UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) - BARCELONATECH

FACULTAT D'INFORMÀTICA DE BARCELONA (FIB)

MASTER IN INNOVATION AND RESEARCH IN INFORMATICS (MIRI)

DATA SCIENCE

---

# Scheduling policies for Big Data workflows

---

FINAL MASTER THESIS (FMT)

2018-2019 | SPRING SEMESTER

*Author:*
Ramon AMELA MILIAN
(ramela@bsc.es)

*Supervisor:*
Dra. Rosa M. BADIA SALA
(rosa.m.badia@bsc.es)

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Choose a job you love, and you will never have to work a day in your life.

---

**Anonymous**

# Dedication

To all my friends, for sharing with me the journey that got me here. Specially to those who have sustained me during my peak stress periods during this master thesis and the last burn out session, thank you very much Marta, Cristian, Fanny, big Pol, Virginia, Joan, Laia, Joana, bourgeois Laia, small Pol, Javi, Josep, Sara, Jordi, Nil and many others.

To my parents, who have always encouraged me to follow my dreams, no matter how stupid they were, and have always supported me, specially when failing. Also thanks to teach me how important is honesty in all areas of life.

Special thanks to you, Josep, for being you and being always there. We are a team, bro.

Wholeheartedly,
Ramon Amela Milian

# Declaration of Authorship

I hereby declare that, except where specific reference is made to the work of others, this Master's thesis has been composed by me and it is based on my own work. None of the contents of this dissertation have been previously published nor submitted, in whole or in part, to any other examination in this or any other university.

Signed: _____

Date: _____

# Acknowledgements

I gratefully thank my supervisor Rosa M. Badia Sala for letting me participate in the projects that have made possible the redaction of this master thesis.

I would also like to thank all my colleagues, current and former members of the *Workflows and Distributed Computing* team from the *Barcelona Supercomputing Center (BSC)* for their comments through the learning process of this Master's thesis: Pol Alvarez, Javier Alvarez, Cristian Ramón Cortés, Sergio Rodriguez, Raul Sirvent, Marc Dominguez, Salvi Sola, Marta Bertan, Hatem Elshazly, Nihad Mammadli, Jorge Ejarque, Francesc Lordan, Francisco Javier Conejero and Daniele Lezzi.

Special thanks to Marta Guindó Martinez for introducing me to the GWAS world and for the infinite patience when everything crushed, crushes and will crush.

Last but not least, thanks to Riccardo Tosi and Riccardo Rossi for the discussions about Multilevel Montecarlo, *on the fly* convergence checking and potential asynchronous execution strategies.

UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC) - BARCELONATECH

Facultat d'Informàtica de Barcelona (FIB)

MASTER IN INNOVATION AND RESEARCH IN INFORMATICS (MIRI)

Data Science

# *Abstract*

**Scheduling policies for Big Data workflows**

by Ramon AMELA MILIAN

While data workflows have some solutions in the market that scale to tens of thousands of cores and are really stable, there are not clear both scalable and stable solutions for arbitrary task flows. Scheduling big and heterogeneous workflows is a big challenge. More precisely, it has been proven to be an NP-hard problem. Nevertheless, the advances in the algorithm designs, the runtime parallelization strategies, the available computational resources and the reduction of network latencies has made possible to attack this kind of research subjects in the last years.

The aim of this master thesis is to both give the programmer some guidelines to change the user code in order to achieve good scalabilities with tasked based programming models and to give to the COMPSs runtime scheduler the capability to handle a high amount of tasks and resources in order to scale out big executions.

**Keywords:** Big Data, HPC, Distributed Computing, Workflows, COMPSs, PyCOMPSs

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| **API** | **A**pplication **P**rogramming **I**nterface |
| **COMPSs** | **COMP S**uperscalar |
| **CPU** | **C**entral **P**rocessing **U**nit |
| **DAG** | **D**irected **A**cyclic **G**raph |
| **FPGA** | **G**rahical **P**rocessing **U**nit |
| **GPU** | **F**ield **P**rogrammable **G**ate **A**rray |
| **GWAS** | **G**enome **W**ide **A**ssociation **S**tudy |

# Glossary

**CPU**
The Central Processing Unit (CPU) is the part of the computer that contains all the elements required to execute the instructions of software programs. Its main components are the main memory, the Processing Unit (PU) and the Control Unit (CU). Modern computers use multi-core processors, which are a single chip containing one or more cores.

**Core**
A core is an individual processor that actually executes program instructions. Current single chip CPUs contain many cores and are referred as multi-processor or multi-cores.

**Node**
A compute node refers to a single system within a cluster of many systems.

**Framework**
Framework stands for a set of standardized concepts, practices or criterias used to face a given problem. Specifically, it defines a set of programs, libraries, languages, and programming models used jointly in a project.

**Workflow**
A workflow is composed of tasks and dependencies between tasks. Workflows are commonly represented as graphs, with the nodes beeing tasks and the arrows representing the dependencies. Somehow, tasks must represent an action that must be done (i.e. the execution of a binary), and the dependencies must represent the requirements that must be satisfied to be able to execute the task (i.e. the machine availability or the required data).

**Binary**
A file containing a list of machine code instructions to perform a list of tasks.

**Parallelization**
Separation of a program in small independent processes that can run simultaneously.

**Runtime**
Metaprogram that controls a program in execution time.

**Granularity**
Average task duration time.

# Chapter 1

# Introduction

## 1.1 Motivation

While data workflows have some solutions in the market that scale to tens of thousands of cores and are really stable [1], there are not clear both scalable and stable solutions for arbitrary task flows. Scheduling big and heterogeneous workflows is a big challenge. More precisely, it has been proven to be an NP-hard problem [2]. Nevertheless, the advances in the algorithm designs [3], the runtime parallelization strategies [4], the available computational resources [5] and the reduction of network latencies [6] has made possible to attack this kind of research subjects in the last years.

The *Workflows and Distributed Computing* group of the *BSC* treats this kind of problems through COMPSs [**compss**] [7], a task based programming model trying to minimize the modifications in the code between a sequential and a distributed execution. This is achieved through decorator definitions following the OpenMP [8] philosophy. Even if its runtime has demonstrated to be more efficient than Spark with task based applications [9], complicated DAGs with a big amount of resources have reached the current limit of the COMPSs scheduling capabilities.

There are different approaches to confront the problem. The first one is to improve the runtime system in such a way that, given a certain DAG, the performance obtained is as good as possible. On the other hand, it is possible to change the user code in such a way that the generated DAG has more potential parallelism and the granularities obtained fits better the characteristics of the used framework.

## 1.2 Context

The current project is conducted as the *Final Master Thesis* in the *Master of Innovation and Research in Informatics - High Performance Computing (MIRI - HPC)* offered by the *Universitat Politecnica de Catalunya (UPC)* [10] and has been funded by the *Barcelona Supercomputing Center (BSC)* [11].

The project has been developed as a junior research engineer in the *Workflows and Distributed Computing* group [12] of the *Computer Science* department at the *BSC*. The main goal of this group is to ease the development of applications in clusters through the *COMP Superscalar (COMPSs)* [**compss**] programming model.

In addition, the projects into which the project has been done were in collaboration with the *Computational Genomics* group [13] of the *BSC* and CIMNE [14] *International Centre for Numerical Methods in Engineering*.

## 1.3 Objectives

This master thesis aims to show how performance can be improved both modifying the applications and improving the scheduling systems, trying to reach the best possible performance for a given application.

### 1.3.1 Detailed Objectives

The following points summarize the main goals of the project:

**O1** Include new features and optimize the code and task definitions in a GWAS workflow

**O2** Design a good methodology to implement a MLMC application

**O3** Introduce improvements in the scheduler to scale as well as possible both when increasing the resources and the amount of tasks in the DAG

In both cases they did the scientific contribution, well defining the workflows to exectute and coding the functionalities to be done in each one of the tasks. My contribution has consisted in studying the workflows, and proposing and implementing the way the tasks are called In the following chapters the precise contributions due to any author will be described as clearly as possible.

## 1.4 Document Structure

The rest of the document is organized as follows. Chapter 2 gives an overview of the current state of the art of GWAS workflows, MLMC distributed excutions and Analytic workflows, describing the most used technologies and comparing them to what is supported in the most common Workflow Managers. Chapter 3 introduces the COMPSs programming model, describing its main features before our contribution. Chapter 4 introduces the workflows and contributions done in this master thesis. Chapter 5 accurately describes our contribution to provide the reader a closer look of the solutions that we have chosen to face each specific challenge. Chapter 6 evaluates the obtained results and the performance of our proposal. Chapter 7 provides a brief summary of the thesis and, finally, in Chapter 8 we discuss the conclusions and state the guidelines for the future work.

# Chapter 2

# State of the art

In this section, the state of the art of all the subjects treated in the document are presented.

## 2.1 Genomic workflows

A straightforward solution to execute genomic workflows in supercomputing infrastructures is by directly attacking the queue manager. Job Arrays are essentially a collection of batch jobs that must have the same initial options (e.g., number of processes, wall-clock time). Thus, the users define genomic workflows as one job array per workflow step. On the one hand, Job Arrays are built directly on top of queue systems (i.e., SLURM, LSF, PBS) providing a very low overhead. For instance, SLURM Job Arrays [15] can submit millions of tasks in milliseconds. However, on the other hand, the users must manually define each job array and the dependencies between them to preserve the full workflow. Moreover, the requirements of each job array (workflow step) are homogeneous for all its tasks. Advanced users may dynamically modify the requirements of each job array job, but it is not a common practice.

Regarding the GWAS workflows, the implementation presented in this master thesis is to my knowledge the one using larger scale systems. In addition, the code has proven its utility since there are already accepted papers that are using it [16]. Other GWAS implementations can be found in the literature, like the one presented in [17] based on NextFlow and on the Common Workflow Language (CWL) [18]. Although the authors claim to use HPC systems, according to the numbers given in the article, the number of CPUs used in the executions do not exceed a ten. Other workflows presented in the same paper use 192 cores. GWASpro [19] is a web server for the analyses of large-scale genome-wide association studies. While the article claim that the server uses around 1000 CPU cores, nor detailed information about the implementation is given, neither information about the efficiency achieved.

## 2.2 Multilevel Monte Carlo

The sequential codes from which we started working of the the Monte Carlo (MC) and Multilevel Monte Carlo (MLMC) algorithms [20], allow to compute statistical analysis of scalar and field Quantity of Interests (QoI). This quantity arise from the solution of a stochastic problem. MLMC presents some issues that will be further discussed that are overcome by the Continuation Multilevel Monte Carlo (CMLMC) algorithm, firstly introduced in [21].

Even if MC [22] is a really old and well known algorithm, multilevel approaches are much more recent [23]. In addition, the way the different moments are computed in the version presented in the previous paragraph are highly suitable to be paralellized.

## 2.3 Workflow managers

Conversely, task-based frameworks can be used as workflow managers to provide a richer set of features. However, this solution requires the applications to be developed specifically for them, and the system administrators to install the full framework stack.

On the one hand, some frameworks force the users to explicitly define the workflow by means of a recipe file or a graphical interface.

FireWorks [24, 25] defines complex workflows using recipe files in Python, JSON, or YAML. It focuses on high-throughput applications, such as computational chemistry and materials science calculations, and provides support arbitrary computing resources (including queue systems), monitoring through a built-in web interface, failure detection, and dynamic workflow management.

Taverna [26, 27] is a suite of tools to design, monitor, and execute scientific workflows. It provides a graphical user interface for the composition of workflows that are written in a Simple Conceptual Unified Flow Language (Scufl) and executed remotely by the Taverna Server to any underlying infrastructure (such as supercomputers, Grids or cloud environments). Similarly, Kepler [28, 29] also provides a graphical user interface to compose scientific frameworks by selecting and connecting pertinent analytical components and data sources. Furthermore, workflows can be easily stored, reuse and shared across the community. Internally, Kepler's architecture is actor-oriented to allow different execution models into the same workflow.

Also, Galaxy [30, 31] is a web-based platform for scientific analysis focused on accessibility and reproducibility of workflows across the scientific community. The users define scientific workflows through the web portal and submit their executions to a Galaxy server containing a full repertoire of tools and reference data.

On the other hand, other frameworks implicitly build the task dependency graph from the user code. Some opt for defining a new scripting language to manage the workflow. These solutions force the users to learn a new language but make a clear differentiation between the workflow's management (the script) and the processes or programs to be executed.

Swift [32, 33] is a parallel scripting language developed in Java and designed to express and coordinate parallel invocations of application programs on distributed and parallel computing platforms. The users only define the main application and the input and output parameters of each program, so that Swift can execute the application in any distributed infrastructure by automatically building the data dependencies. Nextflow [34] enables scalable and reproducible scientific workflows using software containers. It provides a fluent DSL to implement and deploy scientific workflows but allows the adaptation of pipelines written in the most common scripting languages.

Other frameworks opt for defining some annotations on top of an already existing language. These solutions avoid the users from learning a new language but merge the workflow annotations and its execution in the same files.

Dask [35] is a library for parallel computing in Python. Dask follows a task-based approach being able to take into account the data-dependencies between the tasks and exploiting the inherent concurrency. Dask has been designed for computation and interactive data science and integration with Jupyter notebooks. It is based on the dataframe data-structure that offers interfaces to NumPy, Pandas, and Python iterators. Dask supports implicit, simple, task-graphs previously defined by the system (Dask Array or Dask Bag) and for more complex graphs, the programmer can relay in the `delayed` annotation that supports the asynchronous executions of tasks by building the corresponding task-graph. Dask-jobqueue is an interface to execute Dask in large clusters while still supporting interactivity. However, the elasticity needs to be indicated by the programmer in the code. While Dask is very popular in data-science, the authors of this paper are not aware of its application to other areas.

Parsl [36] provides an intuitive way to build implicit dataflows by annotating "apps" in Python codes. In Parsl, the developers annotate Python functions (apps) and Parsl constructs a dynamic, parallel execution graph derived from the implicit linkage between apps based on shared input/output data objects. Parsl then executes apps when dependencies are met. Parsl is resource-independent, that is, the same Parsl script can be executed on a laptop, cluster, cloud, or supercomputer.

Since COMPSs [37] is the solution chosen in this project, a more extended presentation is done in the following chapter.

# Chapter 3

# COMPSs overview

COMPSs is a task-based programming model that aims to make easier the development of parallel applications, targeting distributed computing platforms. It relies on the power of its Runtime to exploit the inherent parallelism of the application at execution time by detecting the task calls and the data dependencies between them.

As shown in Figure 3.1, the COMPSs Runtime allows applications to be executed on top of different infrastructures (such as multi-core machines, grids, clouds or containers) without modifying a single line of the application's code. Thanks to the different connectors, the Runtime is capable of handling all the underlying infrastructure so that the users only define the tasks. It also provides fault-tolerant mechanisms for partial failures (with job resubmission and reschedule when task or resources fail), has a live monitoring tool through a built-in web interface, supports instrumentation using the Extrae [38] tool to generate postmortem traces that can be analyzed with Paraver [39] [40], has an Eclipse IDE, and has pluggable cloud connectors and task schedulers.

