POLYTECHNIC UNIVERSITY OF CATALONIA (UPC) - BARCELONATECH

MASTER THESIS

Improving object management in HPC workflows

Author:
Sergio RODRÍGUEZ GUASCH

Supervisor:
Dra. Rosa Maria BADIA SALA

A thesis submitted in fulfillment of the requirements
for the degree of Master in Innovation and Research in Informatics

in the

Barcelona School of Informatics (FIB)

August 5, 2019

Declaration of Authorship

I, Sergio RODRÍGUEZ GUASCH, declare that this thesis titled, "Improving object management in HPC workflows" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Love is better than hate, because it brings harmony instead of conflict into the desires of the persons concerned. Two people between whom there is love succeed or fail together, but when two people hate each other the success of either is the failure of the other."

Bertrand Russell

POLYTECHNIC UNIVERSITY OF CATALONIA (UPC) - BARCELONATECH

Barcelona School of Informatics (FIB)

MASTER IN INNOVATION AND RESEARCH IN INFORMATICS

Abstract

Improving object management in HPC workflows

by Sergio RODRÍGUEZ GUASCH

Object management represents a substantial fraction of the total computing time in any distributed application, and it also adds complexity in terms of source code. This project proposes and implements a set of features aimed to improve both the usability and performance of distributed applications with heavy object management in task-based parallel and distributed programming models.

Acknowledgements

I would like to thank the whole Workflows and Distributed Systems team for their patience and help, especially Francesc Lordan for guiding me through the complex maze of the COMPSs Runtime. I am also very grateful to Pol Álvarez, Cristián Ramón-Cortés and Ramón Amela for their help, kindness, knowledge and their almost inhuman ability for dealing with my rants, which were not uncommon.

Besides my research team, I would like to express my sincere gratitude and love to Clara, my girlfriend. If it is hard to deal with me at work, just imagine what it is like to live with me.

Finally, I also want to thank my family. Thanks for everything. Jero, we will never forget you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Most modern research fields use computational resources in some way or another. This trend started some decades ago, and it seems to be increasingly accepted and adopted among all the scientific fields. While computer science contributed to solve many research problems, it also created new challenges to researchers, as the increasing difficulty on programming and software development due to the increasing sizes of data sets, experiments, and number and complexity of the available computational resources. This problem becomes even more noticeable when a research group has no computer scientists and all the programming tasks are done by non-experts. This happens to be very common in fields like biology, chemistry or physics.

Non-expert programming has always been an issue, but it was far more manageable when all the programs ran sequentially in a single core machine. Multi-core CPUs allowed a program to run various fragments of its code at the same time and thus increased the difficulty of programming both in a conceptual and in a technical way. Parallel programming introduced concepts such as *race condition*, *data dependency*, *critical region*, *parallelism factor*, *granularity*, *overhead* and many more, and the existing frameworks (e.g: native threads) were too low level to be understood or used by non computer scientists, and implied high development and maintainment costs.

This shift towards more complex computational models and programs due to the presence of parallelism encouraged the industry to create easier frameworks and programming models for parallel computing. One of the greatest examples of these new frameworks is OpenMP[11]. OpenMP simplified the task of writing parallel programs a lot, making it understandable to non computer scientists. As an example, a working sequential matrix multiplication algorithm can be written as follows:

```
#pragma omp parallel for shared(a,b,c) private(i, j, k)
for (i = 0; i < size; ++i) {
    for (j = 0; j < size; ++j) {
        for (k = 0; k < size; ++k) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

Note that all the forks, joins, private copies and similar are just *specified*, but not done explicitly. This simplification allowed the general programming public to take advantage of parallelism. There are many other concurrent and parallel frameworks and models, such as MPI[5], and many programming languages, such as Java, have

a built-in threading library, which is usually simpler to use than native, low-level threads. Even some languages, such as Erlang, are explicitly designed for concurrent and parallel programming.

For many years the computational *growth model* consisted of simply adding more resources to increase the potential degree of parallelism, as the performance improvements of individual processors stagnated. In last years, the parallel growth model also started to show signs of stagnation, but the demand of computational resources and the size of the problems to solve are still increasing. In fact, the sizes of the datasets experimented an exponential growth with the boom of paradigms such as Big Data or Deep Learning, in which it is not strange to deal with datasets that, simply put, do not fit in a single machine.

So, the next step was obvious: make a program run in different machines simultaneously. This implied many new challenges, such as having many different memories and therefore many different versions of the same object allocated in machines with possibly different architectures. Some other problems appeared, as the impossibility to know the exact order in which events happened in a single program, as different machines have different clocks [7], making the task of debugging this kind of programs even harder. The previous problems from parallel programming were inherited or got even worse. Data dependencies, race conditions, and the importance of granularity are still there. Some algorithms are much harder to implement in a distributed fashion due to not having the whole input available at the same time, but only pieces or chunks of it. Also, software developers are now forced to take into account an extra level of parallelism when designing their applications if they want to take advantage of all the available resources.

As happened with parallel programming, the demands of the industry encouraged the development of frameworks, programming models and file systems aimed to make the development of distributed applications easier. These frameworks and programming models usually abstract the user from things such as explicit computational resource management, synchronization between processes, logging, and network and object handling.

One of the most used frameworks is Spark [16] alongside with HDF5 [13]. Usually, a Spark application consists of the repeated application of a set of fixed patterns, such as map-reduce, while making the *hard steps* transparent to the user. Although this is a very powerful tool, it still requires a strong programming knowledge, as it may not be trivial to translate any idea or an already existing application to this set of patterns. It is for this reason that task-based programming models offer a good alternative. A task-based programming model lets the user to select which parts of the code will be tasks, allowing him to take already existing applications and make them run in a distributed environment with minimal effort.

1.2 Objectives

This project focuses on improving the task-based programming model COMP Superscalar (and, from now on, COMPSs) [2] by both adding features aimed to improve usability and performance by focusing on the object management stage. Object management implies a lot of different algorithms and computational problems.

Some of them are:

- Have a quick way to uniquely identify user objects
- Translate user objects into something transferable between different machines with, possibly, different architectures and/or versions of some of its software. Try to cover as much objects as possible
- Maintain consistency between versions of the objects among all the computational resources, keep track of its locations, and use this information to exploit data locality when scheduling tasks in task-based programming models
- Transfer objects between computational resources. Do it as smart as possible to minimize redundant data transfers, bottlenecks, and so on

In this project we will explore some features and paradigms aimed to improve any of these aspects. We will also implement applications and algorithms to test them. Our specific objectives are:

- **Fix PyCOMPSs task Overheads** The current PyCOMPSs version (tagged as 2.4) seems to do some kind of $\mathcal{O}(n)$ computation before sending the nth emitted task to the COMPSs Runtime. Our objective is to detect, analyze and fix the source of this overhead.
- **Collection Parameters** Make COMPSs support arrays of parameters. Compute dependencies between the elements of collections and collections themselves, and exploit the fact that these parameters *go together* to reduce object management overheads.
- **Combine Storage with COMPSs** Explore and provide alternatives to GPFS to avoid COMPSs dealing with the file system.
- **Add threading to PyCOMPSs IO operations** Although Python has a Global Interpreter Lock [1] (GIL) it can still benefit from rearranging non-blocking IO operations. Our objective is to experiment with parallel object (de)serialization.
- **Implement distributed applications as tests** This step consists of implementing applications with specific workflows aimed to test the new features or improvements to the COMPSs programming model. It also contributes to have a bigger repository of use cases and applications.

1.3 Document structure

The rest of the document is organized as follows. Section 3 enumerates the current technologies, frameworks and programming models related to this project and, in general, with HPC and distributed systems. Section 2 enumerates and organizes the tasks to do in this project and the employed methodology. The rest of the sections are devoted to the development of the features, executions and experiments of the project itself. Some source codes can be found as appendices, or as inline comments if they are short enough, but other are too long to be attached in this document, as the COMPSs code itself. In these cases, a link or a reference to a repository will be provided when necessary.

[1] https://wiki.python.org/moin/GlobalInterpreterLock

Chapter 2

Tasks and time planning

This project intends to implement various features aimed to improve the object management stage of the COMPSs programming model. These features may depend on some previous features or they may be totally independent. This project also requires some additional tasks, as writing this document. This section intends to organize the objectives mentioned in section 1.2. Some new objectives and tasks are added, such as learning the COMPSs internals. All tasks, with their dependencies, can be found in figure 2.1.

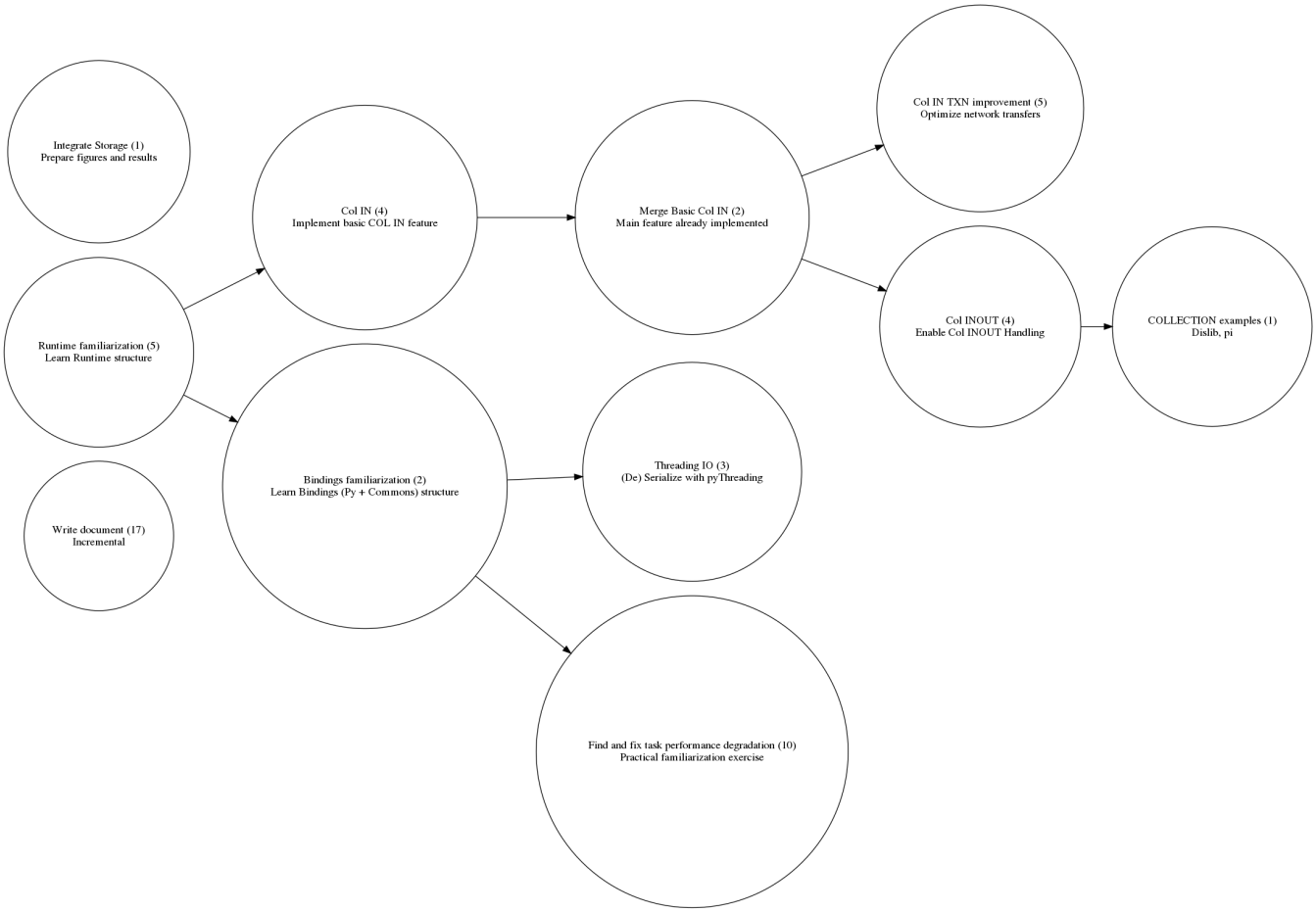

FIGURE 2.1: A dependency graph representation of the different tasks and the dependencies between them. The numbers between parentheses denote the estimated number of needed weeks to do some task

2.1 Project Tasks

A more precise explanation of these tasks (and shortcuts to the corresponding sections) can be found below. It is recommended to read the following sections in order to fully understand all the terms and explanations that will appear in this document.

- **Runtime familiarization** Some features require a deep knowledge of the COMPSs Runtime. COMPSs is written in Java, so this task will mainly consist of learning the class hierarchy and modules of the Runtime, how to build and deploy it, and how to fix and add features to it. This task is explained and developed mainly in section 3.1.3.
- **Bindings familiarization** COMPSs has two bindings that allow the user to write applications for both C/C++ and Python. We will mainly focus on the Python (PyCOMPSs) part. This task will consist of learning the different Python modules, how user code is decorated from there, and how this binding communicates with the COMPSs Runtime via C++ Python extensions [1] and the JNI library [2]. The development of this task can be found in section 3.1.4.
- **Collection IN** This feature will allow the user to deal with multiple COMPSs input parameters at once if he puts them in some container. This task and all the others that have something to do with it is developed and explained in section 5.
- **Merge Collection IN** This task consists of integrating the changes made in COMPSs to support collections into the main branch of the software while guaranteeing that this integration does not affect the stability and performance of the software. This section includes to implement some kind of unit test, and to include it in a continuous integration environment. COMPSs has many concurrent developers attacking many sections of the software at the same time, and the software has many lines of source code, so this task may no be as trivial as it seems.
- **Collection INOUT** Same as collection IN, but allowing inout objects as the content of a collection. The development of this feature can be found in section 5.2.
- **Collection Examples** Improve some existing applications with the COLLECTION feature. These examples can be found in section 5.3.
- **Collection TXN improvement** The fact that two objects belong to the same collection can be used to our advantage to implement some improvements in how this data is transferred to the destination node. For example, they could be transferred together and then split in the destination. This idea may result in better performance (less simultaneous connections, less bandwith bottleneck) or may make things worse (less parallelism when transferring data between nodes). This last section must be considered as an extra, as the difficulty and the required time to implement it is probably out of the scope and resources of this project, and it is more than likely that it will remain as a possible future work line.

[1] https://docs.python.org/3/extending/building.html
[2] https://es.wikipedia.org/wiki/Java_Native_Interface

- **Combine Storage with PyCOMPSs** One of our approaches towards the improvement of object management is to partially delegate it to some *dedicated* storage backend. This includes the development of some PyCOMPSs API that allows the user to use this backend and to integrate it to the *intelligence* of the COMPSs Runtime. All the work related with this task can be found in section 6.

- **Threading IO in PyCOMPSs** Most Python implementations have a Global Interpreter Lock (GIL) that prevent parallelism with Python threads. This does not mean that some speedup can be obtained if IO operations are done with Python threads, and that Python programs are necessary sequential or, at best, concurrent (for example, the Numpy library has many linear algebra operations implemented with OpenMP). This task explores if it is worthy to parallelize IO operations with Python threads.

- **Find and fix task performance degradation** This task will be useful to check if we have actually achieved a good enough understanding of the COMPSs programming model. It consists of dealing with a performance decay in a user's application. By dealing we mean to identify the problem, its sources, think about a solution and implement it (if applies). This task is developed in section 4. This task is intended to serve as an extension of the explanation on how COMPSs works.

As a remark, we consider the development of applications or use cases to be implicit in any task, so there is no "develop practical application" task in the graph.

2.2 Methodology

This project will be developed in a constant-feedback, results-driven model. That is, the outcome of some implementation may make our initial planning change, as these implementations may reveal more interesting lines, or heavy limitations to the current ones.

One of the key aspects of this kind of work is to keep results reachable and easy to reproduce. For this purpose all the contents regarding to this project can be found in two git repositories:

- `https://github.com/srgrr/TFM` The repository with this document, and all the applications, code snippets and figures contained in it

- `https://github.com/bsc-wdc/compss` The public mirror of the COMPSs programming model. All executions and applications will contain a reference to the exact commit or tag we used

This methodology allows our reviewers to easily reproduce all the experiments and to refer to some pieces of source code mentioned here.

All tasks are tracked and annotated in a Trello board. Trello [3] is an online platform that emulates the classical board with post-its on it. A screenshot depicting the Trello board can be found in figure 2.2.

[3] https://trello.com/

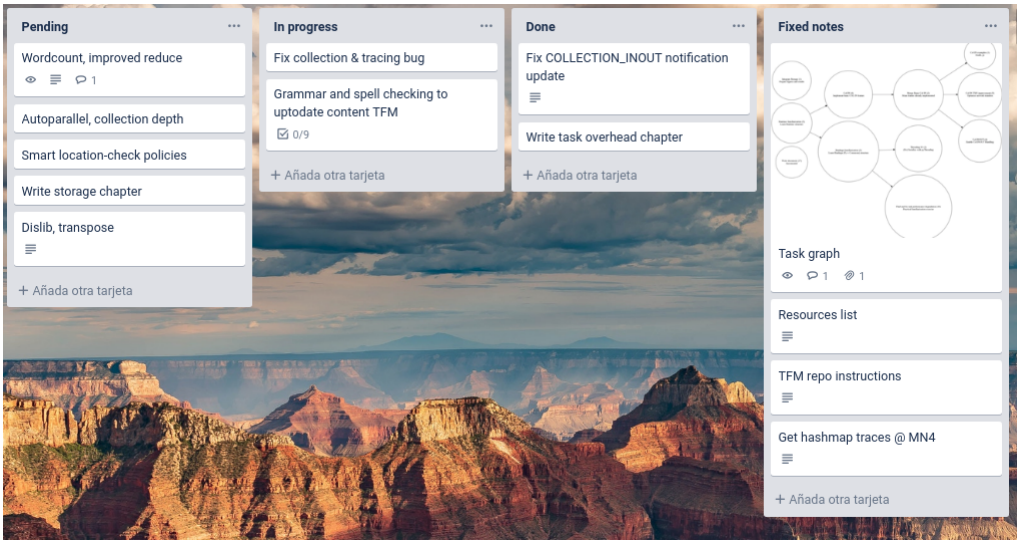

FIGURE 2.2: A screenshot of the Trello board. Tasks are divided in Pending (tasks we want to do but we are not currently doing), In Progress (tasks we want to do and we are currently doing), and Done (tasks we already did but we want to comment them with our supervisors or other members of our team). We also have some fixed notes with links to useful resources, rules of the project, the task graph, and so on

Chapter 3

Background

3.1 COMPSs

COMPSs is a framework aimed to ease the development of applications for distributed infrastructures [2] [9]. A COMPSs application is typically a normal application with some special annotations and a few extra function calls in its code that transform a sequential code into a program that can run in a distributed environment.

COMPSs applications can be written in Java, C/C++, and in Python (both 2 and 3). The Python framework is called PyCOMPSs [14]. All the examples and real-world usages in this project will be developed in the PyCOMPSs framework and in the Python language. However, this does not mean that all the features discussed and developed in this project are only available for PyCOMPSs. In fact, given how COMPSs is designed, the implementation of a feature for PyCOMPSs usually implies to implicitly implement it for any of the programming languages that are supported by COMPSs.

The COMPSs framework also provides users and developers with some tools and data that helps to monitor and to debug the applications and COMPSs itself. From a user point of view, a graph of the workflow and traces of the execution of applications can be generated (figures 3.1 and 3.7). Traces are generated with a combination of Extrae, Paraver, and a custom implementation inside COMPSs itself which merges traces from different processes as a single one. From a developer point of view, many debug information, as logging messages and stack traces, is available if runnning COMPSs with debug flags or in case COMPSs crashes. We must mention that these logs are not always the answer or the solution. For example, if both the master and some worker complain in their respective logs about something some questions should be answered. Some questions, as knowing if some error is the cause or the consequence of some other error that happened elsewhere, are very hard to find and they may take a lot of time to be fixed. Some of these questions, as knowing which error happened first (assuming they are independent), are even harder to answer [7].

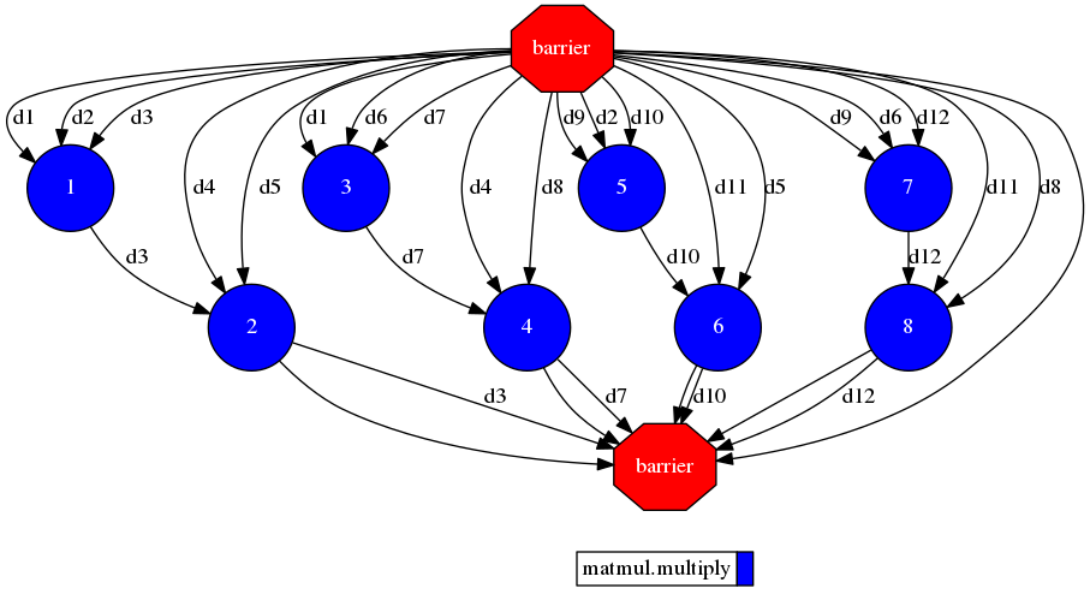

FIGURE 3.1: A dependency graph generated by a COMPSs application. Circular nodes are tasks, octogonal edges are syncpoints, and edges are dependencies between tasks and/or syncpoints caused by some data. The labels of the edges are the identifiers of the data that causes these dependencies.

3.1.1 A Full Example

This section intends to give the reader a more or less extensive insight on what writing a COMPSs application is. We think that this section may help to *materialize* concepts and will avoid to give this document an excessively abstract tone.

Lets suppose that we want to approximate the value of π. For this purpose we have thought on a simple, randomized algorithm:

1. Generate N random 2D points with coordinates between -1 and 1
2. Consider the set of points S within distance 1 or less to the origin
3. Assume that $\frac{|S|}{N} = \frac{\pi}{4}$

A more graphical explanation on why this works can be found in figure 3.2.

We know a little bit of Python, so we have decided to implement this program in it. Basically, our small application will consist of a `test_random_point` function that generates a random point and return 1 if this point lies inside our circle, and 0 otherwise. We will call this function N times, and consider the proportion $\frac{|S|}{N}$ to be equal to $\frac{\pi}{4}$.

