

POLITECNICO DI MILANO

DEPARTMENT OF ELECTRONICS, INFORMATICS AND BIOENGINEERING
M.Sc. COURSE ENGINEERING OF COMPUTING SYSTEMS

ParadisEO vs HyperSpark: Analysis and benchmarking of two frameworks for distributed metaheuristics

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:

Camilo MARTÍNEZ

Matriculation ID 814895

Academic year 2015-2016

*To my loving wife, without her support this would not be possible
To my family which always pointed me in the right direction*

Contents

Contents	iv
List of Tables	vii
List of Listings	viii
1 Introduction	5
2 HyperSpark : A Framework for Parallel Metaheuristics over Spark	9
2.1 Technologies used	9
2.1.1 Apache Spark	9
2.1.2 Scala	10
2.1.3 IntelliJ IDEA	10
2.2 Building blocks	10
2.3 Project website	11
2.4 Repository	11
3 ParadisEO : A Software Framework for Metaheuristics	13
3.1 Technologies used	13
3.1.1 Message Passing Interface (MPI)	14
3.1.2 Programming language: C++	14
3.2 Framework Architecture	15
3.3 Compilation	16
3.4 Executable distribution	16
3.5 Multiobjective optimization	16
3.6 Repository	17
4 Case Study: Permutation Flow Shop Problem	19
4.1 Problem definition	19
4.2 Problem objective: Makespan	20
4.3 Solution evaluation: Relative Percentage Deviation (RPD)	20
4.4 Platform overhead: Relative Execution Overhead (REO)	21

4.5	Problem example	21
4.6	Complexity and metaheuristics	22
5	Implementation	25
5.1	The test algorithm	25
5.1.1	Introduction	25
5.1.2	A simple GA for the PFSP	27
5.2	Distributed execution	29
5.3	Benchmark problems	30
5.4	HyperSpark	31
5.4.1	HyperSpark-PFSP library	31
5.4.2	Distributed execution	33
5.4.3	Framework configuration	33
5.4.4	Improvements	34
5.4.5	Repository	35
5.5	Paradiseo	35
5.5.1	PFSP example	35
5.5.2	Highlighted classes	36
5.5.3	Algorithm configuration	36
5.5.4	Framework extension	38
5.6	Repository	39
6	Experimental Results	41
6.1	Experiments settings	41
6.2	Execution environment	42
6.3	Experiment execution	42
6.3.1	HyperSpark	42
6.3.2	ParadisEO	43
6.4	Experiment results	44
6.4.1	Solution quality	44
6.4.2	Execution time overhead	45
7	Conclusions and Future Work	47
A	Experiment details	49
	Bibliography	71

List of Tables

4.1	Processing times example	22
4.2	Completion times for $s = (2, 1, 3, 0)$	22
A.1	Benchmark instances metadata	49
A.2	Hyperspark benchmark: Relative Percentage Deviation (RPD)	53
A.3	ParadisEO benchmark: Relative Percentage Deviation (RPD)	57
A.4	HyperSpark benchmark: Relative Execution Overhead (REO)	61
A.5	ParadisEO benchmark: Relative Execution Overhead (REO)	65

List of Listings

5.1	Sample input file (benchmark instance 1)	31
5.2	PfsProblem class declaration	32
5.3	PfsProblemParser for benchmark input file	32
5.4	DistributedDatum for encapsulating algorithm and data	33
5.5	Algorithm execution configuration	34
5.6	Benchmark input file parser	37
5.7	Makespan and completion times matrix computation	38
6.1	Command for HyperSpark experiment execution (bash)	43
6.2	Command for ParadisEO experiment execution (bash)	43
6.3	Hostsfile for ParadisEO mpi execution	44
A.1	ParadisEO parameters configuration	69

Abstract

Computing power has grown exponentially following Moore's law and it has given rise to opportunities to leverage commodity hardware by executing computationally intensive operation in a parallel fashion.

Nonetheless, setting up a distributed computation cluster used to be a time consuming process that required expert knowledge. Developing applications capable of running on those clusters also required in-depth understanding of the frameworks available. In this work, I will analyze under several points of view two frameworks; one of them is *ParadisEO*, a C++ white-box object-oriented framework dedicated to design and execution of metaheuristics [11] that supports parallel and distributed architectures using the Message Passing Interface (MPI), a popular standard inside the High Performance Computing (HPC) community.

Recently, there have been important developments in open source systems capable of handling distributed problems in an efficient way with a higher level application programmer interface (API), while providing cluster management tools aiming at reducing the level of expertise required to develop applications for those systems. *HyperSpark*, the second reference framework considered in the comparison, is a general, extensible and portable solution for scalable execution of user-defined, computationally-intensive algorithms build on top of *Apache Spark* and developed at Politecnico di Milano.

For both platforms I will report the efforts required to develop a solution for a well-known scheduling optimisation problem: Permutation Flow Shop Problem (PFSP). I will as well expand into the empirical results obtained on 120 instances of a common benchmark for this type of problem introduced by Talliard [27]. The applications developed have been evaluated using the relative difference to the best solution found. The overhead introduced by each platform will also be analysed.

Abstract in Italian

Negli anni, la potenza di calcolo disponibile in funzione del cost è cresciuta in modo esponenziale seguendo la legge di Moore. In particolare è possibile disporre, a costi relativamente contenuti, di commodity hardware di buona qualità collegato tramite reti ad alta velocità. Questa situazione ha portato alla nascita di piattaforme che semplificano la realizzazione di applicazioni parallele di calcolo intensivo

Ciò nonostante, la realizzazione di un cluster per il calcolo distribuito resta un'operazione tutt'altro che semplice e richiede utenti esperti. Allo stesso modo anche lo sviluppo di applicazioni per questi sistemi è complesso e richiede di l'utilizzo di strumenti (framework) con una curve di apprendimento ripide. Lo scopo di questo lavoro è analizzare sotto diversi punti di vista due piattaforme; la prima, *ParadisEO*, è un framework orientato agli oggetti, sviluppato in C++, dedicato alla progettazione ed esecuzione di metaeuristiche [11] su architetture parallele e distribuite utilizzando la interfaccia per il trasporto dei messaggi (MPI). MPI è da considerarsi uno standard all'interno della comunità di calcolo ad alte prestazioni (HPC).

HyperSpark, il secondo framework considerato, è una soluzione generale, estensibile e portatile per l'esecuzione scalabile e parallela di algoritmi di ottimizzazione definiti dall'utente; è stato realizzato sviluppato presso il Politecnico di Milano e funziona su *Apache Spark*. Recentemente, infatti, ci sono stati importanti sviluppi nei sistemi open source in grado di gestire esecuzioni distribuite in modo efficiente con un interfaccia di programmazione di alto livello (API), fornendo strumenti di gestione dei cluster che riducono il livello di conoscenza e competenza necessarie per sviluppare applicazioni parallele; Spark è uno di questi sistemi, è stato realizzato per supportare applicazioni Big Data ma ha un paradigma abbastanza flessibile da permettere la realizzazione anche di applicazioni generiche.

Per entrambe le piattaforme verranno descritte le operazioni necessarie per sviluppare una soluzione per un problema di ottimizzazione in particolare: Permutation Flow Shop Problem (PFSP). Inoltre verranno presentati e commentati alcuni risultati empirici ottenuti su un benchmark di 120 istanze di riferimento

comune per questo tipo di problema (istanze di Talliard [27]). Le applicazioni sviluppate sono state confrontate usando come indice la differenza percentuale relativa rispetto alla migliore soluzione trovata. L'overhead introdotto da ogni piattaforma verrà anche misurato e analizzato.

CHAPTER 1

Introduction

Finding the best possible solution for a certain problem has always been desired especially when dealing with problems involving (saving or earning) money. However, this is not always possible either owing to time or complexity limitations. Trying to solve problems non-optimally might rely on the exploitation of specific properties of the problem considered (so called “Heuristics”) which often cannot be reusable for other types of problems.

The possibility to apply similar optimization techniques for a variety of problem instances is a desirable property during the design of an optimization framework from a software engineering point of view, because it allows a generalization of the solution strategies and leaves to the users of the framework to develop only the specific details for the problem they want to tackle.

This is the approach followed by “Metaheuristics”, which define optimization procedures in a problem-independent way, often drawing inspiration from natural processes. Of course, even if the optimization process is general, it has to be applied on the specific problem by designing some components considering the problem that is being targeted. Such components are usually devised in a way that is highly problem-specific and might not make sense in another context. However, a “Metaheuristic” does not need to know the details about them but just provides guidance on how to employ lower-abstraction component on optimizing the task at hand.

However, a metaheuristic optimization might give a sub-optimal solution and follow an stochastic behavior which makes harder to evaluate its performance. Nonetheless, this is particularly useful when the optimization problem belongs to the \mathcal{NP} -Complete class such that the complexity of finding the solution is likely to be non-polynomial but a solution can be evaluated in polyno-

mial time.

There are several general purpose algorithms that employ metaheuristics, including techniques falling under the field of Evolutionary computation. We will now list some of them:

- Metropolis-Hastings algorithm[17]
- Ant colony optimization[3]
- Evolutionary algorithms[4]
- Evolutionary programming[4]
- Genetic algorithm[18]
- Particle swarm optimization[13]
- Simulated annealing[22]
- Tabu search[19]

Genetic algorithms is the technique selected for the benchmarks as it has been proven to be effective in the solution of several class of problems. For instance, GAs have been successfully employed to solve the challenging engineering problem of Antenna design with outstanding results [9].

Given the general purpose of this Metaheuristics, they do not constrain the execution of the optimization other than to follow some key steps. This is useful when we want a parallel distributed execution as there are few synchronization points that might be required.

However, early implementations of these metaheuristics were meant for sequential execution, considering the scarcity of parallel execution clusters. This situation changed dramatically during recent years, initially with the commercial release of multiple core processors in stock hardware and later with the development of cluster management technologies for computationally intensive algorithms using a distributed paradigm commonly associated with Apache Hadoop MapReduce [6].

Some initial coupling issues were overcome which led to the creation of the Hadoop Distributed File System (HDFS), a technology that enabled the reuse of part of the Hadoop infrastructure for the generalization of computing paradigms that did not match with the Map Reduce technique.

Such is the case of Apache SparkTM[7] which provides a general data processing environment not constrained by the Map Reduce paradigm. It is on top

of this engine that HyperSpark is introduced to provide an even higher level API tailored to the needs of Metaheuristic optimization.

There are other platforms that were in use before the rise of distributed parallel computing over stock hardware. One of them is MPI which was mainly used for High Performance Computing (HPC) and super computers in the past. On top of this platform it is also possible to run distributed computation as desired. For this purpose we select the ParadisEO[11] framework as a benchmark to HyperSpark solution. ParadisEO is specialized in evolutionary computation and uses MPI as its distributed computation engine.

This work has the following set of objectives:

- Help promoting HyperSpark development, considering it was a project started within Politecnico di Milano, by developing a website where the end-user can get more information about key features and advantages of using HyperSpark.
- Comparing how HyperSpark performs compared to a commonly used solution for problems similar to the ones HyperSpark solves. For this purpose ParadisEO was selected as explained above.
- Providing a detailed assessment over solution quality and platform overhead introduced by HyperSpark compared to the benchmark framework. This requires developing an application solving the case study problem on each platform and executing experimental tests in a controlled execution environment.

This document is organized as follows. Chapter 2 presents an overview of the capabilities of the framework developed within Politecnico di Milano to handle Metaheuristics distributed optimization. Next, Chapter 3 gives an alternative point of view from a high performance computing oriented platform using another technology stack to tackle a similar task. Chapter 4 will describe the problem instance used to test the frameworks and how it is related to the problem of Metaheuristics, including the specific genetic algorithm being used to provide a solution. Chapter 5 provides further details into the development that was required in both platforms to execute the algorithm and how complex it was from a software engineering perspective. We will explore some benchmarks tests that were executed in Chapter 6, and how do they help us understand the tradeoff for each platform. At last, Chapter 7 will summarize the main findings and how it would be possible to overcome some of the limitations and problems that were faced.

HyperSpark : A Framework for Parallel Metaheuristics over Spark

HyperSpark was introduced at Politecnico di Milano as a general framework for scalable and parallel execution of Metaheuristics.

2.1 Technologies used

Hyperspark employs several existing technologies and it is relevant to list them and detail how it differs from them. These technologies were chosen addressing current trend and according to the requirements from the problem. As such they do come with a cost that needs to be highlighted but was deemed worth and attractive to draw the corresponding benefits from them.

2.1.1 Apache Spark

From the name it is evident that HyperSpark relies on Apache Spark for cluster management and computation distribution as well fault tolerance among several useful features that are inherited from it. It is important to establish that Spark is a modern framework initially released in May 2014 [7]. It was originally developed at University of California, Berkeley's AMPLab. Spark main focus was to overcome several limitations that the Map Reduce paradigm had, such as lack of computation caching and limitations of the computation paradigm.

Nonetheless Apache Spark builds on top of Apache Hadoop stack and actually uses Hadoop Distributed File System (HDFS) as distributed file system to

have data locally when required for computation. This has helped to ease Spark adoption even inside corporate environments; where Map Reduce was seemed as both time consuming to setup and verbose to develop.