Moreover, since the COMPSs Runtime is written in Java [41], Python [42] syntax is supported through a binding. This Python binding is supported by a Binding-commons layer which focuses on enabling the functionalities of the Runtime to other languages (currently, Python [43] and C/C++ [44]). It has been designed as an API with a set of defined functions. It is written in C and performs the communication with the Runtime through the JNI [45]. In this master thesis both Java and Python versions has been used. Nevertheless, considering its more compact format, PyCOMPSs has been chosen to present the examples in the overview.

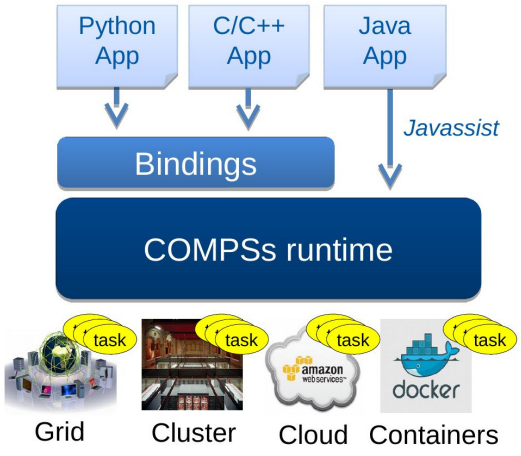



FIGURE 3.1: PyCOMPSs overview

Regarding the programmability, *Tasks* are identified by the programmer using simple annotations in the form of Python decorators, which indicate that invocations of a given method will become tasks at execution time. The `@task` decorator also contains information about the directionality of the method parameters specifying if a given parameter is read (IN), written (OUT) or both read and written in the method (INOUT).

Figure 3.2 shows an example of a task annotation in `Python`. The parameter `c` is of type INOUT, and parameters `a`, `b`, and `MKLProc` are set to the default type IN. The directionality tags are used at execution time to derive the data dependencies between tasks and are applied at an object level, taking into account its references to identify when two tasks access the same object. Furthermore, the `priority` tag is a hint for the PyCOMPSs' scheduler that will force to execute the tasks with this tag earlier, always respecting the data dependencies.

Additionally to the `@task` decorator, the `@constraint` decorator can be optionally defined to indicate some task hardware or software requirements. Continuing with the previous example, the task constraint `ComputingUnits` shows to the Runtime how many CPUs are consumed by each task execution. The available resources are defined by the system administrator in a separated `XML` configuration file. Other constraints that can be defined refer to processor architecture, memory size, etc.

```
@constraint(ComputingUnits="$ComputingUnits")
@task(c=INOUT, priority=True)
def multiply(a, b, c, MKLProc):
    os.environ["MKL_NUM_THREADS"]=str(MKLProc)
    c += a * b
```

FIGURE 3.2: Sample task annotation

A tiny synchronization API completes the PyCOMPSs syntax. As shown in Figure 3.3, the API function `compss_wait_on` waits until all the tasks modifying the `result`'s value are finished and brings the value to the node executing the main program. Once the value is retrieved, the execution of the main program code is resumed. Given that PyCOMPSs is used mostly in distributed environments, synchronizing may imply a data transfer from a remote storage or memory space to the node executing the main program. It is important to realize that when coding with COMPSs in Java, this API is no needed since the code is instrumented with Javassist [46].

```
for block in Data:
    presult = word_count(block)
    reduce_count(result, presult)
finalResult = compss_wait_on(result)
```

FIGURE 3.3: Sample call to synchronization API

## 3.1 COMPSs Runtime

The COMPSs Runtime handles the execution of the applications in the computing infrastructure. The computing infrastructure is composed of several heterogeneous nodes, and the execution is orchestrated following the master-worker paradigm, where the main program is started on the master node and tasks are offloaded to worker nodes. In the most general case, the node allocating the master node will also allocate a worker node.

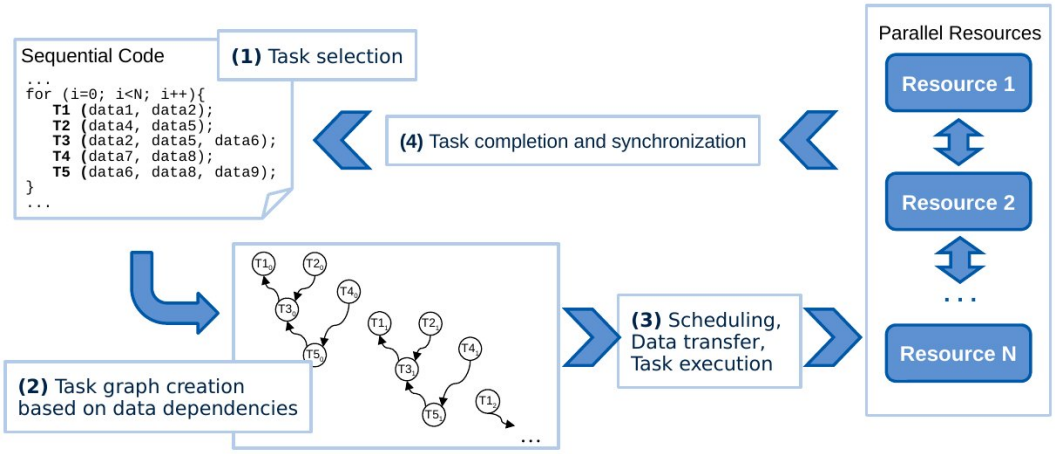



FIGURE 3.4: PyCOMPSs Task life-cycle

As depicted in Figure 3.4, the Runtime first builds a task graph adding a new node to it every time a task is invoked in the application's code. The directionality of the parameters is used to detect the data dependencies between the new task and previous ones. Secondly, the scheduler will analyze the generated graph in a particular way to execute all the workload among the available resources. We must highlight that this analysis is highly dependent to the different scheduler implementations, but the Runtime provides information about all the data locations so that they can exploit the *data locality*. Eventually, when a task is scheduled to a given resource, the required objects and files are transferred between different memory spaces to guarantee that tasks have available their parameters before execution. Finally, the task is executed in the worker resource and, when specified by the synchronization API, the results are gathered back to the master resource (where the main code is running).

Only concerning the PyCOMPSs binding, when a transfer between different memory spaces is required, the binding serializes and writes the object to disk using the standard library `Pickle`. The transfer between different resources is then delegated to the Runtime.

Finally, the available `Computing Units` that each resource can offer to the Runtime is configurable. More specifically, this allows to oversubscribe the amount of work that a resource can receive; meaning that the Runtime can create more threads than the real amount of CPUs that the resource has.

## 3.2 Interaction with external libraries

The PyCOMPSs Runtime supports the execution of multi-threaded tasks using the constraint interface. The number of cores assigned to a multi-threaded task can be indicated by the programmer in the `ComputingUnits` constraint tag. The PyCOMPSs scheduler can

assign several cores to a given multi-threaded task. On the other hand, although support for tasks that use several nodes has been added recently, in this work we only consider tasks executing inside a single node.

Before this work, the cores were assigned blindly to the tasks. The performance results observed were relatively poor when running numerical applications, such as those using the NumPy [47] or SciPy [48] libraries that link to the Intel®MKL library [49]. It has been shown that, by default, Intel®MKL tends to occupy the entire node when the multi-threading is enabled. Not considering this fact can result in a heavy oversubscribing. In addition, each task can be executed in several NUMA sockets. This fact increases the amount of transfers between the different NUMA-nodes, decreasing the performance dramatically. Knowing that this behavior can be found in other libraries, the problem has been solved in a general way.

We have modified the PyCOMPSs task executor in such a way that it is currently able to bind multi-threaded tasks to specific computing units of the infrastructure. This fact, combined with the capability to define the nodes' virtual amount of computing units, allows the user to achieve the desired rate of oversubscribing. However, this is not done blindly: the PyCOMPSs Runtime distributes the tasks evenly between the different NUMA sockets, avoiding at the maximum the transfers between memory spaces.

## 3.3 Scheduling infrastructure

PyCOMPSs Runtime has been extended with a scheduling infrastructure that supports pluggable scheduling policies. Almost all the tests presented in this paper are based on a *data locality* scheduler that takes into account the node that stores the data accessed by the tasks. More precisely, a task will have a score equal to the amount of input data present in a given node.

Defining a new score policy is enough to change the scheduler behavior. It will prioritize the tasks with the highest score for a given combination of resource, implementation, and data. In addition to the *data locality* score, three more policies have been defined: First In First Out (FIFO), Last In First Out (LIFO) and *data locality* with priority to tasks with a shared edge in the dependency graph with the finished task (FIFOData). In this last policy, there are two different scenarios. In the case where there are tasks freed by the job that has just finished, one of them is scheduled in First In First Out order; even before treating the tasks that are already free. Otherwise, *data locality* is considered between all the available tasks. The first two policies (LIFO, FIFO) have served to probe the robustness of the scheduling system. The third one can be seen as a relaxation of the *data locality* scheduler to lighten the amount of needed comparisons to schedule a task.

The available schedulers allow the users to configure the execution depending on the expected load. This master thesis tries to optimize the way the different schedulers are coded internally without changing its current behavior.

## 3.4 Python persistent workers

In previous Runtime versions, COMPSs was enhanced with a persistent Java worker, meaning that a Java worker process was started at the beginning of the application execution, communicating with the master to get information about the tasks to be executed and

data transfers to be performed. However, every time a Python task was invoked, a new Python interpreter was launched. This process has been enhanced with the implementation of Python persistent workers.

More in detail, the PyCOMPSs worker module has been modified on top of the Python's built-in multiprocessing library. When the application execution begins, the primary worker process in each worker node spawns a set of processes that will be responsible for executing the tasks. These processes are kept alive during the whole application execution and communicate with the Java persistent worker through pipes. The messages that they exchange include information about the task execution requests, job parameters, and computation results. This feature improves the overall performance by reducing the overhead of deploying a new Python interpreter per task. Besides, modules loaded by previous tasks are already present in the interpreter and do not need to be reloaded.

## 3.5 Methods' polymorphism

GPUs have demonstrated that can sometimes achieve better performance than CPUs. In fact, it is not always easy to decide whether it is better to use one architecture or the other [50]. Also, FPGAs are gaining some momentum. In this context, projects with the primary focus of interest on heterogeneous architectures are arising [51]. Hence, it seems reasonable to think that, in both HPC and Big Data contexts, we are going towards environments with heterogeneous architectures.

```
@implement(source_class="matmul_objects_MKL", method="multiply")
@constraint (ComputingUnits="${ComputingUnitsKNL}", ProcessorName="KNL")
@task(c=INOUT)
def multiplyKNL(a, b, c, MKLProcXeon, MKLProcKNL):
        os.environ["KMP_AFFINITY"]="disabled"
        os.environ["MKL_NUM_THREADS"]=str(MKLProcKNL)
        c += a * b

@constraint (ComputingUnits="${ComputingUnitsXEON}", ProcessorName="XEON")
@task(c=INOUT)
def multiply(a, b, c, MKLProcXeon, MKLProcKNL):
        os.environ["MKL_NUM_THREADS"]=str(MKLProcXeon)
        c += a * b

```

FIGURE 3.5: Version handling with PyCOMPSs

PyCOMPSs can manage those cases by providing support for the definition of different versions of the same method for different architectures. The programmer can use the `@implements` decorator to indicate that a method implements the same behavior than another. Figure 3.5 shows an example of polymorphism, which together with the `@constraint` decorator allows to indicate to the Runtime that some tasks can only be executed in a given set of computing resources. In fact, using polymorphism and tasks' constraints, the users can define CPU, GPU or FPGA versions of the same task.

Internally, at the beginning of the execution, the Runtime will blindly execute any of the available versions that can run in a given resource in order to obtain an execution profile per version. Afterwards, the Runtime is capable to use the profiled information to choose the implementation with the lowest execution time.

## 3.6 Profiling

PyCOMPSs generates *postmortem* traces under demand using Extrae [38]. These files can be explored with Paraver [39] [40], obtaining visual information to make easier the code performance fine tuning.

Some specific PyCOMPSs events have been added in order to differentiate the different steps done by the master and the workers. More precisely, it is possible to see the different actions performed by a worker each time that a task is executed.

Finally, the dependency graph generated can be plotted at the end of the computation or be explored on runtime with the monitor.

# Chapter 4

# Implementations

The main objective of this chapter is to present the workflows that has been optimized and lately used to test the performance improvements in the scheduler.

## 4.1 GWAS

The first application treated was mainly implemented by the *Computational Genomics* group at *BSC* [52]. It is a genome-wide association study. The exact procedure is still not published since the final results are still being generated. Thus, no further details about the exact content of the tasks are given in this memory. Indeed, it has no fully sense to deeply explain the content of the tasks since it is not my contribution at all. Instead, the way this tasks were called has been modified.

Nevertheless, it has to be noted that all the contributions done are agnostic from the exact application and could be applied to any kind of scientific workflow having similar characteristics to the ones described in this memory. This fact gives more strength to the contributions done since they can be applied elsewhere without any problem. Hence, the key points and common problems has been identified and solved as generally as possible.

Although the dependency graph of the application is not really complicated, the executed workflow is quite complex and difficult to debug for two main reasons:

1. It is not possible to have small debug executions
   With small datasets, some parts of the workflow are not useful since they rely on statistical studies that are not valid with a small population. Hence, the application must be debugged in production environments.

2. There is a big amount of different binaries implied
   A wide range of binaries are used in the workflow. This fact, combined with the previous point, makes really difficult to detect why some binaries get stuck at some point given the amount of files and data involved.

In the following subsections, the main characteristics of the application are described as long as the related contributions done in this master thesis.

### 4.1.1 Application characteristics

The first step was to really understand how the workflow behaved. Figure 4.1 shows a simplified representation of the DAG executed that already contains all the important information necessary to understand the application morphology. Not being able to have debug

executions, it became crucial to understand the different steps of the application. The application has three different input levels, which are the points of the workflow in which new input files are added. It is a simplification since, for instance, if several inputs are defined at the beginning (the red ones), the results are combined at the end. Nevertheless, we can consider that the different workflows depending on each of the red files are independent.

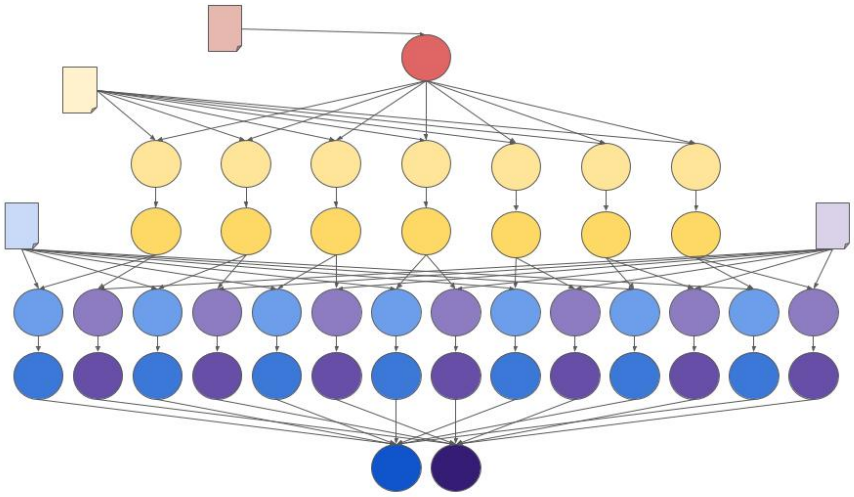


FIGURE 4.1: GWAS DAG

The workflow will have the following size depending on the amount of inputs:

1. Red inputs
   There will be as many DAGs as input red files.

2. Yellow inputs
   For each one of the red inputs, a variable amount of chunks is created. Each one of the chunks must perform some operations with each yellow input. Hence, considering $N_i$ the amount of chunks for the red input file $i$, the total amount of yellow pipelines will be $\sum_i N_i \cdot j$ where $j$ is the amount of input files in the second level.

