Scala Days 2011*, 2011.

[28] Shantenu Jha, Judy Qiu, André Luckow, Pradeep Kumar Mantha, and Geoffrey Charles Fox. A Tale of Two Data-Intensive Paradigms: Applications, Abstractions, and Architectures. *CoRR*, abs/1403.1528, 2014.

[29] Chao Jin, C. Vecchiola, and R. Buyya. MRPGA: An Extension of MapReduce for Parallelizing Genetic Algorithms. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 214–221, Dec 2008.

[30] J. Kennedy and R. Eberhart. Particle Swarm Optimization. In *Proceedings of IEEE International Conference on Neural Networks*, pages 1942–1948. IEEE, 1995.

[31] Samia Kouki and Mohamed Jemni. Solving the Permutation Flow Shop Problem with Makespan Criterion using Grids. *International Journal of Grid and Distributed Computing*, 4(2), 2011.

[32] Samia Kouki, Mohamed Jemni, and Talel Ladhari. Solving the Permutation Flow Shop Problem with Makespan Criterion using Grids. *International Journal of Grid and Distributed Computing*, 4, 2011.

[33] Marcel Kronfeld, Hannes Planatscher, and Andreas Zell. The eva2 optimization framework. In *Proceedings of the 4th International Conference on Learning and Intelligent Optimization*, LION'10, pages 247–250, Berlin, Heidelberg, 2010. Springer-Verlag.

[34] Kuhn, Roland and Nordwall, Patrik and Varga, Endre and Malawski, Konrad and Mickevicius, Martynas. Akka Actor system - Scala library for asynchronous communication. http://doc.akka.io/docs/akka/2.4.1/scala/actors.html. Accessed: 2016-03-13.

[35] Solving Hard Problems with Lots of Computers. https://cs.brown.edu/research/pubs/theses/ugrad/2012/ryza.pdf.

[36] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[37] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, Alan J. Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1):4–15, Feb 2002.

[38] Parallel Monte Carlo using Scala. https://darrenjw.wordpress.com/2014/02/23/parallel-monte-carlo-using-scala/. Accessed: 2015-11-01.

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

[40] MIKE NOVELS. A new approach to capacity planning and scheduling. *International Journal of Dairy Technology*, 49(2):49–52, 1996.

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

[42] Nsf data intensive computing research program. http://www.nsf.gov/funding/pgm_summ.jsp?pims_id=503324&org=IIS. Accessed: 2015-10-30.

[43] Ih Osman and Cn Potts. Simulated annealing for permutation flow-shop scheduling. *Omega*, 17(6):551–557, 1989.

[44] Paradiseo - A software framework for metaheuristics. http://paradiseo.gforge.inria.fr/. Accessed: 2016-03-11.

[45] Jose Parejo, Antonio Ruiz-Cortes, Sebastian Lozano, and Pablo Fernandez. Metaheuristic Optimization Frameworks: A Survey and Benchmarking. *Soft Comput.*, 16(3):527–561, March 2012.

[46] Columbia University - Lecture Notes on Production Scheduling. www.columbia.edu/~cs2035/courses/ieor4405.S09/perm.pdf. Accessed: 2015-11-20.

[47] Comaprison between the fastest benchmark programs for selected programming languages. http://benchmarksgame.alioth.debian.org/u64q/which-programs-are-fastest.html. Accessed: 2016-01-08.

[48] Atanas Radenski. Distributed simulated annealing with mapreduce. In *Proceedings of the 2012T European Conference on Applications of Evolutionary Computation*, EvoApplications'12, pages 466–476, Berlin, Heidelberg, 2012. Springer-Verlag.

[49] Chandrasekharan Rajendran and Hans 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.

[50] Deepak Ravichandran, Patrick Pantel, and Eduard Hovy. The terascale challenge. In *Proceedings of KDD Workshop on Mining for and from the Semantic Web (MSW-04)*, pages 1–11, 2004.

[51] Big Data Processing in Spark. http://horicky.blogspot.it/2015/02/big-data-processing-in-spark.html. Accessed: 2015-11-03.