HyperSpark aims to have a much reduced scope compare to this general computation problem and for this it tries to establish a higher level API which can be used to develop general-purpose algorithms (as, in this case, metaheuristics) in an object oriented way using the Scala language.

The advantage that HyperSpark provides is handling the computational details using Spark while leaving to the user to implement only the application specific code as we will investigate further in Chapter 4.

2.1.2 Scala

Scala is an object-oriented and functional static typed programming language designed with the aim of building components and components systems [21]. Odersky et al. [21] postulate “that a component software programming language needs to be scalable in the sense that the same concepts can describe small as well as large parts”.

Part of Scala success could be attributed to the fact that it has embraced existing technologies as it can seamless interact with Java code. Actually, it executes inside the Java Virtual Machine (JVM). However, early in the design of the language the authors decided to part ways with some of the Java conventions and mechanism, therefore, it is not a strict superset of Java. Scala provides an uniform object model that enables a fully functional programming paradigm where functions are first-class values

2.1.3 IntelliJ IDEA

IntelliJ IDEA was the Integrated Development Framework (IDE) used during development of the solution to the problem instances that will be investigated. Eclipse was also a valid option, that was in fact used during the main development of HyperSpark. Both offer extended support for Scala on top of being JAVA development environments.

2.2 Building blocks

To draw clarity onto the API offered by HyperSpark it is useful to list some of the basic classes that were used while developing a solution.

- **Problem** - Base class that encapsulates all the parameter and variables corresponding to the user-defined problem instance.

- **Solution** - Base class for solution instances independent on how well they might perform. *Solution* is a descriptive solution, while *EvaluatedSolution* is a solution that has been assigned a metric that is comparable among solutions and indicates how well it performs in the problem instance. Currently, HyperSpark comes with a library of algorithms that implement only single objective optimization but this is not an intrinsic limitation of the framework.
- **Algorithm** - User defined procedure to solve the problem instance. It can contain data relevant only during execution such as stopping condition or convergence criteria. It is completely up to the user to define the specific procedure to solve the problem. It is treated as data that is effectively serialized to be sent to other nodes once the job is submitted to the cluster.
- **Seed** - It is an starting point solution provided to the *Algorithm*. It is optional to provide it initially but can be used in subsequent iterations to speed up the algorithm convergence. As such it is extensively employed to provide a best performing solution on each iteration of the execution during the benchmark.

2.3 Project website

As part of the promotion effort for the development done for HyperSpark, the project website was developed and it is part of the deliverables of this work. The site contains general information about the project, its vision and purpose, and how it can be used depending on the specific needs.

The website was built using *GitHub Pages* and it is related to the official code repository of the project. You can access the site at:

<http://deib-polimi.github.io/hyperspark/>

2.4 Repository

HyperSpark is an open source project and it is available at the popular development platform GitHub [8]. The official development for this work happened under the deib-polimi account, hyperspark repo and the ga (genetic algorithm) branch.

The complete code is available at:

<https://github.com/deib-polimi/hyperspark/tree/ga>

ParadisEO : A Software Framework for Metaheuristics

ParadisEO was introduced at *INRIA*, the French Institute for Research in Computer Science and automation, as a white-box object-oriented for flexible design of metaheuristics [11].

ParadisEO builds on top of Evolving Objects (EO), which is a template-based ANSI C++ evolutionary computation library. It was developed aiming at reusability and portability as main concerns. We need to bear in mind that the main focus of the authors was having a highly performance implementation for state of the art metaheuristics algorithms that allowed to reproduce the latest developments of the field [1]. Accordingly, when the project was started around 2003 it was implementing several well know metaheuristic algorithms including Genetic Algorithm (GA), Particle Swarm Optimization (PSO) and Ant Colony Optimization (ACO).

3.1 Technologies used

ParadisEO is a C++ white-box portable object-oriented framework. It compiles to native code for the major OSs: Windows, Unix and MacOS. Given that the development of ParadisEO started around 2003 and the latest stable release was in 2012, it is fair to say that the technology landscape has changed from the one faced by the developers of the project. Nonetheless, it is a striking contrast with HyperSpark as the latter is inspired in the latest trends in both technologies and computation paradigms while ParadisEO is firmly rooted in the idea

of achieving performance through the usage of native code compilation for the target platform.

An important aspect to mention about distributed execution in ParadisEO is that it is handled as an add-on to the framework. Actually, the library that handles parallel and distributed architectures by providing a wrapper on top of MPI (conveniently called EO-MPI) is not compiled by default with the framework as it requires the additional dependencies on MPI libraries.

3.1.1 Message Passing Interface (MPI)

The Message Passing Interface (MPI) is a standard message based protocol devised to encourage the development of portable parallel applications. It is still seen as a dominant model in the High Performance Computing (HPC) community today [25]. MPI is different but complementary to shared memory programming parallel libraries (such as *OpenMP*) as those only cover parallel but not distributed execution. However, it is common to use both interfaces for the same application depending on the availability of multiple cores per worker machine (as it is often the case nowadays).

MPI provides basic communication primitives for both point-to-point communications and collective functions such as broadcast and scatter-gather mechanisms. By default, the communication is synchronous and does not provide fault tolerance mechanisms built-in, so exceptions are meant to be handled by the application developer, which is a major difference with modern approaches to cluster management that try to hide this complexity by providing fail-safe interfaces.

MPI also requires full knowledge of the size, identity and location of the nodes that are part of the cluster as part of its initialization. Given this constraints, it performs better in controlled cluster environments where node failure is unlikely [25], this is usually not the case when running a cluster of commodity hardware (or virtual machines) but more frequent in HPC execution environments.

3.1.2 Programming language: C++

C++ is an imperative, object-oriented programming language which compiles to native code on the target platforms, providing memory manipulation facilities. It was conceived as an extension to C, to aid in program organization while achieving similar execution efficiency. It is an ISO standard, the latest of which

came out in 2014 and added several features while including additional implementations in its standard library.

C++ is highly sought after when performance is an adamant concern, considering that its compiler for different platforms and the fact that it executes native code is a hard to beat baseline. However, several improvements in alternative platforms such as the Java Virtual Machine (JVM) have ease this concern. In any case it is important to realize that the performance benefit comes at the cost of added development complexities such as compilation and linking process that are a frequent source of issues and might impact significantly the development time.

3.2 Framework Architecture

ParadisEO support a broad range of features for developing metaheuristics algorithms. The main components of ParadisEO can be seen in Figure 3.1:

- **Evolving Objects (EO)** supports population-based metaheuristics like Genetic Algorithm (GA), while **Moving Objects (MO)** targets solution-based metaheuristics such as Particle Swarm Optimization (PSO).
- **Multi-Objective EO (MOEO)** is used for multiobjective optimization, implementing several Pareto-based fitness assignment strategies.
- **MPI** and others are third party libraries used for parallel and distributed execution. ParadisEO provides wrappers on top of them targeted for distributed metaheuristics common communication and coordination scenarios.

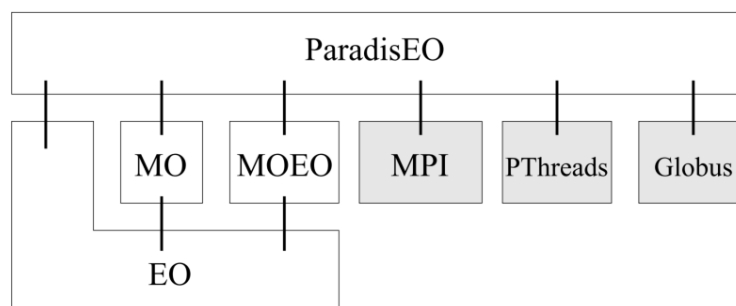


Figure 3.1 – ParadisEO architecture. Taken from [15]

3.3 Compilation

Before actually being able to develop and execute an application within ParadiseO, compiling the framework was required as the binaries available [11] were not updated and available for the targeted platforms.

This turned out to be surprisingly not trivial as several dependencies have changed and given that the codebase has been in maintenance mode, some work was required to fix issues in the build scripts and the framework's source code.

This is also a time consuming process specially when done from the scratch for development purposes. However, no compilation is required when the binaries are deployed to the cluster as the ParadiseO's libraries are linked statically (i.e. included as part of the executable).

3.4 Executable distribution

There is a major difference between ParadiseO and HyperSpark regarding code distribution. While this is handled in HyperSpark as part of the job submission, in ParadiseO is done offline before execution.

It is up to the user how to handle this distribution. It could be automated to some degree via scripts (for this purpose during experiments we used Ansible). However, it is still a significant burden because it also leaves up to the user to configure properly the ParadiseO libraries and make sure all the executables are in the same file path so they can be reached when execution starts.

3.5 Multiobjective optimization

ParadiseO supports multi-objective optimization [15], which opens up the possibility to tackle a completely different set of problems compared to the ones that are covered by the current implementation of HyperSpark. For instance, the fitness assignment strategies might be Pareto-based if the goal is to obtain the Pareto efficient frontier.

The base implementation considered for solving the problem introduced later on in the case study (Chapter 4) was performing multi-objective optimization, but this approach was incomparable with the current development status of HyperSpark. For a fair comparison, the original solution was ported to a single objective optimization as explained in Chapter 5.

3.6 Repository

ParadisEO is an open source project available at SourceForge. However, a more recent fork was available at GitHub [8] and was the one used as basis for the required extensions of the framework to solve the case study.

The initial fork from nojhan/paradiseo is available at:

`https://github.com/camilomartinez/paradiseo`

Case Study: Permutation Flow Shop Problem

In this section I will present an introduction to the problem instance considered for the benchmark, how it relates to the type of problems addressed by the platforms and which algorithm was used for solving it during the tests.

4.1 Problem definition

The Permutation Flow Shop Problem (PFSP) is a well know problem in scheduling theory that seeks to establish the proper schedule for jobs in a workshop. Formally defined under the following conditions:

- There are n machines and m jobs.
- Each job contains n operations that must be executed in the same order, starting from machine 0 till machine $n - 1$
- For each operation an execution time is specified
- Only one operation can be executed on a machine simultaneously. Conversely, execution of an operation in a machine for one job cannot be started before the machine finishes executing an operation for a different job or the previous machine finishes executing the previous operation for the same job
- The problem is to determine the optimal arrangement such that a performance metric on the scheduling is minimized

- For our case the performance arrangement that will be used is makespan, which measures the latest completion time among the job in the scheduling problem.

Mathematically we index each machine $i = 0, 1 \dots m - 1$, and each job $j = 0, 1 \dots n - 1$. A valid solution s is represented as a vector of size n which indicates the order of execution of the jobs. The processing times is a matrix P of size $m \times n$ where position $P_{i,j}$ indicates how much time it takes to finish the corresponding part of job j in machine i .

4.2 Problem objective: Makespan

We will use the makespan as the objective to minimize. This is a commonly used objective [28] and it is the one available for the benchmark instances that will be used during the experiments [27].

It is defined as the latest completion time of the last job executed. Formally, given a solution s we define the completion times matrix C as:

$$C_{0,s_0} = P_{0,s_0} \quad (4.1)$$

$$C_{i,s_0} = C_{i-1,s_0} + P_{i,s_0} \quad \forall i = 1 \dots m - 1 \quad (4.2)$$

$$C_{0,s_j} = C_{0,s_{j-1}} + P_{0,s_j} \quad \forall j = 1 \dots n - 1 \quad (4.3)$$

$$C_{i,s_j} = \max(C_{i-1,s_j}, C_{i,s_{j-1}}) + P_{i,s_j} \quad \forall i = 1 \dots m - 1, j = 1 \dots n - 1 \quad (4.4)$$

Which means that job j finishes its execution in machine i at time $C_{i,j}$. Once this completion matrix is computed, the makespan is easily defined as the maximum of the matrix:

$$C_{max} = \max(C) = C_{m-1,s_{j-1}} \quad (4.5)$$

4.3 Solution evaluation: Relative Percentage Deviation (RPD)

Comparing two different solutions using the makespan is only meaningful if both belong to the same problem instance. Otherwise, the comparison is meaningless because of variations in the processing times. We need to rely on a different evaluation metric for comparison across different problem instances.

Ideally this metric should give us a hint how good is the solution relative to the intrinsic properties of the instance it belongs to and not related to other possible solutions that could be found. This leads to defining performance relative to the optimum solution. Where no global optimum has been found or verified we rely on the best solution ever found[27].

Formally the Relative Percentage Deviation (RPD) is defined as:

$$rpd(s) = \frac{C_{max}(s) - C_{max}(s_{best})}{C_{max}(s_{best})} \times 100 \quad (4.6)$$

where both s and s_{best} are specific for a given problem instance [28].

4.4 Platform overhead: Relative Execution Overhead (REO)

To have an idea of the overhead introduced by the cluster management provided by each platform we mean to measure how much of time is spent in activities other than actually executing the algorithm. For this purpose we introduce the Relative Execution Overhead (REO) as a metric relative to the total execution time (which depends on the size of the problem, see Section 6.1)

The REO of a run is the time spent in non-computing activities as a share of the wallclock time of the run. Formally defined in Equation 4.7 where t_{algo} is an input (see Section 6.1). $t_{overhead}$ is considered to be the part of the execution time exceeding the given time.

$$reo = \frac{t_{overhead}}{t_{wallclock}} \quad (4.7)$$

$$t_{wallclock} = t_{overhead} + t_{algo} \quad (4.8)$$

4.5 Problem example

It is useful to consider an example to illustrate the previous definitions. Let us consider a toy problem with number of jobs $n = 4$, number of machines $m = 5$ and processing times matrix P given in Table 4.1.