3. Blue/purple inputs
   In the next step, each output of the previous pipeline must do some computations with the new inputs. Considering that there are $k$ inputs at this point, the amount of pipelines of the third kind is $\sum_i N_i \cdot j \cdot k$.

In fact, the key point here is that both the input and generated files are pretty big, so the interest is to make as many chunks as possible. On the other side, having too small chunks make the results incorrect, so there is a lower bound in the chunk size. Even with this bound, the amount of chunks is quite big. Depending on the $i$, $N_i$ goes from $51$ to $252$. Furthermore, the amount of input files in the first level goes from $1$ to $25$. Finally, the amount of inputs in the last step can be as big as wanted.

### 4.1.2 Binary complexity

As it has been previously said, the application is strongly binary based. More precisely, the following binaries have some use all along the workflow:

- SHAPEIT [53]
- Eagle [54]
- Impute [55]
- Minimac3 [56]
- Minimac4 [57]
- snptest [58]
- PLINK [59]
- QCTool [58]
- BCFTools [60]
- SAMTools [61]
- GTool [62]
- 4 different custom scripts in R [63]
- 3 different custom java functions

Although the exact function of each binary is not important regarding the contribution done in this master thesis, the binaries calls has been added to allow the lecturer understand its heterogeneity and complexity. Indeed, the amount of different functionalities orchestrated is pretty high. In addition, in the elaboration of this thesis, some binary version have been upgraded. This fact has implied changes both in the installations (compilation and dependency handling of the new versions) and the related code that was relying of concrete version formats.

#### 4.1.2.1 Heterogeneous Task Requirements

The different binaries of the workflow can have different hardware and software requirements. In general, this requirements are related to the amount of memory needed depending on the input. On change applied to the workflow is to put some of the constraints as global variables in such a way that they can be changed depending on the execution without needing to recompile the Java code. For instance, Figure 4.2 shows two different tasks from the workflow's annotated interface that have different CPU and memory requirements. The annotation to the `phasingBed` task indicates that any invocation to `phasingBed` will require 48 cores and 50.0 GB of memory. In production runs, a specific configuration file is exported depending on the used inputs to optimise the available resources as well as possible.

During this master thesis, both the constraints definitions and the configuration files used in the diferent production executions has been coded.

```
@Method(declaringClass = "guidance.GuidanceImpl")
@Constraints(computingUnits = "${snptestCU}",
   memorySize = "${snptestMem}")
void snptest(
    ...
);

@Method(declaringClass = "guidance.GuidanceImpl")
@Constraints(memorySize = "6.0")
void qctoolS(
    ...
);

```

FIGURE 4.2: Task annotations with different CPU and memory constraints

#### 4.1.2.2 Heterogeneous Binary Invocations

There are three main reasons that make the parametrisation of the binaries very complex. First, the file formats accepted by each binary are usually different because binaries are developed by different institutions and there is no standard file format. Although the differences are not significant (i.e., compression, number of columns, column order), the input files must be adapted before each binary execution.

Second, many binaries do not generate the output files when there is no content to write on them. This can cause cascading failures when being part of a static workflow since, typically, the output files of one binary are input files of another one.

Third, binaries may alter their behaviour depending on the provided command line arguments. Although this is not a problem itself, having executions of the same binary with a different number of parameters is, because application users want to invoke the same task definition regardless of the number of arguments.

Another example of this behaviour is the different merging operations to be done at the end. Depending on the ones that are added by the user, the call to the `R` scripts have a different amount of parameters.

In addition, it has to be noted that scientists are not necessarily computer science experts. Thus, it is a common practice to rely on state of the art software packages to perform specific operations efficiently. For instance, a workflow orchestrated in Python or Java can execute filtering and merging operations in the native language, imputations or associations using binary tools (written in C or C++), and generate plots using R.

To hide this heterogeneity a `Binary Wrapper` has been used. As shown in Figure 4.3, first, the wrapper parses the input parameters and builds the command line arguments; allowing to switch between different binary calls inside the same task definition. Second, checks the input files and re-formats them if necessary; allowing the task definition to accept

any input file format. Next, executes the binary command and, finally, checks the output files; generating any missing output file to prevent cascade failures. In order to avoid later failures, the correct headers for each type of binary are added when needed.

This encapsulation acts as an interface to design the workflow orchestration entirely in the same language, freeing the scientists from dealing explicitly with inter-language issues (such as binary spawning, and process input and output redirection).

The different Binary Wrappers can be encapsulated in a library as a set of building blocks and offered to the final programmer that defines the workflows. This methodology provides a unified interface that simplifies the definition of the workflows, while the complexity of the different components and its orchestration is performed behind the scenes. Even if this has not been done in this master thesis, the demonstration of the usefulness of this approach has been probed and will be taken into account in future developments.

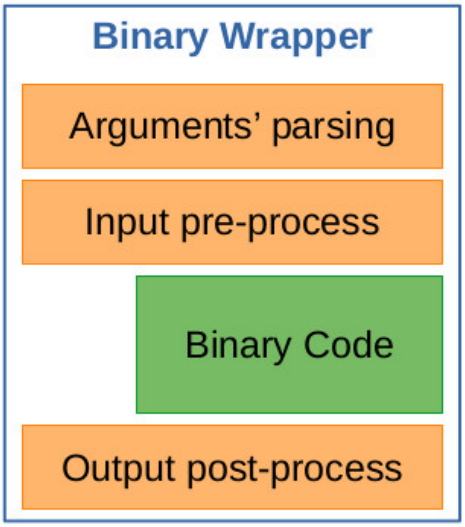



FIGURE 4.3: Binary wrapper to homogenise binary invocations

Regarding this aspect, the main contribution of this master thesis has been in the arguments parsing, pre and post process. Even if the binary calls were defined by the scientific experts, the changes in the binary versions and the addition of new functionalities have implied lots of changes in this envelope binary code.

### 4.1.3 Intelligent Workflow Execution

At this point, it is important to justify why a task based approach has been chosen instead of choosing a data centered framework.

As presented at the beginning of this section, applications composed by different analysis steps where different data-set are partially analysed against different input files has been considered. Each analysis step requires a different number of jobs, and each job can have different resource constraints and duration depending on the size of the analysed data-set and the size and complexity of the chromosome. This implies that the execution of the different partial analysis is unbalanced and the execution has to wait for the larger computation in each step. Implementing the different binary executions as tasks allows the runtime to detect the data dependencies between the different partial analysis. Hence, the application

can advance with the partial analysis from other steps that depend on these tasks without having to wait until the whole step is completed. The runtime can detect if the analysis of different steps can run in parallel allowing to interleave executions from these steps to balance executions and increase performance. Figure 4.4 shows the execution traces of the same execution. In the superior trace, all the tasks are presented. In the lower one, only the merging tasks are shown. Even if the merge is the step done in the last part of the study, it can be seen that the merging is performed during all the execution.

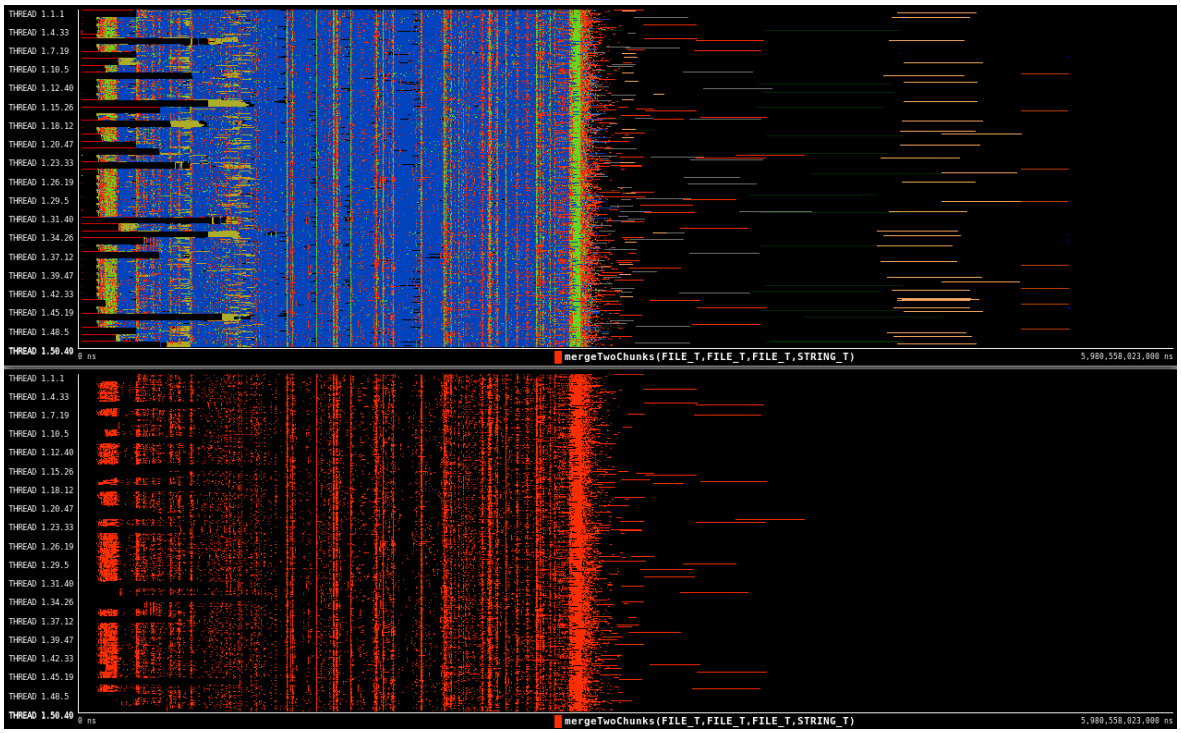



FIGURE 4.4: Execution trace of an execution showing all the tasks and only the last phase merging tasks

Furthermore, when looking at the duration of the tasks in Figure 4.5 we see a clear heterogeneity. Scheduling the tasks dynamically it is possible to better fill the gaps without the need to predict this stochastic durability in advance. This kind of graph show all the execution threads in the vertical axis. The horizontal axis is the execution time. The different taks are put in the graph depending on the thread that executed them and the elapsed execution time. In this case there are 4900 threads and the right limit corresponds to 4 minutes, that are 240 seconds.

### 4.1.4 Pipeline refactor

Looking at the heterogeneity on the duration of the tasks, the whole workflow has been called into question. Hence, the duration of the tasks has been deeply studied. Figure 4.6 shows the tasks that last from 0 to 13 seconds. It can be seen that there is a large group of tasks than last less than two seconds. Considering the amount of resources to handle, this is a really small granularity that may not well fit the optimum characteristics for COMPSs.

The original code of the workflow was developed considering as many tasks as possible in order to tune the exact granularity. This fact, theoretically, should allow to better fill the

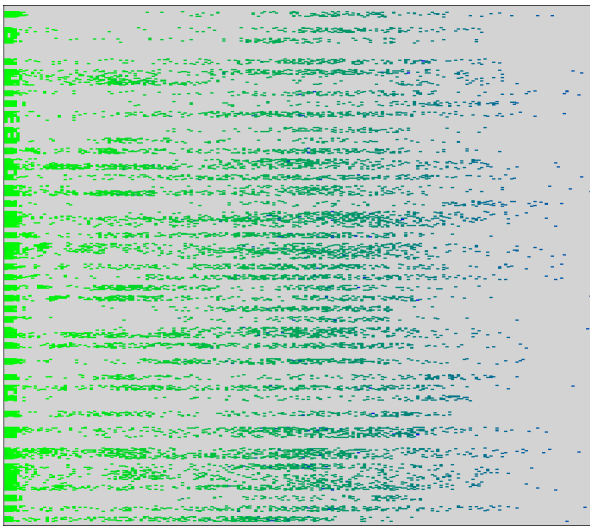


FIGURE 4.5: Main workflow task duration

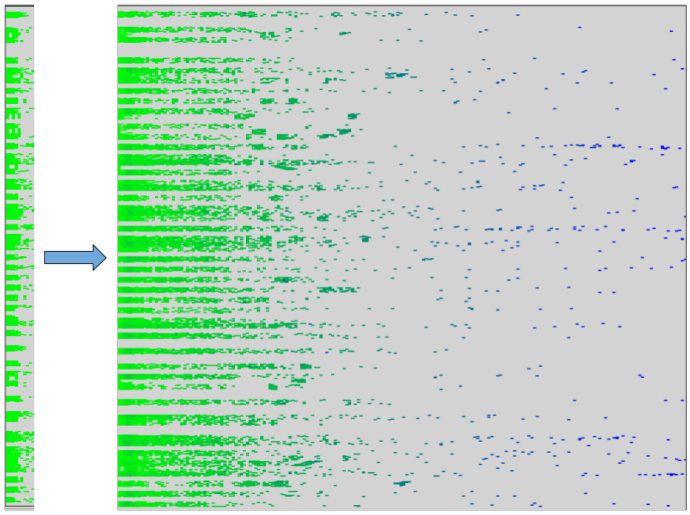


FIGURE 4.6: Shorter task's duration

available resources. Nevertheless, too small tasks can be imply scheduling problems when growing the amount of computational capabilities.

Hence, the granularity of the different tasks has been studied. Given that in the second and third pipelines almost all the time is due to the main functionality, it has been decided to merge all the pipeline in a single task when all the tasks must be executed. Figure 4.7 shows squares indicating the different tasks that have been encapsulated in single remote calls.

It is important to realize that it is possible to launch parts of the workflow depending on the input configuration file. This improvement does not prevent from launching partial execution of the unified pipelines. Instead, when launching partial executions the tasks are called independently to allow stopping the pipelines exactly at the desired point.