```
def test_random_point():
  import numpy as np
  # p is a 1x2 vector with two numbers in [-1, 1]
  p = 2.0 * np.random.rand(2) - 1.0
  # count this point iff it is within one unit or less
  # within the origin
```

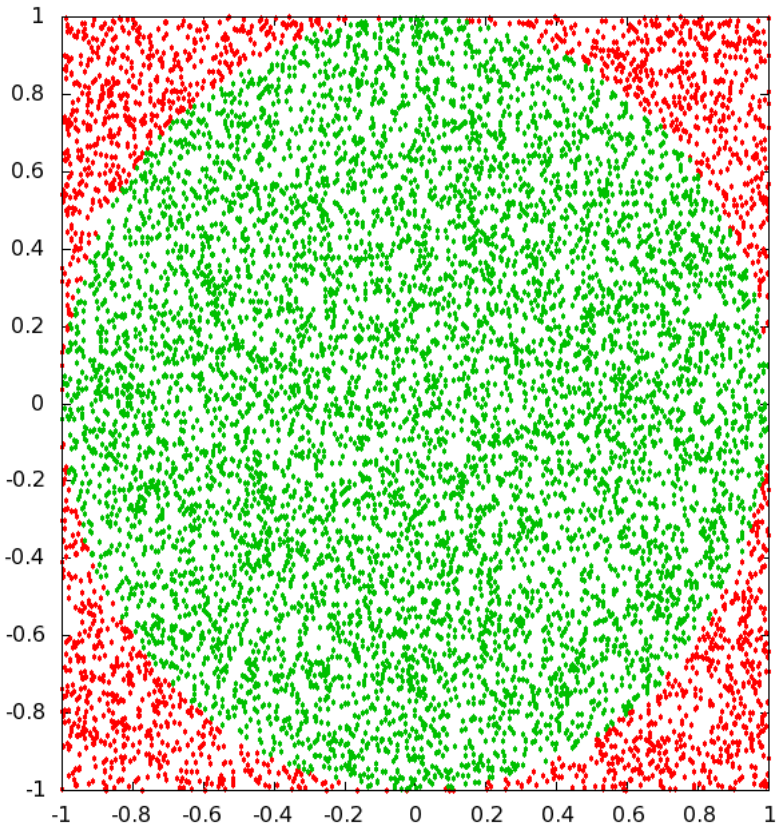

FIGURE 3.2: A graphical representation of the random experiment. The square has side length 2, so the circle has radius 1, and therefore area π. In ratio terms, $\frac{\pi}{4}$ of the points belong to the circle

```
  if np.linalg.norm(p) <= 1.0:
    return 1.0
  return 0.0

def main():
  import sys
  N = int(sys.argv[1])
  proportion = sum([test_random_point() / float(N) for _ in range(N)])
  # Print our pi approximation with up to 16 decimal places
  print("%.16f" % (4.0 * proportion))

if __name__ == "__main__":
  main()
```

This code can be straightforward *optimized* by transforming the `test_random_point` function into a COMPSs task and syncing the results in the main procedure.

```
from pycompss.api.task import task
from pycompss.api.parameter import *

@task(returns = 1)
def test_random_point():
  import numpy as np
```

```
  # p is a 1x2 vector with two numbers in [-1, 1]
  p = 2.0 * np.random.rand(2) - 1.0
  # count this point iff it is within one unit or less
  # within the origin
  if np.linalg.norm(p) <= 1.0:
    return 1.0
  return 0.0

def main():
  import sys
  N = int(sys.argv[1])
  # Note that test_random_point is now a PyCOMPSs task
  # This means that calling this task does not imply immediate results
  # So, a future object will be returned instead
  future_proportions = [test_random_point() for _ in range(N)]
  from pycompss.api.api import compss_barrier, compss_wait_on
  # We will wait for all tasks to end before adding these values to our
  # final accumulator
  compss_barrier()
  proportion = sum([compss_wait_on(x) / float(N) for x in future_proportions])
  # Print our pi approximation with up to 16 decimal places
  print("%.16f" % (4.0 * proportion))

if __name__ == "__main__":
  main()
```

Although this may be a good approach to *parallelize* this application we must note that we want to make it run in a distributed environment. The main difference we can appreciate is that a COMPSs task may run in a different machine than the master code, so some coordination between two processes in different machines and the transfer of potentially big amounts of data are necessary. In other words, the tradeoff between task granularity and performance is much more punishing in distributed computing than in single-machine parallel cases.

Another important thing to note is that a distributed application can still exploit lower level parallelism in each of its tasks. In our case, we can transform our `test_random_point` function into a `test_random_points` procedure that generates and tests various random points at the same time.

```
from pycompss.api.task import task
from pycompss.api.parameter import *

@task(returns = 1)
def test_random_points(M):
  import numpy as np
  # p is a Mx2 matrix with two numbers in [-1, 1]
  p = 2.0 * np.random.rand(M, 2) - 1.0
  # count these points iff they are within one unit or less
  # within the origin
  return np.sum(np.linalg.norm(p, axis = 1) <= 1.0) / float(M)

def main():
```

```
  import sys
  N = int(sys.argv[1])
  M = int(sys.argv[2])
  # Note that test_random_point is now a PyCOMPSs task
  # This means that calling this task does not imply immediate results
  # So, a future object will be returned instead
  future_proportions = [test_random_points(M) for _ in range(N)]
  from pycompss.api.api import compss_barrier, compss_wait_on
  # We will wait for all tasks to end before adding these values to our
  # final accumulator
  compss_barrier()
  proportion = sum([compss_wait_on(x) / float(N) for x in future_proportions])
  # Print our pi approximation with up to 16 decimal places
  print("%.16f" % (4.0 * proportion))

if __name__ == "__main__":
  main()
```

This last approach is what we consider a well *COMPSsfied* application: it has a reasonable task count and granularity, and it exploits various levels of parallelism at the same time. This application also delegates most of the work to `numpy` procedures, which are mainly written in C++ and OpenMP. This aspect is also important in PyCOMPSs, as Python is, by nature, a very slow programming language and it should be only used as an orchestrator.

COMPSs is mainly designed to run in HPC environments. Most HPC machines integrate some sort of queue system to manage its resources among all the demanding users. Our previous example can be run as a job in a queue system with the following command:

```
#!/bin/bash

num_nodes=$1
num_experiments=$2
points_per_experiments=$3

enqueue_compss \
  --qos=debug \
  --num_nodes=$1 \
  --worker_working_dir=scratch \
  pycompss_vectorized.py $2 $3
```

The `enqueue_compss` command refers to a generic queueing script (see section 3.3) which translates our request to enqueue this COMPSs job to a specific queue system. Some of the most common parameters of a COMPSs job can be found in table 3.1.

3.1.2 COMPSs Components

COMPSs is designed, developed, and deployed in a modular way. This has some advantages:

Argument name	Description
`exec_time`	Job time limit
`num_nodes`	Number of computing nodes
`cpus_per_node`	Number of cores per computing node
`constraints`	Additional constraints (e.g: highmem nodes)

TABLE 3.1: Some example configuration parameters of the queue system. These parameters are usually passed as flags to the enqueue_compss script.

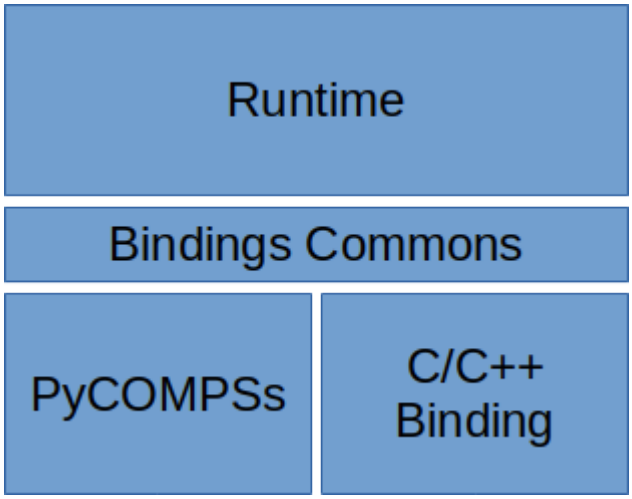

FIGURE 3.3: Overview of the main COMPSs components.

- Easier isolation of features
- Partial COMPSs installations are possible (e.g: install COMPSs without PyCOMPSs)
- Components can be individually replaced, leading to faster deployments

An overview of the main COMPSs components can be found in figure 3.3. These components are also modularized, as seen in figures 3.5 and 3.4.

This design choice also brings some unwanted problems. The main issue is isolation and concentration of knowledge of some parts in some developers, which leads to unnecessary code replication, lack of coherence of design and implementation choices between different modules, partial feature implementations (e.g: a feature that is only available in PyCOMPSs because it was developed by someone who did not know how to implement it in the Java runtime), and many other things. All these issues will be adressed and referred to in this document, as they appear and play an important role in our own design choices and implementations.

3.1.3 Runtime Structure

The COMPSs Runtime is the *brain* of the programming model. It receives the generated task parameters, computes dependencies between them, decides how to distribute the workload of the tasks among the available resources, and so on. As we

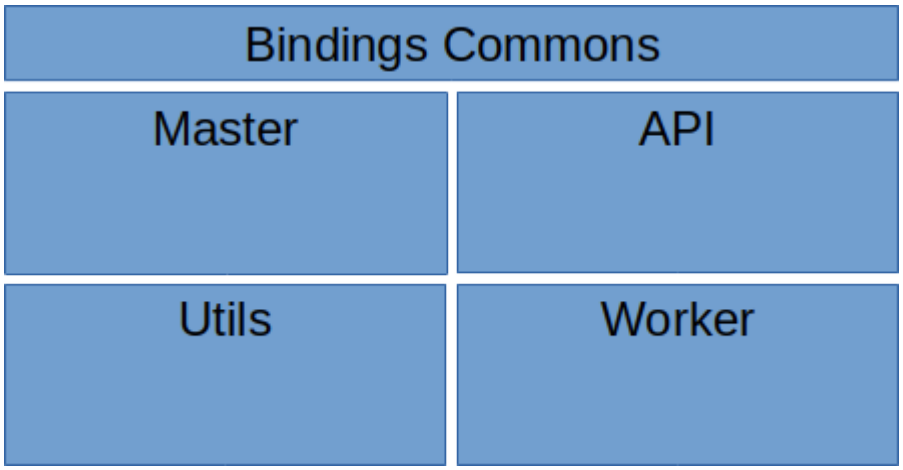

FIGURE 3.4: Overview of the main PyCOMPSs components.

can see in figure 3.5, the COMPSs Runtime is divided in five layers. The roles of these layers are:

- **Engine** Receive a task from the bindings/loader, process its parameters, register them if necessary, compute dependencies between tasks. Keep track of this task graph, and of tasks with in-degree zero in this graph. Send executable tasks to the scheduler.
- **Scheduler** Receive a executable task from the COMPSs Engine. Keep track of the available resources, the locations of all data units, the current load of these resources, and decide a location to execute this task according to these parameters
- **Adaptors** Provide an intermediate layer between the communications library and other layers. Responsible of sending and receiving requests between the computing nodes. A request can be about some data unit or about executing a task in some resource.

We have omitted the Resources and Communication Library because they are out of the scope of this project. The Resources layer will not affect any of our work, and the Communication Library needs no modifications, as it is a very low level library, and we can construct any of the new communication features we need by combining the already existing primitives offered by it.

3.1.4 PyCOMPSs Structure

PyCOMPSs can be summarized as a Python Binding for COMPSs. It gives the user a way to annotate his Python code, and it internally transforms and forwards all the derived task creation requests and data to the COMPSs Runtime. Its role can be summarized as follows:

1. Execute the user code, both the master and the worker part
2. Implement code annotations, such as `@task`, `@binary`, etc
3. Implement wrappers to flow control mechanisms, such as `compss_wait_on`, `compss_barrier`, etc

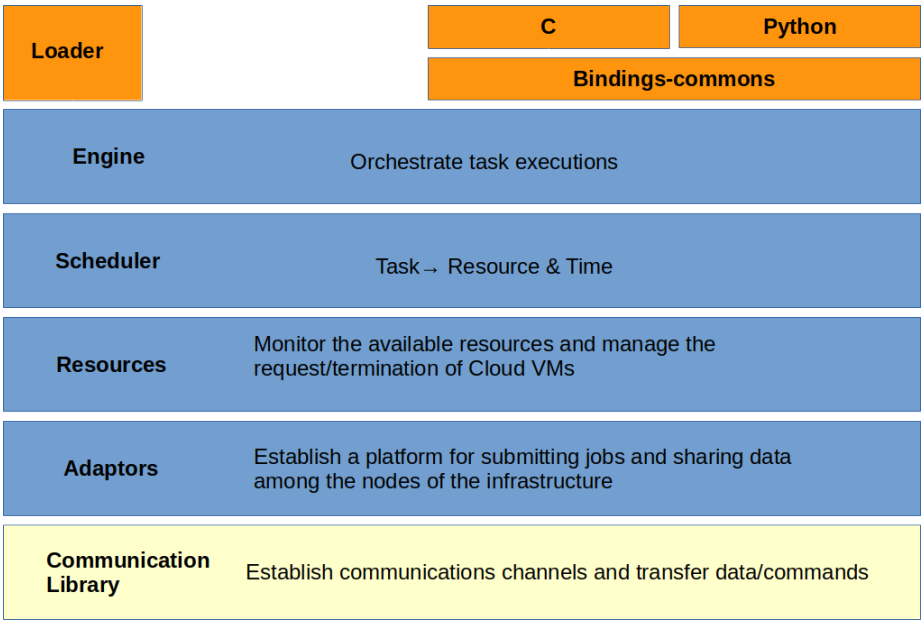

FIGURE 3.5: Overview of the main Runtime components.

4. Transform the user data into something easy to transport between different machines

The binding implementation can be divided into two big parts: master and worker. The master executes the user's code, captures tasks calls, forwards the data to the COMPSs Runtime, offers future objects to the user and the possibility to wait for some object to be available (synchronization points). The worker is just a Python process constantly listening to a pipe and obeying the orders it receives from this pipe. These orders can consist of executing a task or to end the process. When executing a task, the worker must deserialize the parameters, which are usually stored as files in the filesystem, fetch the function, reconstruct its signature, rearrange the parameters according to it, execute the user's function, capture the results (or the modification of the parameters if they were *INOUTs*), serialize and store them in the filesystem, and communicate success or failure to the COMPSs Runtime.

PyCOMPSs makes a strong emphasis on usability. This means that a lot of effort is put on minimizing the necessary changes to make a sequential program run in PyCOMPSs. We also try to take advantage of any new usability feature to try to introduce improvements in the programming model performance. This is not always possible and, in fact, usability implies a tradeoff with performance.

3.1.5 Usability vs Performance

COMPSs has two goals: to give the not-so-expert user an easy way to make their sequential applications run in distributed environments, and to do it as efficiently as possible. Many improvements in the COMPSs framework are aimed to improve only one of these two aspects. For example, any improvement in the communication library may improve the performance of the user application, but the user will still face the same usability limitations when using COMPSs. Adding an automatic return completion, to handle the case when the user forgot to annotate the return value of some task, may save the user a lot of debugging time, but it will have no

impact in the performance of the user application.

The COMPSs software is developed and mantained by a research team in a research center, so it may be natural to think that most of the efforts and improvements are aimed to test and develop methods, models, and algorithms that improve performance, memory usage, minimize network transfers and so on. However, COMPSs is also used by other research teams as a *tool* for their own purposes. Some of these teams intend to run exotic, old, complicated applications in distributed environments. Also, these teams are usually composed of researchers from different fields than computer science, so a lack of knowledge in parallel and distributed applications should be expected. The user-oriented features intend to help these research teams, and to make their life easier in the very complicated world of distributed computing. These two big forces (being a research team and having *clients*) act as the main source of ideas and features in the COMPSs environments, and they are not always acting towards the same direction.

This project tries to bring something that improves COMPSs in these two directions: give something to the user that makes his life easier while making COMPSs more efficient. For example, we do not intend to limit ourselves to give the user a way to pack some parameters in a collection. We see this feature as an opportunity to give COMPSs additional intelligence that may help to improve the performance of the framework. The same applies with the storage interface. Our goal is twofold: to give the user a way to make his or her COMPSs application run with other storage systems and to take advantage of these systems in terms of performance.

3.2 GPFS

GPFS [12] is a distributed file system developed by IBM. It gives a perceived behaviour of a regular POSIX file system, while guaranteeing consistency between different computational resources, and a correct parallel access to its files. Under this model, any node has access any file at any location. As we can see in figure 3.6 a node accesses data through a switching fabric. A switching fabric is a kind network topology in which any two nodes connect between each other through a series of switches. This topology allows a more efficient communication between nodes than other topologies such as broadcast networks. GPFS is available at the Mare Nostrum

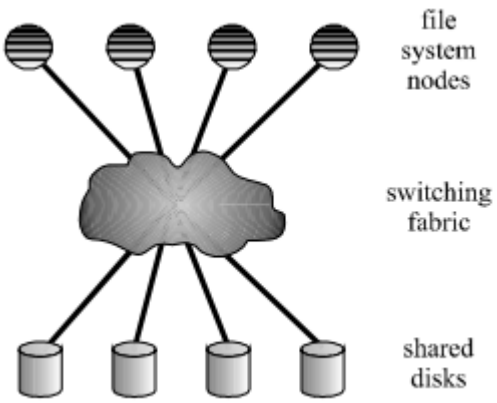

FIGURE 3.6: GPFS Shared disk environment. Figure 1 from [12]

IV supercomputer, and COMPSs takes advantage of it by delegating the file system many tasks such as file transfers, consistency across computational resources, and so on. It also makes task scheduling easier, as the data locality factor can be ignored by the COMPSs scheduler, focusing only on load balancing. Being more specific, if COMPSs runs under a GPFS file system it will consider that any piece of data is available anywhere, instead of explicitly keeping track of its locations.

3.3 Queue Systems - SLURM and LSF

Most supercomputers have many concurrent users. All of these users want to use some of the resources of the supercomputer, and usually in a selfish manner. This situation creates a lot of conflicts between users, and even some unethical behaviors such as some user killing the processes of other users. Also, many benchmarks and experiments require no noise introduced by concurrent, unrelated processes running in the same machine, so resource exclusivity must be guaranteed in these cases.

The most common solution to the two aforementioned problems is to divide the different nodes of a supercomputer into login nodes and computing nodes. When a user opens a session in some supercomputer he will *land* into some login node. Computing nodes are unreachable or even not visible by regular users, and the only way to have access to them is to ask the system for resources and wait until the system lends them to the user. The most common implementation of this resource assignment mechanism is a queue system. A queue system processes all the requests from the users, gives them a priority as a function of various parameters and lends them the requested resources according to these priorities, as a process scheduler does with processes in an operative system.

Two of the most common queue systems are LSF [17] and SLURM [15]. All the experiments of this project will be done in the Mare Nostrum 4 supercomputer, which uses SLURM.

Although SLURM has its own micro-language and instructions, such as `srun`, and submissions scripts, most of the experiments done in this project will not need them, as we will have generic queueing scripts available to us. A generic queueing script is a script capable to work with various queue systems to generate the corresponding specific queueing scripts. In our case, our script will translate our orders into a bunch of `srun` commands and similar.

3.4 Extrae and Paraver

Extrae [1] and Paraver [6] are two profiling tools developed at the BSC. Extrae is an instrumentation software to trace programs. A program instrumented with Extrae usually emits events. An event usually consists of an identifier or label, and a timestamp indicating the exact moment of its emission (according to the clock of the machine that executed the program). These events can be later visualized with Paraver as what is known as a *trace*. An example of a trace can be found in figure 3.7.

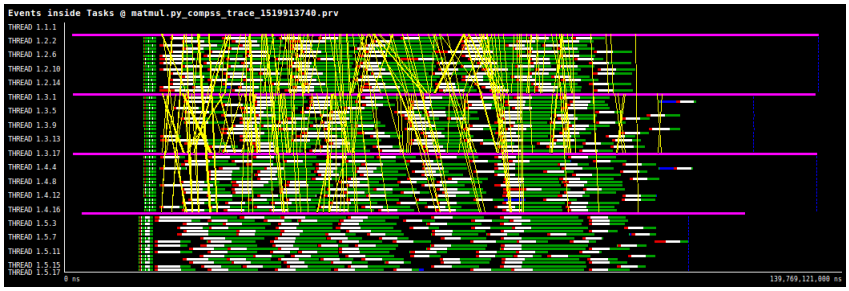

FIGURE 3.7: An example of a trace. Each row represents a thread (or a process depending on the case), colored segments are different tasks executed by the thread, and yellow lines are network transfers between different computing nodes. Time is represented as the horizontal axis, from left to right.

Traces can be visualized in many different ways, depending on the needs of the user. This project will only use traces like the one from figure 3.7, and any relevant information about traces will be mentioned in the caption of the corresponding figure.

In this project no explicit use of Extrae will appear, as we will work with a framework which already has Extrae instrumentation.

3.5 Hecuba and DataClay

Hecuba [1] is a distributed non-relational database. It implements a runtime which coordinates various independent databases to make them work as a cluster, and provides a set of functions to allow the user make queries to this cluster.

[1] https://tools.bsc.es/doc/pdf/extrae.pdf

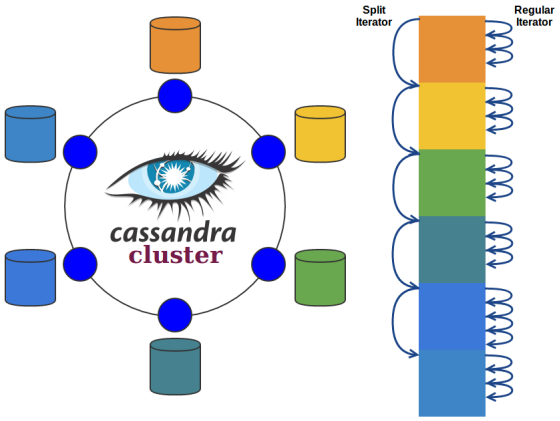

FIGURE 3.8: Graphic description of a Hecuba cluster. Source: [1]

In a Hecuba cluster each instance is able to accept and to compute queries, allowing the user to make all his queries to the nearest node without needing to worry about the underlying topology of the cluster.

DataClay [10] is another distributed database implementation. In a DataClay cluster, each individual instance is a Postgres SQL database to which the user has no direct access. Instead, DataClay implements a series of methods and functions to interact with the whole cluster.

Both storage implementations follow the Storage Object model. A Storage Object is an instance of an OOP language class which has some of its attributes marked as class fields. A class field is automatically accessed and synchronized with the storage backend. This paradigm allows the user to avoid any kind of direct query and to have any explicit database knowledge to interact with the storage backend.

Hecuba supports data partitioning via the special *split* method. If a piece of data with key k is split and queried then an iterator is returned instead. An iterator is just a key and a reference to another iterator, like a linked list. As we can see in figure 3.8, this means that a piece of data can be split among all the nodes of a cluster, and thus improving the balancing.

Chapter 4

Improving object identification in PyCOMPSs

This section describes a performance improvement in the COMPSs programming model. It also gives a practical example of how important is to properly manage objects in distributed programming models and to show how complex finding a bug in this kind of software can be.

4.1 Problem description

A COMPSs user reported via mailing list that his application showed a gradual performance degradation over time. A source code that reproduces the *guilty* workflow is the following:

```
from pycompss.api.task import task
from pycompss.api.api import compss_barrier

NUM_ITERATIONS = 10
NUM_OBJECTS = 1000

@task(returns = 1)
def f(x):
  return x

def main():
  for i in range(NUM_ITERATIONS):
    l = []
    for j in range(NUM_OBJECTS):
      l.append(f(object()))
    compss_barrier()

if __name__ == "__main__":
  main()
```

The user was able to detect this performance degradation thanks to the tracing tools. Two traces showing this issue can be found in figures 4.1 and 4.2.

4.2 Analysing and narrowing down the problem

The trace from figure 4.2 shows us that there is something wrong about how COMPSs manages tasks. Note that this fact only gives us a very rough hint on where to start

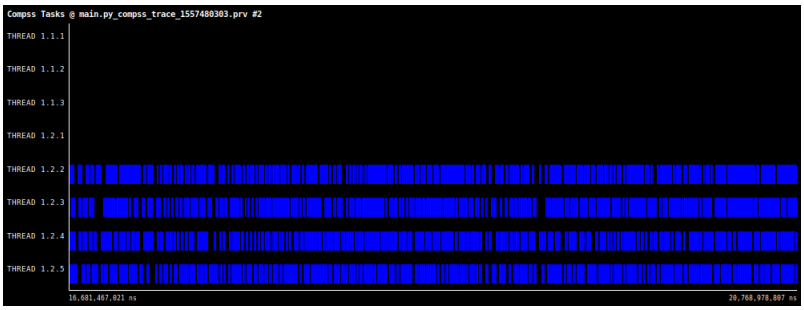

FIGURE 4.1: A trace showing the first tasks of an execution. Each row represents a thread, blue represents a task being executed and black that either the thread is waiting for the next task or it is doing something else

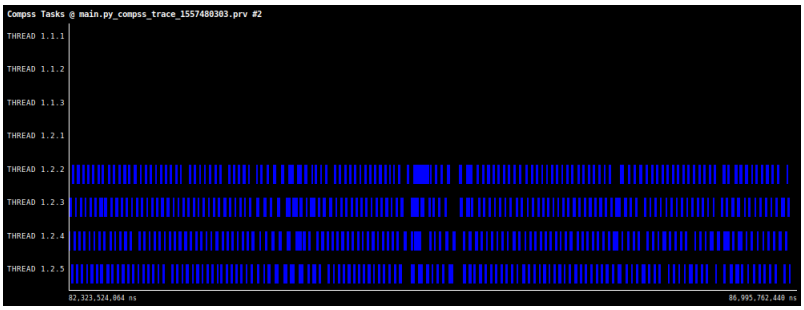

FIGURE 4.2: A trace showing the last tasks of an execution. The meaning of the trace is the same as in figure 4.1, but now the spacing between tasks has increased a lot

looking for, but there are still many possible candidates and places to look at. Some of these places are:

- PyCOMPSs `@task` decorator
- PyCOMPSs object serialization
- PyCOMPSs-to-COMPSs parameter forwarding
- COMPSs Runtime task registering
- COMPSs Runtime dependency computation
- COMPSs Runtime data transfer
- COMPSs Worker task reception
- ...

The list may contain 30 additional items before the last step(s) *user's task is executed, results are serialized and the master gets a notification about it*. Some sections are arguably skippable, as our knowledge and experience tells us they cannot have anything to do with our issue, but many others must be reviewed. The natural way to proceed is to simply follow the PyCOMPSs and COMPSs source code in the *natural order*. That is, start from the user's source code, detect a task, go to the `@task` decorator, and so on.

Fortunately for us, the problem was on a quite early stage: the handling of the object identifiers in the Python binding.

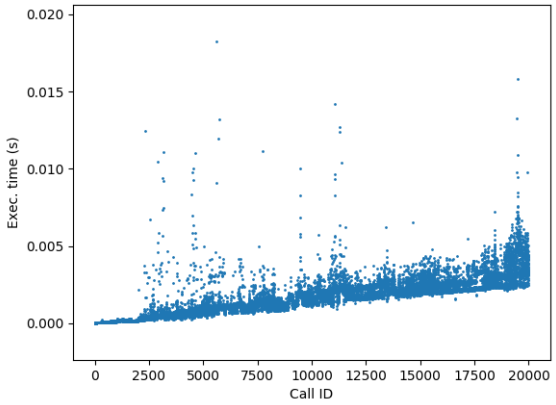

FIGURE 4.3: Time required to complete a function call. Calls are arranged chronologically. Although there is a lot of noise, a linear behavior can be observed

4.3 Object identification and mapping in PyCOMPSs

Our first (and last) bet was on the `get_object_id` function from the PyCOMPSs source code. Let's take a look at this function:

```
def get_object_id(obj, assign_new_key = False, force_insertion = False):
    global _runtime_id
    # Force_insertion implies assign_new_key
    assert not force_insertion or assign_new_key
    for identifier in _id2obj:
        if _id2obj[identifier] is obj:
            if force_insertion:
        return new_id
    return None
```

This function iterates over potentially all of the tracked objects just to get the identifier of some object (or to assign it one). This is done this way because an object needs to be hashable for being used as a key in a dictionary. Hashable usually means immutable, and no user can guarantee us that his objects will fulfill these properties. In fact, the programming model supports INOUT objects, which are, by nature, mutable objects. Some examples of mutable Python objects are `[1, 2, 3, "hello"]`, `object()` and `numpy.random.rand(5)`.

If the Python binding is tracking n objects then this function may iterate $\mathcal{O}(n)$ times just to retrieve (or to give) a single identifier. It is not hard to see that an application with n simultaneous PyCOMPSs objects will have to do $\mathcal{O}(n^2)$ iterations, which is consistent with what we observed at the traces.
Even if the previous analysis tells us that there is something very wrong with this function, we still need to make sure that this is the main source of our current problem. For that purpose, we have decided to time all the calls to this function. As we can see in figure 4.3, this function scales pretty poorly with the number of tracked object. All this evidence can be considered more than enough to start thinking about possible fixes and improvements. Our idea A possible fix may consist of using the `id` function. This function accepts an object as its unique argument and returns its

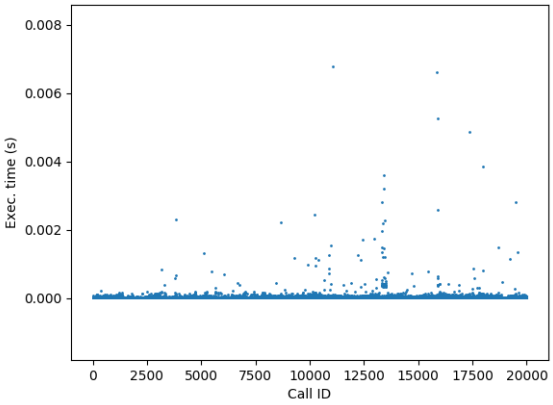

FIGURE 4.4: Time required to complete a function call after the fix. Call are arranged chronologically.

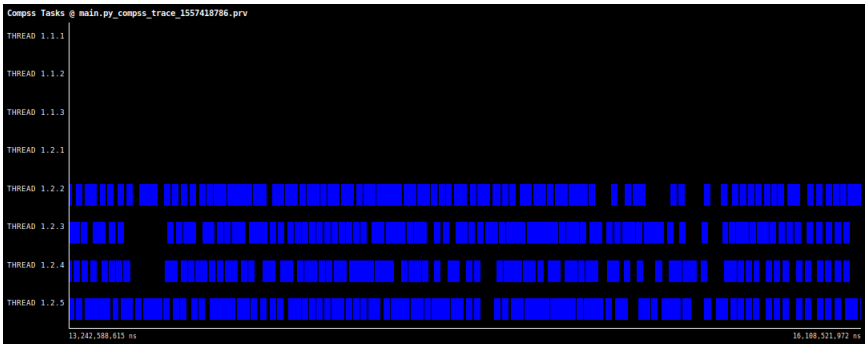

FIGURE 4.5: A trace showing the first tasks of an execution after the fix. Each row represents a thread, blue represents a task being executed and black that either the thread is waiting for the next task or it is doing something else

memory address. Note that this identifier is not entirely unique for a single object, as two objects that do not coexist may have the same memory address. However, this way to identify objects is enough for our use case. After applying this fix the `get_object_id` function behaves as seen in figure 4.4, which is the normal and expected behavior when accessing to a hashmap.

Additionally, figures 4.5 and 4.6 show us that the time between tasks is now constant.

The overall performance gain was enourmous. Although the final implementation consisted of a few lines of source code, it was arguably hard to find where this improvement should be made. We must note that COMPSs has, according to the `cloc` tool, around 300000 source lines of code (sloc).

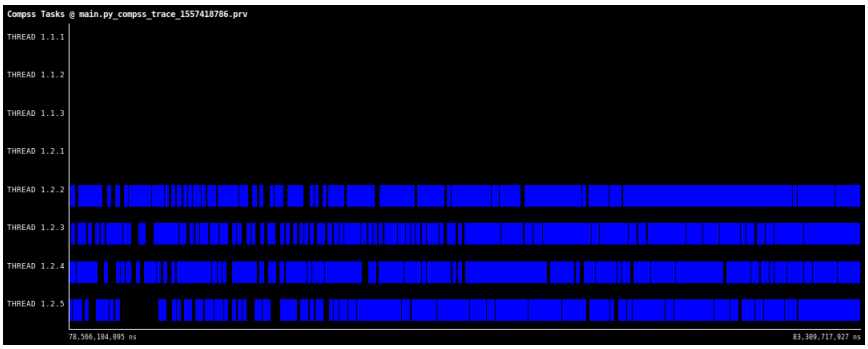

FIGURE 4.6: A trace showing the last tasks of an execution after the fix. As we can see, the gaps between tasks are quite similar to the ones from figure 4.5

Chapter 5

Collections in COMPSs

As we have seen in previous sections and examples, a COMPSs parameter is basically a regular user-code object, as a `numpy.ndarray`, with additional metadata to help the COMPSs Runtime to compute any dependency between tasks induced by this particular object, keep track of its locations, and so on.

A very common issue reported by COMPSs users is that the programming model is not able to detect dependencies induced by attributes or contents. Many examples are valid: an array `[object(), some_future_object]`, an instance of a class with some attribute that is a future object... or some object that has been used in a super-object. If the container is used as a COMPSs parameter, no process of the sub-object will ever happen, as the programming model won't know about it, so the user should expect tasks to receive outdated or future objects.

The ideal solution, a generic introspection algorithm, is very hard, if not impossible, to implement. Python is a dynamically typed language, some objects can be modified if iterated, many others have no easy way to list its internal attributes, circular references can happen... the list is almost endless. Another obstacle is object reconstruction. Let's consider the following code:

```
A = MyClass()
A.attribute = some_pycompss_task()
another_pycompss_task(A)
```

Ideally, we would like to detect the dependency induced by `A.attribute` with no synchronizations in the master, and then get the full object in the worker. This means that the programming should:

1. Detect the data dependency (introspection)
2. Ask for `A`, and `A.attribute`
3. Deserialize `A` and `A.attribute`, realize that one object is an attribute of the other, and add it

These steps require a heavy implementation with a noticeable performance impact. For example, a 2000×2000 `numpy.matrix` can make the programming model iterate through 4000000 elements unnecessarily.

It was decided that support for arrays should be implemented in the COMPSs Programming Model. Given that many COMPSs users find words like *array, hash map, reflection, inheritance* complicated and misleading it was decided to call this feature as *support for collections*, as collection is a word that seemed more understandable by non computer science researchers. This name also gives the opportunity to extend

this implementation to other iterable data structures such as sets, hash maps and so on.

This feature should cover these two cases:

```
L = [future_object_1, future_object_2, ...]
y = f(L) # Future objects should be synced and available

L = [object_1]
modify_object_1()
f(L) # object_1 should be updated and synced properly
```

In other words, collections should support both IN and INOUT objects.

This feature is especially interesting because its implementation serves two purposes: usability and performance. With no collections users are forced to use some *alternative tricks* such as functions with the signature `f(*args)` to deceive the programming model into believing that it is receiving multiple, distinct arguments instead of an array of them. As we will see later, this particular trick has its own problems and issues.

5.1 Collections as Input Parameters

The first step consists of enabling the COMPSs Programming Model to accept collections composed of input parameters. This step will also help us to identify all the COMPSs components that require some implementation and/or modification when dealing with this feature.

The easiest way to implement this feature is with recursion. Note that, in terms of dependencies, if some object x is contained in some collection C then any dependency that affects C must also affect x, and vice versa. Let's consider the following pseudo-code:

```
c1 = f() # c1 is a future object
C = [c1, ...] # C contains a future object and possibly more things
g(C)
```

There is a clear dependency between the tasks $g(C)$ and $f()$. The easiest way to implement this is to recursively iterate collection objects and to process each object as a single parameter. This approach offers some room to improve performance. For example, it is not necessary to transfer certain metadata of each single element of a collection, as it can be deduced or inherited from the global collection object. In this first case, it is not necessary to specify that the direction of all the elements $c_1, ..., c_n$ of some input collection C is `IN`, as it can be deduced from the fact that C is an input collection. The same applies with some other information such as locations, and so on.

Our chosen approach is something similar to what is called TDD (Test Driven Development) [1]: we wrote a PyCOMPSs app that uses collections as input parameters and now we want to make it work. The source code can be found in appendix A.

[1] https://en.wikipedia.org/wiki/Test-driven_development

As we mentioned in section 3.1.2, the design of the programming model forces the developer to go through many layers of the software just to implement a single feature. Any PyCOMPSs parameter will go through the pipeline shown in figure 5.1.

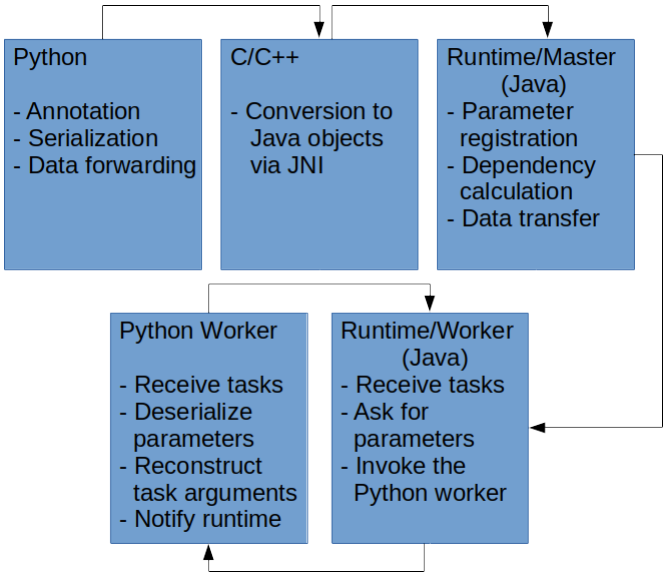

FIGURE 5.1: The journey of a Python parameter, from the user's function call until the task is finished in the worker

Our implementation can be generalized with this pattern:

```
def some_parameter_processing_function(p):
        if p.type == COLLECTION:
                for elem in p.content:
                        some_parameter_processing_function(elem)
        process_parameter(p)
```

This pattern allows us to implicitly define collections of collections. In fact, a `Depth` field can be defined when decorating a task. This field has a default value of 1, and it determines the allowed levels of recursion before considering any object a regular COMPSs parameter. For example, if `Depth = 2` and a $2 \times 2 \times 2$ matrix is passed as a parameter, COMPSs will interpret it as 2×2 collection.

Following the schema from figure 5.1, a collection parameter will be processed as follows:

- **Python** When a collection is detected it is recursively processed. This means that all of its objects are individually identified and serialized. However, we will only forward the collection parameter to the COMPSs Runtime. This parameter will contain an array of identifiers of its contents. This saves us some memory and computing time, as the contents of a collection will inherit many attributes from the collection parameter itself, such as direction, name and so on.
- **C/C++** This step is very straightforward as our only concern is to convert the Python String which contains all the identifiers of the contents to a Java Array.

- **Runtime/Master (Java)** Parameter registration and dependency calculation are done by recursively calling already existing code (see figure 5.2). However, data transfers are optimized. More specifically, only the collection parameter with the list of its content is marked as a transferrable unit here. This saves us many unnecessary connections between nodes.
- **Runtime/Worker (Java)** It receives a collection parameter and is responsible of asking the master for the corresponding resources. This approach allows us to ask for collection parameters lazily, as an eager approach may cause a huge bottleneck if many tasks are simultaneously launched. It also constructs the *collection file structure* from figure 5.3, which will be used later by the Python Worker.
- **Python Worker** When a collection parameter is received it will simply traverse the file hierarchy from figure 5.3 and reconstruct the collection, while checking that no file is desearialized twice by using the resource memoizer we have mentioned before.

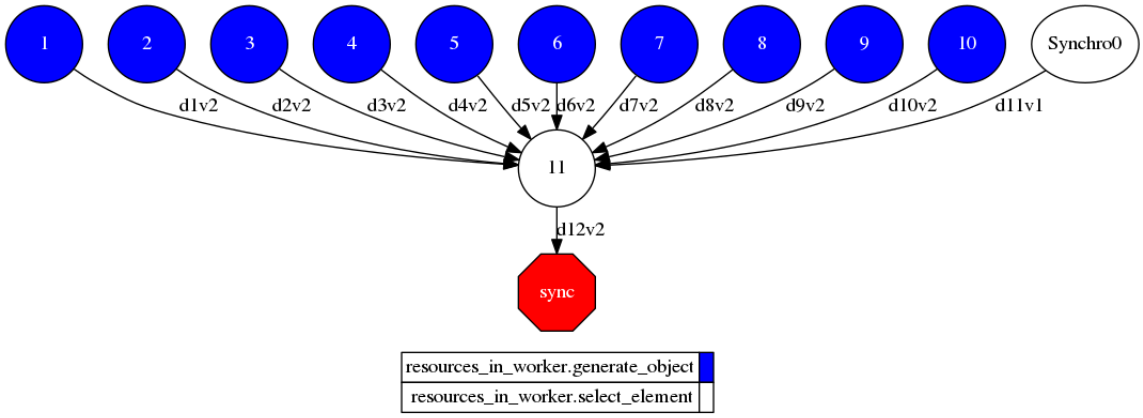

FIGURE 5.2: A dependency graph generated by a task with a COLLECTION_IN parameter with 10 elements. Nodes are tasks, an edge from A to B means that needs some data generated or modified by A. As we can see, COMPSs will see 11 dependencies, making no difference between collections and parameters.

We have also implemented a *serialization memoizer* to avoid serializing and deserializing parameters more than once. Let's consider the following code:

```
obj1, obj2 = object(), object()
A = [obj1, obj2]
f(obj1, A)
```

After the improvement from section 4 all COMPSs objects are identified by their memory address, so we are able to realize that `obj1` and `A[0]` are the same object. We have taken advantage of this knowledge to avoid doing extra serializations and deserializations by keeping track of the objects that are already processed. Some stranger cases such as `f([obj1, [obj1]])` also benefit from it.

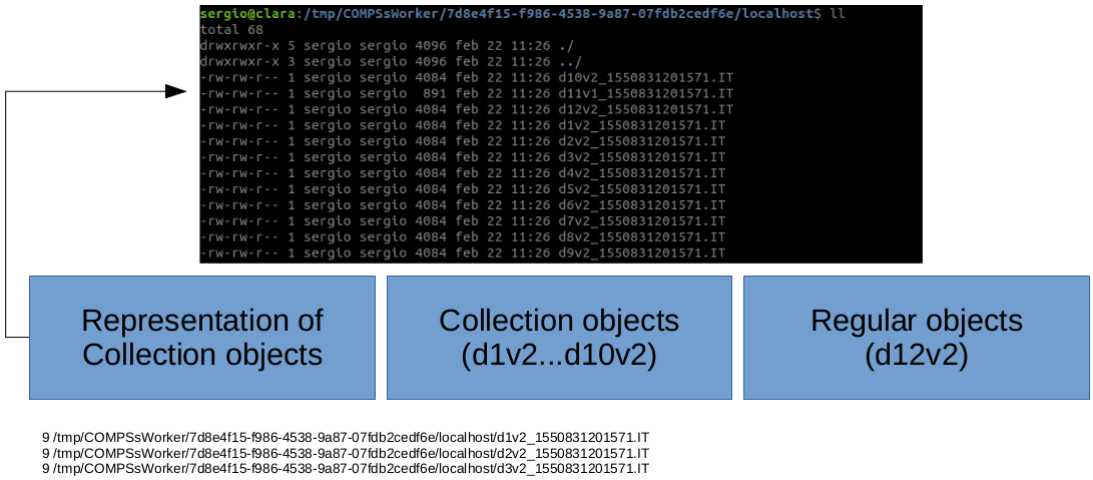

FIGURE 5.3: The in-worker representation of a collection. If a collection contains another collection, a reference to this file will appear, forming a DAG.

5.2 Collections as INOUT Parameters

Once we have collections of input parameters the next logical step consists of allowing COMPSs have collections of parameters that can be updated inside some task. Apart from adding some usable notation we needed to implement some extra steps such as notifying the runtime that the versions of the contents have been updated. The same optimization tricks apply here, as many metadata can be deduced from the parent data structure. Two example applications can be found in appendix B.

5.3 Practical Applications

5.3.1 Approximating cardinalities of huge sets

The Count-Distinct or Word Count Problem can be formulated as follows: given a sequence of elements $s_1, ..., s_n$ compute the amount of **distinct** elements in it. For example, for the sequence dog, cat, dog, bird, bird the answer is 3 (the distinct elements are bird, cat, and dog).

If the sequence is not too large this problem can be easily solved in expected linear time and space using hash tables, or $\mathcal{O}(nlogn)$ time and linear space using some data structures as Red-Black trees. However, this bound on space starts to become unacceptable when datasets are too large. In this section we will describe a probabilistic algorithm named as HyperLogLog [4].

This algorithm is very simple and yet very powerful. The core idea is the following: for each element s_i of our sequence, use a hash function $h : \{0,1\}^* \mapsto \{0,1\}^b$ to compute a value $h(s_i)$ and estimate the cardinality as 2^m, where m is the maximum number of leading zeros among all $h(s_i)$. We must note that if all $h(x)$ have the same probability $\frac{1}{2^b}$ then the probability for some value to have k leading zeros is 2^{-k}. This means that the expected number of observations that are needed to find a number with k leading zeros is 2^k. Given that having a single hash value is not precise enough but computing multiple hash functions is too expensive, what is done is the following:

1. Given a token t, compute $h(t)$
2. Take the first p bits and use them to refer to a position in an array consisting of 2^p elements $a_0, ..., a_{2^p-1}$
3. Update this position according to the other $b - p$ bits so it keeps the maximum amount of leading zeros seen so far
4. Once all tokens are processed output the harmonic mean of $2^{a_0}, ..., 2^{a_{2^p-1}}$ as the answer.

An interesting trivia fact is that if we need $\mathcal{O}(\log n)$ bits for our hash function to be able to count until n then we only need $\mathcal{O}(\log \log n)$ to store the number of leading zeros of some hash value. This is why HyperLogLog is called that way.

A very nice property of HyperLogLog is that two distinct runs on two different datasets can be merged if they have used the same parameters (hash function, b, and p) in such a way that it approximates the cardinality of the union of the two datasets. Given two arrays a and b, each corresponding to a run of HyperLogLog we can get a fictional run of HyperLogLog c that represents the union of both datasets by computing $c_i = \max(a_i, b_i)$ for all of the 2^p positions. This makes sense, as it produces the same result as running a single HyperLogLog on the concatenation of the two datasets. This property allows us to parallelize or to distribute this algorithm, giving us a potential performance boost. The source code we will use for our experiments can be found in appendix C.

Note that this application is a classical *map-reduce* workflow. Without collections we are forced to implement any reduce function as `reduce(f, *args)`. Each extra argument is passed as an input parameter via socket and pipe, implying a huge overhead. With collections only the collection object, and the list of identifiers, are transferred. The other properties of the contents, such as direction, locations and so on, are deduced or requested in the destination node.

The elimination of this overhead is noticeable even with a very small number of parameters. As we can see in figure 5.4, the collection feature reduces the overhead drastically. An important observation is that a PyCOMPSs task of the form `f(*args)` usually starts to show problems and crashes when more than 60 arguments are passed, as each argument represents a lot of metadata to be transferred via socket and pipe.

A comparison between the amount of meta-data generated and sent by the classical reduce implementation and by the collections version can be found in appendix D.

This improvement benefits many applications, as the map-reduce scheme is very common.

5.3.2 Usage of collections in other projects

The collection feature was very welcome by some other research groups and projects. Some of these research groups are:

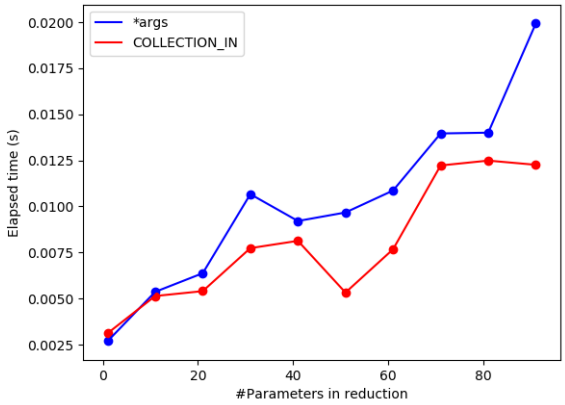

FIGURE 5.4: Execution time of the reduce functions with and without collections. Each point is the average of 5 executions. Although the samples are noisy, as they are small, a consistent improvement by the collection feature can be appreciated. The non-collections versions started to crash and to show strange behaviours around the 60 parameters

1. Dislib, a distributed computing library highly focused on machine learning on top of PyCOMPSs [2]

2. DDS (Distributed Data Set) [3], a PyCOMPSs implementation of Google Spark's RDDs. It also allows the user to write Spark-like codes for PyCOMPSs .

3. exaQute (**Exa**scale **Q**uantification of Uncertainties for **Te**chnology and Science Simulation)[4].

[2]https://www.bsc.es/research-and-development/software-and-apps/software-list/dislib/related-software

[3]https://github.com/bsc-wdc/compss/tree/stable/compss/programming_model/bindings/python/src/pycompss/dds

[4]http://exaqute.eu/

Chapter 6

Combining Storage Systems with COMPSs

All COMPSs objects are created by the user and managed by the Runtime. The data transferring software is a home-made library based on NIO [1]. Although this is usually a good enough solution for most use cases, there are three scenarios in which it may be a disadvantage to use this library:

1. The objects are the output of some previous application
2. The outputs of the COMPSs application are the input of some other application
3. The filesystem and/or the network presents huge bandwidth limitations

The idea of replacing files with databases or other alternative storage systems such as NVRAM helps with these three items when the available file system is not distributed like GPFS. Custom storage systems can help the user to avoid dealing with the file system and execute complex, distributed workflows using only the RAM of all the machines (and some additional swap space if necessary). Another advantage is that these storage systems are specialized, and therefore have better implementations than us on transferring objects between nodes, so delegating object management to these systems may give us a little performance boost. In this chapter we explore the possibility of relegating COMPSs as a simple task orchestrator by replacing the object management stage by specialized storage systems. As usually happens, the implementation of this feature can give some usability problems. We will also deal with this aspect of the implementation.

6.1 Defining a Storage API

The need of defining a storage API comes from the fact that some research groups are interested in mixing some of their tools with the COMPSs Programming Model. The two main tools or products from these research groups are DataClay [10] and Hecuba [1].

We decided that storage systems should implement all the same API. This would make implementations simpler and would allow the users to run the same application with different storage implementations by just changing the chosen API implementation in their configuration. The existence of an API should also make the user's life easier, so it must be developed taking into account the needs of the research groups and the usability of the final product. All API implementations must define a `StorageObject` class from which any object that interacts with the database must inherit. A list of these methods can be found in table 6.1.

[1] https://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html

Method name	Description
`make_persistent(id = None)`	Make the Storage Object persistent
`delete_persistent()`	Delete the Storage Object from the Storage System
`getID()`	Get the identifier of the Storage Object
`getByID(*objects)` (static)	Retrieve the objects with the given identifiers

TABLE 6.1: Public methods of the storage API

The `makePersistent` method is called by the user and stores the in-memory object in the database. This method is entirely dependent of the storage implementation. Some implementations just serialize the whole in-memory object and store the byte array in the database, while other implementations store classfields separately. The only constraint about this method is that it must guarantee that the object will be fully stored in the storage system before allowing the user code to continue, and that the in-memory version must be exactly equal to the storage version of the object.

There are also some additional internal methods that are optional to implement, as `getLocations`, which allows the COMPSs Runtime to know the locations of some piece of data. This `getLocations` is implicit when using the default file system of COMPSs, but needs to be defined when using a storage system in which COMPSs has little to no control on where the objects are stored.

COMPSs can be configured to let it know that the user is using some storage system. In that case, a path with a *implementation bundle* should be provided. COMPSs will add to the `CLASSPATH` and `PYTHONPATH` variables the necessary paths.

The target storage system can be an already running database, or can be created by COMPSs if a script to do that is specified. In the first case, a list with the addresses of the storage node(s) should be provided. Note that this list can refer to nodes that have no COMPSs worker instances running on it. This is ideal for COMPSs applications which take the outputs of some other applications as their inputs. In the second case a `storage_init.sh` script should be provided. This script receives the list of the COMPSs nodes as command line arguments and it is responsible to create the corresponding storage backend. It is also advisable to implement a `storage_stop.sh` script if the created storage system by COMPSs is not necessary after the COMPSs execution.

It must be noted that COMPSs is still responsible of dependency calculation and transferring metadata between nodes (e.g: the parameters of a task). This means that our collections feature still offers some of the advantages introduced by it, such as a metadata transfer optimizations.

6.2 A Practical Implementation: Redis

The first step towards validating this storage API consisted of providing a valid, functional implementation of it. For this purpose Redis was chosen.

Redis [2] is a simple Key-Value distributed storage database. Redis can be seen as a distributed hash map with $2^{14} = 16384$ slots. Each key is either chosen by the

[2] https://redis.io/

user or randomly assigned, and it determines the position of this object in the hash table. More precisely, given a key k, and a value v, v will be stored in the position $\text{CRC16}(k) \mod 16384$. CRC16[3] is a known checksum-like method used by many devices and network protocols to check that a message has been received with no errors, and it can also be used as a quick hash function.

Our implementation serializes in-memory objects and stores them as byte arrays in the database. Although this does not save us from serializing objects it is enough to avoid us to deal with the filesystem, allowing us to do all the operations in-memory. Huge byte arrays are split in distributed blocks to avoid long-term load imbalances and to increase long-term data locality, in a similar fashion to Hecuba. Another important detail is that Redis offers the possibility to have replicas. A replica is a Redis instance that mirrors the behavior of some other Redis instance and is usually located in a different node/machine than the original one. This reduces even more the expected transfer time, but it introduces a dangerous tradeoff between time and space.

Another detail about our Redis implementation is that Redis offers no direct way to make some piece of data be stored in some node. However, the Storage API defines a method which allows the user to do that. Our solution to this problem consisted of simply generating random keys until one of them mapped to a valid slot. This adds almost no overhead, as the number of nodes tends to be a very manageable number such as 16, 32, 64, or at most 128 in most practical cases. We also introduced an optimization for static cluster topologies which consist of precomputing all the possible locations for all the 16384 slots. This way, `getLocations` can be computed in constant time with no online queries to the storage backend. This gives us a small edge with respect to other storage implementations (or COMPSs itself when handling files), which do an explicit computation each time they call `getLocations`.

Redis also supports pipelining. Pipelining consists of joining various different queries into a single macro-query, allowing the storage backend optimize the order and internal commands. This pipelining feature also represents a small edge with respect to classical files, which are handled separately.
Our Redis implementation can be used with an already existing storage backend by just specifying the list of storage nodes and it is also capable to create a Redis Cluster with the specified COMPSs Nodes. A Redis cluster is created by launching three or more Redis instances and then joining them into a cluster with a Redis command. This storage implementation, and its user's manual, can be found in the project's repository [4]. The two most importants pieces of it, the Java and Python core API implementations, can be found in appendix E.

6.3 Practical Applications

6.3.1 K-Means

K-Means [8] is a clustering algorithm which, given N k-dimensional points and an integer c, assigns each point a label between 1 and c. The idea is that these labels

[3] https://en.wikipedia.org/wiki/Cyclic_redundancy_check
[4] https://github.com/srgrr/TFM/tree/master/applications/STORAGE

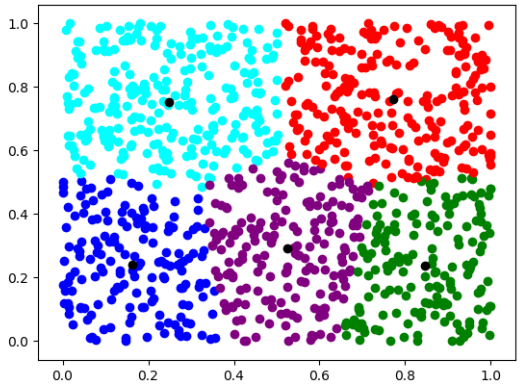

FIGURE 6.1: A set of points grouped by the K-Means algorithm. Black points represent centroids, colours represent different groups

represent groups of *similar* points. An example of what this algorithm computes can be found in figure 6.1.

The algorithm can be sumarized as follows:

1. Generate c random d dimensional points, call them *centroids*
2. For each point of the input data, compute the nearest centroid, assign them labels according to which centroid is the closest
3. For each group, compute the mean of its members. Use this mean point as the new centroid
4. Repeat step 2 until the new centroids are *equal enough* to the old ones

This algorithm can be easily run in a distributed environment by dividing the input points into chunks and replicating the centroids in each computing node. Note that the input points will not vary during all the execution, and that the centroids usually represent a very small amount of data, so no big network transfers should be expected here, and therefore no huge improvements should be observed with the storage implementation. Our PyCOMPSs implementation is the following can be found in appendix F.

Some results of how this application scales and behaves with various, different storage implementations and with files can be found in figures 6.2 6.3 6.4, 6.5, and 6.6.

As we can see, our storage implementation does not improve the overall performance of this application. This applications has little to no heavy transfers, only at the beggining, so these results are more or less expected.

6.3.2 Matrix Multiplication

The matrix multiplication is a very common algorithm and it is usually the preferred example of what an embarrassingly parallel application is (i.e: a parallel application with no dependencies). Its distributed version is also interesting but due to other property: the enormous amount of required data transfers. Let's take a look to the following code:

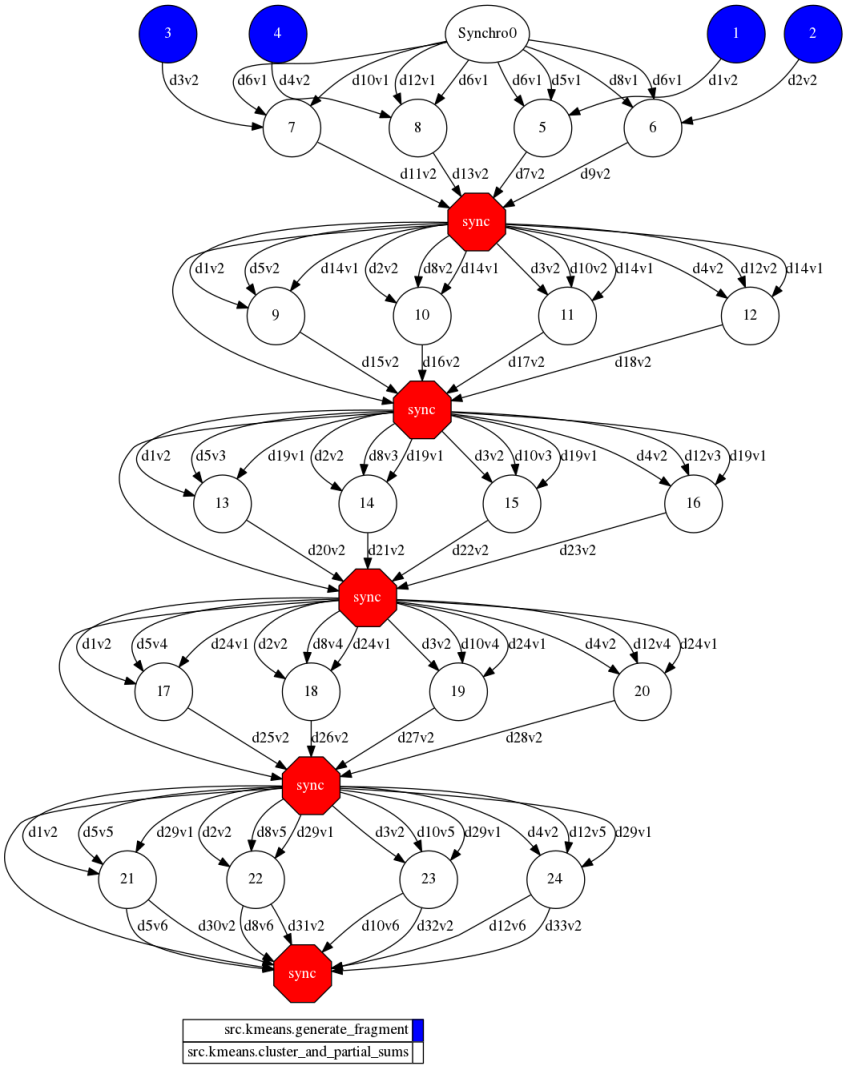

FIGURE 6.2: Dependency graph of a 6-iteration K-Means execution with 4 point fragments.

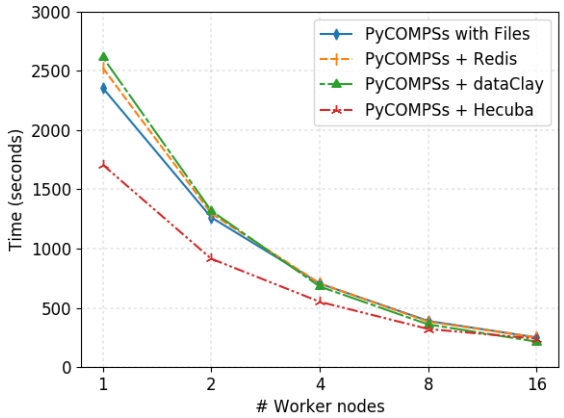

FIGURE 6.3: Strong scaling graph of our various storage implementations

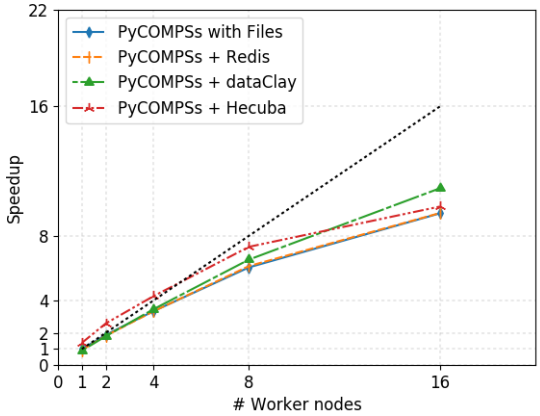

FIGURE 6.4: Strong scaling speedup graph of our various storage implementations

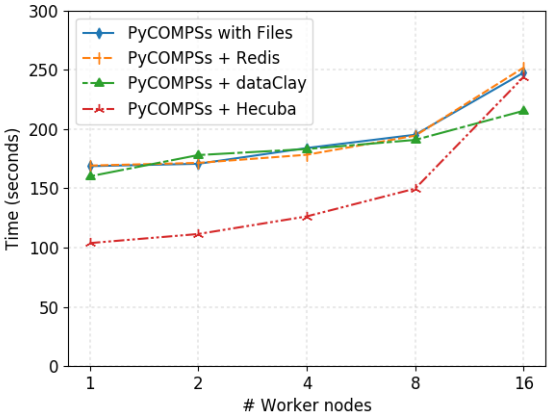

FIGURE 6.5: Weak scaling graph of our various storage implementations

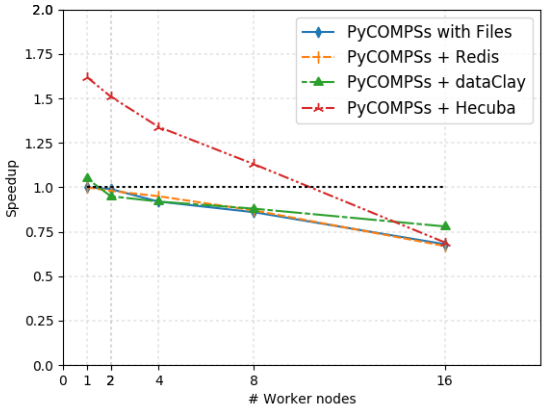

FIGURE 6.6: Weak scaling speedup graph of our various storage implementations

```
for i in range(N):
  for j in range(N):
```

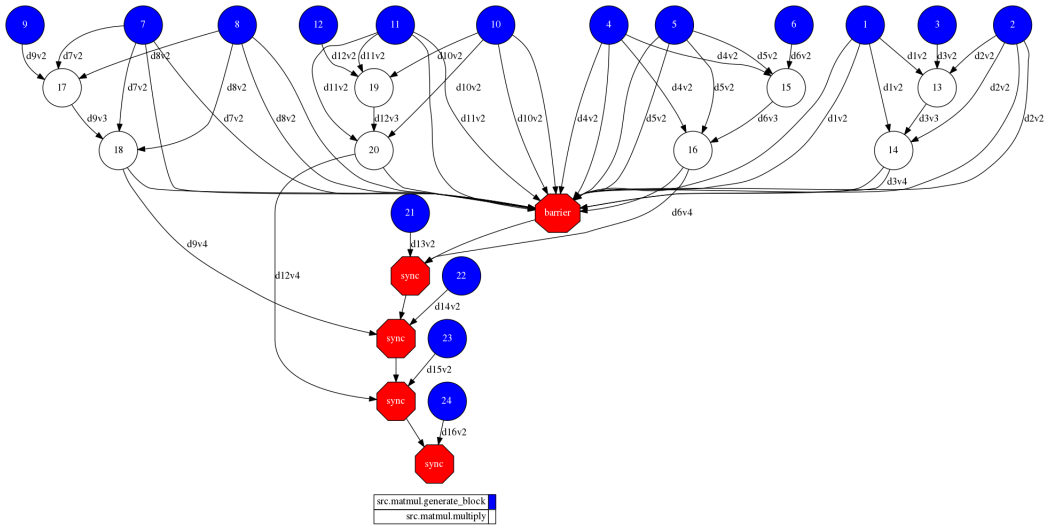

FIGURE 6.7: Dependency graph of a 2x2 matrix multiplication.

```
for k in range(N):
  C[i, j] += A[i, k] * B[k, j]
```

This code can be parallelized in any of the three loops. The only special requirement is to make sure that no pair of additions over the same $C_{i,j}$ are done concurrently. Our COMPSs implementation can be found in appendix G. As we can see, we *solve* the problem by arranging all the tasks that involve some $C_{i,j}$ in a dependency chain. We also consider the members of the matrices to be sub-matrices instead of single numbers in order to keep some balance between task count and task granularity. These dependencies can be observed in figure 6.7. As we can observe in the labels of edges, this application has far less tasks than pieces of data, and tasks have a huge *data variety*. This means that most tasks will require to transfer at least one of its parameters.

As we can see in figures 6.8 and 6.9 the Matrix Multiplication algorithm gets a huge benefit from our storage systems. This is expected, as this application is a great example of a very transfer-intensive workflow. Although this application is syntethic and has little to no usage in real scenarios, it still represents and reproduces many realistic workflows. Usually, an improvement on this application implies improvements on many real life applications, as they tend to be very transfer intensive.

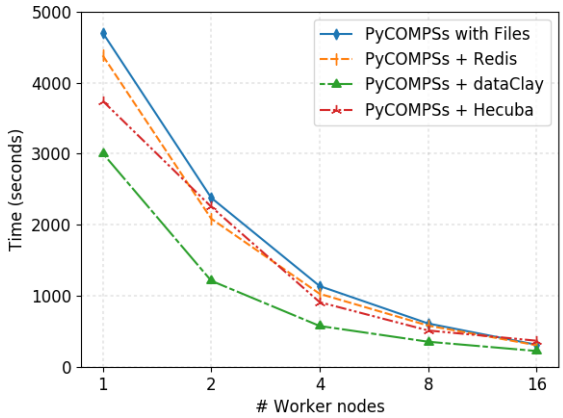

FIGURE 6.8: Strong scaling graph of our various storage implementations

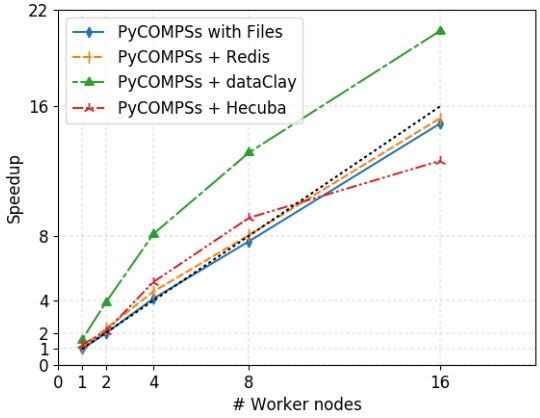

FIGURE 6.9: Strong scaling speedup graph of our various storage implementations

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Our contributions have improved both the usability and the performance of the COMPSs programming model. Our features have been succesfully tested and used with real use-cases written by real users, not only with our own test, synthetic programs.

We have removed two big flaws in the COMPSs programming model: the impossibility to run tasks with many parameters due to the huge overhead from the metadata of parameters (and thus affecting map-reduce workflows, which happen to be very common) and the performance issues when running COMPSs applications in non-shared filesystems such as GPFS by offering the user an alternative based on storage implementations.

7.2 Future Work

Let's consider the application from section 6.3.2. In the product of two 2×2 matrices the dependency $mul(A_{1,1}, B_{1,1}, C_{1,1}) \implies mul(A_{1,2}, B_{2,1}, C_{1,1})$ appears, when it is actually enough to ensure that no two tasks involving the same $C_{i,j}$ are executed concurrently. An open research line consists of developing a distributed mechanism to allow task commutativity. It was discussed to implement the option to assign each task a commutativity group, meaning that two tasks that belong to this group are mutually commutative. Task commutativity is already implemented in the OMPSs programming model (a COMPSs-like software but optimized for making applications run in a single node) [3], but it still remains as a challenge to implement an equivalent feature for a distributed programming model.

As we mentioned in section 6, Redis allows to have replicas of some instances. Replicas introduce a dangerous tradeoff between time and space. A naive usage of replicas quickly leads to program crashes derived from not having enough RAM, or even disk space. However, this does not mean that it is entirely useless. During the development of this project this line was quickly discarded, as it was considered to be too challenging and time-consuming. This idea cannot be properly tested locally, so a lot of time should be devoted to development and debugging, and even more supercomputer time should be requested. The replica feature was left unexplored and marked as a possible future work line.

Among all the proposed implementations regarding to collections (see section 2),

the most important one that was left undone is the collection transfer optimization. This was left as future work for two reasons: it seems hard and it is a vague idea. The general notion is that the COMPSs Runtime can exploit the fact that a group of parameters go together in a collection to optimize transfers. For example, they could be packed, sent in a single connection and unpacked in their destination. However, it was not possible to reach a concrete idea or implementation, leaving this as a possible future work line. Another possible work line regarding collections is dynamic size. Dynamic, and possibly unknown, sizings bring many new challenges. For example, the future or promise representation of the contents of a dynamically sized collection is not as trivial as fixed size arrays, as we cannot know in advance how many future objects we must generate.

Finally, the last future work line to mention is I/O and Threading. The main idea was to parallelize I/O operations when serializing and deserializing objects. The main problem with this idea is that I/O parallelization created a serious problem with memory usage. Serializing and deserializing an object can take twice the size of the object of memory. This issue does not affect most regular COMPSs applications if done sequentally, but causes a lot of crashes when done in parallel, as twice the size of *all* objects is significantly more than twice the size of *some* object. However, this does not mean that this idea should be totally discarded. Some kind of *smart* system that parallelizes I/O while avoiding these issues with memory can be designed, and we leave this as a another possible future work line.

Appendix A

Collections as Input Parameters