If we analyze an arbitrary solution $s = (3, 2, 4, 1)$ then we can compute the completion times matrix C as seen in Table 4.2. From C it is easily verifiable that the makespan for this solution is $C_{max}(s) = 36$

It is possible to find out that the best solution for this example $s_{best} = (3, 2, 0, 1)$ has makespan $C_{max}(s_{best}) = 28$.

TABLE 4.1
PROCESSING TIMES EXAMPLE

P	J_0	J_1	J_2	J_3
M_0	7	10	3	2
M_1	6	5	2	3
M_2	8	1	9	2

TABLE 4.2
COMPLETION TIMES FOR $s = (2, 1, 3, 0)$

C	J_0	J_1	J_2	J_3
M_0	22	13	3	15
M_1	28	18	5	21
M_2	36	19	14	23

If we now apply Equation 4.6 to the sample solution $s = (2, 1, 3, 0)$ that we are considering it give us a RPD of $rp d(s) = 28.57\%$

4.6 Complexity and metaheuristics

What makes the PFSP a popular test case for metaheuristics is the fact that for more than two machines (i.e. $m > 2$) is NP-Complete. There is no algorithm that can guarantee to efficiently find an optimal solution as, in fact, it becomes more and more expensive to find one for large values of m .

For this reason we need to rely on using metaheuristics on practice, which might lead to sub-optimal solutions in bounded time. Some of this metaheuristics rely on some sort of local search such that local minima might be reached (e.g. Hill Climbing). Another strategy, used by evolutionary computation is to explore the solution space by introducing some stochastic decision processes during the execution of the algorithms such that it does not get trapped into local optima but it is also able to explore other regions of the solution space. Properly tuned, these algorithms are also able to optimize the best solutions they found so that these are close the the global optima. Specifically, these algorithms use representations of the solutions that are modified following a criteria that should move them closer to the optima in the solution space.

4.6. Complexity and metaheuristics

Metaheuristics that exhibit such behavior are Iterated Greedy [24], Ant Colony Optimization [3], Simulated Annealing [22], Tabu Search [26], Particle Swarm Optimization [13] and Cuckoo Search [29] among others

Implementation

Once we have a clear understanding about what is the case study that is being targeted, we can now speak about the actual algorithm that is executed during the benchmarks. In particular, we will also see how this algorithm has been implemented on each platform and how we have used the tools provided by them. The idea behind this implementation is to perform a fair comparison among the platforms by executing an algorithm as similar as possible on each one of them.

5.1 The test algorithm

In order to compare the two frameworks the implementation of a specific algorithm is needed. To this end we selected a simple Genetic Algorithm (GA) and we implemented it on both platforms. Similar GAs are commonly used as a baseline for solving PFSP and specifically for evaluating distributed optimization algorithms in the literature [23][28][10][5]. The rest of this section is divided into two part, in the first we present in general the elements of the algorithm, whereas in the second, we explain the specific way it has been applied to our case study.

5.1.1 Introduction

A Genetic Algorithm (GA) represents elements in the solution space as *candidate solutions* and tries to “evolve” them towards better solutions. On each step of this “evolution” a set of candidate solutions is kept in a set often referred to as *population*.

Each candidate solution has certain attributes and invariants (*genes*) that are encoded in a genetic representation called an *individual*. During the evolution process these individuals are mutated or altered by changing their encoding using *genetic operators*. The most common genetic operators are:

- **Mutation:** Takes one individual as input and gives as output a slight modification upon it.
- **Crossover:** Takes two individuals as “parents” and combines them to generate two new individuals, called “offspring”.

It is important to notice that the actual implementation of these operators depends on the encoding that is used as it might not be meaningful otherwise. This means that is it problem specific and the quality of the solution returned by the algorithm greatly depends on the operators applied.

Whether to apply this genetic operators is decided stochastically to be able to explore the solution space. This means that there is some variation on the operator behavior, on deciding over which individuals to apply it, or whether to apply it at all. This is usually configured using parameters such as the rate (probability) on how frequently that operator is used.

In any case, we would like to apply the operators over “good” individuals on average, as this will lead in the long run to exploring mostly good solutions. We would also like to discard not so good solutions so that we keep exploring only promising ones. In order to compare individuals we need what is called a *fitness* function that given an individual returns a value that represents how well the considered solution fits in the “environment”. In many cases the fitness coincides with the objective function the algorithm aims at optimizing. As mentioned before, this fitness is used to select individuals from the population (i.e. *selection* process) to apply on them the genetic operators.

A GA can be briefly described through the following steps:

1. **Initialization:** Initially we need to create as many individuals as the desired population size, usually they are initialized at random but in a meaningful way for the encoding being used.
2. **Selection:** On each iteration some individuals are stochastically selected to *breed* them by applying the genetic operators.

3. **Offspring:** The most common way to generate the offspring is by stochastically applying the crossover operator on the individuals selected in the previous step, and then stochastically applying the mutation operator on them.
4. **Replacement:** Once the offspring has been generated we need to decide which individuals to discard if we want to keep the population size constant (which is often the case). For that purpose a replacement policy must be defined so that the algorithm can perform a selection of individuals to drop by the end of each iteration. This selection usually follows a mechanism inverse to the one used to select the parents in second step (e.g. We might drop the worst solutions).
5. **Termination:** After each iteration we need to evaluate the termination criterion, also called *stopping condition*, to decide whether to keep evolving the population (jumping back to step 2) or report the best individual as the final outcome of the algorithm.

In the next section we will discuss the specific definition for the PFSP optimization of the general concepts that we have seen before.

5.1.2 A simple GA for the PFSP

In this section we show how each one of the concepts presented in the previous section can be applied to the particular case of the PFSP. This algorithm was introduced by Liefoghe et al. [16] for a multi-objective PFSP optimization problem and was adapted in this work to a single objective PFSP optimization problem.

- **Genetic representation or Individual:** We represent a candidate solution as an array of integers with size equal to the number of jobs n . The elements of this array is the set of integers $J = \{x \mid 0 \leq x \leq n\}$. This representation is called a *permutation encoding*.
The order of the elements in this array indicates the sequence of execution of the jobs. This means job j_0 at position 0 is executed first, then job j_1 at positions 1 is executed, and so on until job j_{n-1} , which is executed last.
- **Initialization:** We initialize the population by generating random individuals, that is, given an array of ordered elements in the range 0 to $n - 1$, we initialize this array by shuffling its elements.
- **Selection:** The selection of the “parents” is implemented by means of the *deterministic tournament* with size 2. This means that each parent is

selected in the following way: two individuals are picked at random from the population and are compared to each other in terms of the fitness. The individual with the best fitness value is selected as parent. The size of the tournament indicates how many individuals are picked at random before selecting the best one. In our case we selected two, so only one comparison had to be made.

- **Crossover:** We set the crossover rate to 0.5, meaning around half of the time we apply the crossover operator on the selected parents. The other half of the time they were directly used as offspring.

We used a *Two point* crossover operator [16] [12] which basically samples two random points within the range of the array size with at least two positions of separation. It takes the elements in position 0 till point 1 from those positions in parent 1, and the elements from point 2 till the end from parent 1 as well. For the remaining elements to fill the positions between point 1 and point 2 it takes them in the order they appear in parent 2.

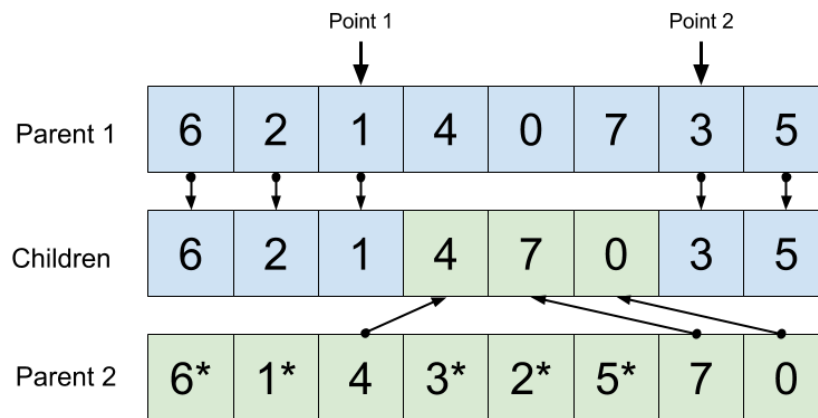


Figure 5.1 – Two point crossover operator example

In Figure 5.1 we can see an example of an application of the two point crossover operator on parents $p_1=(6,2,1,4,0,7,3,5)$ and $p_2=(6,1,4,3,2,5,7,0)$ considering a case when we randomly sample points 2 and 6.

The positions 0 to 2 and 6 to 7 are taken directly from parent 1. To fill positions 3 to 5 we scan parent 2 and take the remaining elements as they appear. Elements that were already taken are marked with an asterisk. This is why we take 4, 7, 0 in that order for positions 3 to 5.

- **Mutation:** We set the mutation probability to 0.7 as we actively want to explore the solution space and avoid getting trapped into local minima. Once we decide to apply the mutation operator we decided randomly with equal probability between the two following operators.
 - **Shift** mutation which selects two different random points. If point 1 is lower than point 2 we apply a right circular shift to the elements between point 1 and point 2. If point 1 is greater than point 2 we apply a left circular shift in the same way.
 - **Exchange** mutation which selects two different random points and swaps the elements in those positions.
- **Replacement policy:** We used an inverse deterministic tournament of size 2 to decide which elements to discard. It is similar to the deterministic tournament used for parent selection with the difference that we select the worst element to be discard from the random ones that were picked. We discarded two elements (the size of the offspring set) in order to keep the population size constant.

We also enforced *weak elitism* which guarantees that we keep the best solution found so far at the end of the replacement. It does this by replacing it with the worst individual in the population if the best solution in the population is worse than the best one found so far.
- **Stopping condition:** We used elapsed time as the termination criteria. Specifically, we set the amount of time the algorithm can run before reporting the best solution found.

5.2 Distributed execution

Up to this point we have not considered the parallel distributed execution of the algorithm. We will now explain how it was implemented to allow a distributed and collaborative execution.

In order to coordinate the distributed execution we rely in a master-workers paradigm such that the master process is only executing cluster management and coordination tasks but not executing the algorithm itself. The workers follow the execution instructions provided by the master and report back with the results.

A worker is nothing more than a process dedicated to the execution of the algorithm on a machine different from the one where the master is located. The

worker execution is started once the master indicates so. After finishing each iteration it becomes idle while waiting for the next instruction. As we will see in Section 6.2 for our execution environment each worker runs on a separate CPU core.

The core idea of the parallel execution of the GA is that we will execute the algorithm several times in about the same time it would take to execute it once hence. Ideally we would also share good intermediate results so that all the workers can also improve their solutions while they algorithm still running.

Considering that the GA follows an stochastic execution, we could potentially reach better solutions by running the algorithm several times rather than only once, as different executions might lead to different results. This is accomplished by running concurrent executions of the same algorithm using different random seeds.

We leverage the cooperative distributed execution by dividing the total execution time into several iterations such that each iteration is completed as an independent execution path except for initialization of the population. In fact, at the end of each iteration all the workers report the best solution found to the master which keeps record of them and at the start of the next iteration provides the best global solution found so far to each worker to include it by default in the population for the next iteration of the algorithm. This way, the independent executions of the algorithm actually share some information but can still try to explore the solution space in different ways so that they are not limited by the best performing individuals found so far.

During the explanation for the specific implementation for each platform we will see more details on how this distributed execution has been actually developed.

5.3 Benchmark problems

It is better if we describe briefly the input that were used as benchmark, which come from Talliard problem instances [27]. Every instance of the benchmark has the following schema:

- First line specifies the number of jobs n followed by the number of machines of the problem m separated by a space.
- Each one of the following lines corresponds to the processing times for each job for the corresponding machine starting with job 0 until job $n - 1$ separated by spaces.