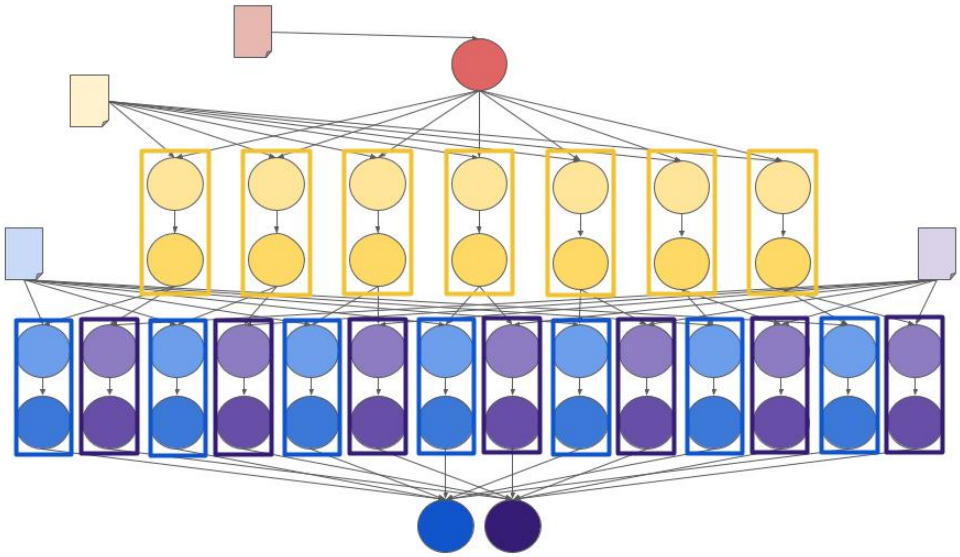
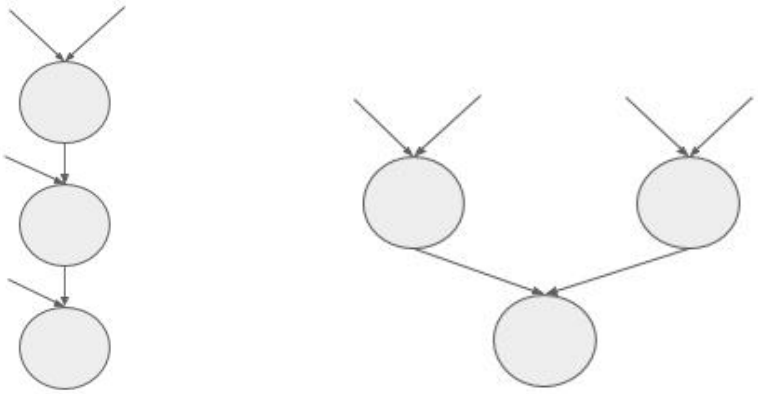


FIGURE 4.7: Resulting workflow after pipeline regrouping



(A) Sequential merge (B) Tree merge

FIGURE 4.8: Merge strategies

### 4.1.5 Merge refactor

Regarding the merging process of the different tasks, in the original code it was done in a sequential way as shown in Figure 4.8a. This fact created large tails at the end of the execution. In order to solve this fact, the merge is now done in a tree manner as shown in Figure 4.8b.

### 4.1.6 Containerization

Considering the amount of binaries implied, the deployment of the applications has demonstrated to be really complicated in the past. In order to mitigate this problem, one of the contributions done in the master thesis is the containerization of the whole workflow. First of all, a docker [64] image has been created. Nevertheless, it has to be kept in mind that this workflow is intended to run in HPC clusters. Hence, there is no `sudo` [65] in the environments where the workflow is supposed to run. This is why once the image is

created, it is pushed into a local repository. From this docker image, a singularity [66] image is created. Finally, this is the container that has been successfully used to run the production executions.

### 4.1.7 Cloud execution

Finally, even if the main usage of the workflow has been done in HPC facilities. The bioinformatics community is switching progressively from clusters to the cloud. An other contribution of this master thesis is the execution of the workflow in a cloud computing [67] provider.

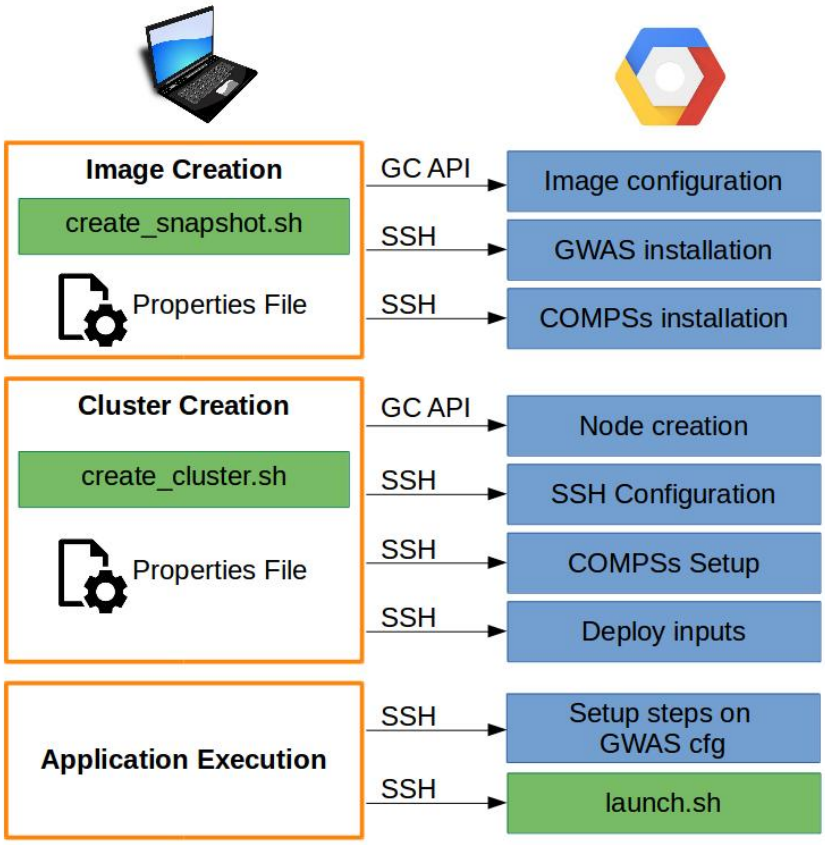



FIGURE 4.9: Cloud setup for the execution of GUIDANCE with COMPSs.

I have created a set of scripts to create a base instance containing GUIDANCE, COMPSs, and all its dependencies, create and set up a cluster of virtual machines, and execute the workflow. Figure 4.9 shows the architecture of these scripts when using the Google Cloud Platform [68] as cloud backend. Although this set of scripts is created to ease the configuration of cloud environments, the strong point is that once the virtual machines are configured, the same application that was running in supercomputing architectures can be executed on the cloud environment.

Special thanks to Cristian Ramon-Cortés Vilarrodona who took my separated and merely working scripts and refactored them to reach a much more professional, modular and, reusable result.

## 4.2 MLMC

Similarly to the previous section, this section has also been done in the context of a collaboration with an external institution. More precisely, both MC and MLMC has been reviewed, understood and coded in a sequential way by Riccardo Tosi under the supervision of Riccardo Rossi in the context of the exaQUte project [69]. In addition, all the simulations are done with the Kratos [70] solver. All the resulting code is already pushed and available in github [71].

The contribution of this master thesis has consisted in modify the code to enhance its parallel capabilities. It is important to note that the comprehension level needed to fresh code the presented algorithms and the one needed to modify them is quite different. Hence, the contribution of this master thesis consists in understanding the code enough to be able to propose modifications and enhancements while keeping the correctness of the results.

In order to better understand the improvements proposed, a short overview of both MC and MLMC algorithms is done at the beginning of this section.

### 4.2.1 Monte Carlo algorithm overview

The MC method is the reference technique in the stochastic analysis of multi-physics problems with uncertainties in the data parameters. This technique gives origin to a wide class of different algorithms, whose main idea is to repeat many times the simulation with different known r.v. $w$ each time; this leads to an accurate estimation of the statistics of the QoI. We consider the approximation $QoI \simeq QoI_M$, moreover, since the relation $QoI_M = f(u_M)$ holds, the MC estimator for the expected value of the QoI $\mathbb{E}[QoI_M]$ is:

$$\mathbb{E}^{\text{MC}}[QoI_M] := \frac{1}{N}\sum_{i=1}^{N} QoI_M(w^{(i)}) \,, \tag{4.1}$$

where $QoI_M(w^{(i)})$ and $i = 1, \dots, N$ are $N$ independent, identically distributed (i.i.d.) values of the QoI computed for the mesh $\Omega_M$. The MC potential lies in its basic property of convergence to the exact statistics of the solution as the number of input samples tends to infinity, independently from the dimensionality of the stochastic space and mostly independently from the physics of the problem under consideration. It also has the advantage of being considered as a "black box", since it is non-intrusive and directly applicable to any simulation code.

The MC estimation accuracy of the expectation can be evaluated through the mean square error, that reads as follows

$$\begin{aligned} \text{mse}^2_{\text{MC}} :=& \mathbb{E}[(\mathbb{E}^{\text{MC}}[QoI_M] - \mathbb{E}[QoI])^2] \\ =& (\mathbb{E}[QoI_M - QoI])^2 + \frac{\mathbb{V}\text{ar}[QoI_M]}{N} \,, \end{aligned} \tag{4.2}$$

where $\mathbb{V}\text{ar}[QoI] = \mathbb{E}[QoI^2] - \mathbb{E}[QoI]^2$ stands for the variance of the QoI. The term $(\mathbb{E}[QoI_M - QoI])^2$ is the bias or discretization error (B), it is independent from the statistics of the QoI and only depends on the level of accuracy of the grid we are exploiting to approximate $QoI$ with $QoI_M$. On the other hand, $\frac{\mathbb{V}\text{ar}[QoI_M]}{N}$ is the statistical error (SE), which decreases as long as the number of samples grows, and is an indicator of the variance of our estimator.

Unfortunately, one of the main drawbacks of the MC method is its too high computational cost for the stochastic analysis of industrial problems with complex geometries.

### 4.2.2 Multilevel Monte Carlo algorithm overview

MLMC algorithm gives origin to a broad class of algorithms, which try to overcome the limitations of the father MC. The main idea of the MLMC algorithm is to draw many MC instances simultaneously on a set of grids with increasing accuracy. The different grid refinement generates levels of accuracy. This means that the mesh parameter $M$ grows as long as the level increases, i.e. $M_0 < M_1 < \cdots < M_L$, where $L$ is the maximum number of levels the current simulation may reach. Due to the linearity of the expectation operator, the mean of the QoI may be written as a telescopic sum of the expectations of the QoI on the coarser levels. In fact, the expected value of the QoI of mesh $\Omega_{M_L}$ is:

$$\begin{aligned}\mathbb{E}[QoI_{M_L}] &= \mathbb{E}[QoI_{M_0}] + \sum_{l=1}^{L} \mathbb{E}[QoI_l - QoI_{l-1}] \\ &= \sum_{l=1}^{L} \mathbb{E}[Y_l] \,,\end{aligned} \tag{4.3}$$

where $Y_l = QoI_{M_l} - QoI_{M_{l-1}}$ and $Y_0 = QoI_{M_0}$. Similarly to the MC case, the MLMC estimator for the expected value of the QoI is:

$$\begin{aligned}\mathbb{E}^{\text{MLMC}}[QoI_M] &:= \sum_{l=0}^{L} \frac{1}{N_l} \sum_{i=1}^{N_l} Y_l(w^{(i,l)}) \\ &= \sum_{l=0}^{L} \mathbb{E}^{\text{MC}}[QoI_{M_l} - QoI_{M_{l-1}}] \,.\end{aligned} \tag{4.4}$$

One important observation is that the two QoI $QoI_{M_l} - QoI_{M_{l-1}}$ are computed using the same sample $w$. Analogously to the MC algorithm, the mean square error of the MLMC expectation estimator is the sum of a discretization error and a statistical error, in fact

$$\begin{aligned}\text{mse}^2_{\text{MLMC}} &:= \mathbb{E}[(\mathbb{E}^{\text{MLMC}}[QoI_M] - \mathbb{E}[QoI])^2] \\ &= (\mathbb{E}[QoI_M - QoI])^2 + \sum_{l=0}^{L} \frac{\mathbb{V}\text{ar}[Y_l]}{N_l} \,,\end{aligned} \tag{4.5}$$

where $(\mathbb{E}[QoI_M - QoI])^2$ is the bias, and $\sum_{l=0}^{L} \frac{\mathbb{V}\text{ar}[Y_l]}{N_l}$ the statistical error.

We can observe matching equations equations (4.2) and (4.5) that the only difference in the $\text{mse}$ evaluation is the statistical contribute, which is supposed to decrease as long as the mesh parameter $M$ grows. In fact, three important considerations lie at the basis of both algorithms:

i) the cost of computing one sample $QoI_{M_l}$ grows with the level accuracy $M_l$,

ii) $|\mathbb{E}[QoI_{M_l} - QoI]|$ decreases as $M_l$ grows,

iii)
- MC: $\mathbb{V}\text{ar}[QoI_M]$ more or less constant w.r.t. $M$,
- MLMC: $\mathbb{V}\text{ar}[Y_l]$ decreases as $M_l$ grows.

The evaluation of the QoI's cost, bias and variance builds a list $\mathcal{P}$ of parameters required to compute the optimal hierarchy, i.e. number of levels $L$ and number of samples per level $N_l$ , $l = 0, \cdots, L$. The analysis of the $\mathrm{mse}$ was useful to highlight the differences between the two algorithms analyzed, and how differently the variance behaves. On the other hand, this implementation checks different ideas to verify the convergence of the algorithms.

### 4.2.3 Convergence criteria

Convergence is accomplished if the estimator of the expected value ($\mathbb{E}^{\mathrm{MC}}[QoI_M]$ or $\mathbb{E}^{\mathrm{MLMC}}[QoI_M]$) achieves a desired tolerance $\varepsilon$ w.r.t. the true estimator $\mathbb{E}[QoI]$ with a confidence of $1-\phi$. In other words, we define a probability of failure (or error probability), and we want this probability to fail with a certain confidence.

For the MC algorithm, the probability of failure is defined by:

$$\mathbb{P}[|\,\mathbb{E}^{\mathrm{MC}}[QoI_M]| - \mathbb{E}[QoI] < \varepsilon] \leq \phi\,, \quad \phi << 1\,. \tag{4.6}$$

Two different convergence criteria for the MC algorithm are considered, the first one [72] arises from the Central Limit Theorem and relies only on the sample variance and the sample mean. Convergence is achieved when

$$2(1 - \frac{\Phi(\sqrt{N}\varepsilon)}{\bar{\sigma}_N}) < \phi\,, \tag{4.7}$$

where $\Phi$ is the Cumulative Distribution Function (CDF) of the standard normal distribution[1], $N$ the number of i.i.d. samples available, $\bar{\sigma}_N$ the sample variance[2] and $\phi$ the confidence of achieving the desired tolerance. This stopping criteria works well in the asymptotic regime, when $\varepsilon \to 0$, but in the non-asymptotic regime, when both $\varepsilon$ and $\phi$ are greater than $0$, this second moment criteria may fail. In [72], the authors propose an improvement of the second order stopping criteria of equation (4.7), and to evaluate the convergence they exploit also higher order moments (up to the fourth central moment). The goal is still to accomplish equation (4.6), but now a penalty term is added to equation (4.7). In fact, the stopping criteria based on higher moments reads as

$$2(1 - \frac{\Phi(\sqrt{N}\varepsilon)}{\bar{\sigma}_N}) + \text{penalty} < \phi\,, \tag{4.8}$$

where the new penalty term is function of higher moments. This gives an increasing reliability to the sequential MC algorithm for the non-asymptotic regime.