```
from pycompss.api.task import task
from pycompss.api.parameter import *

# A small, simple program to show the COLLECTION_IN feature
# Its workflow can be summarized as follows:
# 1) Ten random vectors with 5 elements are created in Python functions
# 2) A task receives these ten random vectors packed in a COLLECTION_IN
#    parameter
# 3) The fifth of these vectors is chosen
# 4) This same vector is created in the master program, and a check is
 ↪  performed


# This program should be enough to test that this feature
# works, as the main aspects are tested with it.

def generate_object(seed):
  """Given an integer, create a random numpy vector of 5 elements
  using this integer as the random seed.
  """
  import numpy as np
  np.random.seed(seed)
  return np.random.rand(5)

@task(c = COLLECTION_IN, returns = 1)
def select_element(c, i):
  """Given a collection and an integer "i", return the ith element
  of this collection
  """
  return c[i]

def main():
  from pycompss.api.api import compss_wait_on
  # Generate ten random vectors with pre-determined seed
  ten_random_vectors = [generate_object(i) for i in range(10)]
  # Pick the fifth vector from a COLLECTION_IN parameter
  fifth_vector = compss_wait_on(select_element(ten_random_vectors, 4))
  print("My chosen vector is \t %s" % str( fifth_vector ))

  # Recreate this vector locally
```

```
  import numpy as np
  np.random.seed(4)
  master_vector = np.random.rand(5)

  print("The correct vector is \t %s" % str( master_vector ))

  # Check that they match
  assert np.allclose(fifth_vector, master_vector), "Vectors do not
   ↪  match"

  # They did match, we can now end the program
  print("Congratulations! Both vectors match!")

if __name__ == "__main__":
  main()
```

Appendix B

Collections as INOUT Parameters

```
from pycompss.api.task import task
from pycompss.api.parameter import *

# A small, simple program to show the COLLECTION_INOUT feature
# Its workflow can be summarized as follows:
# 1) Ten random vectors with 5 elements are created in COMPSs tasks
# 2) A task receives these ten random vectors packed in a COLLECTION_IN
#    parameter
# 3) A task receives this collection as a COLLECTION_INOUT, and
 ↪  increases all entries
#    of all vectors by exactly one
# 4) The fifth of these vectors is chosen
# 5) This same vector is created in the master program, and a check is
 ↪  performed


# This program should be enough to test that this feature
# works, as the main aspects are tested with it.

@task(returns = 1)
def generate_object(seed):
  """Given an integer, create a random numpy vector of 5 elements
  using this integer as the random seed.
  """
  import numpy as np
  np.random.seed(seed)
  return np.random.rand(5)

@task(c = COLLECTION_INOUT)
def increase_elements(c):
  for elem in c:
    elem += 1.0

@task(c = COLLECTION_IN, returns = 1)
def select_element(c, i):
  """Given a collection and an integer "i", return the ith element
  of this collection
  """
  return c[i]
```

```
def main():
  from pycompss.api.api import compss_wait_on
  # Generate ten random vectors with pre-determined seed
  ten_random_vectors = [generate_object(i) for i in range(10)]
  increase_elements(ten_random_vectors)
  # Pick the fifth vector from a COLLECTION_IN parameter
  fifth_vector = compss_wait_on(select_element(ten_random_vectors, 4))
  print("My chosen vector is \t %s" % str( fifth_vector ))

  # Recreate this vector locally
  import numpy as np
  np.random.seed(4)
  master_vector = np.random.rand(5) + 1.0

  print("The correct vector is \t %s" % str( master_vector ))

  # Check that they match
  assert np.allclose(fifth_vector, master_vector), "Vectors do not
   ↪  match"

  # They did match, we can now end the program
  print("Congratulations! Both vectors match!")

if __name__ == "__main__":
  main()
```

The same application but adding a task which changes a single element of a collection, proving that our recursion pattern is enough to catch the two possible ways in which dependencies are generated (from collection to content, and from content to collection).

```
from pycompss.api.task import task
from pycompss.api.parameter import *

# A small, simple program to show the COLLECTION_INOUT feature
# Its workflow can be summarized as follows:
# 1) Ten random vectors with 5 elements are created in COMPSs tasks
# 2) A task receives these ten random vectors packed in a
 ↪  COLLECTION_INOUT
#    parameter
# 3) A task receives the fifht element of this collection as an INOUT
 ↪  and increases all
#    its entries by one
# 4) A task receives this collection as a COLLECTION_INOUT, and
 ↪  increases all entries
#    of all vectors by exactly one
# 5) The fifth of these vectors is chosen
# 6) This same vector is created in the master program, and a check is
 ↪  performed
```

```
# This program should be enough to test that this feature
# works, as the main aspects are tested with it.

@task(returns = 1)
def generate_object(seed):
  """Given an integer, create a random numpy vector of 5 elements
  using this integer as the random seed.
  """
  import numpy as np
  np.random.seed(seed)
  return np.random.rand(5)

@task(c = COLLECTION_INOUT)
def increase_elements(c):
  for elem in c:
    elem += 1.0

@task(e = INOUT)
def increase_element(e):
  e += 1.0

@task(c = COLLECTION_IN, returns = 1)
def select_element(c, i):
  """Given a collection and an integer "i", return the ith element
  of this collection
  """
  print("Received collection value is %s" % str(c))
  return c[i]

def main():
  from pycompss.api.api import compss_wait_on
  # Generate ten random vectors with pre-determined seed
  ten_random_vectors = [generate_object(i) for i in range(10)]
  increase_elements(ten_random_vectors)
  increase_element(ten_random_vectors[4])
  # Pick the fifth vector from a COLLECTION_IN parameter
  fifth_vector = compss_wait_on(select_element(ten_random_vectors, 4))
  print("My chosen vector is \t %s" % str( fifth_vector ))

  # Recreate this vector locally
  import numpy as np
  np.random.seed(4)
  master_vector = np.random.rand(5) + 2.0

  print("The correct vector is \t %s" % str( master_vector ))

  # Check that they match
  assert np.allclose(fifth_vector, master_vector), "Vectors do not
   ↪  match"

  # They did match, we can now end the program
```

```
  print("Congratulations! Both vectors match!")

if __name__ == "__main__":
  main()
```

Appendix C

HyperLogLog

Main application:

```
from pycompss.api.task import task
from pycompss.api.parameter import *
from HyperLogLog import HyperLogLog

def parse_arguments():
        import argparse
        parser = \
                argparse.ArgumentParser(
                        description = "Count the number of distinct
                         ↪  words of a set of text files"
                )
        parser.add_argument("files", metavar="N", type = str, nargs =
         ↪  "+",
                help="(Absolute or relative) paths to files")
        parser.add_argument("--bits", default = 64, type = int, help =
         ↪  "Bits per hash")
        parser.add_argument("--buckets", default = 11, type = int, help
         ↪  = "Bits for bucket indexing")
        parser.add_argument("--collections", action = "store_true")
        return parser.parse_args()

@task(filename = FILE_IN, returns = 1)
def count_distinct(filename, bits, buckets):
        h = HyperLogLog(b = bits, p = buckets)
        with open(filename, "r") as f:
                for line in f:
                        for w in line.strip().split():
                                h.add_word(w)
        return h

@task(hloglogs = COLLECTION_IN, returns = 1)
def reduce_hloglog_collections(hloglogs, bits, buckets):
        h = HyperLogLog(b = bits, p = buckets)
        for hloglog in hloglogs:
                h.add_hyperloglog(hloglog)
        return h.get_estimation()

@task(returns = 1)
def reduce_hloglog(bits, buckets, *args):
```

```
        h = HyperLogLog(b = bits, p = buckets)
        for hloglog in args:
                h.add_hyperloglog(hloglog)
        return h.get_estimation()

def elapsed_time(name, start, end):
        print("%s=%.08f" % (name, end - start))

def main(files, bits, buckets, collections):
        import time
        N = len(files)
        hloglogs =\
                list(map(count_distinct, files, [bits] * N, [buckets] *
                 ↪  N))
        from pycompss.api.api import compss_barrier, compss_wait_on
        # Easier isolation of reduce time
        compss_barrier()
        start_t = time.time()
        if collections:
                estimation = reduce_hloglog_collections(hloglogs, bits,
                 ↪  buckets)
        else:
                estimation = reduce_hloglog(bits, buckets, *hloglogs)
        end_t = time.time()

        print("Estimated cardinality: %d" % compss_wait_on(estimation))

        elapsed_time("reduce", start_t, end_t)

if __name__ == '__main__':
        opts = parse_arguments()
        main(**vars(opts))
```

HyperLogLog class

```
import numpy as np


class HyperLogLog(object):
        """An implementation of an HyperLogLog memory object, part of
         ↪  the
        wordcount-collections example source code
        """
        def __init__(self, b, p, plot = False):
                """Constructor method.
                Parameters:
                b, integer: total number of bits (b - p bits per
 ↪  bucket)
                p, integer: exponent of the number of buckets (2^p)
                """
                self.b = b
                self.p = p
```

```
                self.buckets = np.zeros(2 ** p, dtype = np.uint8)
                self.plot = plot

                if self.plot:
                        self.histogram = []

        def hash(self, w):
                """Get the hash of a token.
                Maybe sha256 is an overkill for this situation, but we
↪  prefer to err
                in the cautious side
                """
                import hashlib
                return
                 ↪  int(hashlib.sha256(w.encode("utf-8")).hexdigest(),
                 ↪  16)

        def leading_zeroes(self, n, num_bits):
                """Get the number of leading zeroes in binary
                 ↪  representation
                """
                for i in range(num_bits - 1, -1, -1):
                        if n & (1 << i):
                                return num_bits - i - 1
                return num_bits

        def add_word(self, w):
                """Process a token, update the corresponding register
                 ↪  (if necessary)
                """
                bpp = self.b - self.p
                w_hash = self.hash(w) & ((1 << self.b) - 1)
                bucket_index = w_hash >> bpp
                hash_value = w_hash & ((1 << bpp) - 1)
                self.buckets[bucket_index] = \
                        max(self.buckets[bucket_index],
                         ↪  self.leading_zeroes(hash_value, bpp))
                if self.plot:
                        self.histogram.append(bucket_index)

        def add_hyperloglog(self, h):
                assert self.b == h.b, "Cannot merge HLogLogs with
                 ↪  different number of bits"
                assert self.p == h.p, "Cannot merge HLogLogs with
                 ↪  different number of buckets"
                # The maximum can be expressed as the componentwise
                self.buckets = np.maximum(self.buckets, h.buckets)

        def get_estimation(self):
                """Return the estimation of the cardinality as the
                 ↪  harmonic mean
```

```
            of the registers
            """
            return float(self.buckets.shape[0]) / np.mean(1.0 /
             ↪  (2.0 ** self.buckets))

    def plot_result(self):
            if self.plot:
                    import pylab as plt
                    plt.figure()
                    plt.plot(self.histogram)
                    plt.savefig("hloglog.png")
```

Appendix D

Metadata generation comparison

A task `f(a, b, c, d, e)` sends this information through a socket:

```
[BINDING-COMMONS]  -  @GS_RegisterCE  -  Task registered: metadata.f
[BINDING-COMMONS]  -  @GS_ExecuteTaskNew - Processing task execution in bindings-common.
[BINDING-COMMONS]  -  @GS_ExecuteTaskNew  -  Processing parameter 0
[BINDING_COMMONS]  -  @process_param
[BINDING-COMMONS]  -  @process_param  -  ENUM DATA_TYPE: 9
[BINDING-COMMONS]  -  @process_param  -  File: ...
[BINDING-COMMONS]  -  @process_param  -  ENUM DIRECTION: 0
[BINDING-COMMONS]  -  @process_param  -  ENUM STREAM: 3
[BINDING-COMMONS]  -  @process_param  -  PREFIX: null
[BINDING-COMMONS]  -  @process_param  -  NAME: *args*_0
[BINDING-COMMONS]  -  @GS_ExecuteTaskNew  -  Processing parameter 1
[BINDING_COMMONS]  -  @process_param
[BINDING-COMMONS]  -  @process_param  -  ENUM DATA_TYPE: 9
[BINDING-COMMONS]  -  @process_param  -  File: ...
[BINDING-COMMONS]  -  @process_param  -  ENUM DIRECTION: 0
[BINDING-COMMONS]  -  @process_param  -  ENUM STREAM: 3
[BINDING-COMMONS]  -  @process_param  -  PREFIX: null
[BINDING-COMMONS]  -  @process_param  -  NAME: *args*_1
[BINDING-COMMONS]  -  @GS_ExecuteTaskNew  -  Processing parameter 2
[BINDING_COMMONS]  -  @process_param
[BINDING-COMMONS]  -  @process_param  -  ENUM DATA_TYPE: 9
[BINDING-COMMONS]  -  @process_param  -  File: ...
[BINDING-COMMONS]  -  @process_param  -  ENUM DIRECTION: 0
[BINDING-COMMONS]  -  @process_param  -  ENUM STREAM: 3
[BINDING-COMMONS]  -  @process_param  -  PREFIX: null
[BINDING-COMMONS]  -  @process_param  -  NAME: *args*_2
[BINDING-COMMONS]  -  @GS_ExecuteTaskNew  -  Processing parameter 3
[BINDING_COMMONS]  -  @process_param
[BINDING-COMMONS]  -  @process_param  -  ENUM DATA_TYPE: 9
[BINDING-COMMONS]  -  @process_param  -  File: ...
[BINDING-COMMONS]  -  @process_param  -  ENUM DIRECTION: 0
[BINDING-COMMONS]  -  @process_param  -  ENUM STREAM: 3
[BINDING-COMMONS]  -  @process_param  -  PREFIX: null
[BINDING-COMMONS]  -  @process_param  -  NAME: *args*_3
[BINDING-COMMONS]  -  @GS_ExecuteTaskNew  -  Processing parameter 4
[BINDING_COMMONS]  -  @process_param
[BINDING-COMMONS]  -  @process_param  -  ENUM DATA_TYPE: 9
[BINDING-COMMONS]  -  @process_param  -  File: ...
[BINDING-COMMONS]  -  @process_param  -  ENUM DIRECTION: 0
```

```
[BINDING-COMMONS]  -  @process_param  -  ENUM STREAM: 3
[BINDING-COMMONS]  -  @process_param  -  PREFIX: null
[BINDING-COMMONS]  -  @process_param  -  NAME: *args*_4
[BINDING-COMMONS]  -  @GS_ExecuteTaskNew  -  Processing parameter 5
[BINDING_COMMONS]  -  @process_param
[BINDING-COMMONS]  -  @process_param  -  ENUM DATA_TYPE: 9
[BINDING-COMMONS]  -  @process_param  -  File: ...
[BINDING-COMMONS]  -  @process_param  -  ENUM DIRECTION: 1
[BINDING-COMMONS]  -  @process_param  -  ENUM STREAM: 3
[BINDING-COMMONS]  -  @process_param  -  PREFIX: #
[BINDING-COMMONS]  -  @process_param  -  NAME: $return_0
```

While the collection equivalent generates this data:

```
[BINDING-COMMONS]  -  @GS_RegisterCE  -  Task registered: metadata.g
[BINDING-COMMONS]  -  @GS_ExecuteTaskNew - Processing task execution in bindings-commor
[BINDING-COMMONS]  -  @GS_ExecuteTaskNew  -  Processing parameter 0
[BINDING_COMMONS]  -  @process_param
[BINDING-COMMONS]  -  @process_param  -  ENUM DATA_TYPE: 26
[BINDING-COMMONS]  -  @process_param  -  Collection: ...
[BINDING-COMMONS]  -  @process_param  -  ENUM DIRECTION: 0
[BINDING-COMMONS]  -  @process_param  -  ENUM STREAM: 3
[BINDING-COMMONS]  -  @process_param  -  PREFIX: null
[BINDING-COMMONS]  -  @process_param  -  NAME: c
[BINDING-COMMONS]  -  @GS_ExecuteTaskNew  -  Processing parameter 1
[BINDING_COMMONS]  -  @process_param
[BINDING-COMMONS]  -  @process_param  -  ENUM DATA_TYPE: 9
[BINDING-COMMONS]  -  @process_param  -  File: ...
[BINDING-COMMONS]  -  @process_param  -  ENUM DIRECTION: 1
[BINDING-COMMONS]  -  @process_param  -  ENUM STREAM: 3
[BINDING-COMMONS]  -  @process_param  -  PREFIX: #
[BINDING-COMMONS]  -  @process_param  -  NAME: $return_0
```

Appendix E

Redis Storage API implementation

Python:

```
#
#  Copyright 2017 Barcelona Supercomputing Center (www.bsc.es)
#
#  Licensed under the Apache License, Version 2.0 (the "License");
#  you may not use this file except in compliance with the License.
#  You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
#  Unless required by applicable law or agreed to in writing, software
#  distributed under the License is distributed on an "AS IS" BASIS,
#  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
 ↪  implied.
#  See the License for the specific language governing permissions and
#  limitations under the License.
#
'''API for Redis PSCO handling
This class is responsible to establish and to mantain a Redis
 ↪  connection to
the backend, also it is responible of retrieving objects from it.
As a reminder, objects are stored as a serialized byte array.
@author: srodrig1 < sergio dot rodriguez at bsc dot es >
'''
import uuid
import redis
import rediscluster
from pycompss.util.serializer import serialize_to_string,
 ↪  deserialize_from_string, deserialize_from_handler
__name__ = "redispycompss"