Listing 5.2: PfsProblem class declaration

```
package pfsp.problem

import it.polimi.hyperh.problem.Problem

class PfsProblem(
  val numOfWorks: Int,
  val numOfWorks: Int,
  val jobTimesMatrix: Array[Array[Int]]
) extends Problem { ... }
```

Listing 5.3: PfsProblemParser for benchmark input file

```
package pfsp.util

import scala.util.parsing.combinator.RegexParsers
import pfsp.problem.PfsProblem

object PfsProblemParser extends RegexParsers {
  def number: Parser[Int] = "\\d+".r ^^ { _.toInt }
  def params: Parser[(Int,Int)] = number ~ number <~ "x" ^^ { case x ~ y =>
    (x,y) }
  def row: Parser[Array[Int]] = number.+ <~ "x" ^^ { _.toArray }
  def matrix: Parser[Array[Array[Int]]] = row.+ ^^ { _.toArray }
  def problem: Parser[PfsProblem] = params ~ matrix ^^ {
    case p ~ m => new PfsProblem(p._1,p._2,m)
  }
  def apply(input: String): Option[PfsProblem] = parseAll(problem, input)
  match {
    case Success(result, _) => Some(result)
    case NoSuccess(_, _) => None
  }
}
```

- **PfsProblemParser:** In order to build the `PfsProblem` corresponding to the benchmark instance we are running, we implemented a specific parser for the Taillard file format by leveraging a Scala class called `RegexParsers` that allows the programmer to easily implement a context free grammar based parser as seen in Listing 5.3
- **PfsEvaluatedSolution:** Using the `PfsProblem` we can calculate the fitness of a given `PfsSolution` and encapsulate this value together with the related solution within the `PfsEvaluatedSolution`, which extends the

Listing 5.4: DistributedDatum for encapsulating algorithm and data

```

package it.polimi.hyperh.spark

class DistributedDatum(ind: Int, alg: Algorithm, seedOption:
  Option[Solution], stopCond: StoppingCondition) extends Serializable {...}

```

EvaluatedSolution from HyperSpark basecode.

- **GAAAlgorithm:** Once the building blocks are established we can actually implement the algorithm that was discussed in Subsection 5.1.2, including the specific initialization, selection, replacement procedures and the desired genetic operators. The code for this algorithm is extensive and it is not shown here for brevity, but it is available in the repository within the package `pfsp.algorithms`.

5.4.2 Distributed execution

As explained in Chapter 2, HyperSpark relies on Apache Spark to handle the distributed execution of algorithms. The way it does it is by exploiting Spark's Resilient Distributed Dataset (RDD), which is an immutable collection of objects that can be partitioned to be distributed across several machines.

In this case the dataset is composed by a collection of `DistributedDatum` which is essentially a class containing the algorithm to run along with all its necessary data. In our case this is a `GAAAlgorithm` instances, along with the seed solution and a stopping condition. In Listing 5.4 it is possible to observe that the algorithm to execute is passed as an object to `DistributedDatum` and as such is treated as “data” to be distributed as well.

5.4.3 Framework configuration

It is important to mention the specific HyperSpark configuration that has been used for experimentation purposes (summarized in Listing 5.5). We receive from the command line the instance number for which to run the experiment and the number of workers to execute, called here `parallelism`. We build the problem instance passing the corresponding input file to the parser in line 8.

In line 9, we specify an algorithm builder function with the proper parameters for the `GAAAlgorithm` in line 9, the first three are: Population size (20), Mutation rate (0.5) and Crossover rate (0.7).

Listing 5.5: Algorithm execution configuration

```

1 package it.polimi.hyperh.experiments
2
3 class Experiment1(instance: Int, parallelism: Int) extends
  Experiment(instance, parallelism) {
4
5   override def run() {
6     val runs = 1
7     val problem =
8       PfsProblem.fromResources(filename("inst_ta", instance, ".txt"))
9     val makeAlgo = () => new GAAAlgorithm(20, 0.5, 0.7, 1, 1, None)
10    val numOfAlgorithms = parallelism
11    val totalTime = problem.getExecutionTime()
12    val numOfIterations = 10
13    val iterTimeLimit = totalTime / numOfIterations
14    val stopCond = new TimeExpired(iterTimeLimit)
15    val strategy = new SameSeeds()
16
17    val conf = new FrameworkConf()
18      .setProblem(problem)
19      .setNAlgorithms(makeAlgo, numOfAlgorithms)
20      .setNDefaultInitialSeeds(numOfAlgorithms)
21      .setNumberOfIterations(numOfIterations)
22      .setStoppingCondition(stopCond)
23      .setSeedingStrategy(strategy)
24    val resultStr = testInstance(instance, runs, conf, true)
25  }
26 }

```

Lines 11 to 14 set the proper stopping condition by allocating equal amount of time to each iteration depending on the total execution time, which in turns depend on the problem instance size as will be explained in Section 6.1. The total execution time is computed using the formula described in Equation 6.1. Line 15 specify that all the workers should receive the same best solution as seed (i.e. starting solution) for the next population. Finally, these options are used to build the FrameworkConf instance that is used to indicate how to test the instance in line 24.

5.4.4 Improvements

In order to guarantee a fair comparison with ParadisEO later on, a major revision of the HyperSpark internal implementation details was conducted, which found out an important bug on how the workers were being initialized that was

rendering pointless the distributed parallel execution as it was causing all the workers to exhibit the same stochastic behavior, as all of them were using the same seed for random generation of numbers.

This issue was fixed by making sure that during the Framework configuration, the master sets different random seeds for the corresponding random number generator on the workers. A mechanism to set the random seed in the master is provided in order to secure some reproducibility of the results given a random number generator seed.

The specific mechanisms being used are available in the FrameworkConf class method `setNAlgorithms` in the package `it.polimi.hyperh.spark`

5.4.5 Repository

The complete development done for the tests is available at GitHub [8], under the `deib-polimi` account, `hyperspark` repo and the `ga` (genetic algorithm) branch.

<https://github.com/deib-polimi/hyperspark/tree/ga>

5.5 Paradiseo

As discussed in Chapter 3, ParadisEO provides out of the box implementation for several metaheuristics algorithms including Genetic Algorithm (GA).

However, problem specific classes are left to be developed by the user. Nonetheless, examples solving PFSP were available and were used as basis to develop the complete solution for our case study. As a reminder, ParadisEO is implemented in C++ and this is the language used for the codes samples in this section.

5.5.1 PFSP example

An example PFSP multi-objective optimization was available in ParadisEO's website, implementing the representation dependent classes to solve the problem [14]. It was developed precisely as a way to prove ParadisEO's applicability to solve scheduling problems such as PFSP [16].

However, the example was optimizing a multiobjective function with another part of the library that we did not intend to use in order to be able to compare this implementation with HyperSpark, considering that the latter does not feature any multiobjective optimization yet.

For this reason a considerable amount of modifications and refactoring was required to adapt this example to our case study so that it targeted single objective optimization. Additionally, we also needed to adapt the implementation for distributed execution which was not supported in the base implementation.

5.5.2 Highlighted classes

We will now review some of the ParadisEO specific classes that needed to be developed to solve our case study as they were adapted from the base example.

- **FlowShop**: It is the individual or genetic representation using permutation encoding. For distributed execution purpose we need to be able to exchange individuals by means of messages (e.g. to indicate which has been the best solution found so far). Consequently, we needed to implement the `eoserial::Persistent` pack and unpack methods in order to indicate how to serialize our individual to JSON and how to recover it from JSON correspondingly.
- **FlowShopBenchmarkParser**: The input file parser was also modified to parse single objective benchmark files. It is striking to contrast the heavy buffer manipulation and memory allocation required for this task in C++ (Listing 5.6) with the simplicity of the grammar based higher level parser in Scala (Listing 5.3)
- **FlowShopEval**: ParadisEO requires a fitness evaluation function that given an individual returns its fitness. Its parameters are set after parsing the file being M the number of machines, N the number of jobs and p the processing times matrix for which $p[i][j]$ is the processing time of job j on machine i . In Listing 5.7 we can see how the makespan and completion times matrix are computed exactly as it is formulated in Section 4.2 Equation 4.1

5.5.3 Algorithm configuration

ParadisEO offers runtime configuration by leveraging the `eoParser` class that allows to retrieve information from a parameters file. This is information ranges from the benchmark instance to run, to the crossover and mutation rate to use. It configures several aspects of the execution of the algorithm, as such it allows to modify the execution without recompiling, which might actually not be possible in the targeted system.

Listing 5.6: Benchmark input file parser

```

using namespace std;

void FlowShopBenchmarkParser::init(const string _benchmarkFileName)
{
    string buffer;
    string::size_type start, end;
    ifstream inputFile(_benchmarkFileName.data(), ios::in);
    // opening of the benchmark file
    if (!inputFile)
        throw runtime_error("ERROR: Unable to open the benchmark file");
    getline(inputFile, buffer, '\n');
    start = buffer.find_first_not_of(" ");
    end = buffer.find_first_of(" ", start);
    // number of jobs (N)
    N = atoi(buffer.substr(start, end-start).data());
    start = buffer.find_first_not_of(" ", end);
    end = buffer.find_first_of(" ", start);
    // number of machines M
    M = atoi(buffer.substr(start, end-start).data());
    // processing times
    p = vector< vector<unsigned int> >(M, vector<unsigned int>(N));
    // for each machine...
    for (unsigned int i=0 ; i<M ; i++) {
        // processing times of the machine i for each job
        getline(inputFile, buffer, '\n');
        start = buffer.find_first_not_of(" ");
        for (unsigned int j=0 ; j<N ; j++) {
            end = buffer.find_first_of(" ", start);
            p[i][j] = atoi(buffer.substr(start, end-start).data());
            start = buffer.find_first_not_of(" ", end);
        }
    }
    inputFile.close();
}

```

Listing 5.7: Makespan and completion times matrix computation

```
#include <vector>
#include <FlowShop.h>

using namespace std;

class FlowShopEval : public eoEvalFunc<FlowShop>
{
...

    double makespan(const FlowShop & _flowshop)
    {
        vector< vector<unsigned int> > C = completionTime(_flowshop);
        return C[M-1][_flowshop[N-1]];
    }

    vector< vector<unsigned int> > completionTime(
        const FlowShop & _flowshop) {
        vector< vector<unsigned int> > C(M, vector<unsigned int>(N));
        C[0][_flowshop[0]] = p[0][_flowshop[0]];
        for (unsigned int j=1; j<N; j++)
            C[0][_flowshop[j]] = C[0][_flowshop[j-1]] + p[0][_flowshop[j]];
        for (unsigned int i=1; i<M; i++)
            C[i][_flowshop[0]] = C[i-1][_flowshop[0]] + p[i][_flowshop[0]];
        for (unsigned int i=1; i<M; i++)
            for (unsigned int j=1; j<N; j++)
                C[i][_flowshop[j]] = max(
                    C[i][_flowshop[j-1]],
                    C[i-1][_flowshop[j]] + p[i][_flowshop[j]]);
        return C;
    }
}
```

The complete configuration parameters file used for the ParadisEO experiments is available at Listing A.1 where you can see the variety of parameters and possible values that the framework supports

5.5.4 Framework extension

Several parts of the framework needed to be extended in order to support the desired implementation. For once, how to handle exchange of individuals between master and workers is left up to the user by the framework so we had to implement several extensions to cope with our use case of sending the best global solution found so far at the beginning of the iteration for each worker

and keeping a registry of the best solutions found in the master once the workers report their progress at the end of the iteration.

Another extension point that was implemented was to support time stopping conditions in the range of the milliseconds because the timer that was already implemented has second resolution and it was not satisfactory for our benchmarks considering that the small instances might execute an iteration for 300 milliseconds. This was achieved by using the chrono time library available since C++11.

We also needed to modify the stopping condition tracking by allowing to reset it once we restart the next iteration. This stopping condition work for a single execution by default in the framework.

5.6 Repository

The complete code developed for tests in ParadisEO is available at GitHub [8] under my personal account (camilomartinez), paradiseo repo and the flowshopEo branch.

<https://github.com/camilomartinez/paradiseo/tree/flowShopEo>

Experimental Results

Several tests were conducted to evaluate the relative performance of each of the platforms, both in terms of quality of the solution and cluster coordination overhead given the distributed nature of their execution.

6.1 Experiments settings

As explained in Chapter 4 we will look into solving the Permutation Flow Shop Problem (PFSP). For this we selected the instances of the problem presented in [27]. In the benchmark there are 120 instances providing different processing times, number of jobs (ranging from 20 to 500) and number of machines (from 5 to 20). For each number of jobs/number of machines combination there are 10 instances with different processing times matrices. For each one of the 120 instances the makespan for the optimal solution or the best solution known is available at [27]. This information is an aggregation of several empirical results previously obtained by several authors [24][19][20].

The genetic algorithms (one for each considered platform), implemented as explained in Chapter 5, have been executed 5 times for each one of the instances on both HyperSpark and ParadisEO. The total execution time (in milliseconds) was decided using the following formula used, among others, in Vallada et al. [28]:

$$t_T = n \cdot (m \cdot 2) \cdot 60ms \quad (6.1)$$

On each run $n_{iter} = 10$ equal-timed iterations have been executed, each one lasting a tenth of the total execution time (t_{iter}) as in Equation (6.2). After each iteration, all the workers report back to the master the best solution found and

the master distributes the best global solution to all the workers at the start of the next iteration. The comparison to establish the best solution was done considering that the optimization objective was to minimize the makespan, as discussed in Section 4.2.

$$t_{iter} = \frac{t_T}{n_{iter}} \quad (6.2)$$

The complete metadata about the test instances is available in Appendix A Table A.1

6.2 Execution environment

The experiments for both platforms have been run in the same environment, using the same virtual machines for fair comparison. The machines are hosted on *PoliCloud*, an Infrastructure as a Service (IaaS) cloud computing platform available for research at Politecnico di Milano [2].