On the other hand, we refer to [21, 20] for the MLMC convergence criteria, which satisfies equation (4.6) in the asymptotic regime. The total error (te) can be bounded, with probability $(1-\phi)$, by the sum of the bias and the statistical error:

$$\begin{aligned} \text{te} :=& |\,\mathbb{E}^{\mathrm{MLMC}}[QoI_M] - \mathbb{E}[QoI]| \\ \leq& |\,\mathbb{E}[QoI - QoI_M]| + |\,\mathbb{E}^{\mathrm{MLMC}}[QoI_M] - \mathbb{E}[QoI_M]| \\ \leq& |\,\mathbb{E}[QoI] - \mathbb{E}[QoI_M]| + \mathcal{C}_\phi\, \mathbb{V}\mathrm{ar}[\mathbb{E}^{\mathrm{MLMC}}[QoI_M]]\,, \end{aligned} \tag{4.9}$$

[1] $\Phi(x) = \frac{1}{2}[1 + \mathrm{erf}(\frac{x}{\sqrt{2}})]$, where $\mathrm{erf}(x)$ is the error function is the CDF function for the normal distribution $\mathcal{N}(0,1)$.

[2] $\bar{\sigma}_N = \frac{1}{N-1}\sum_{i=1}^{N}(QoI(w^{(i)}) - \mathbb{E}[QoI])^2$

where

$$\mathcal{C}_\phi = \Phi^{-1}(1 - \frac{\phi}{2}) \tag{4.10}$$

$$\mathbb{V}\text{ar}[\mathbb{E}^{\text{MLMC}}[QoI_M]] = \sqrt{\sum_{l=0}^{L} \frac{\mathbb{V}\text{ar}[Y_l]}{N_l}} \,. \tag{4.11}$$

Since the te can be seen as the sum of discretization and statistical errors, we can set an upper bound for both. Therefore we introduce a splitting parameter $\theta \in (0, 1)$ s.t.

$$\text{B} = |\,\mathbb{E}[QoI] - \mathbb{E}[QoI_M]| \leq (1 - \theta)\phi \tag{4.12}$$

$$\text{SE} = \mathbb{V}\text{ar}[\mathbb{E}^{\text{MLMC}}[QoI_M]] \leq (\frac{\theta\phi}{\mathcal{C}_\phi}) \,, \tag{4.13}$$

and we remark that $\theta = \theta(\varepsilon_{it}, L)$ for the CMLMC algorithm, thus it is not constant and user-defined as for the MC and the MLMC algorithms.

### 4.2.4 Description of the algorithms

The sequential MC algorithm behaves as follows:

```
while loop until convergence:
  if iteration = 1:
    set initial hierarchy
  elif iteration > 1:
    update number of samples
  generate QoI values (re-use active)
  update expectations and variances
  check stopping criterion
```

FIGURE 4.10: Sequential MC algorithm

The CMLMC algorithm behaves as follows:

The most important thing to realize at this point is that considering the previous subsections and the results in which they are based [20], the first four moments considered to compute the convergence can be expressed as a combination of the following parameters:

- $S_1 = \sum_i QoI_i$
- $S_2 = \sum_i QoI_i^2$
- $S_3 = \sum_i QoI_i^3$
- $S_4 = \sum_i QoI_i^4$

This is indeed an embarrassingly parallel operation depending on the simulation results that can be performed in whichever order [73].

```
screening phase:
  set initial hierarchy
  loop on levels:
    generate QoI values for initial hierarchy
  compute costs, expectations and variances
  fit cost, bias and statistical error models (LSQ fit)
  compute Bayesian variance estimation
while loop until convergence:
  update tolerance
  compute optimal levels, splitting parameter and optimal number samples per level
  loop on levels:
    generate QoI values (re-use active)
  update costs, expectations and variances
  fit cost, bias and statistical error models (LSQ fit)
  update Bayesian variance estimation
  estimate total error = bias + statistical error
  check stopping criterion

```

FIGURE 4.11: CMLMC algorithm

### 4.2.5 Improvements

Indeed, once the algorithms have been understood, the contribution of this master thesis has consisted in proposing improvements in the code in order to better take advantage of the potential parallelism. The starting point of the work done is shown in Figure 4.12.

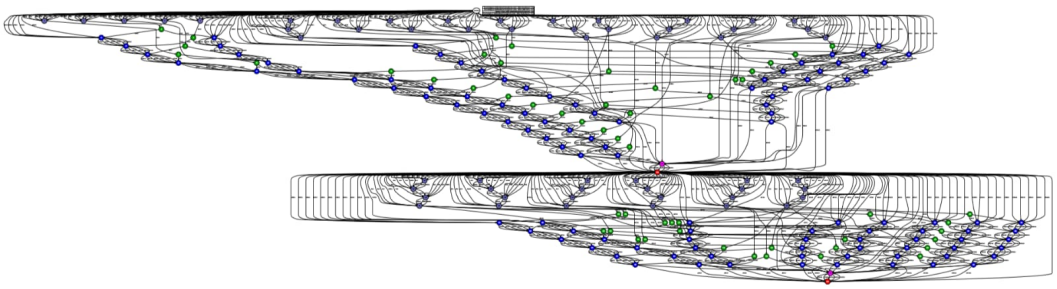


FIGURE 4.12: Initial Multilevel Montecarlo dependency graph

All the reduces are done following a sequential schema and the convergence is checked once all the executions of a single batch are finished. In this case, two iterations are computed. In addition, the computation of the convergence must wait until all the simulations have finished. This fact can lead to resource wasting in case an other iteration is needed since no new simulations are started until the convergence has been computed.

All the improvements has been introduced in the MC implementation. Having a single level it was much more easy to debug. Nevertheless, the code has been written considering levels. This way, all the improvements done are much more easy to port to the multilevel case. Indeed, we have considered all the data structures needed. Nevertheless, only one level is present in each one of them.

#### 4.2.5.1 Tree merge

First of all, the sequential merges has been changed to tree merges, in an analogous way to what has been explained in subsection 4.1.5. In this case, and since the programming language chosen is Python, its `*args` [74] capabilities has been taken into account. This fact

can increase both the parallelism and the granularity to better fit the PyCOMPSs needed characteristics.

#### 4.2.5.2 Batch design

Once the accumulation was done in a wiser way, the next step has consisted in launch different execution batches at once, accumulating the result as they finished. Figure 4.13 shows a dependency graph with three different batches.

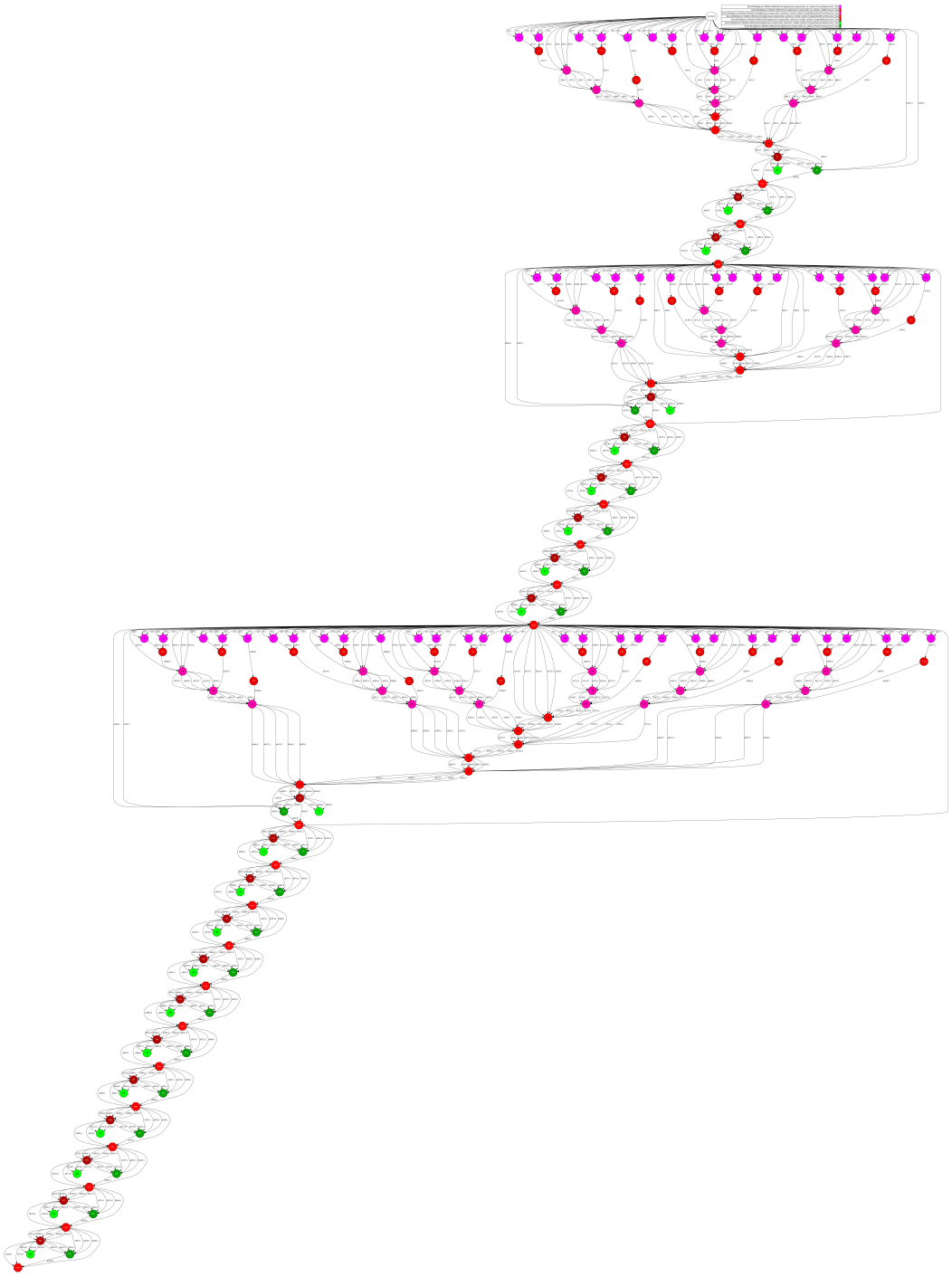


FIGURE 4.13: Batched Multilevel Montecarlo dependency graph

Although the spawning of the tasks was already done by batches, at this level the convergence of all the batches spawned at once was done in a sequential code without the capability to spawn new tasks. Figure 4.14 shows this fact. This is mainly because this code was coded in a really objected oriented manner. Each one of the modifications done implied a full refactor of the code.

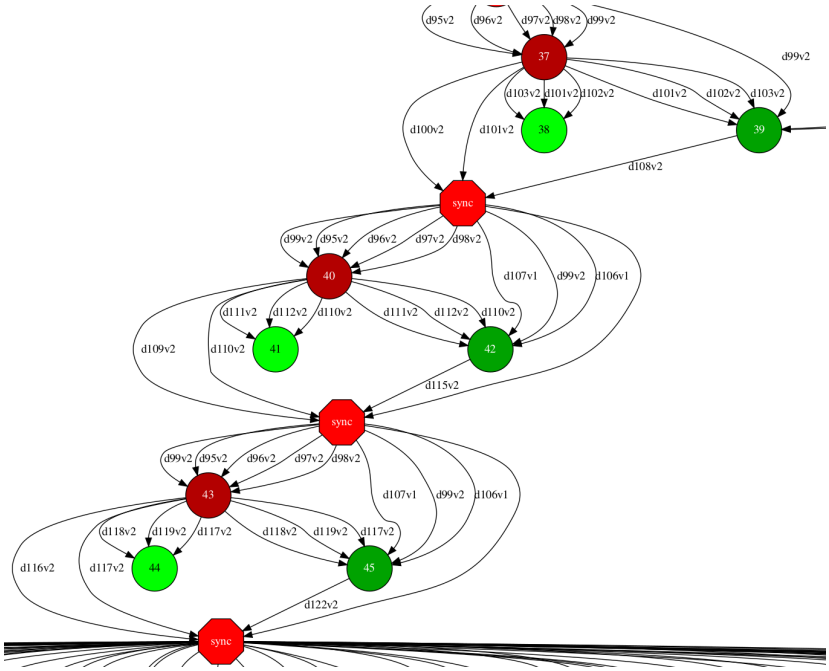



FIGURE 4.14: Convergence check of the batched Multilevel Montecarlo

Finally, and in order to be able to spawn new batches as the partial convergences where reached, an other refactor has been done. This way, we can consider that there are two levels of parallelism. The parallelism achieved inside each batch that allow to independently launch the different simulations and the parallelism achieved between independent batches.

Figure 4.15 shows the final workflow, where convergence is checked as long as the batches finish their execution so new simulations can be launched in case the convergence is not achieved in the next iteration. Since the reservations are static, we launch preventive computations in order to better use the allocated resources.

#### 4.2.5.3 Full stack deployment

The last contribution of this master thesis regarding this section concerns the testing of the resulting code. In order to test its behavior, the full stack containing Kratos, PyCOMPSs and all its dependencies has been deployed both in MareNostrum IV [75] and Salomon [76]. For PyCOMPSs, all the dependencies can be found in appendix A.1. In the Kratos side, the most important dependencies are Boost [77], MMG [78], Metis [79] and Trilinos [80]. In this stage, some problems with the OpenMP pragmas has been encountered with Intel's c++ [81] compiler [82] so the compilation scripts has been modified in order to solve this problems. At this point, the help from MareNostrum IV support team [83] has been crucial.

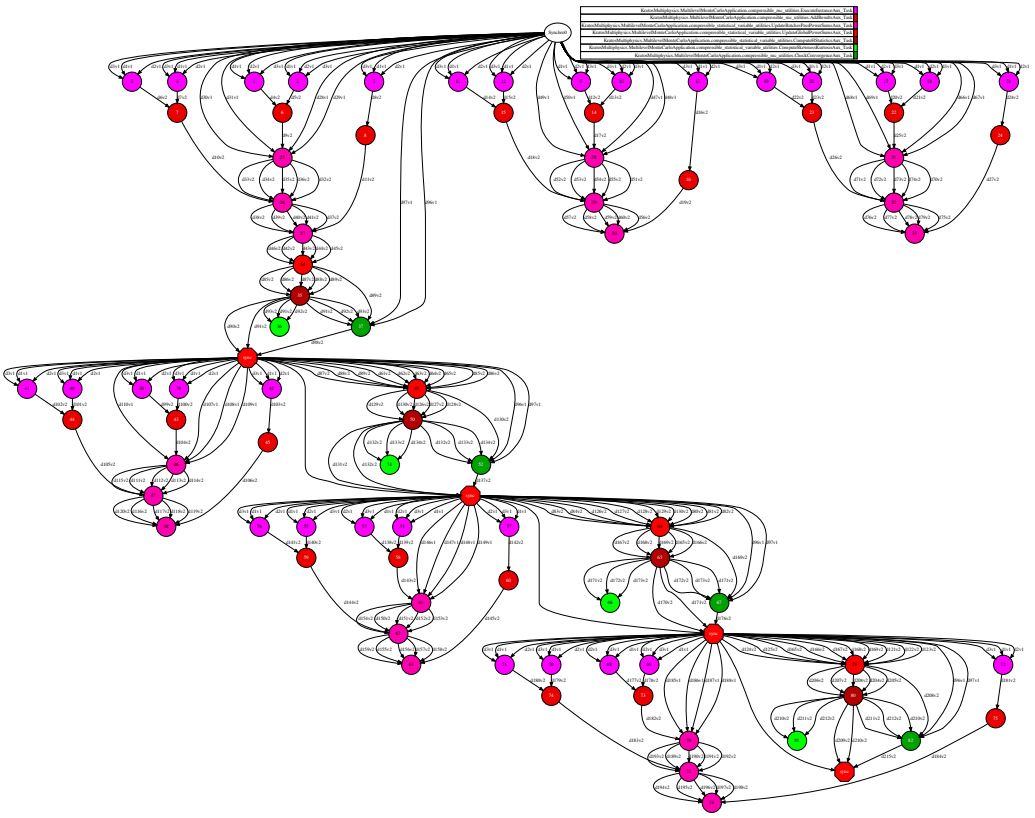


FIGURE 4.15: Final batched Montecarlo dependency graph

## 4.3 Runtime improvements

Finally, once all the user code had been improved, execution as large as possible have been launched. At this point, some performance issues has been detected. For example, Figure 4.16 shows an execution of the MC algorithm with 7920 simulations with 15 worker nodes. It is clear that the resources stay idle during an astonishingly high amount of time. This behavior is reduced when changing the task granularities and can be shown in Figure 4.17. Nevertheless, the execution trace still shows a lot of black spaces were resources are being wasted.