'''Constants
'''
REDIS_PORT = 6379
MAX_BLOCK_SIZE = 510 * 1024 * 1024
#MAX_BLOCK_SIZE = 16

'''Global variables
They are declared only for visibility purposes
'''
```

```
redis_connection = None
hosts = None


class StorageException(Exception):
    '''StorageException class
    '''
    pass


def init(config_file_path=None, **kwargs):
    '''Init the storage client. Basically, we set the Redis client and
     ↪  connects it
    to the instance/cluster
    '''
    global redis_connection
    global hosts
    # If config_file_path is None we will assume that we only have
     ↪  localhost
    # as storage node
    if config_file_path is None:
        try:
            import StringIO as sio
        except ImportError:
            from io import StringIO as sio
        config_file_handler = sio.StringIO('localhost\n')
    else:
        config_file_handler = open(config_file_path)
    # As accorded in the API standar, this file must contain all the
     ↪  hosts names
    # with no port, one per line
    hosts = [x.strip() for x in config_file_handler.readlines()]
    config_file_handler.close()
    # If we have more than one host then we will assume that our
     ↪  backend is a Redis
    # cluster. If not, we will assume that we are dealing with a Redis
     ↪  standalone
    # instance
    if len(hosts) > 1:
        # Given that cluster clients are capable to perform master
        # slave hierarchy discovery, we will simply connect to the
         ↪  first
        # node we got
        redis_connection = \
            rediscluster.StrictRedisCluster(host=hosts[0],
             ↪  port=REDIS_PORT)
    else:
        # We are in standalone mode
        redis_connection = \
            redis.StrictRedis(host=hosts[0], port=REDIS_PORT)
```

```
    # StrictRedis is not capable to know if we had success when
     ↪ connecting by
    # simply calling the constructor. We need to perform an actual
     ↪ query to
    # the backend
    # If we had no success this first line should crash
    redis_connection.set('PYCOMPSS_TEST', 'OK')
    # Beware py2 vs py3 - b'string' works for both.
    assert redis_connection.get('PYCOMPSS_TEST') == b'OK'
    redis_connection.delete('PYCOMPSS_TEST')

def initWorker(config_file_path=None):
    '''Per-worker init function
    '''
    init(config_file_path)

init_worker = initWorker

def finishWorker(*args, **kwargs):
    '''Same as finish. No additional actions are needed
    '''
    pass

finish_worker = finishWorker

def finish(**kwargs):
    '''Finish the storage: Nothing to do, as Python redis clients have
     ↪ no
    close method.
    '''
    pass

def getByIDOld(identifier):
    '''Retrieves the object that has the given identifier from the
     ↪ Redis database.
    That is, given an identifier, retrieves the contents from the
↪   backend
    that correspond to this key, deserializes it and returns the
↪   reconstructed
    object.
    '''
    global redis_connection
    import io
    with io.BytesIO() as bio:
        num_blocks = int(redis_connection.llen(identifier))
        for l in redis_connection.lrange(identifier, 0, num_blocks):
            bio.write(l)
        # In case that we have read a None then it means that the
         ↪ requested object
        # was not present in the Redis backend
        bio.seek(0)
```

```
        ret = deserialize_from_handler(bio)
    return ret

def getByID(*identifiers):
    '''Retrieves a set of objects from their identifiers by pipelining
     ↪  the get commands
    '''
    global redis_connection
    p = redis_connection.pipeline()
    # Stack the pipe calls
    for identifier in identifiers:
        num_blocks = int(redis_connection.llen(identifier))
        p.lrange(identifier, 0, num_blocks)
    # Get all the objects
    ret = p.execute()
    # Deserialize and delete the serialized contents for each object
    for i in range(len(identifiers)):
        ret[i] = deserialize_from_string(
                b''.join(
                    ret[i]
                )
            )
        ret[i].pycompss_mark_as_unmodified()
    return ret[0] if len(ret) == 1 else ret

get_by_ID = getByID

def makePersistent(obj, identifier = None):
    '''Persists an object to the Redis backend. Does nothing if the
     ↪  object
    is already persisted.
    '''
    if obj.pycompss_psco_identifier is not None:
        # Non null identifier -> object is already persisted
        return
    # The object has no identifier, we need to assign it one
    obj.pycompss_psco_identifier = str(uuid.uuid4()) if identifier is
     ↪  None \
    else identifier
    # Serialize the object and store the pair (id, serialized_object)
    serialized_object = serialize_to_string(obj)
    bytes_size = len(serialized_object)
    num_blocks = (bytes_size + MAX_BLOCK_SIZE - 1) // MAX_BLOCK_SIZE
    for block in range( num_blocks ):
        l = block * MAX_BLOCK_SIZE
        r = (block + 1) * MAX_BLOCK_SIZE
        redis_connection.rpush(
            obj.pycompss_psco_identifier, serialized_object[l : r]
        )
    # Object is now synced with backend
    obj.pycompss_mark_as_unmodified()
```

```
make_persistent = makePersistent

def deletePersistent(obj):
    '''Deletes a persisted object. If the object was not persisted,
    ↪ then
    nothing will be done.
    '''
    if obj.pycompss_psco_identifier is None:
        # The object was not persisted, there is nothing to do
        return
    # Delete the object from the backend
    redis_connection.delete(obj.pycompss_psco_identifier)
    # Set key to None
    obj.pycompss_psco_identifier = None
    # Mark as unmodified
    obj.pycompss_mark_as_unmodified()

delete_persistent = deletePersistent

class TaskContext(object):
    '''Here for compatibility purposes
    '''
    def __init__(self, logger, values, config_file_path=None):
        self.logger = logger
        self.values = values
        self.config_file_path = config_file_path

    def __enter__(self):
        # Do something prolog

        # Ready to start the task
        self.logger.info("Prolog finished")
        pass

    def __exit__(self, type, value, traceback):
        # Update all modified objects
        for obj in self.values:
            try:
                if obj.pycompss_is_modified():
                    print('Repersisting object %s' % obj)
                    old_id = obj.getID()
                    obj.delete_persistent()
                    obj.make_persistent(old_id)
            except:
                pass
        # Finished
        self.logger.info("Epilog finished")
        pass

task_context = TaskContext
```

Java:

```
package storage;

import java.io.*;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.*;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import redis.clients.jedis.*;
import redis.clients.jedis.exceptions.JedisDataException;
import redis.clients.util.JedisClusterCRC16;
import storage.utils.Serializer;

public final class StorageItf {

    // Logger According to Loggers.STORAGE
    private static final Logger LOGGER =
     ↪  LogManager.getLogger("es.bsc.compss.Storage");

    // Error Messages
    private static final String ERROR_HOSTNAME = "ERROR_HOSTNAME";

    private static final String MASTER_HOSTNAME;

    // Redis variables

    // This port is the official Redis Port
    // See
     ↪  https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers
    // The storage API will assume that, given a hostname, there is a
     ↪  Redis Server listening there
    private static final int REDIS_PORT = 6379;
    private static final int REDIS_MAX_CLIENTS = 1<<19;
    // Number of Redis hash slots. This is fixed, and official. See
     ↪  redis.io tutorials
    private static final int REDIS_MAX_HASH_SLOTS = 16384;
    // Client connections
    // Given that the client classes that are needed to establish a
     ↪  connection with a Redis backend are
    // different for standalone and cluster cases, we are going to
     ↪  first try to establish a connection with
    // the cluster client, and, if it fails, with the standalone client
    // Given that JedisCluster and Jedis are classes that share no
     ↪  common ancestor, this is the cleanest way I can
    // come up with.
    private static JedisCluster redisClusterConnection;
    private static JedisPool    redisConnection;
    private static boolean      clusterMode = true;
```

```
    private static List<String> hosts = new ArrayList<>();

    private static HashMap<String, String> previousVersion = new
     ↪  HashMap<>();

    // Given a hash slot, return a list with the hosts that contain at
     ↪  least one instance that includes
    // this slot in its slot interval
    private static ArrayList< String >[] hostsBySlot = new
     ↪  ArrayList[REDIS_MAX_HASH_SLOTS];

    static {
        String hostname = null;
        try {
            InetAddress localHost = InetAddress.getLocalHost();
            hostname = localHost.getCanonicalHostName();
        } catch (UnknownHostException e) {
            System.err.println(ERROR_HOSTNAME);
            e.printStackTrace();
            System.exit(1);
        }
        MASTER_HOSTNAME = hostname;
    }

    /**
     * Constructor
     */
    public StorageItf() {
        // Nothing to do since everything is static
    }

    /**
     * Initializes the persistent storage
     * Configuration file must contain all the worker hostnames, one by
↪   line
     * @param storageConf Path to the storage configuration File
     * @throws StorageException
     */
    public static void init(String storageConf) throws
     ↪  StorageException, IOException {
        LOGGER.info("[LOG] Configuration received: " + storageConf);
        try (BufferedReader br = new BufferedReader(new
         ↪  FileReader(storageConf))) {
            String line;
            while ((line = br.readLine()) != null) {
                hosts.add(line.trim());
                LOGGER.info("Adding " + line.trim() + " to list of
                 ↪  known hosts...");
            }
        } catch (FileNotFoundException e) {
```

```
            throw new StorageException("Could not find configuration
             ↪ file", e);
        } catch (IOException e) {
            throw new StorageException("Could not open configuration
             ↪ file", e);
        }
        assert(!hosts.isEmpty());
        clusterMode = hosts.size() > 1;
        if(clusterMode) {
            LOGGER.info("More than one host detected, enabling Client
             ↪ Cluster Mode");
            // TODO: Ask Jedis guys why JedisCluster needs a
             ↪ HostAndPort and why Jedis needs a String and an Integer
            redisClusterConnection = new JedisCluster(new
             ↪ HostAndPort(hosts.get(0), REDIS_PORT));
            // Precompute host hashmap
            preComputeHostHashMap();
        }
        else {
            LOGGER.info("Only one host detect, using standalone
             ↪ client...");
            JedisPoolConfig poolConfig = new JedisPoolConfig();
            poolConfig.setMaxTotal(REDIS_MAX_CLIENTS);
            redisConnection = new JedisPool(poolConfig, hosts.get(0),
             ↪ REDIS_PORT);
        }
    }

    // Temporary representation of a host
    static private class Host {
        // Host (name)
        String host;
        // Hash slot endpoints
        int l, r;

        Host(String clusterInfoLine) {
            String[] tokens = clusterInfoLine.split(" ");
            this.host = tokens[1].split("@")[0].split(":")[0];
            InetAddress addr = null;
            try {
                addr = InetAddress.getByName(this.host);
            } catch (UnknownHostException e) {
                e.printStackTrace();
            }
            this.host = addr.getHostName();
            String[] interval = tokens[tokens.length - 1].split("-");
            this.l = Integer.parseInt(interval[0]);
            this.r = Integer.parseInt(interval[1]);
        }

        void printHostInfo() {
```

```
            System.out.printf("Host %s covers slots [%d, %d]\n", host,
             ↪  l, r);
        }

    }

    private static void preComputeHostHashMap() {
        String someHost =
         ↪  (String)redisClusterConnection.getClusterNodes().keySet().toArray()[0];
        String clusterInfo =
         ↪  redisClusterConnection.getClusterNodes().get(someHost).getResource().clusterNod
        String[] clusterLines = clusterInfo.split("\n");
        ArrayList< Host > clusterHosts = new ArrayList<>();
        for(String line : clusterLines) {
            Host h = new Host(line);
            clusterHosts.add(h);
        }
        for(int i = 0; i < REDIS_MAX_HASH_SLOTS; ++i) {
            ArrayList< String > validHosts = new ArrayList<>();
            for(Host h : clusterHosts) {
                if(h.l <= i && i <= h.r) {
                    validHosts.add(h.host);
                }
            }
            hostsBySlot[i] = new ArrayList<>(new
             ↪  TreeSet<>(validHosts));
        }
    }

    /**
     * Stops the persistent storage
     * StorageItf
     * @throws StorageException
     */
    public static void finish() throws StorageException {
        if(clusterMode) {
            try {
                redisClusterConnection.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        else {
            redisConnection.close();
        }
    }

    /**
     * Returns all the valid locations of a given id
     * @param id Object identifier
     * @return List of valid locations for given resource
```

```
     * @throws StorageException
     */
    public static List<String> getLocations(String id) throws
    ↪  StorageException {
        if(id != null && clusterMode) {
            int slot = JedisClusterCRC16.getSlot(id);
            return hostsBySlot[slot];
        }
        else {
            return hosts;
        }
    }

    /**
     * Creates a new replica of PSCO id @id in host @hostname
     *
     * @param id
     * @param hostName
     * @throws StorageException
     */
    public static void newReplica(String id, String hostName) throws
    ↪  StorageException {
        throw new StorageException("Redis does not support this
        ↪  feature.");
    }

    private static void putInRedis(byte[] serializedObject, String id)
    ↪  throws StorageException {
        String result = clusterMode ?
                redisClusterConnection.set(id.getBytes(),
                ↪  serializedObject) :
                redisConnection.getResource().set(id.getBytes(),
                ↪  serializedObject);
        if(!result.equals("OK")) {
            throw new StorageException("Redis returned an error while
            ↪  trying to store object with id " + id);
        }
    }

    /**
     * Create a new version of the PSCO id @id in the host @hostname
↪   Returns the id of the new version
     *
     * @param id
     * @param hostName
     * @return
     * @throws StorageException
     */
    public static String newVersion(String id, boolean preserveSource,
    ↪  String hostName) throws StorageException, IOException,
    ↪  ClassNotFoundException {
```

```
        byte[] obj = getBytesByID(id);
        String newId = UUID.randomUUID().toString();
        previousVersion.put(newId, id);
        putInRedis(obj, newId);
        if(!preserveSource) {
            consolidateVersion(newId);
        }
        return newId;
    }

    /**
     * Returns the object with id @id This function retrieves the
↪   object from any location
     *
     * @param id
     * @return
     * @throws StorageException
     */
    public static Object getByID(String id) throws StorageException,
     ↪  IOException, ClassNotFoundException {
        byte[] serializedObject = clusterMode ?
                redisClusterConnection.get(id.getBytes()) :
                redisConnection.getResource().get(id.getBytes());
        if(serializedObject == null) {
            throw  new StorageException("Object with id " + id + " is
             ↪  not in Redis!");
        }
        Object ret = Serializer.deserialize(serializedObject);
        ((StorageObject)ret).setID(id);
        return ret;
    }

    private static byte[] getBytesByID(String id) throws
     ↪  StorageException {
        byte[] ret = clusterMode ?
                redisClusterConnection.get(id.getBytes()) :
                redisConnection.getResource().get(id.getBytes());
        if(ret == null) {
            throw  new StorageException("Object with id " + id + " is
             ↪  not in Redis!");
        }
        return ret;
    }

    /**
     * Executes the task into persistent storage
     *
     * @param id
     * @param descriptor
     * @param values
     * @param hostName
```

```
     * @param callback
     * @return
     * @throws StorageException
     */
    public static String executeTask(String id, String descriptor,
     ↪  Object[] values, String hostName, CallbackHandler callback)
            throws StorageException {
        throw new StorageException("Redis does not support this
         ↪  feature.");
    }

    /**
     * Retrieves the result of persistent storage execution
     *
     * @param event
     * @return
     */
    public static Object getResult(CallbackEvent event) throws
     ↪  StorageException {
        throw new StorageException("Redis does not support this
         ↪  feature.");
    }

    /**
     * Consolidates all intermediate versions to the final id
     *
     * @param idFinal
     * @throws StorageException
     */
    public static void consolidateVersion(String idFinal) throws
     ↪  StorageException {
        LOGGER.info("Consolidating version for " + idFinal);
        // Skip final version
        idFinal = previousVersion.get(idFinal);
        while(idFinal != null) {
            LOGGER.info("Removing version " + idFinal);
            removeById(idFinal);
            String oldId = idFinal;
            idFinal = previousVersion.get(idFinal);
            previousVersion.remove(oldId);
        }
    }

    /*
     *
↪   ********************************************************************************
     * SPECIFIC IMPLEMENTATION METHODS

↪   ********************************************************************************
    /**
     * Stores the object @o in the persistent storage with id @id
```

```
     *
     * @param o
     * @param id
     * @throws StorageException
     */
    public static void makePersistent(Object o, String id) throws
    ↪  StorageException, IOException {
        byte[]  serializedObject = Serializer.serialize(o);
        String result = clusterMode ?
                redisClusterConnection.set(id.getBytes(),
                ↪  serializedObject) :
                redisConnection.getResource().set(id.getBytes(),
                ↪  serializedObject);
        if(!result.equals("OK")) {
            throw new StorageException("Redis returned an error while
            ↪  trying to store object with id " + id);
        }
    }

    /**
     * Removes all the occurrences of a given @id
     *
     * @param id
     */
    public static void removeById(String id) {
        if(clusterMode) {
            redisClusterConnection.del(id.getBytes());
        }
        else {
            redisConnection.getResource().del(id.getBytes());
        }
    }

    // ONLY FOR TESTING PURPOSES
    static class MyStorageObject extends StorageObject implements
    ↪  Serializable {
        private String innerString;

        public MyStorageObject(String myString) {
            innerString = myString;
        }

        public String getInnerString() {
            return innerString;
        }

        public void setInnerString(String innerString) {
            this.innerString = innerString;
        }
    }
```

```
    /**
     * ONLY FOR TESTING PURPOSES
     * @param args
     */
    public static void main(String[] args) throws
     ↪  ClassNotFoundException {
        try {

             ↪  init("/home/sergiorg/git/framework/utils/storage/redisPSCO/scripts/samp
            if(clusterMode) {
                // let's do getByID stuff
                MyStorageObject myObject = new MyStorageObject("This is
                 ↪  an object");
                myObject.makePersistent();
                Map< String, JedisPool > m =
                 ↪  redisClusterConnection.getClusterNodes();
                for(String s : m.keySet()) {
                    JedisPool jp = m.get(s);
                    Jedis j = jp.getResource();
                    //System.out.println(j.info());
                    //System.out.println(j.clusterInfo());
                    //System.out.println(j.clusterNodes());
                    break;
                }
            }
            else {
                // let's do standalone stuff
                MyStorageObject myObject = new MyStorageObject("This is
                 ↪  an object");
                StorageItf.makePersistent(myObject, "prueba");
                Object retrieved = StorageItf.getByID("prueba");

                 ↪  System.out.println(((MyStorageObject)retrieved).getInnerString());
                myObject.updatePersistent();
                StorageItf.removeById("prueba");

            }
        } catch(StorageException | IOException e) {
            e.printStackTrace();
        }
    }

}
```

The `storage_init.sh` script, used to build Redis clusters on demand:

```
#!/bin/bash

  ############################################################
  # Name: storage_init.sh
  # Description: Storage API script for COMPSs
  # Parameters: <jobId>              Queue Job Id
```