For our experiments 6 machines have been set up in the cluster, each one of type *m1.xlarge* with 8-core CPU (Intel Core i7 9xx), 16GB RAM and 500 GB of disk capacity. However one machine was used exclusively as master without executing any workers locally to avoid unfair local data transfer which might speed up the computations for workers co-located with the master compared to workers located elsewhere, just out of avoiding incurring in network transfer delay.

On each run 20 workers are executing in parallel, equally distributed across the remaining 5 nodes so that each executes 4 workers at a time. Theoretically, this is equivalent to executing 20 times the algorithm with cooperation on each iteration, in the same time that 1 serial execution would take.

6.3 Experiment execution

We comment here briefly the precise commands and scripts that have been used to execute the experiments on each platform.

6.3.1 HyperSpark

In Listing 6.1 we can see the exact command that has been used to execute the experiments in HyperSpark. It is worth mentioning that the job is being submitted to the master (line 6) so the code does not need to be distributed previously as is the case for ParadisEO.

Listing 6.1: Command for HyperSpark experiment execution (bash)

```

1 cores=20;
2 for instance in $(seq 1 120); do
3   for run in $(seq 1 5); do
4     /opt/spark/bin/spark-submit
5     --conf spark.root.logger=INFO,console
6     --master spark://master:7077
7     --conf spark.cores.max=$cores
8     --conf spark.executor.cores=1
9     --conf spark.executor.memory=2G
10    --class it.polimi.hyperh.experiments.Experiment1
11    hyperh-0.0.1-SNAPSHOT.jar $instance $cores
12    2>&1
13  done
14 done

```

Listing 6.2: Command for ParadisEO experiment execution (bash)

```

1 { time
2   mpirun
3   --hostfile hosts -np 21
4   --bynode --nooversubscribe
5   FlowShopEO @FlowShopEO.param >> log_fitness.txt ;
6 } 2>> log_time.txt

```

The number of cores is set to 20 while only 1 core is used for the master, independent from the worker ones (lines 7 and 8). We are executing Experiment 1 class from the provided jar package and pass as command line arguments the instance number to execute and the fixed number of cores (lines 10 and 11). The execution overhead time was measured inside the application.

As this was a long running job (it executed for about 11 and a half hours) we collected logs for both standard output and error (line 12), while running the job with inside a screen session to avoid the hangup signal on session log out.

6.3.2 ParadisEO

In Listing 6.2 we show the command used for executing experiments in ParadisEO. First of all the command line utility `time` (line 1) was used to obtain

the wallclock execution time (defined as the elapsed time from process start till its end) to measure the execution overhead time.

`mpi run` (line 2) is used to submit a job to the MPI cluster. Nonetheless the application binary (`FlowShopEO`) needed to be distributed on each node offline before submitting the job and its execution is relative to the path where it is being submitted which introduces burden for the developer for taking care of those details instead of handling the application distribution as Spark does.

Using command line options we indicate that we want to run 21 MPI slots (line 3), 1 master and 20 workers. Line 4 indicates that we want the workers to be equally distributed across nodes so that each node has the same workload, and in any case warn if we are trying to execute more slots than the number of physical cores available on each machine as this will significantly degrade performance.

The hostfile that follows explicitly lists the slots available on each node and forces the exclusive execution of processor 0 (master) on a separate node, while otherwise distributing equally the workers load on the computing nodes.

Listing 6.3: Hostsfile for ParadisEO mpi execution

```
master          slots=1 max_slots=1
computenode1    slots=8 max_slots=8
computenode2    slots=8 max_slots=8
computenode3    slots=8 max_slots=8
computenode4    slots=8 max_slots=8
computenode5    slots=8 max_slots=8
```

The execution for ParadisEO lasted for about 9 and a half hours and it was also a long running job inside a screen session.

6.4 Experiment results

We will now discuss the results from the experiments conducted both in terms of quality and overhead introduced by the cluster coordination.

6.4.1 Solution quality

The comparison of the solutions obtained by each platform is done using the Relative Percentage Deviation (RPD) as explained in Section 4.3. For both platforms this required logging best solution found (minimum makespan) at the end of the 10 iterations for each instances on every run. There are in total 1200 data points. The complete RPD results are available in the Appendix A: Table A.2 for HyperSpark and Table A.3 for ParadisEO

The overall average RPD for ParadisEO was 0.94%, which is lower than the one for HyperSpark that was 1.27%. It could be explained partly because ParadisEO keeps the population from the previous iteration when starting the new one while HyperSpark uses only the best solution but initialize the other elements at random. While the latter allows for more exploration, in the cases where a good enough solution has been found, it is not exploited.

Some highlights of the results are the following:

- **Worst instance:** For both platforms the highest RPD was for instance 111 which has 500 jobs and 20 machines, 3.8% for ParadisEO and 4.31% for HyperSpark.
- **Optimum found:** ParadisEO was able to find the optimum solution for 24% of the runs (144) while HyperSpark found it 16.17% of the time (97 runs).
- **Statistical test for the difference:** Running a Wilcoxon signed rank test for the differences between ParadisEO and HyperSpark (using as paired samples the average RPD for each problem instance, 120 pairs in total) lead us to conclude that the observed difference is significant with p-value $p < 0.0001$. This is understandable considering that only for instance 67 the average RPD for ParadisEO is worse than the one for HyperSpark.

6.4.2 Execution time overhead

In order to have an idea about the overhead introduced by using each platform we will look into the Relative Execution Overhead (REO) as explained in Section 4.4. The idea is to compare the overall execution wallclock time with the one given as input for each run considering the size of the instance using the Equation 6.1. Intuitively this makes sense as we want to consider the cost that is incurred to handle the distributed computation that should be reflected in non-computing activities taking time apart from the actual algorithm execution.

For both platforms the wallclock time from submitting the job to the cluster to getting the final solution has been measured for each instance on every run, so we have the same 1200 datapoints. The complete REO results are available in the Appendix A: Table A.4 for HyperSpark and Table A.5 for ParadisEO.

The overall average REO for ParadisEO was 10.73% much lower than the one for HyperSpark which was 38.09%. The standard deviation for this measure is also lower in ParadisEO 10.01% against 20.56% in HyperSpark which might be due to the greater support for cluster management that it provides as we will point out shortly.

However, it is important to say that this comparison might not be fair among the two platforms if we consider that they provide different benefits from using them. For once, ParadisEO does not provide any guarantee regarding fault tolerance, although the failure rate observed in the execution environment was rather low: 0.33%, just 2 out of the 600 runs failed.

HyperSpark has also a much more general scope as a metaheuristics optimization framework which might lead to additional costs by handling problems generically enough. Spark is also a modern general computing platform taking care of aspects such as communication, data distribution and execution coordination that are up to the developer in MPI. This reduces the execution burden at the cost of a significant increase in the development time handling the platform primitive or wrapper in the case of ParadisEO. Another key difference to consider here is that HyperSpark is taking care of the code distribution as well, while ParadisEO relies on the user distributing the code before starting the execution.

Important points to bear in mind regarding the execution overhead are the following:

- **Lower overhead on bigger problems:** For both platforms the REO decreased as the problem size increased. Which follows from the fact that more execution time was also allocated as this size increased. It suffice to say that the distributed computation benefits introduced by the platforms are meaningful for longer runs. This effect is even more visible for HyperSpark was designed with Big Data calculations in mind and does not perform very well for smaller problems as the setup time is considerable.
- **Overhead range:** Both platforms had the smallest overhead for 500 jobs and 20 machines (instances 111 to 120) and the highest for the simplest instances with 20 jobs and 5 machines. For HyperSpark the smallest overhead was 6.4% on instance 111, while the largest was 77.76% for instance 6. Conversely, for ParadisEO the smallest overhead was 0.59% for instance 113, with the largest one being 36.54% for instance 10.
- **Statistical test for the difference:** Running a Wilcoxon signed rank test for the differences between ParadisEO and HyperSpark (using as paired samples the average REO on each instance) lead us to conclude that the observed difference is significant with p-value $p < 0.0001$
This is understandable considering that for every instance the overhead for ParadisEO is much lower (one order of magnitude for some) than HyperSpark.

Conclusions and Future Work

We set to compare HyperSpark implementation of distributed metaheuristics against a relevant benchmark of an established framework using different technologies. During this work we have highlighted key differences between the two frameworks and how they impact the development time, the required knowledge, the learning curve of the framework and finally the solution quality and time overhead introduced by the platform.

ParadisEO was a much more complex solution to develop requiring extensive understanding of the internal mechanisms of the framework in order to implement the problem specific representations in the proper way. The fact that it issues a compiled language also difficult the development cycle and was a frequent source of issues. However, once this investment was made the effort paid off in terms of achieving a better solution in terms of quality with less overhead introduced by the framework with the disadvantage of the lack of cluster management help.

On the other hand, the development time for HyperSpark was much less compared to ParadisEO, aided by the fact that the extension points are well defined with a clear interface for the user that does not require internal knowledge of the framework. The framework was also easier to understand as it provides a higher level interface. Managing the execution of the optimization is an important advantage of using Spark as distribution computation technology but comes at a significant cost of more overhead that might not have only if the job being executed is rather big either requiring a significant amount of execution time or processing bigger amounts of data. Considering that HyperSpark is a much younger project compared to an establish product such as ParadisEO it is fair to say that the results obtained are promising and further development could reduce the observed differences in performance.

7. CONCLUSIONS AND FUTURE WORK

Specifically to reduce uncertainties about the small differences between the algorithms implemented on each platform it would be useful to extend HyperSpark to be able to keep the whole population between iterations and not only the best individual.

Another interesting extension for HyperSpark will be the support for multiobjective optimization as is currently the case in ParadisEO. This will make more attractive the platform for end users by allowing them to tackle a more diverse range of optimization problems.

Experiment details

TABLE A.1
BENCHMARK INSTANCES METADATA

Instance number	Number of jobs	Number of machines	Lowest makespan	Execution time (s)
1	20	5	1278	3
2	20	5	1359	3
3	20	5	1081	3
4	20	5	1293	3
5	20	5	1235	3
6	20	5	1195	3
7	20	5	1234	3
8	20	5	1206	3
9	20	5	1230	3
10	20	5	1108	3
11	20	10	1582	6
12	20	10	1659	6
13	20	10	1496	6
14	20	10	1377	6
15	20	10	1419	6
16	20	10	1397	6
17	20	10	1484	6
18	20	10	1538	6
19	20	10	1593	6

Continued on next page

Table A.1 – continued from previous page

Instance number	Number of jobs	Number of machines	Lowest makespan	Execution time (s)
20	20	10	1591	6
21	20	20	2297	12
22	20	20	2099	12
23	20	20	2326	12
24	20	20	2223	12
25	20	20	2291	12
26	20	20	2226	12
27	20	20	2273	12
28	20	20	2200	12
29	20	20	2237	12
30	20	20	2178	12
31	50	5	2724	7.5
32	50	5	2834	7.5
33	50	5	2621	7.5
34	50	5	2751	7.5
35	50	5	2863	7.5
36	50	5	2829	7.5
37	50	5	2725	7.5
38	50	5	2683	7.5
39	50	5	2552	7.5
40	50	5	2782	7.5
41	50	10	2991	15
42	50	10	2867	15
43	50	10	2839	15
44	50	10	3063	15
45	50	10	2976	15
46	50	10	3006	15
47	50	10	3093	15
48	50	10	3037	15
49	50	10	2897	15
50	50	10	3065	15
51	50	20	3850	30
52	50	20	3704	30
53	50	20	3640	30
54	50	20	3723	30
Continued on next page				

Table A.1 – continued from previous page

Instance number	Number of jobs	Number of machines	Lowest makespan	Execution time (s)
55	50	20	3611	30
56	50	20	3681	30
57	50	20	3704	30
58	50	20	3691	30
59	50	20	3743	30
60	50	20	3756	30
61	100	5	5493	15
62	100	5	5268	15
63	100	5	5175	15
64	100	5	5014	15
65	100	5	5250	15
66	100	5	5135	15
67	100	5	5246	15
68	100	5	5094	15
69	100	5	5448	15
70	100	5	5322	15
71	100	10	5770	30
72	100	10	5349	30
73	100	10	5676	30
74	100	10	5781	30
75	100	10	5467	30
76	100	10	5303	30
77	100	10	5595	30
78	100	10	5617	30
79	100	10	5871	30
80	100	10	5845	30
81	100	20	6202	60
82	100	20	6183	60
83	100	20	6271	60
84	100	20	6269	60
85	100	20	6314	60
86	100	20	6364	60
87	100	20	6268	60
88	100	20	6401	60
89	100	20	6275	60
Continued on next page				

Table A.1 – continued from previous page

Instance number	Number of jobs	Number of machines	Lowest makespan	Execution time (s)
90	100	20	6434	60
91	200	10	10862	60
92	200	10	10480	60
93	200	10	10922	60
94	200	10	10889	60
95	200	10	10524	60
96	200	10	10329	60
97	200	10	10854	60
98	200	10	10730	60
99	200	10	10438	60
100	200	10	10675	60
101	200	20	11195	120
102	200	20	11203	120
103	200	20	11281	120
104	200	20	11275	120
105	200	20	11259	120
106	200	20	11176	120
107	200	20	11360	120
108	200	20	11334	120
109	200	20	11192	120
110	200	20	11168	120
111	500	20	25931	300
112	500	20	26520	300
113	500	20	26371	300
114	500	20	26456	300
115	500	20	26334	300
116	500	20	26477	300
117	500	20	26389	300
118	500	20	26560	300
119	500	20	26005	300
120	500	20	26457	300