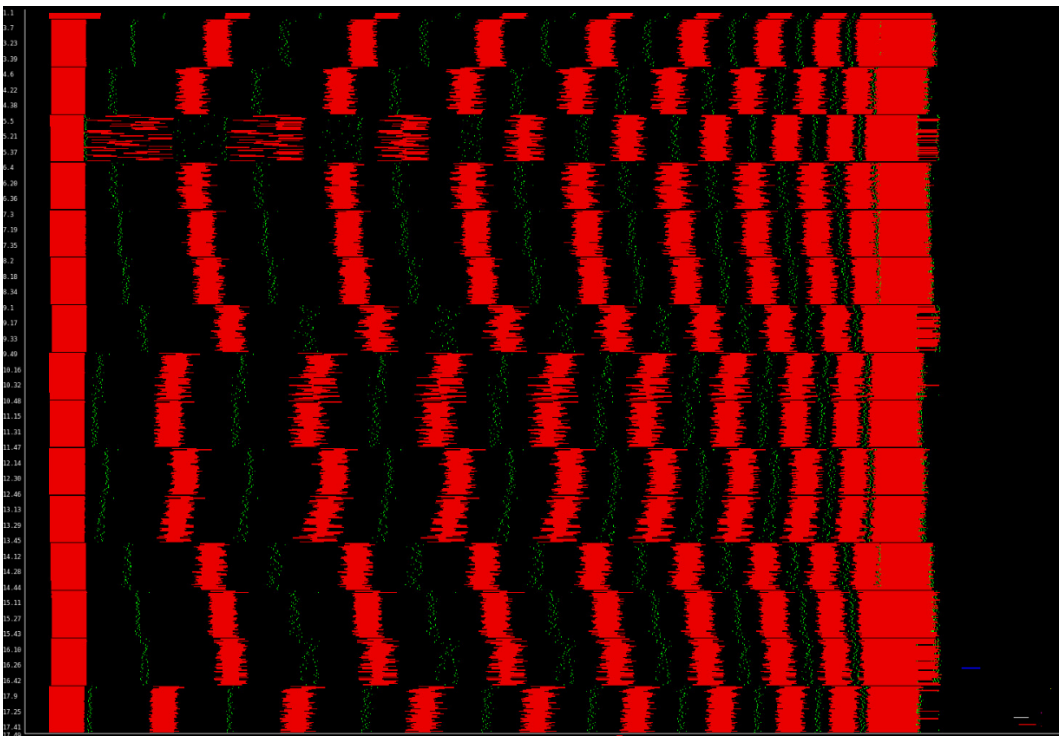


FIGURE 4.16: 7920 simulations MC execution with 15 MN IV worker nodes

As it has been previously said, big data executions have a huge amount of tasks. In addition, HPC environments network connections are really fast and there are underlying systems that can handle the data sharing in a really efficient way [84] [85]. Thus, the most important thing is to reduce scheduling latencies as much as possible.

### 4.3.1 Problem diagnosis

On the first hand, the current scheduler has been studied in order to detect where it could be improved. COMPSs has several scheduling policies available. The focus has been put in the ready scheduler, which only takes into account the tasks that have already been freed from dependencies, that are ready to be executed in a given moment. Figure 4.18 shows the code corresponding to the starting point at this stage. This function is called every time that a tasks frees an execution slot.

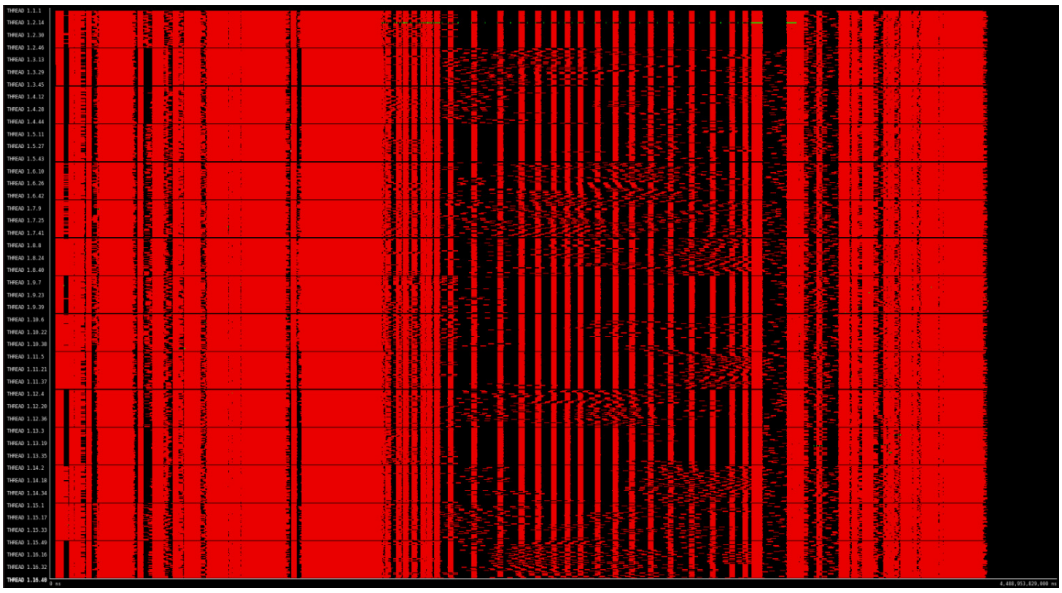


FIGURE 4.17: 30000 simulations MC execution with 15 MN IV worker nodes

```
private <T extends WorkerResourceDescription> void tryToLaunchFreeActions(List<
    AllocatableAction> dataFreeActions,
        List<AllocatableAction> resourceFreeActions, List<AllocatableAction>
    blockedCandidates,
        ResourceScheduler<T> resource) {

    // Try to launch all the data free actions and the resource free actions
    PriorityQueue<ObjectValue<AllocatableAction>> executableActions = new PriorityQueue
    <>();
    for (AllocatableAction freeAction : dataFreeActions) {
        Score actionScore = generateActionScore(freeAction);
        Score fullScore = freeAction.schedulingScore(resource, actionScore);
        ObjectValue<AllocatableAction> obj = new ObjectValue<>(freeAction, fullScore);
        executableActions.add(obj);
    }
    for (AllocatableAction freeAction : resourceFreeActions) {
        Score actionScore = generateActionScore(freeAction);
        Score fullScore = freeAction.schedulingScore(resource, actionScore);
        ObjectValue<AllocatableAction> obj = new ObjectValue<>(freeAction, fullScore);
        if (!executableActions.contains(obj)) {
            executableActions.add(obj);
        }
    }

    while (!executableActions.isEmpty()) {
        ObjectValue<AllocatableAction> obj = executableActions.poll();
        AllocatableAction freeAction = obj.getObject();
        Score actionScore = obj.getScore();

        // LOGGER.debug("Trying to launch action " + freeAction);
        try {
            scheduleAction(freeAction, actionScore);
            tryToLaunch(freeAction);
        } catch (BlockedActionException e) {
            blockedCandidates.add(freeAction);
        }
    }
}
```

FIGURE 4.18: Original COMPSs ready scheduler

There are three main things that could be improved:

- Keep track of the available worker nodes
  The scheduler try to execute the free task in all the resources, verifying the available slots for each one of them. Hence, the complexity increases linearly with the amount of available resources. In addition, the scheduler tries to execute all the available tasks even when no available resources are filled.
- Parallelize the scheduling process
  The scheduling process is done in a single thread. Hence, all the tasks are treated sequentially.
- Avoid computing the ordered list with the actions regarding each resource each time that a slot is freed
  Assuming that the scores associated to each pair resource - task do not vary all along the execution, this ordered list could be maintained between calls. This fact can avoid reordering a list as long as the amount of available tasks each time. For big executions, this list can contain tens of thousands of tasks. Thus, this fact cannot be overlooked.

### 4.3.2 Implementation proposed

Considering the problems detected in the previous section, a solution has been proposed that both included a multi-threaded treatment that keeps the ordered lists and a control of the available resources in order to stop the scheduling operations once all the resources were already filled. The main code corresponding to this part can be found in the appendix C.

Figure 4.19 shows how the main `while` has been changed and Figure 4.20 shows the chosen mechanism to asynchronously update the scheduler structures. This should be enough to briefly understand the basis on which the full implementation is based.

The implemented solution is based in three main ideas:

- Keep track of the available workers in a HashMap [86]
  This data structure has been chosen in order to guarantee a constant complexity access to this information.
- Keep a different list for each one of the resources
  This fact allows to store a list with the priority order of the available tasks for each one of the resources considering the chosen policy. Currently, they are FIFO, LIFO, data locality and load balancing (in case of equal data locality, tasks are sent to the workers with less workload).
- Spawn threads to update the scheduling structures
  Since each one of the resources has its own priority queue [87], its update can be done asynchronously and just wait for the result in case a certain resource frees a slot to perform some computations.

In order to achieve the desired behavior without making the code too complicated, a strategy based on tokens has been followed. This way, each time that a list must be modified, the thread in charge of this modification wait for the token corresponding to the last modification of each one of the lists. In addition, each time that a modification is added to the thread scheduler, the token is update so the next time that a new thread is spawned it

```
1
    Future<?> lastToken = this.resourceTokens.get(resource);
3   if (lastToken != null) {
      try {
5       lastToken.get();
      } catch (InterruptedException | ExecutionException e) {
7       e.printStackTrace();
        LOGGER.fatal("Unexpected thread interruption");
9       ErrorManager.fatal("Unexpected thread interruption");
      }
11  }
    this.resourceTokens.put(resource, null);
13
    Iterator<ObjectValue<AllocatableAction>> executableActionsIterator = this.
    unassignedReadyActions.get(resource)
15      .iterator();
    HashSet<ObjectValue<AllocatableAction>> objectValueToErase = new HashSet<
    ObjectValue<AllocatableAction>>();
17  while (executableActionsIterator.hasNext() && !this.availableWorkers.isEmpty()) {
      ObjectValue<AllocatableAction> obj = executableActionsIterator.next();
19    AllocatableAction freeAction = obj.getObject();
      try {
21      if (Tracer.isActivated()) {
          Tracer.emitEvent(Tracer.Event.TRY_TO_SCHEDULE.getId(), Tracer.Event.
    TRY_TO_SCHEDULE.getType());
23      }
        freeAction.tryToSchedule(obj.getScore(), this.availableWorkers);
25      if (Tracer.isActivated()) {
          Tracer.emitEvent(Tracer.EVENT_END, Tracer.Event.TRY_TO_SCHEDULE.getType());
27      }
        ResourceScheduler<? extends WorkerResourceDescription> assignedResource =
    freeAction
29          .getAssignedResource();
        tryToLaunch(freeAction);
31      if (!assignedResource.canRunSomething()) {
          this.availableWorkers.remove(assignedResource);
33      }
        objectValueToErase.add(obj);
35    } catch (BlockedActionException e) {
        ...
37    } catch (UnassignedActionException e) {
        ...
39    }
    }
41
```

FIGURE 4.19: Asynchronous scheduling structures update

waits to exactly the previous modifying thread. Afterwards, when the queue needs to be accessed, the main thread wait to the token corresponding to the last modification. This way, it is guaranteed that the queue contains all the modifications needed until this moment. This modifications are basically erasing tasks that are already running on an other resource and adding tasks that has been freed from the last resource scheduling. This way, the modification of the scheduling structures is removed from the scheduling critical path.

All the new implementation and small bug fixes done during the process can be found on the *exaQUte*'s branch of the COMPSs official github repository [88].

```
  private Runnable createAddRunnable(
2     final Map.Entry<ResourceScheduler<?>, TreeSet<ObjectValue<AllocatableAction>>>
    currentEntry,
      final AllocatableAction action, final Future<?> token) {
4   Runnable addRunnable = new Runnable() {
      public void run() {
6       if (token != null) {
          try {
8           token.get();
          } catch (InterruptedException | ExecutionException e) {
10          e.printStackTrace();
            LOGGER.fatal("Unexpected thread interruption");
12          ErrorManager.fatal("Unexpected thread interruption");
          }
14      }
        addActionToResource(currentEntry, action);
16    }
    };
18  return addRunnable;
  }
20
```

FIGURE 4.20: Asynchronous scheduling structures update

# Chapter 5

# Results and evaluation

To validate the proposed COMPSs features for supporting complex task based workflows, some experiments have been launched in order to evaluate the following features: (i) How efficient is the runtime on managing heterogeneity in the application; (ii) Scalability of the approach; and (iii) Portability of this approach.

At this stage, both applications presented in the previous chapter are used in order to demonstrate how the scheduler had an effect on both of them and how the modifications have affected the performance obtained. The improvements originated by the changes in the workflows and the scheduler are always mixed. This is mainly because since the applications scaled pretty well, the executions in production environments required a lot of resources. It was thought to be a useless waste of computing hours to launch worst executions just to find the contribution due to one factor and the other.

Nonetheless, the contributions of this master thesis were used as soon as they were ready. More precisely, a custom branch with the content of this master thesis has been installed in MareNostrum IV during all the development. This has allowed several users to enhance their computing capabilities when encountering scheduling difficulties. Since this users generated execution traces, it is possible to show the effect of the improvements done in production executions.

The COMPSs version used is the previously mentioned *exaQUte* branch (available at [88]). All the stack used in each experiment is the one presented in the corresponding sections of the previous chapter.

This chapter is organized as follows. First of all, the improvements that can only be assumed by the scheduling changes are presented. For this sake, executions where the user code has not changed are used. Afterwards, the obtained performances with the codes presented previously regarding several aspects are shown.