[52] Spark Programming Guide - RDD Transformations. http://spark.apache.org/docs/latest/programming-guide.html#transformations. Accessed: 2015-11-03.

[53] Colin R. Reeves. A Genetic Algorithm for Flowshop Sequencing. *Comput. Oper. Res.*, 22(1):5–13, Jan 1995.

[54] Gerhard Reinelt. *The Traveling Salesman: Computational Solutions for TSP Applications*. Springer-Verlag, Berlin, Heidelberg, 1994.

[55] Umit Rencuzogullari and Sandhya Dwardadas. Dynamic adaptation to available resources for parallel computing in an autonomous network of workstations. In *Proceedings of the Eighth ACM SIGPLAN Symposium on*

Principles and Practices of Parallel Programming, PPoPP '01, pages 72–81, New York, NY, USA, 2001. ACM.

[56] Jorge L. Reyes-Ortiz, Luca Oneto, and Davide Anguita. Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf. *Procedia Computer Science*, 53:121–130, 2015. INNS Conference on Big Data 2015 Program San Francisco, CA, USA 8-10 August 2015.

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

[58] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (3rd Edition)*. Prentice Hall, 3 edition, December 2009.

[59] MLLib - Apache Spark's Machine Learning Library. `http://spark.apache.org/mllib/`. Accessed: 2015-07-06.

[60] Spark Programming Guide - Configuration properties. `http://spark.apache.org/docs/latest/configuration.html#application-properties`. Accessed: 2015-07-30.

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

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

[63] Taillard's Instances for Flow Shop Scheduling. `http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/ordonnancement.html`. Accessed: 2016-02-02.

[64] Qing Tan, Qing He, and Zhongzhi Shi. Parallel Max-Min Ant System Using MapReduce. In Ying Tan, Yuhui Shi, and Zhen Ji, editors, *Advances in Swarm Intelligence*, volume 7331 of *Lecture Notes in Computer Science*, pages 182–189. Springer Berlin Heidelberg, 2012.

[65] Vincent T'Kindt, Jean-Charles Billaut, and Henry Scott. *Multicriteria scheduling : theory, models and algorithms*. Springer, Berlin, 2002.

[66] Project Tungsten: Bringing Spark Closer to Bare Metal. `https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html`. Accessed: 2016-03-11.

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

[68] John Paul Walters and Vipin Chaudhary. Replication-Based Fault Tolerance for MPI Applications. *IEEE Trans. Parallel Distrib. Syst.*, 20(7):997–1010, July 2009.

[69] Darrell Whitley, Soraya Rana, and Robert B. Heckendorn. The Island Model Genetic Algorithm: On Separability, Population Size and Convergence. *Journal of Computing and Information Technology*, 7:33–47, 1998.

[70] Seung Woo, Son Samuel, Lang Robert, Latham Robert, and Ross Rajeev Thakur. Reliable MPI-IO through Layout-Aware Replication. In *Proceedings of 7th IEEE International Workshop on Storage Network Architecture and Parallel I/O*, 2011.

[71] B. Wu, G. Wu, and M. Yang. A MapReduce based Ant Colony Optimization approach to combinatorial optimization problems. In *Natural Computation (ICNC), 2012 Eighth International Conference on*, pages 728–732, May 2012.

[72] Bihan Wu, Gang Wu, and Mengdong Yang. A MapReduce based Ant Colony Optimization approach to combinatorial optimization problems. In *Natural Computation (ICNC), 2012 Eighth International Conference on*, pages 728–732, May 2012.

[73] Lei Xu and Erkii Oja. Improved Simulated Annealing, Boltzmann Machine, and Attributed Graph Matching. In *Proceedings of the EURASIP Workshop 1990 on Neural Networks*, pages 151–160, London, UK, UK, 1990. Springer-Verlag.

[74] X. S. Yang and Suash Deb. Cuckoo Search via Levy flights. In *Nature Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*, pages 210–214, Dec 2009.

[75] 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.