TABLE A.2
HYPERSPARK BENCHMARK: RELATIVE PERCENTAGE DEVIATION (RPD)

Instance	RPD (%)					
	Average	Run				
		1	2	3	4	5
1	0.00	0.00	0.00	0.00	0.00	0.00
2	0.00	0.00	0.00	0.00	0.00	0.00
3	0.11	0.00	0.56	0.00	0.00	0.00
4	0.28	0.31	0.00	0.62	0.00	0.46
5	0.00	0.00	0.00	0.00	0.00	0.00
6	0.00	0.00	0.00	0.00	0.00	0.00
7	0.63	0.57	1.38	0.41	0.41	0.41
8	0.00	0.00	0.00	0.00	0.00	0.00
9	0.16	0.00	0.00	0.00	0.81	0.00
10	0.00	0.00	0.00	0.00	0.00	0.00
11	0.47	0.19	0.25	0.76	0.25	0.88
12	0.83	0.78	0.66	1.15	0.72	0.84
13	0.96	1.00	1.07	0.87	0.67	1.20
14	0.52	0.07	0.94	1.09	0.36	0.15
15	0.75	1.06	0.85	1.13	0.35	0.35
16	0.39	0.00	0.79	0.29	0.21	0.64
17	0.18	0.13	0.00	0.61	0.00	0.13
18	0.92	0.65	0.98	1.56	0.65	0.78
19	0.63	1.19	0.38	1.19	0.00	0.38
20	0.84	0.44	0.82	1.07	0.82	1.07
21	0.68	0.74	0.52	0.65	0.74	0.74
22	0.37	0.10	0.95	0.62	0.10	0.10
23	0.68	0.86	0.47	0.52	0.90	0.64
24	0.74	0.76	0.81	0.27	0.76	1.08
25	0.44	0.39	0.35	0.57	0.39	0.48
26	0.48	0.45	0.13	0.45	0.45	0.90
27	0.56	0.53	0.22	0.53	0.88	0.66
28	0.61	0.95	0.64	0.50	0.41	0.55
29	0.37	0.63	0.22	0.22	0.22	0.54
30	0.61	0.46	0.92	0.64	0.69	0.32
31	0.00	0.00	0.00	0.00	0.00	0.00
32	0.16	0.00	0.00	0.14	0.49	0.14
33	0.02	0.00	0.00	0.00	0.04	0.04
Continued on next page						

Table A.2 – continued from previous page

Instance	Relative Percentage Deviation (RPD) (%)					
	Average	Run				
		1	2	3	4	5
34	0.27	0.40	0.40	0.40	0.07	0.07
35	0.00	0.00	0.00	0.00	0.00	0.00
36	0.00	0.00	0.00	0.00	0.00	0.00
37	0.00	0.00	0.00	0.00	0.00	0.00
38	0.13	0.00	0.00	0.00	0.63	0.00
39	0.31	0.00	0.35	0.35	0.35	0.47
40	0.00	0.00	0.00	0.00	0.00	0.00
41	1.95	2.21	1.64	1.91	2.47	1.54
42	2.04	1.95	1.92	1.99	2.79	1.53
43	2.03	3.06	1.13	1.69	2.57	1.69
44	0.57	0.23	0.03	1.27	0.03	1.27
45	2.12	1.92	2.15	1.71	2.76	2.08
46	1.16	0.96	1.03	1.00	1.50	1.33
47	1.64	1.97	2.33	1.29	1.29	1.29
48	0.47	0.30	0.92	0.23	0.30	0.63
49	1.10	1.10	1.07	0.83	1.07	1.45
50	1.42	1.53	2.12	0.85	1.14	1.47
51	2.44	2.42	2.16	2.68	2.55	2.39
52	2.70	3.21	3.13	3.00	2.46	1.70
53	3.23	3.96	3.43	2.97	3.35	2.42
54	2.64	3.09	2.61	2.95	2.55	2.01
55	3.41	3.77	3.27	2.80	3.41	3.82
56	2.74	2.39	2.09	3.42	2.39	3.40
57	3.20	3.46	1.97	3.10	4.35	3.10
58	3.51	3.39	3.66	3.20	3.98	3.33
59	3.09	3.07	2.86	3.21	2.97	3.34
60	2.55	2.77	2.13	2.21	2.77	2.88
61	0.01	0.00	0.00	0.04	0.00	0.04
62	0.30	0.30	0.30	0.30	0.30	0.30
63	0.19	0.19	0.43	0.00	0.35	0.00
64	0.18	0.18	0.18	0.18	0.18	0.18
65	0.08	0.10	0.06	0.10	0.10	0.06
66	0.06	0.00	0.18	0.08	0.00	0.04
67	0.00	0.00	0.00	0.00	0.00	0.00
Continued on next page						

Table A.2 – continued from previous page

Instance	Relative Percentage Deviation (RPD) (%)					
	Average	Run				
		1	2	3	4	5
68	0.30	0.08	0.59	0.24	0.59	0.00
69	0.00	0.00	0.00	0.00	0.00	0.00
70	0.35	0.38	0.38	0.23	0.38	0.38
71	0.31	0.52	0.24	0.29	0.24	0.23
72	0.73	0.52	0.79	1.01	0.79	0.52
73	0.35	0.26	0.26	0.67	0.26	0.26
74	1.16	0.85	2.14	0.78	0.83	1.21
75	1.06	1.06	0.66	1.30	0.82	1.45
76	0.32	0.09	0.47	0.34	0.34	0.34
77	0.87	0.93	0.97	1.14	0.64	0.66
78	0.92	1.14	0.85	1.14	0.85	0.59
79	0.53	0.34	0.34	0.60	0.97	0.39
80	0.58	0.99	0.62	0.62	0.05	0.62
81	3.25	3.35	3.40	3.77	2.84	2.90
82	3.24	3.69	2.99	3.25	3.33	2.93
83	2.82	1.74	3.32	3.59	2.77	2.68
84	2.24	2.58	2.49	2.31	2.19	1.61
85	2.70	2.66	2.26	2.57	3.50	2.52
86	2.60	3.21	3.13	2.66	2.01	1.98
87	3.25	3.69	3.16	3.22	3.00	3.19
88	3.55	2.94	3.66	4.20	3.12	3.84
89	3.02	3.49	2.22	3.11	2.95	3.33
90	2.12	2.04	2.47	1.90	1.63	2.58
91	0.38	0.37	0.37	0.37	0.21	0.61
92	0.88	0.94	0.74	0.86	0.88	0.98
93	0.58	0.67	0.47	0.50	0.87	0.39
94	0.46	0.46	0.46	0.46	0.60	0.35
95	0.50	0.48	0.48	0.28	1.12	0.12
96	0.57	0.70	0.77	0.77	0.17	0.45
97	0.69	0.54	1.15	0.54	0.70	0.50
98	0.63	0.63	0.63	0.63	0.62	0.63
99	0.41	0.34	0.38	0.31	0.38	0.62
100	0.70	1.29	0.49	0.73	0.49	0.49
101	2.62	2.91	2.44	2.55	2.26	2.95

Continued on next page

Table A.2 – continued from previous page

Instance	Relative Percentage Deviation (RPD) (%)					
	Average	Run				
		1	2	3	4	5
102	3.65	4.18	3.56	3.71	3.37	3.45
103	3.18	3.37	3.24	3.33	3.08	2.91
104	3.16	3.27	2.89	2.91	3.15	3.56
105	2.44	2.51	3.32	2.37	2.34	1.68
106	2.97	3.24	3.07	2.59	3.16	2.81
107	3.29	3.84	3.23	3.24	2.95	3.20
108	2.95	3.26	3.05	2.31	3.03	3.11
109	3.26	3.09	3.14	3.58	3.06	3.42
110	4.31	4.45	4.61	4.09	4.65	3.75
111	2.82	2.82	2.87	2.83	2.71	2.88
112	2.27	1.97	2.30	2.51	2.52	2.06
113	2.11	2.10	2.45	1.90	1.82	2.28
114	1.62	1.61	1.40	1.72	1.50	1.87
115	1.62	1.70	1.55	1.94	1.50	1.42
116	1.94	1.98	1.87	1.72	2.20	1.94
117	1.60	1.53	1.54	1.30	1.86	1.75
118	1.84	1.72	1.55	1.98	1.87	2.06
119	2.04	2.20	2.03	2.11	1.55	2.31
120	1.96	1.96	2.18	1.92	1.67	2.07

TABLE A.3
PARADISEO BENCHMARK: RELATIVE PERCENTAGE DEVIATION (RPD)

Instance	RPD (%)					
	Average	Run				
		1	2	3	4	5
1	0.00	0.00	0.00	0.00	0.00	0.00
2	0.00	0.00	0.00	0.00	0.00	0.00
3	0.00	0.00	0.00	0.00	0.00	0.00
4	0.00	0.00	0.00	0.00	0.00	0.00
5	0.00	0.00	0.00	0.00	0.00	0.00
6	0.00	0.00	0.00	0.00	0.00	0.00
7	0.60	0.41	0.41	0.41	1.38	0.41
8	0.00	0.00	0.00	0.00	0.00	0.00
9	0.00	0.00	0.00	0.00	0.00	0.00
10	0.00	0.00	0.00	0.00	0.00	0.00
11	0.11	0.19	0.19	0.00	0.19	0.00
12	0.47	0.60	0.60	0.06	0.48	0.60
13	0.32	0.27	0.00	0.27	0.74	0.33
14	0.41	0.07	0.15	0.51	0.36	0.94
15	0.20	0.21	0.00	0.00	0.35	0.42
16	0.09	0.00	0.00	0.00	0.43	0.00
17	0.00	0.00	0.00	0.00	0.00	0.00
18	0.70	0.78	0.78	0.52	0.59	0.85
19	0.11	0.56	0.00	0.00	0.00	0.00
20	0.53	0.00	0.82	0.44	0.82	0.57
21	0.52	0.52	0.52	0.30	0.48	0.78
22	0.12	0.10	0.29	0.10	0.05	0.10
23	0.37	0.39	0.47	0.69	0.17	0.13
24	0.32	0.27	0.27	0.27	0.54	0.27
25	0.38	0.31	0.44	0.57	0.22	0.39
26	0.36	0.40	0.36	0.40	0.27	0.36
27	0.46	0.57	0.22	0.22	0.70	0.57
28	0.29	0.41	0.55	0.18	0.32	0.00
29	0.13	0.22	0.22	0.22	0.00	0.00
30	0.22	0.28	0.32	0.28	0.23	0.00
31	0.00	0.00	0.00	0.00	0.00	0.00
32	0.03	0.00	0.00	0.07	0.07	0.00
33	0.00	0.00	0.00	0.00	0.00	0.00
Continued on next page						

Table A.3 – continued from previous page

Instance	Relative Percentage Deviation (RPD) (%)					
	Average	Run				
		1	2	3	4	5
34	0.09	0.00	0.00	0.07	0.40	0.00
35	0.00	0.00	0.00	0.00	0.00	0.00
36	0.00	0.00	0.00	0.00	0.00	0.00
37	0.00	0.00	0.00	0.00	0.00	0.00
38	0.00	0.00	0.00	0.00	0.00	0.00
39	0.21	0.00	0.35	0.35	0.35	0.00
40	0.00	0.00	0.00	0.00	0.00	0.00
41	1.72	1.60	1.94	1.54	1.91	1.60
42	1.81	1.67	1.95	1.53	1.95	1.95
43	1.66	1.48	1.97	1.80	1.48	1.55
44	0.27	0.56	0.26	0.26	0.03	0.26
45	1.51	1.01	1.58	1.51	1.98	1.44
46	0.92	1.00	0.96	1.20	0.43	1.00
47	1.01	1.10	1.23	1.29	0.71	0.71
48	0.34	0.36	0.36	0.30	0.36	0.30
49	0.78	0.45	0.97	0.97	0.97	0.55
50	1.11	0.85	0.88	1.11	0.88	1.83
51	1.89	1.87	1.19	2.10	2.44	1.82
52	2.41	2.51	2.59	2.13	2.08	2.73
53	2.87	2.88	3.41	2.58	2.25	3.21
54	2.11	2.52	1.58	1.88	1.99	2.55
55	2.84	3.02	2.60	2.94	2.71	2.91
56	2.00	2.04	1.49	2.25	2.04	2.17
57	2.35	2.89	1.86	2.11	2.89	2.02
58	2.53	2.87	2.00	2.87	1.82	3.06
59	2.15	2.03	2.24	2.56	2.22	1.68
60	1.82	1.89	2.48	1.86	0.96	1.92
61	0.00	0.00	0.00	0.00	0.00	0.00
62	0.14	0.13	0.28	0.13	0.13	0.00
63	0.00	0.00	0.00	0.00	0.00	0.00
64	0.08	0.06	0.14	0.08	0.08	0.06
65	0.03	0.04	0.04	0.04	0.00	0.06
66	0.00	0.00	0.00	0.00	0.00	0.00
67	0.10	0.00	0.25	0.00	0.00	0.25
Continued on next page						