## 5.1 Experimental Setup

The results presented in this chapter have been obtained using the MareNostrum IV Supercomputer located at the Barcelona Supercomputing Center (BSC). Its current peak performance is 11.15 Petaflops, ten times more than its previous version, MareNostrum III [89]. The supercomputer is composed by 3456 nodes, each of them with two Intel®Xeon Platinum 8160 (24 cores at 2,1 GHz each). It has 384.75 TB of main memory, 100Gb Intel®Omni-Path [90] Full-Fat Tree Interconnection, and 14 PB of shared disk storage managed by the Global Parallel File System (gpfs).

Furthermore, COMPSs version used is the previously mentioned *exaQUte* branch (available at [88]). All the stack used in each experiment is the one presented in the corresponding sections of the previous chapter.

## 5.2 Scheduling performance

An application doing geospatial computations for the Institute of Political Economy and Governance [91] has been used as reference. Figure 5.1 shows the execution traces of both executions. The elapsed times goes from 14800 seconds to 3600, that is a speedup of 4.11 without changing a single line of code. The application has 75259 with different CPU constraints that go from tasks (the blue ones) which need the whole node to tasks (the yellow ones) that only need a single CPU. One thing to take into account is that the scheduling of the first part, with tasks colored in red and white, is more or less the same. This fact is because of the duration of the tasks is superior to the latency of the old scheduler, so any improvement can be detected. When zooming into the improvable zone, where the duration of the tasks seem to be inferior to the scheduling latency, we obtain the zoom shown in Figure 5.2. If we compute the execution times of both traces, we get 11981 seconds for the zoomed section with the old scheduler and 951 seconds with the new one, that is a speed up of 12.59. Or what is the same, the second execution takes the 7.9% of the time compared to the first execution.

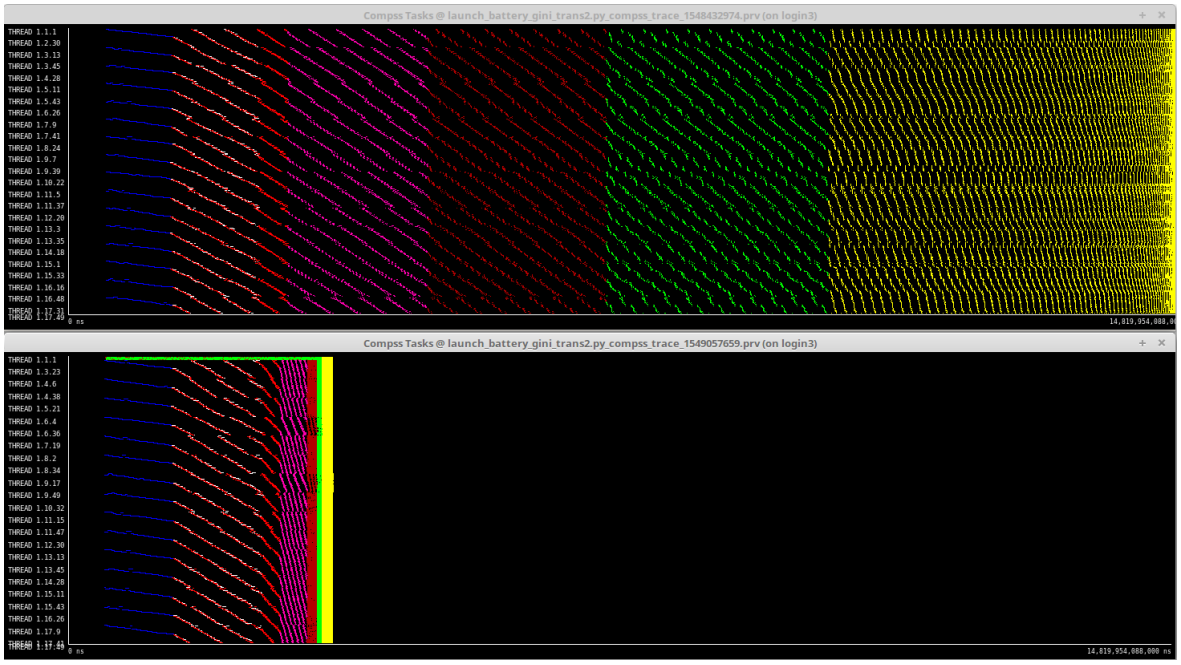



FIGURE 5.1: Execution trace of the same application with the old and the new scheduler

The first thing that we could think on is that the tasks last more or less in one case or the other. Figure 5.3 show the duration of the tasks executed in each one of the processors. It is clear that the execution times are similar. More precisely, the time scale goes from 0 to 60.

Nevertheless, the two representations that show more clearly the reason of the speed-up are Figure 5.4 and Figure 5.5. The first one shows the elapsed time between consecutive

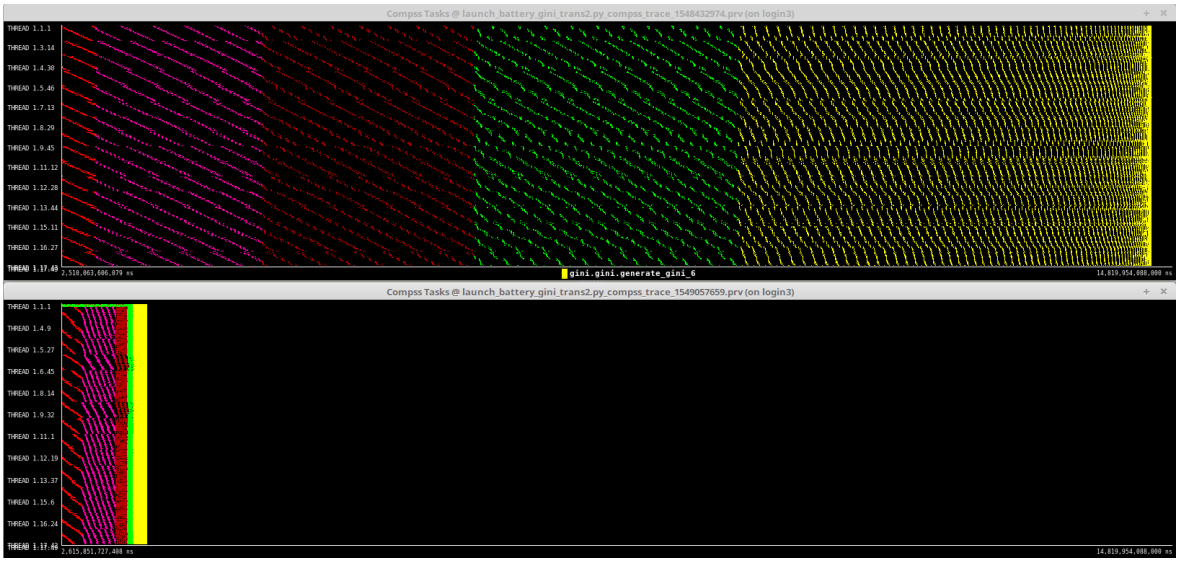


FIGURE 5.2: Zoom on the execution trace of the same application with the old and the new scheduler

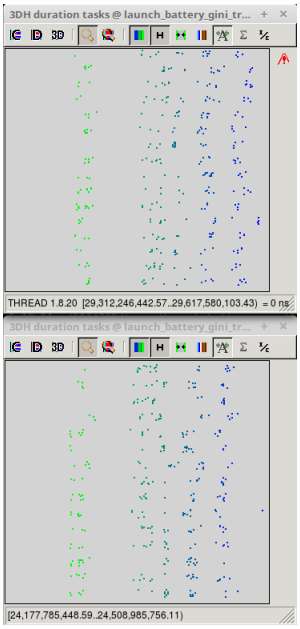


FIGURE 5.3: Duration of the tasks when executing with the old and the new scheduler

tasks. With the old scheduler, this time decreases progressively while the tasks start finishing, so the amount of available ready tasks decreases. The code color goes from blue (more ellapsed time) to green (less elapsed time). With the new scheduler, as long as the duration of the tasks and its constraints makes it possible, the elapsed time between executions decreases dramatically.

Finally, we can see this latency impact on the amount of concurrent tasks that are executed by the application. In the old scheduler, only at the end a high amount of tasks is running at the same time. This is mainly because the amount of free tasks when ordering the priority queue is already small. With the new version, this number grows steadily as the constraints of the tasks allow it.

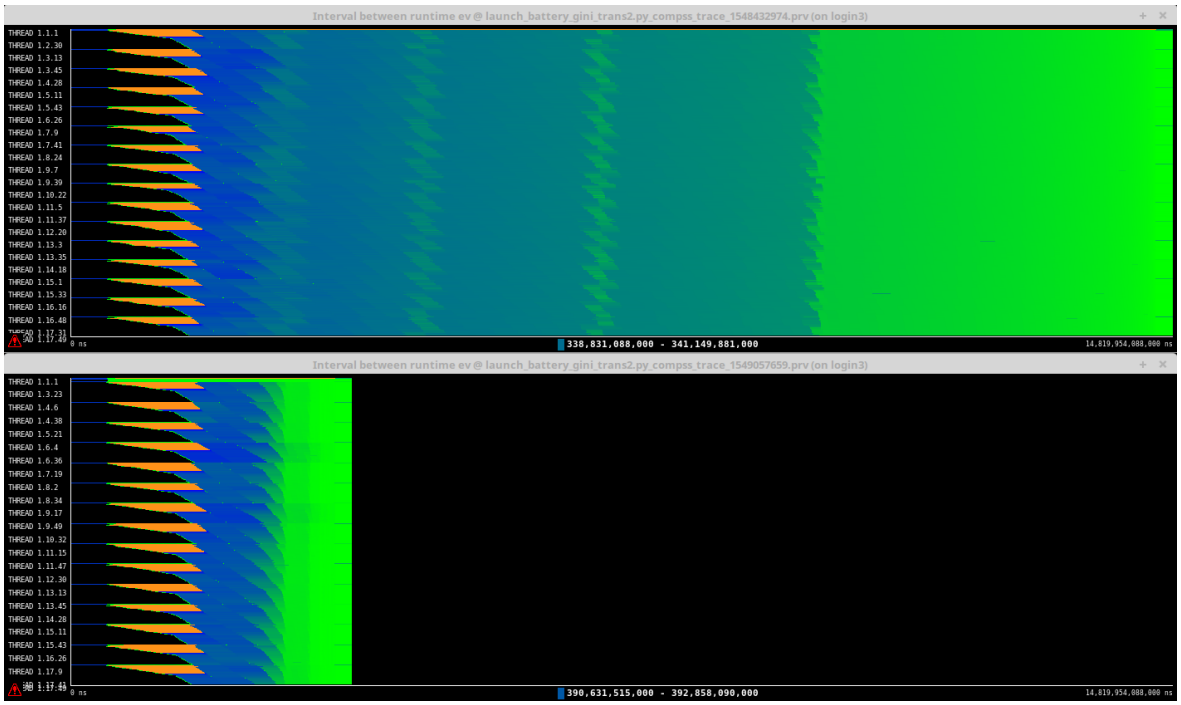



FIGURE 5.4: Elapsed time between tasks when executing with the old and the new scheduler

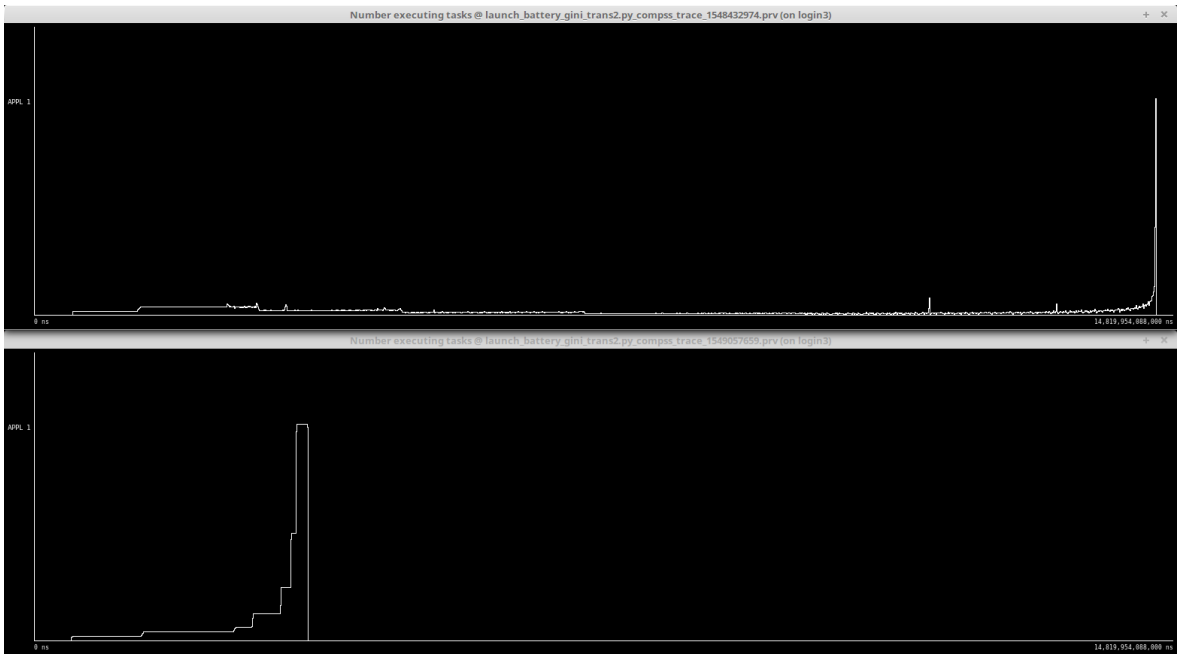



FIGURE 5.5: Amount of executing tasks with the old and the new scheduler

This scheduling traces contain all the improvements detailed in the previous chapter (a queue per resource that is kept between different scheduling and a hashmap with the available resources) unless the multithreaded scheduler. This improvement has been added lately when it has been shown that this improvements were not enough for certain applications. At this point it is really important to keep in mind that the improvements presented

does not contain the multithreading in the management of the different resource queues since this will have an impact later.

Hence, it is demonstrated that the performance has been dramatically increased with this master thesis.

## 5.3 Dynamic scheduling with different tasks' constraints evaluation of the GAWS workflow

Figure 5.6 shows four Paraver traces to evaluate the performance of the GWAS worklow using COMPSs. The time is represented in the horizontal axis, and its scale is the same for the three of them. The available cores are represented in the vertical axis, and the different colours represent different task types being executed in a given core during a certain time. All the executions perform the same run using 30 computing nodes (1,440 cores), spawning 93,858 tasks, generating 120,018 files (217.68 Gb), and analyzing 2,860 inputs in the first level with 1 single input in the second level and 3 different inputs in the third level.

The top trace (a) emulates an execution with SLURM Job Arrays with fixed job requirements, executing each step separately and using static constraints. Trace (b) emulates the same run but with dynamic job constraints. Also, Trace (c) emulates the behaviour of a state of the art task-based framework, merging all the steps by building a data dependence graph but keeping the static constraints. Finally, the bottom trace (d) shows the execution using dynamic constraints and without any barrier. Notice that the benefits of COMPSs handling automatically the tasks' data dependencies and taking into account the tasks' constraints is able to speed-up 2.24 times the execution. The behavior of the dynamic scheduling had already been shown on Figure4.8. At this point the performance interest on this approach is demonstrated.