```
#             <masterNode>          COMPSs Master Node
#             <storageMasterNode>   Node reserved for Storage Master
 ↪  Node (if needed)
#             "<workerNodes>"       Nodes set as COMPSs workers
#             <network>             Network type
#             <storageProps>        Properties file for storage
 ↪  specific variables
#############################################################


#-----------------------------------------------------------
# ERROR CONSTANTS
#-----------------------------------------------------------
ERROR_PROPS_FILE="Cannot find storage properties file"
ERROR_GENERATE_CONF="Cannot generate conf file"


#-----------------------------------------------------------
# HELPER FUNCTIONS
#-----------------------------------------------------------

####################
# Function to display usage
####################
usage() {
  local exitValue=$1

  echo " Usage: $0 <jobId> <masterNode> <storageMasterNode>
   ↪  \"<workerNodes>\" <network> <storageProps>"
  echo " "

  exit $exitValue
}

####################
# Function to display error
####################
display_error() {
  local errorMsg=$1

  echo "ERROR: $errorMsg"
  exit 1
}


#-----------------------------------------------------------
# MAIN FUNCTIONS
#-----------------------------------------------------------

####################
# Function to get args
```

```
####################
get_args() {
  NUM_PARAMS=6

  # Check parameters
  if [ $# -eq 1 ]; then
    if [ "$1" == "usage" ]; then
      usage 0
    fi
  fi
  if [ $# -ne ${NUM_PARAMS} ]; then
    echo "Incorrect number of parameters"
    usage 1
  fi

  # Get parameters
  jobId=$1
  master_node=$2
  storage_master_node=$master_node
  worker_nodes=$4
  network=$5
  storageProps=$6
}

####################
# Function to check and arrange args
####################
check_args() {
  # Check storage Props file exists
  if [ ! -f ${storageProps} ]; then
    # PropsFile doesn't exist
    display_error "${ERROR_PROPS_FILE}"
  fi
  source ${storageProps}

  # Convert network to suffix
  if [ "${network}" == "ethernet" ]; then
    network=""
  elif [ "${network}" == "infiniband" ]; then
    network="-ib0"
  elif [ "${network}" == "data" ]; then
    network=""
  fi
}

####################
# Function to log received arguments
####################
log_args() {
  echo "--- STORAGE_INIT.SH ---"
  echo "Job ID:              $jobId"
```

```
  echo "Master Node:          $master_node"
  echo "Storage Master Node: $storage_master_node"
  echo "Worker Nodes:         $worker_nodes"
  echo "Network:              $network"
  echo "Storage Props:        $storageProps"
  echo "-----------------------"
}
####################
# Function to resolve a hostname to an IP
####################
resolve_host_name() {
  echo `getent hosts $1 | awk '{ print $1 }'`
}

####################
# Function to check if an argument is an IP
####################
function valid_ip() {
  local  ip=$1
  local  stat=1
  if [[ $ip =~ ^[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}$ ]];
  ↪ then
    OIFS=$IFS
    IFS='.'
    ip=($ip)
    IFS=$OIFS
    [[ ${ip[0]} -le 255 && ${ip[1]} -le 255 \
    && ${ip[2]} -le 255 && ${ip[3]} -le 255 ]]
    stat=$?
  fi
  return $stat
}

####################
# Given a node identifier, resolves it to an IP if it not already an
↪ IP
####################
get_ip_address() {
  local output=$1;
  if ! valid_ip $1;
  then
    output=$(resolve_host_name $1);
  fi
  echo $output
}

####################
# Returns the command that must be executed in order to create a
↪ redis instance on a given location, on a given port
####################
get_redis_instantiation_command() {
```

```
    # $1 = host
    # $2 = path

    # SSH: This command simply opens an SSH connection to the target
     ↪  node and executes the command
    # SSH assumes the following:
    #     1) There is passwordless access to the destination
    #     2) The target machine has no zombie-killer mechanism
    if [ "$REDIS_REMOTE_COMMAND" == "ssh" ];
    then
      echo "ssh $1 \"cd $2; redis-server redis.conf\""
    elif [ "$REDIS_REMOTE_COMMAND" == "blaunch" ];
    then
      echo "BLAUNCH Pending to implement!"
    elif [ "$REDIS_REMOTE_COMMAND" == "srun" ];
    then
      echo "SRUN Pending to implement!"
    else
      echo "ERROR: \"$REDIS_REMOTE_COMMAND\" is not a valid
       ↪  instantiation command or it is not supported"
      echo "Supported commands are: ssh, blaunch, and srun"
      exit 1
    fi
}

# Default values
# The idea is to replace these values by the desired ones in the
 ↪  --storage_conf file
# The values below form a valid sample configuration anyway
REDIS_PORT=6379
REDIS_NODE_TIMEOUT=5000
REDIS_REPLICAS=0
REDIS_REMOTE_COMMAND=ssh
REDIS_HOME=/tmp/redis_cluster

get_args "$@"
check_args
log_args

#------------------------------------------------------------
# MAIN FUNCTIONS
#------------------------------------------------------------
REDIS_TEMPLATE="bind 0.0.0.0\ndaemonize yes\nprotected-mode no\nport
 ↪  REDIS_PORT\ncluster-enabled yes\ncluster-config-file
 ↪  nodes.conf\ncluster-node-timeout REDIS_NODE_TIMEOUT\nappendonly
 ↪  no"
REDIS_SANDBOX=$REDIS_HOME/${jobId}

STORAGE_HOME=$(dirname $0)/../

echo "REDIS CONFIGURATION PARAMETERS"
```

```
echo "REDIS_HOME:            $REDIS_HOME"
echo "REDIS_SANDBOX:         $REDIS_SANDBOX"
echo "REDIS_PORT:            $REDIS_PORT"
echo "REDIS_NODE_TIMEOUT:    $REDIS_NODE_TIMEOUT"
echo "REDIS_REPLICAS:        $REDIS_REPLICAS"
echo "REDIS_REMOTE_COMMAND: $REDIS_REMOTE_COMMAND"
echo "-----------------------"

# These paths are needed for COMPSs because the runtime
# will systematically look for a storage.cfg file here
# if storage_home has been defined
COMPSS_STORAGE_DIR=$HOME/.COMPSs/${jobId}/storage
COMPSS_STORAGE_CFG_DIR=$COMPSS_STORAGE_DIR/cfgfiles
COMPSS_STORAGE_CFG_FILE=$COMPSS_STORAGE_CFG_DIR/storage.properties

echo "COMPSS PATH PARAMETERS"
echo "COMPSS_STORAGE_DIR:      $COMPSS_STORAGE_DIR"
echo "COMPSS_STORAGE_CFG_DIR:  $COMPSS_STORAGE_CFG_DIR"
echo "COMPSS_STORAGE_CFG_FILE: $COMPSS_STORAGE_CFG_FILE"
echo "-----------------------"
############################
## STORAGE DEPENDENT CODE ##
############################

# These paths are needed to be available because COMPSs will look
# for storage stuff there, and there is no way to change it
# Note that REDIS_HOME is a different variable
# This is intentional because the $HOME directory may be shared, so
# we may end up creating Redis instances sandboxes for two different
 ↪  instances
# in a same location if they have the same port
mkdir -p $COMPSS_STORAGE_DIR
mkdir -p $COMPSS_STORAGE_CFG_DIR

# Write the nodes to the storage config that is needed for COMPSs
echo ${storage_master_node}${network} >>  $COMPSS_STORAGE_CFG_FILE
for worker_node in $worker_nodes
do
  echo ${worker_node}${network} >> $COMPSS_STORAGE_CFG_FILE
done

# Pre-step: Resolve the nodes names to IPs (if needed)
# This is due to the Redis backend limitation that imposes that it
 ↪  only works well when IPs are
# passed. Given that we have no guarantee about the locations format
 ↪  (i.e: we do not know if they are
# going to be hostnames of IPs) then we must check if we got an IP
 ↪  and, if not, resolve it
# see get_ip_address, valid_ip, and resolve_host_name to see what is
 ↪  being done here
```

```
storage_master_node=$(get_ip_address
 ↪  ${storage_master_node}${network});
worker_nodes=$(
  for worker_node in $worker_nodes
    do
      echo $(get_ip_address ${worker_node}${network})
    done
);


echo "RESOLVED REDIS NODES"
echo "MASTER NODE: $storage_master_node"
for worker_node in $worker_nodes
do
  echo "WORKER NODE: $worker_node"
done

echo "WORKING DIRECTORY: $(pwd)"

echo "------------------------"

# Compute the amount of needed redis instances
# The amount of needed instances can be computed as follows:
# max(3, (replicas + 1)*num_locations))
# Redis needs at least three instances to work well as a cluster
 ↪  (otherwise they suggest you to switch to
# standalone mode). Also, we need to have a Redis master in all of
 ↪  our nodes and we need their replicas too
all_instances_locations=("${storage_master_node}
 ↪  ${worker_nodes[@]}");
num_locations=$(wc -w <<< "${all_instances_locations}");
eff_locations=$num_locations
if (($eff_locations < 3));
then
  eff_locations=3;
fi
needed_instances=$(($((REDIS_REPLICAS + 1))*eff_locations));

# Create the Redis sandboxes for our instances
# A Redis instance is located at a given host and listens to a given
 ↪  port
# If we are forced to launch more than one instance then we are going
 ↪  to need more
# ports than the given one to listen. Our choice is simple, if our
 ↪  original port is 6379 and it is not
# available, then we will assign the port 6380, 6381... and so on.
 ↪  This idea is taken from the redis cluster
# tutorial. We assume that these ports (alongside with these ports +
 ↪  10000) are free. The default and official
# port for Redis 6379, and the next official port is 6389 by some
 ↪  strange software, so it is generally safe
```

```
# to establish ports this way
# See https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers
# A redis sandbox consists of a folder named after the port we are
 ↪  going to use that contains a redis.conf file
# See REDIS_TEMPLATE and the line that seds it to redis_conf below to
 ↪  see what it is being done here
current_instances=0;
node_ids="";
while (($current_instances < $needed_instances))
do
  for instance_location in ${all_instances_locations}
  do
    if (($current_instances == $needed_instances))
    then
      break
    fi
    # Compute the port
    redis_port=$((REDIS_PORT + $((current_instances/num_locations))))
    # Replace the configuration template parameters with their values
    redis_conf=$(echo $REDIS_TEMPLATE | sed -s
     ↪  s/REDIS_PORT/$redis_port/ | sed -s
     ↪  s/REDIS_NODE_TIMEOUT/$REDIS_NODE_TIMEOUT/)
    # Compute the path of the sandbox for this instance
    redis_path=${REDIS_SANDBOX}/${redis_port};
    # Create the folder structure (and remove the previous one if
     ↪  needed). This part can be done with ssh
    # It is needed to remove the old storage version because they may
     ↪  contain nodes.conf files of old clusters
    # that may not coincide with the configuration we want in this
     ↪  execution
    ssh $instance_location "rm -rf ${redis_path}; mkdir -p
     ↪  ${redis_path}; echo -e \"${redis_conf}\" >
     ↪  ${redis_path}/redis.conf;";
    # Launch the redis instance
    # This part is on an specific function because the needed command
     ↪  may vary from one queue system to another
    eval $(get_redis_instantiation_command $instance_location
     ↪  ${redis_path})
    current_instances=$((current_instances+1));
    node_name=${instance_location}:${redis_port}
    node_ids="${node_ids} $node_name"
    echo "CREATED REDIS INSTANCE IN $node_name"
  done
done
# Create a cluster with the instances
# We should detect failures when trying to create the cluster
echo "yes" | redis-trib.rb create --replicas $REDIS_REPLICAS
 ↪  $node_ids

############################
## END                    ##
```

```
############################
exit
```

Appendix F

K-Means + Storage Implementation

```
@task(returns = 1, labels = INOUT)
def cluster_and_partial_sums(fragment, labels, centres, norm):
  '''Given a fragment of points, declare a CxD matrix A and, for each
  ↪  point p:
  1) Compute the nearest centre c of p
  2) Add p / num_points_in_fragment to A[index(c)]
  3) Set label[index(p)] = c
  '''
  ret = np.matrix(np.zeros(centres.shape))
  n = fragment.mat.shape[0]
  c = centres.shape[0]
  # Check if labels is an empty list
  if not labels:
    # If it is, fill it with n zeros.
    for _ in range(n):
      # Done this way to not lose the reference
      labels.append(0)
  # Compute the big stuff
  associates = np.zeros(c)
  # Get the labels for each point
  for (i, point) in enumerate(fragment.mat):
    distances = np.zeros(c)
    for (j, centre) in enumerate(centres):
      distances[j] = np.linalg.norm(point - centre, norm)
    labels[i] = np.argmin(distances)
    associates[labels[i]] += 1
  # Add each point to its associate centre
  for (i, point) in enumerate(fragment.mat):
    ret[labels[i]] += point / associates[labels[i]]
  return ret


def kmeans_frag(fragments, dimensions, num_centres = ...):
  '''A fragment-based K-Means algorithm.
  Given a set of fragments (which can be either PSCOs or future objects
 ↪  that
  point to PSCOs), the desired number of clusters and the maximum
 ↪  number of
```

```
    iterations, compute the optimal centres and the index of the centre
    for each point.
    PSCO.mat must be a NxD float np.matrix, where D = dimensions
    '''
    import numpy as np
    # Choose the norm among the available ones
    norms = {
      'l1': 1,
      'l2': 'fro'
    }
    # Set the random seed
    np.random.seed(seed)
    # Centres is usually a very small matrix, so it is affordable to have
     ↪  it in
    # the master.
    centres = np.matrix(
      [np.random.random(dimensions) for _ in range(num_centres)]
    )
    # Make a list of labels, treat it as INOUT
    # Leave it empty at the beggining, update it inside the task. Avoid
    # having a linear amount of stuff in master's memory unnecessarily
    labels = [[] for _ in range(len(fragments))]
    # Note: this implementation treats the centres as files, never as
     ↪  PSCOs.
    for it in range(iterations):
      partial_results = []
      for (i, frag) in enumerate(fragments):
        # For each fragment compute, for each point, the nearest centre.
        # Return the mean sum of the coordinates assigned to each centre.
        # Note that mean = mean ( sum of sub-means )
        partial_result = cluster_and_partial_sums(frag, labels[i],
         ↪  centres, norms[norm])
        partial_results.append(partial_result)
      # Bring the partial sums to the master, compute new centres when
       ↪  syncing
      new_centres = np.matrix(np.zeros(centres.shape))
      from pycompss.api.api import compss_wait_on
      for partial in partial_results:
        partial = compss_wait_on(partial)
        # Mean of means, single step
        new_centres += partial / float( len(fragments) )
      if np.linalg.norm(centres - new_centres, norms[norm]) < epsilon:
        # Convergence criterion is met
        break
      # Convergence criterion is not met, update centres
      centres = new_centres
    # If we are here either we have converged or we have run out of
     ↪  iterations
    # In any case, now it is time to update the labels in the master
    ret_labels = []
    for label_list in labels:
```

```
    from pycompss.api.api import compss_wait_on
    to_add = compss_wait_on(label_list)
    ret_labels += to_add
  return centres, ret_labels
```

Appendix G

Matmul + Storage Implementation

```
@task(C = INOUT)
def multiply(A, B, C):
  '''Multiplies two blocks and acumulates the result in an INOUT
  matrix
  '''
  C += A.block * B.block

def dot(A, B, C, set_barrier = False):
  '''A COMPSs-PSCO blocked matmul algorithm
  A and B (blocks) are PSCOs, while C (blocks) are objects
  '''
  n, m = len(A), len(B[0])
  # as many rows as A, as many columns as B
  for i in range(n):
    for j in range(m):
      for k in range(n):
        multiply(A[i][j], B[i][j], C[i][j])
  if set_barrier:
    from pycompss.api.api import compss_barrier
    compss_barrier()
```

Bibliography

[1] Guillem Alomar, Yolanda Becerra Fontal, and Jordi Torres Viñals. "Hecuba: Nosql made easy". In: *BSC Doctoral Symposium (2nd: 2015: Barcelona)*. Barcelona Supercomputing Center. 2015, pp. 136–137.

[2] Rosa M. Badia et al. "COMP Superscalar, an interoperable programming framework". In: *SoftwareX* 3 (Nov. 2015). DOI: 10.1016/j.softx.2015.10.004.

[3] Alejandro Duran et al. "Ompss: a proposal for programming heterogeneous multi-core architectures". In: *Parallel processing letters* 21.02 (2011), pp. 173–193.

[4] Philippe Flajolet et al. "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm". In: *IN AOFA '07: PROCEEDINGS OF THE 2007 INTERNATIONAL CONFERENCE ON ANALYSIS OF ALGORITHMS*. 2007.

[5] Message P Forum. *MPI: A Message-Passing Interface Standard*. Tech. rep. Knoxville, TN, USA, 1994.

[6] Jesús Labarta et al. "Scalability of Tracing and Visualization Tools". In: Sept. 2005, pp. 869–876.

[7] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563.

[8] Stuart P. Lloyd. "Least squares quantization in pcm". In: *IEEE Transactions on Information Theory* 28 (1982), pp. 129–137.

[9] Francesc Lordan et al. "ServiceSs: An Interoperable Programming Framework for the Cloud". In: *Journal of Grid Computing* 12.1 (2014), pp. 67–91. ISSN: 1572-9184. DOI: 10.1007/s10723-013-9272-5. URL: https://doi.org/10.1007/s10723-013-9272-5.

[10] Jonathan Martí et al. "Dataclay: A distributed data store for effective inter-player data sharing". In: *Journal of Systems and Software* 131 (May 2017). DOI: 10.1016/j.jss.2017.05.080.

[11] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.0*. May 2008. URL: http://www.openmp.org/mp-documents/spec30.pdf.

[12] Frank B Schmuck and Roger L Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters." In: *FAST*. Vol. 2. 19. 2002.

[13] Konstantin Shvachko et al. "The hadoop distributed file system." In: *MSST*. Vol. 10. 2010, pp. 1–10.

[14] Enric Tejedor et al. "PyCOMPSs: Parallel computational workflows in Python". In: *The International Journal of High Performance Computing Applications* 31.1 (2017), pp. 66–82. DOI: 10.1177/1094342015594678. eprint: https://doi.org/10.1177/1094342015594678. URL: https://doi.org/10.1177/1094342015594678.

[15] Andy B Yoo, Morris A Jette, and Mark Grondona. "Slurm: Simple linux utility for resource management". In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2003, pp. 44–60.

[16] Matei Zaharia et al. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=1863103.1863113.

[17] Songnian Zhou. "Lsf: Load sharing in large heterogeneous distributed systems". In: *I Workshop on cluster computing*. Vol. 136. 1992.