Table A.3 – continued from previous page

Instance	Relative Percentage Deviation (RPD) (%)					
	Average	Run				
		1	2	3	4	5
68	0.00	0.00	0.00	0.00	0.00	0.00
69	0.00	0.00	0.00	0.00	0.00	0.00
70	0.09	0.11	0.00	0.11	0.11	0.11
71	0.27	0.24	0.24	0.16	0.24	0.45
72	0.47	0.79	0.52	0.24	0.79	0.00
73	0.22	0.26	0.26	0.26	0.26	0.05
74	0.86	0.90	0.90	0.78	0.93	0.78
75	0.65	0.57	0.64	0.66	0.73	0.66
76	0.19	0.34	0.09	0.09	0.09	0.34
77	0.55	0.66	0.93	0.21	0.48	0.48
78	0.54	0.59	0.59	0.59	0.34	0.59
79	0.31	0.34	0.34	0.15	0.39	0.34
80	0.05	0.05	0.05	0.05	0.05	0.05
81	2.70	2.90	2.56	2.11	2.64	3.27
82	2.62	3.06	2.68	2.05	2.46	2.85
83	2.38	2.06	2.50	2.20	2.49	2.66
84	1.68	1.83	1.74	1.63	1.61	1.60
85	2.33	1.66	2.45	2.44	2.60	2.52
86	2.36	2.40	2.33	2.11	2.40	2.55
87	2.67	3.13	2.68	2.60	2.49	2.44
88	2.87	2.37	2.86	3.23	2.39	3.48
89	2.32	2.34	2.28	2.28	2.26	2.45
90	1.80	1.73	2.13	1.76	1.85	1.55
91	0.13	0.21	0.09	0.17	0.09	0.09
92	0.61	0.68	0.50	0.74	0.53	0.60
93	0.18	0.05	0.20	0.39	0.01	0.24
94	0.03	0.00	0.04	0.04	0.04	0.04
95	0.12	0.12	0.12	0.12	0.12	0.12
96	0.32	0.38	0.45	0.45	0.27	0.05
97	0.29	0.26	0.28	0.41	0.26	0.26
98	0.45	0.23	0.21	0.63	0.55	0.63
99	0.26	0.26	0.26	0.26	0.26	0.26
100	0.29	0.09	0.49	0.49	0.09	0.29
101	1.95	1.89	2.00	1.74	1.82	2.29
Continued on next page						

Table A.3 – continued from previous page

Instance	Relative Percentage Deviation (RPD) (%)					
	Average	Run				
		1	2	3	4	5
102	3.02	2.62	3.09	3.27	3.21	2.91
103	2.74	3.08	3.02	2.86	2.35	2.36
104	2.79	2.61	2.77	2.94	3.07	2.59
105	1.82	1.63	1.75	1.87	1.63	2.22
106	2.54	2.42	2.51	2.75	2.66	2.40
107	2.24	2.19	2.36	2.19	2.48	1.98
108	2.27	2.07	2.74	2.10	2.46	1.99
109	2.47	2.56	2.71	2.50	2.47	2.14
110	3.80	4.25	3.57	3.48	4.21	3.47
111	2.38	2.75	2.23	2.41	2.48	2.05
112	1.86	1.69	1.93	1.98	1.84	1.89
113	1.55	1.60	1.83	1.38	1.44	1.49
114	1.26	1.30	1.36	1.36	1.12	1.17
115	1.28	1.48	1.18	1.10	1.31	1.32
116	1.49	1.57	1.31	1.36	1.53	1.67
117	1.07	1.09	1.11	0.98	1.17	0.99
118	1.51	1.39	1.39	1.72	1.55	1.49
119	1.61	1.65	1.56	1.74	1.84	1.24
120	1.45	1.30	1.41	1.45	1.60	1.50

TABLE A.4
HYPERSPARK BENCHMARK: RELATIVE EXECUTION OVERHEAD (REO)

Instance	REO (%)					
	Average	Run				
		1	2	3	4	5
1	76.55	77.17	76.45	72.98	79.48	76.68
2	76.54	77.40	77.44	75.19	76.02	76.64
3	75.76	76.60	76.37	74.60	76.91	74.33
4	76.02	76.53	76.81	76.97	76.25	73.53
5	76.30	76.31	76.79	77.18	77.20	74.00
6	77.76	78.86	77.34	77.31	79.74	75.53
7	76.56	77.37	77.37	76.85	76.95	74.27
8	76.84	77.24	75.87	76.66	77.09	77.34
9	75.88	75.69	72.58	77.01	77.12	76.99
10	76.94	77.52	75.05	77.60	76.73	77.81
11	62.22	62.66	63.08	63.65	58.87	62.83
12	63.69	62.74	63.73	64.92	62.26	64.81
13	61.96	62.32	64.21	60.58	62.81	59.88
14	63.26	64.50	60.74	63.40	63.98	63.66
15	63.36	63.55	64.01	62.14	63.36	63.76
16	62.33	61.83	63.27	59.59	64.20	62.73
17	65.07	63.56	62.20	70.43	64.01	65.14
18	63.51	64.77	63.04	64.19	61.03	64.51
19	62.81	64.10	62.70	62.78	63.76	60.71
20	62.46	63.14	60.06	63.48	63.63	61.99
21	46.93	46.95	48.58	48.55	46.77	43.79
22	47.52	47.17	45.73	47.81	47.95	48.95
23	47.19	49.23	45.04	48.56	45.84	47.29
24	46.51	46.56	45.65	47.79	43.69	48.88
25	47.01	46.22	47.19	47.59	46.60	47.46
26	46.32	41.09	47.65	45.54	48.94	48.38
27	45.46	47.16	47.71	45.65	44.93	41.84
28	46.57	46.38	44.22	47.03	46.96	48.25
29	47.49	48.60	47.17	48.42	44.34	48.91
30	46.86	47.23	46.46	46.25	46.42	47.91
31	56.40	52.32	59.31	53.70	58.43	58.23
32	60.19	70.21	64.03	54.50	58.98	53.21
33	59.82	58.52	57.79	68.08	57.02	57.68

Continued on next page

Table A.4 – continued from previous page

Instance	Relative Execution Overhead (REO) (%)					
	Average	Run				
		1	2	3	4	5
34	57.80	54.74	58.61	57.53	59.19	58.93
35	58.74	57.55	59.38	58.94	58.76	59.05
36	57.64	56.74	58.99	57.42	58.70	56.35
37	64.52	59.02	67.54	59.74	68.85	67.47
38	61.38	59.17	67.63	68.39	53.67	58.02
39	57.06	56.76	58.75	53.30	57.68	58.81
40	58.31	58.87	58.56	58.15	57.71	58.26
41	41.82	41.21	43.76	43.60	38.02	42.53
42	41.80	41.84	40.69	42.74	43.48	40.25
43	42.00	43.00	38.47	43.16	42.75	42.60
44	42.23	41.59	41.93	41.39	42.60	43.62
45	42.35	40.20	42.77	43.33	42.50	42.95
46	41.57	40.97	42.86	44.75	37.51	41.75
47	42.13	43.19	42.34	41.97	40.31	42.85
48	51.41	62.16	49.96	48.59	47.59	48.76
49	43.42	48.70	41.12	42.63	43.33	41.34
50	40.45	43.31	39.21	41.26	42.44	36.03
51	28.17	28.85	28.34	26.76	27.47	29.45
52	30.99	34.04	28.53	33.55	24.87	33.97
53	33.73	34.13	33.52	33.99	33.18	33.81
54	33.25	33.37	33.20	31.63	34.35	33.70
55	33.23	33.90	33.23	33.43	32.19	33.39
56	33.54	33.00	34.08	34.03	33.25	33.32
57	33.14	33.36	32.56	32.63	33.68	33.46
58	33.29	33.20	31.47	32.04	35.05	34.66
59	33.58	33.75	34.30	32.76	33.24	33.87
60	32.72	34.17	33.31	33.02	32.43	30.67
61	49.09	48.64	49.98	48.37	50.23	48.23
62	45.19	47.47	49.28	46.71	40.62	41.86
63	43.60	40.90	42.05	42.97	43.59	48.51
64	48.11	48.32	48.25	48.57	49.15	46.24
65	43.91	41.30	48.51	45.35	41.94	42.43
66	41.90	41.66	42.60	39.32	43.45	42.45
67	43.21	47.64	41.82	42.02	42.52	42.05
Continued on next page						

Table A.4 – continued from previous page

Instance	Relative Execution Overhead (REO) (%)					
	Average	Run				
		1	2	3	4	5
68	40.16	41.39	40.80	41.24	39.74	37.59
69	42.06	41.93	43.08	42.24	42.37	40.68
70	41.13	41.73	43.24	36.46	43.26	40.99
71	28.25	28.62	29.29	28.37	27.29	27.70
72	29.42	26.84	26.90	32.90	27.70	32.77
73	32.07	33.29	28.61	33.07	31.71	33.68
74	32.74	33.38	33.92	31.99	30.90	33.49
75	33.58	33.96	33.29	33.22	34.44	32.98
76	32.17	34.01	32.94	32.75	31.64	29.50
77	33.23	33.00	33.32	32.66	34.36	32.82
78	33.35	33.72	33.60	33.25	32.18	34.02
79	33.64	34.18	33.76	33.19	32.91	34.17
80	32.49	33.30	32.11	33.63	31.24	32.16
81	20.40	20.18	19.94	20.88	19.98	21.01
82	20.41	21.19	21.28	19.66	19.87	20.07
83	20.26	20.19	20.76	20.74	19.27	20.35
84	20.50	20.72	20.85	19.92	20.89	20.09
85	20.78	20.69	21.38	20.74	20.55	20.52
86	20.71	20.90	20.93	20.82	19.66	21.24
87	20.30	20.30	19.80	20.70	20.65	20.03
88	20.36	20.56	20.52	20.17	20.73	19.82
89	20.44	21.29	20.84	21.54	20.40	18.15
90	20.33	21.43	20.78	20.22	20.90	18.30
91	20.62	21.13	20.25	20.68	19.73	21.32
92	20.38	21.16	20.37	20.83	21.08	18.46
93	20.60	20.60	21.13	20.01	20.41	20.84
94	20.65	20.27	21.18	20.90	20.11	20.78
95	21.13	21.54	20.84	20.66	21.31	21.29
96	20.28	21.83	20.15	20.50	18.82	20.11
97	20.90	20.72	21.47	21.00	20.08	21.22
98	20.35	20.87	20.88	20.67	19.64	19.69
99	21.29	20.97	22.35	21.04	21.48	20.63
100	20.63	21.13	20.90	19.81	20.93	20.36
101	12.34	12.66	12.46	11.83	12.31	12.46

Continued on next page

Table A.4 – continued from previous page

Instance	Relative Execution Overhead (REO) (%)					
	Average	Run				
		1	2	3	4	5
102	12.38	12.20	12.28	13.00	12.71	11.72
103	12.51	12.04	12.72	12.08	12.81	12.90
104	12.69	12.49	12.88	12.72	12.02	13.33
105	12.60	12.88	12.42	12.56	12.78	12.39
106	12.58	12.46	12.09	13.10	12.93	12.31
107	12.37	12.24	12.02	12.14	12.63	12.83
108	12.37	11.84	12.08	12.73	12.77	12.44
109	12.12	12.43	12.20	10.98	12.53	12.47
110	12.33	12.85	12.93	12.42	11.76	11.70
111	6.42	6.45	6.34	6.39	6.14	6.81
112	6.66	6.46	6.77	6.68	6.42	6.94
113	6.58	6.51	6.42	6.58	6.85	6.53
114	6.73	6.72	6.67	6.64	6.86	6.75
115	6.64	6.79	6.66	6.70	6.69	6.37
116	6.53	6.55	6.73	6.22	6.71	6.43
117	6.72	6.45	7.23	6.38	6.91	6.64
118	6.65	6.77	6.78	6.66	6.39	6.63
119	6.61	6.53	6.69	6.56	6.70	6.59
120	6.47	6.30	6.69	6.40	6.34	6.63

TABLE A.5
PARADISEO BENCHMARK: RELATIVE EXECUTION OVERHEAD (REO)

Instance	REO (%)					
	Average	Run				
		1	2	3	4	5
1	36.10	36.03	36.62	36.18	35.73	35.94
2	35.63	35.86	35.61	35.48	35.11	36.08
3	35.93	35.16	36.88	35.64	35.98	35.99
4	36.25	36.51	36.70	36.25	36.13	35.65
5	35.68	35.47	35.55	35.82	35.37	36.21
6	36.15	35.18	35.68	35.86	37.43	36.60
7	36.08	35.88	36.01	35.29	37.42	35.80
8	35.87	36.33	35.90	35.51	35.71	35.91
9	36.38	35.72	37.04	37.77	35.66	35.72
10	35.74	36.53	36.05	35.23	35.44	35.44
11	22.02	21.85	21.54	23.19	21.88	21.67
12	21.90	21.44	21.94	21.72	22.08	22.34
13	21.87	22.57	21.50	21.69	21.68	21.91
14	24.10	22.64	30.24	21.85	23.55	22.24
15	21.89	21.80	21.79	21.59	22.35	21.94
16	22.11	22.19	22.14	22.56	21.80	21.83
17	22.04	21.79	21.89	21.71	22.76	22.05
18	21.66	21.53	21.96	21.79	21.38	21.66
19	22.03	21.94	22.25	22.46	21.79	21.70
20	22.28	22.14	21.79	21.54	22.50	23.45
21	12.46	12.27	12.52	12.42	12.48	12.61
22	12.77	12.53	12.48	13.62	13.04	12.19
23	12.42	12.28	12.28	12.29	12.69	12.56
24	12.89	12.59	12.33	13.14	13.49	12.88
25	12.83	12.56	13.31	13.16	12.68	12.47
26	12.32	12.07	12.89	12.12	12.30	12.22
27	12.45	12.68	12.31	12.27	12.42	12.59
28	13.22	12.88	14.37	12.36	14.19	12.27
29	12.39	12.50	12.57	12.35	12.49	12.02
30	12.40	12.74	11.91	12.82	12.48	12.08
31	19.27	19.33	18.18	18.51	20.25	20.09
32	18.53	18.85	18.14	18.02	18.38	19.25
33	18.86	18.87	21.13	18.14	17.96	18.23

Continued on next page

Table A.5 – continued from previous page

Instance	Relative Execution Overhead (REO) (%)					
	Average	Run				
		1	2	3	4	5
34	18.41	18.55	18.58	17.96	18.60	18.35
35	18.13	18.07	17.93	18.50	18.37	17.79
36	18.65	18.45	18.15	18.49	18.91	19.27
37	18.97	18.55	18.22	18.27	20.89	18.93
38	18.55	19.55	18.74	18.43	18.02	18.00
39	18.52	18.56	18.29	18.59	18.81	18.33
40	18.39	18.10	18.14	18.28	19.47	17.99
41	10.01	9.86	10.10	9.99	10.11	9.97
42	10.12	10.08	10.18	10.02	10.06	10.28
43	10.21	10.37	10.20	10.01	10.29	10.16
44	10.03	10.05	10.15	9.92	9.98	10.03
45	10.25	10.39	10.15	9.87	10.13	10.69
46	10.23	9.89	9.95	10.21	10.32	10.76
47	10.00	9.95	10.32	9.95	9.89	9.89
48	10.13	10.50	10.08	9.88	10.02	10.17
49	10.01	10.03	9.85	10.32	9.90	9.96
50	10.21	9.96	10.69	10.07	10.14	10.20
51	6.09	7.24	5.64	5.62	6.46	5.51
52	5.34	5.34	5.25	5.46	5.19	5.45
53	5.44	5.47	5.48	5.32	5.46	5.48
54	5.31	5.21	5.36	5.44	5.28	5.25
55	5.26	5.23	5.17	5.26	5.38	5.28
56	5.43	5.29	5.44	5.37	5.53	5.52
57	5.52	5.65	5.94	5.34	5.34	5.32
58	5.35	5.30	5.36	5.28	5.30	5.50
59	5.27	5.34	5.23	5.29	5.31	5.20
60	5.44	5.57	5.29	5.46	5.50	5.37
61	10.08	10.19	10.08	10.02	9.93	10.19
62	10.13	10.53	9.89	9.97	10.19	10.10
63	10.15	10.46	9.95	10.39	10.07	9.87
64	10.02	9.93	9.93	9.86	10.19	10.17
65	10.14	9.91	10.07	10.24	10.62	9.85
66	10.17	10.39	10.23	10.07	10.17	9.97
67	10.28	10.51	10.47	9.94	10.14	10.32
Continued on next page						

Table A.5 – continued from previous page

Instance	Relative Execution Overhead (REO) (%)					
	Average	Run				
		1	2	3	4	5
68	10.05	9.96	10.05	10.21	10.02	10.01
69	10.26	10.19	10.64	10.04	10.07	10.34
70	10.22	10.80	9.96	10.10	10.10	10.14
71	5.97	5.41	5.59	5.37	8.03	5.43
72	5.72	5.56	5.31	5.26	5.38	7.11
73	5.47	5.98	5.18	5.30	5.30	5.58
74	5.46	5.36	5.61	5.51	5.50	5.34
75	5.65	5.25	5.68	5.44	5.51	6.38
76	5.74	6.49	6.18	5.41	5.31	5.28
77	5.54	5.47	5.48	5.69	5.74	5.31
78	5.42	5.39	5.25	5.48	5.51	5.46
79	5.39	5.40	5.60	5.31	5.28	5.35
80	5.38	5.31	5.44	5.43	5.38	5.33
81	2.88	2.79	2.78	2.92	2.81	3.09
82	2.80	2.82	2.75	2.74	2.89	2.77
83	2.83	2.79	2.77	2.72	3.11	2.76
84	2.75	2.76	2.70	2.73	2.79	2.77
85	2.75	2.74	2.73	2.73	2.79	2.75
86	2.76	2.72	2.78	2.79	2.77	2.71
87	3.32	2.70	2.75	2.87	2.82	5.44
88	2.78	2.75	2.72	2.68	2.79	2.95
89	2.79	2.72	2.76	2.82	2.73	2.92
90	2.86	2.78	2.86	2.94	2.79	2.93
91	2.85	2.81	2.91	2.91	2.82	2.81
92	2.82	2.98	2.79	2.82	2.80	2.70
93	3.90	6.40	3.29	3.99	2.76	3.04
94	2.84	2.87	2.81	2.80	2.87	2.87
95	2.86	2.75	2.81	2.77	3.10	2.87
96	2.80	2.78	2.76	2.76	2.91	2.78
97	2.94	2.87	2.94	3.09	2.90	2.90
98	2.79	2.86	2.75	2.86	2.72	2.78
99	2.83	2.97	2.75	2.80	2.77	2.83
100	2.92	2.77	2.84	2.82	2.79	3.38
101	1.54	1.86	1.47	1.43	1.45	1.47
Continued on next page						

Table A.5 – continued from previous page

Instance	Relative Execution Overhead (REO) (%)					
	Average	Run				
		1	2	3	4	5
102	1.44	1.49	1.43	1.41	1.43	1.43
103	1.47	1.44	1.43	1.45	1.47	1.55
104	1.46	1.50	1.55	1.44	1.41	1.41
105	1.43	1.46	1.43	1.43	1.40	1.43
106	1.49	1.42	1.43	1.42	1.76	1.43
107	1.55	1.45	2.01	1.40	1.41	1.46
108	1.50	1.43	1.44	1.39	1.41	1.81
109	1.43	1.45	1.42	1.41	1.43	1.43
110	1.51	1.41	1.55	1.51	1.53	1.57
111	0.61	0.63	0.62	0.60	0.62	0.59
112	0.62	0.60	0.59	0.60	0.72	0.61
113	1.33	0.59	0.59	1.04	1.06	3.38
114	1.16	3.34	0.59	0.63	0.62	0.63
115	0.63	0.66	0.63	0.64	0.62	0.62
116	0.73	0.60	0.58	0.60	0.73	1.12
117	0.62	0.60	0.63	0.60	0.64	0.61
118	0.61	0.64	0.62	0.59	0.60	0.61
119	0.62	0.60	0.66	0.59	0.59	0.65
120	0.61	0.60	0.61	0.63	0.63	0.58

Listing A.1: ParadisEO parameters configuration

```

# Lines commented out are using default values

### GENERAL #####
# --help=0                # -h : Prints this message
# --stopOnUnknownParam=1  # Stop if unknwn param
#--seed=118337975        # -S : Random number seed

### EVOLUTION ENGINE #####
# --popSize=20            # -P : Population Size
--selection=DetTour(2)    # -S : Selection:
                          # DetTour(T), StochTour(t)
                          # Roulette , Ranking(p,e)
                          # Sequential
--nbOffspring=2          # -O : No of offspring
--replacement=SSGADet(2) # -R : Replacement:
                          # Comma, Plus or
                          # EPTour(T), SSGAWorst,
                          # SSGADet(T), SSGAStoch(t)
--weakElitism=1         # -w : Old best parent
                          # replaces new worst
                          # offspring *if necessary*

### MULTISTART #####
--totalTime=0            # Desired wallclock time
--iterations=10         # Number of times the
                          # algo is started on each
                          # worker. missing = 2

### OUTPUT #####
--useEval=0              # Use nb of eval. as
                          # counter (vs nb of gen.)
# --useTime=1            # Display time (s)
--printBestStat=0        # Print Best/avg/stddev
# --printPop=0           # Print sorted pop

### OUTPUT - DISK #####
# --resDir=Res           # Directory to store out
# --eraseDir=1           # erase files in dirName
# --fileBestStat=0       # Output bes/avg/std

```

A. EXPERIMENT DETAILS

```
### OUTPUT - GRAPHICAL #####
# --plotBestStat=0          # Plot Best/avg Stat
# --plotHisto=0           # Plot fitness histogram

### PERSISTENCE #####
# --Load=                  # -L : A save file
# --recomputeFitness=0    # -r : Recompute fitness
# --saveFrequency=0       # Save every F generation
# --saveTimeInterval=0    # Save every T seconds
# --status=FlowShopEO.status # Status file

### REPRESENTATION #####
#--BenchmarkFile=instances/ta023.txt # -B : File name

### STOPPING CRITERION #####
--maxGen=0                # -G: Maximum generations
# --steadyGen=100         # -s: Number of
                           #       generations with no
                           #       improvement
# --minGen=0              # -g : Minimum number of
                           #       generations
--maxEval=0               # -E : Maximum number of
                           #       evaluations
# --targetFitness=0       # -T : Stop fitness
--maxTime=5               # -T : Maximum running
                           #       time per iteration
                           #       in seconds

### VARIATION OPERATORS #####
# --crossRate=1           # Relative rate for the
                           #       only crossover
# --shiftMutRate=0.5      # Relative rate for shift
                           #       mutation
# --exchangeMutRate=0.5   # Relative rate for
                           #       exchange mutation
--pCross=0.5              # -c : Probability of
                           #       Crossover
--pMut=0.7                # -m : Probability of
                           #       Mutation
```

Bibliography

- [1] S. Cahon, N. Melab, and E.-G. Talbi. Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10(3):357–380, 2003.
- [2] Politecnico di Milano. Policloud.
- [3] 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.
- [4] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer, Berlin, 2007.
- [5] 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.
- [6] Apache Software Foundation. Apache Hadoop.
- [7] Apache Software Foundation. Apache spark official website.
- [8] GitHub. Github: How people build software.
- [9] Al Globus, Greg Hornby, Derek Linden, and Jason Lohn. Automated antenna design with evolutionary algorithms.
- [10] 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.
- [11] INRIA. Paradiseo. a software framework for meta heuristics.

- [12] H. Ishibuchi and T. Murata. Multi-objective genetic local search for minimizing the number of fuzzy rules for pattern classification problems. In *Fuzzy Systems Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, volume 2, pages 1100–1105 vol.2, 1998.
- [13] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of IEEE International Conference on Neural Networks*, pages 1942–1948. IEEE, 1995.
- [14] Arnaud Liefoghe. Paradiseo. use paradiseo-moeo to solve the flowshop problem.
- [15] Arnaud Liefoghe, Matthieu Basseur, Laetitia Jourdan, and El-Ghazali Talbi. *Evolutionary Multi-Criterion Optimization: 4th International Conference, EMO 2007, Matsushima, Japan, March 5-8, 2007. Proceedings*, chapter ParadisEO-MOEO: A Framework for Evolutionary Multi-objective Optimization, pages 386–400. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
- [16] Arnaud Liefoghe, Matthieu Basseur, Laetitia Jourdan, and El-Ghazali Talbi. *Evolutionary Multi-Criterion Optimization: 4th International Conference, EMO 2007, Matsushima, Japan, March 5-8, 2007. Proceedings*, chapter Combinatorial Optimization of Stochastic Multi-objective Problems: An Application to the Flow-Shop Scheduling Problem, pages 457–471. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
- [17] David D. L. Minh and Do Le (Paul) Minh. Understanding the hastings algorithm. *Communications in Statistics - Simulation and Computation*, 44(2):332–349, 2015.
- [18] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, 1996.
- [19] 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.
- [20] Eugeniusz Nowicki and Czeslaw Smutnicki. Permutation flow shop scheduling. benchmarks results. Technical report, Institute of Engineering Cybernetics, Technical University of Wroclaw., 1996.
- [21] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

-
- [22] Ih Osman and Cn Potts. Simulated annealing for permutation flow-shop scheduling. *Omega*, 17(6):551–557, 1989.
- [23] Colin R. Reeves. A genetic algorithm for flowshop sequencing. *Comput. Oper. Res.*, 22(1):5–13, Jan 1995.
- [24] 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.
- [25] Sayantan Sur, Matthew J. Koop, and Dhabaleswar K. Panda. High-performance and scalable mpi over infiniband with reduced memory usage: An in-depth performance analysis. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC '06*, New York, NY, USA, 2006. ACM.
- [26] E. Taillard. Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research*, 47(1):65 – 74, 1990.
- [27] Éric Talliard. Benchmarks for basic scheduling problems. *ORWP89*, December 1989.
- [28] Eva Vallada and Rubén Ruiz. Cooperative metaheuristics for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 193(2):365 – 376, 2009.
- [29] 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.