The trace files from the previous figure also provide valuable information to understand the parallelism and computational complexity of the different steps. The complexity of the first step (phasing, shown in green in the traces) is proportional to the number of subjects in the input, and the amount of spawned tasks is constant and equal to the number of chromosomes in the study.

Next, the complexity of the second step is proportional to the size of the input in the first level and the second level. Moreover, the amount of spawned tasks is proportional to the number of inputs considered in the second level. In opposition to the previous step, the size of the different inputs in the second level directly affects the memory requirements of each task in this step. Hence, a good configuration of the requirements is crucial. Too large constraints result in resource wasting. Too small constraints ends up with error executions by memory space overflow.

The complexity and number of tasks of the third section (shown in purple in the traces) are proportional to both the number of inputs in the second and third level considered.

Finally, at the end of the execution, some single node scripts in R are executed without further dependencies. Since there are lots of resources and the R scripts can always be run in parallel, this phase does not scale at all in the strong scaling but scales perfectly in the

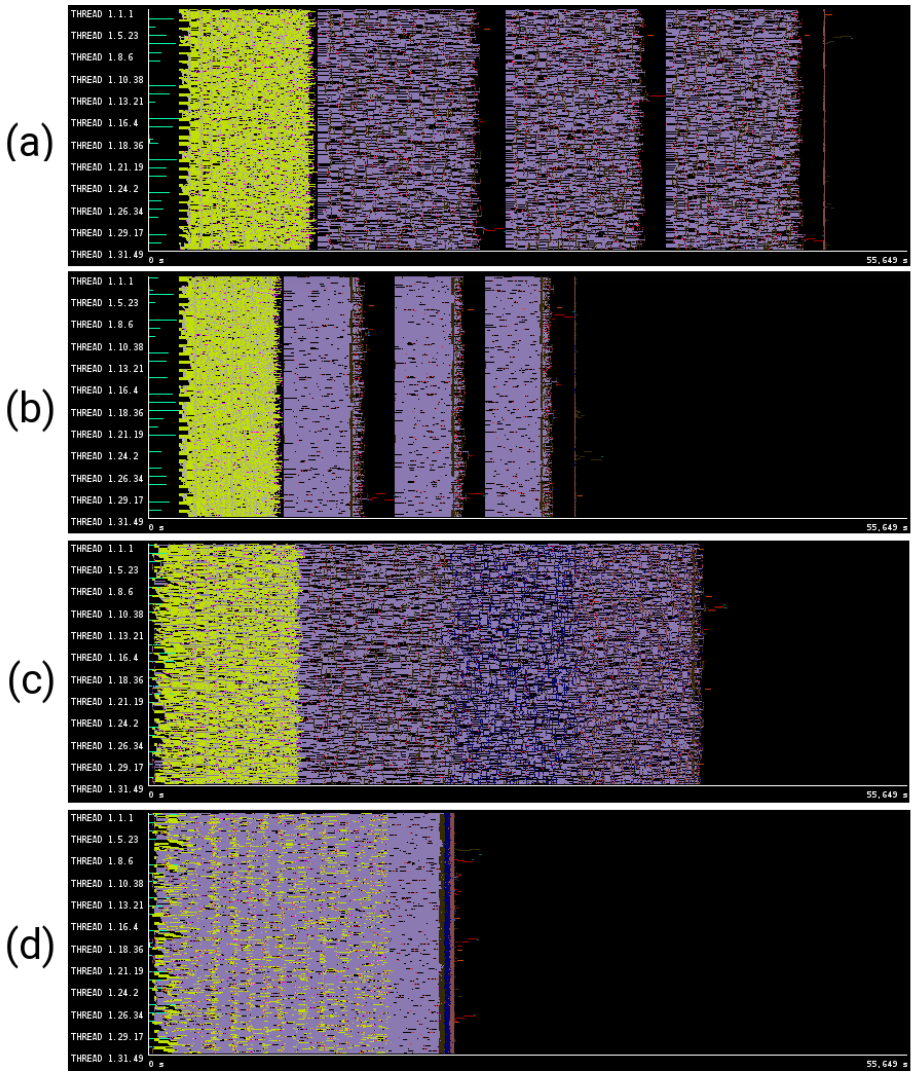



FIGURE 5.6: GWAS executions using COMPSs. (a) Separated steps with static constraints. (b) Separated steps with dynamic constraints. (c) Merged steps with static constraints. (d) Merged steps with dynamic constraints.

weak scaling.

From the content of this section, it seems clear the importance of the improvements done in the scheduling, since low latencies and the ability to schedule dynamically the tasks has a high impact on the execution time. On the other hand, the dynamic control of the constraints also have a high impact on the execution time.

## 5.4 Scheduling and application improvements in the GWAS code

With the scheduler improvements previously presented (unless the multithreading version) and the GWAS workflow version of the previous section, an execution with 50 nodes (2400 cores), 2 inputs in the second phase and 4 inputs in the third phase has been done. This execution is presented in Figure 5.7. This will be the reference point to illustrate the improvements achieved with the modification in the user code and the multithreading in the scheduler. It seems clear that the execution no longer scales. This has been considered

as the departure point since the strong scaling of the same version of the code and same scheduler scaled until 100 nodes with 2 inputs in the third level and the execution with 25 nodes and 4 inputs in the third level went also really well.

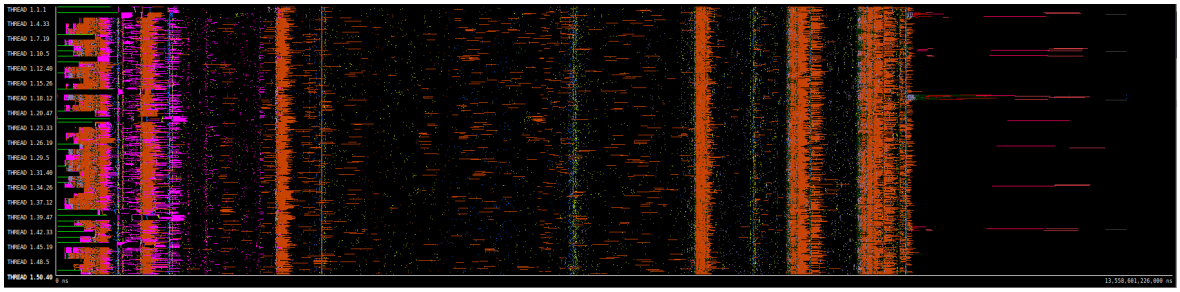


FIGURE 5.7: Execution trace with all the scheduling improvements unless the multithreading one and all the GWAS improvements unless the merging of the fine tasks.

There are two images that can clearly show the impact of the combination of this improvements. The first one is the launch of the same execution, that is the one with the same inputs but with the small tasks merged into the big ones and the multithreaded scheduler. Both execution traces are shown in Figure 5.8.

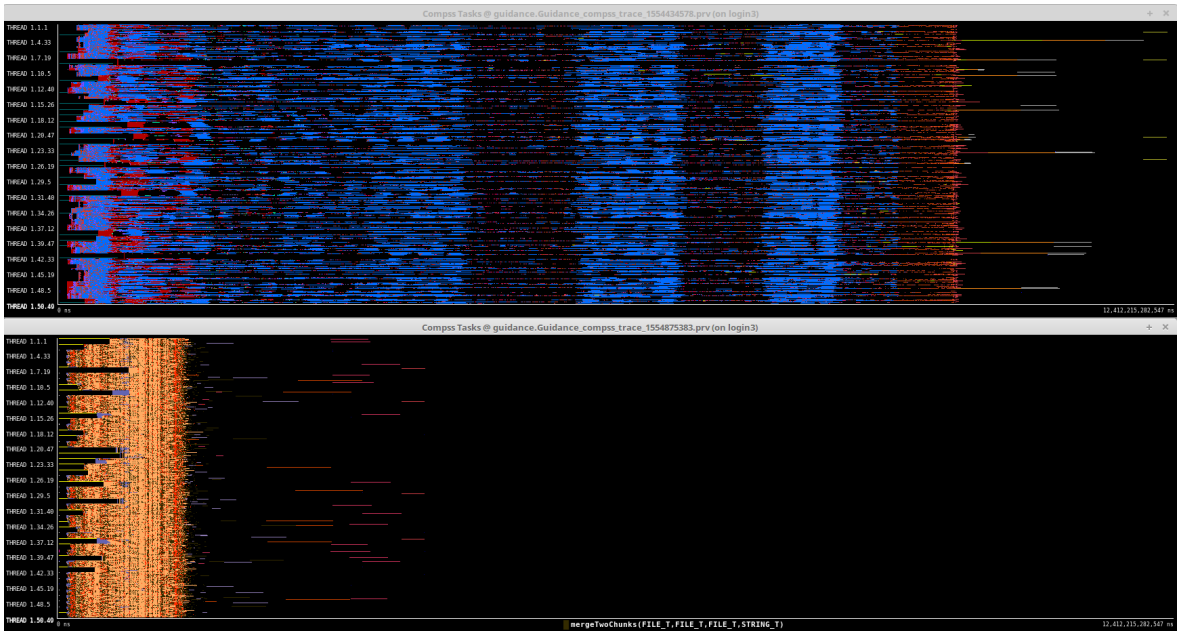



FIGURE 5.8: Execution traces comparing the execution shown in Figure 5.7 with only partial improvements and the one including all the improvements in the scheduler and the GWAS workflow

On the other hand, when launching the execution with eight inputs in the third part and all the improvements, we obtain the comparison shown in Figure 5.9. The new combination, even if having almost the double complexity, outperforms the already improved version.

Finally, the same execution has been done with 100 nodes (4800 execution CPUs). The execution traces are shown in Figure 5.10.

The improvement achieved seems clear, there is not much more to say. The summary Table 5.1 shows the specific details of the executions. Final code means that it considers all

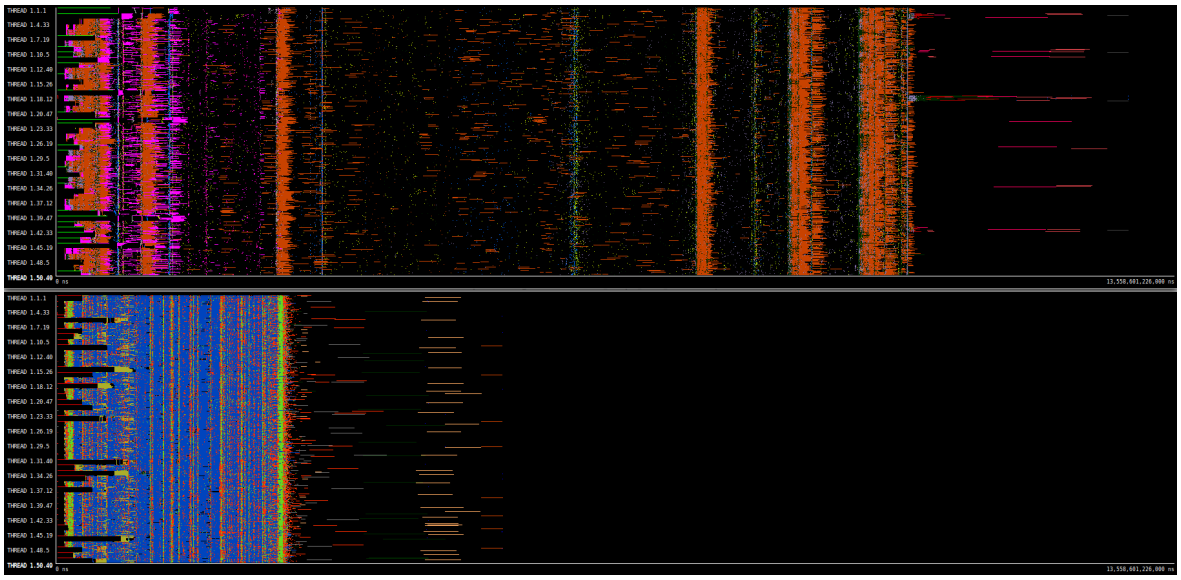


FIGURE 5.9: Execution traces comparing the execution shown in Figure 5.7 with only partial improvements and the one including all the improvements in the scheduler and the GWAS workflow with two times the amount of inputs in the third level

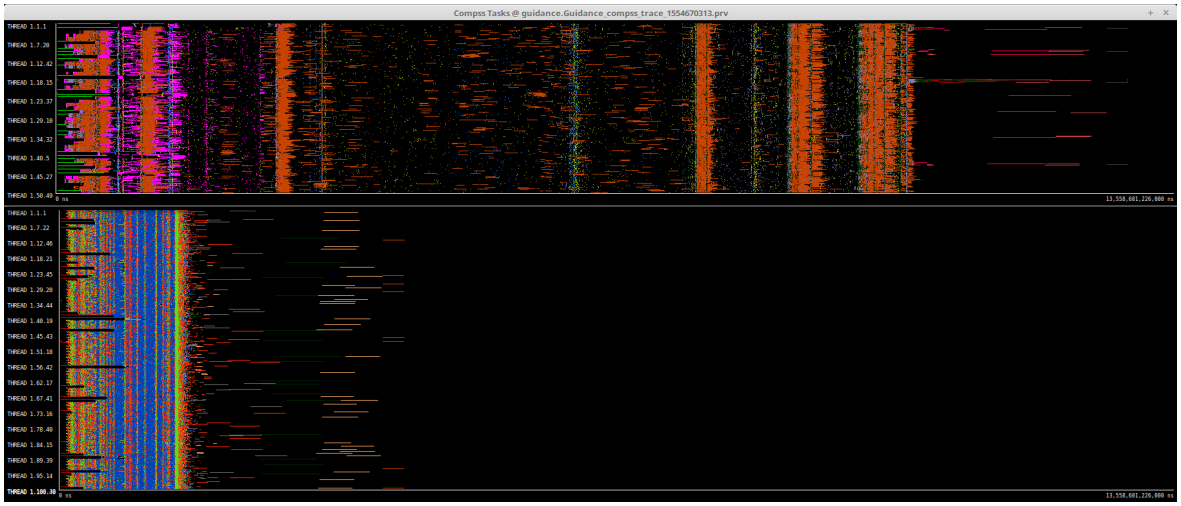


FIGURE 5.10: Execution traces comparing the execution shown in Figure 5.7 with only partial improvements and the one including all the improvements in the scheduler and the GWAS workflow with two times the amount of inputs in the third level and twice the amount of available resources

the improvements. Partial means that only considers some scheduling improvements.

## 5.5 Scalability

### 5.5.1 Strong Scaling

| Code | Cores | Inputs in third level | Total exec time (s) | Parallel exec time (s) |
|---|---|---|---|---|
| Partial | 2400 | 4 | 13558 | 10562 |
| Final | 2400 | 4 | 4146 | 1605 |
| Final | 2400 | 8 | 5446 | 2971 |
| Final | 4800 | 8 | 4221 | 1651 |

TABLE 5.1: Summary of the executions used to demonstrate the improvements achieved

For the strong scaling analysis, I have run the GWAS workflow with 300 inputs in the first input level, 2 in the second and, 8 in the third one with 1200, 2400, and 4800 cores respectively (25, 50, and 100 nodes).

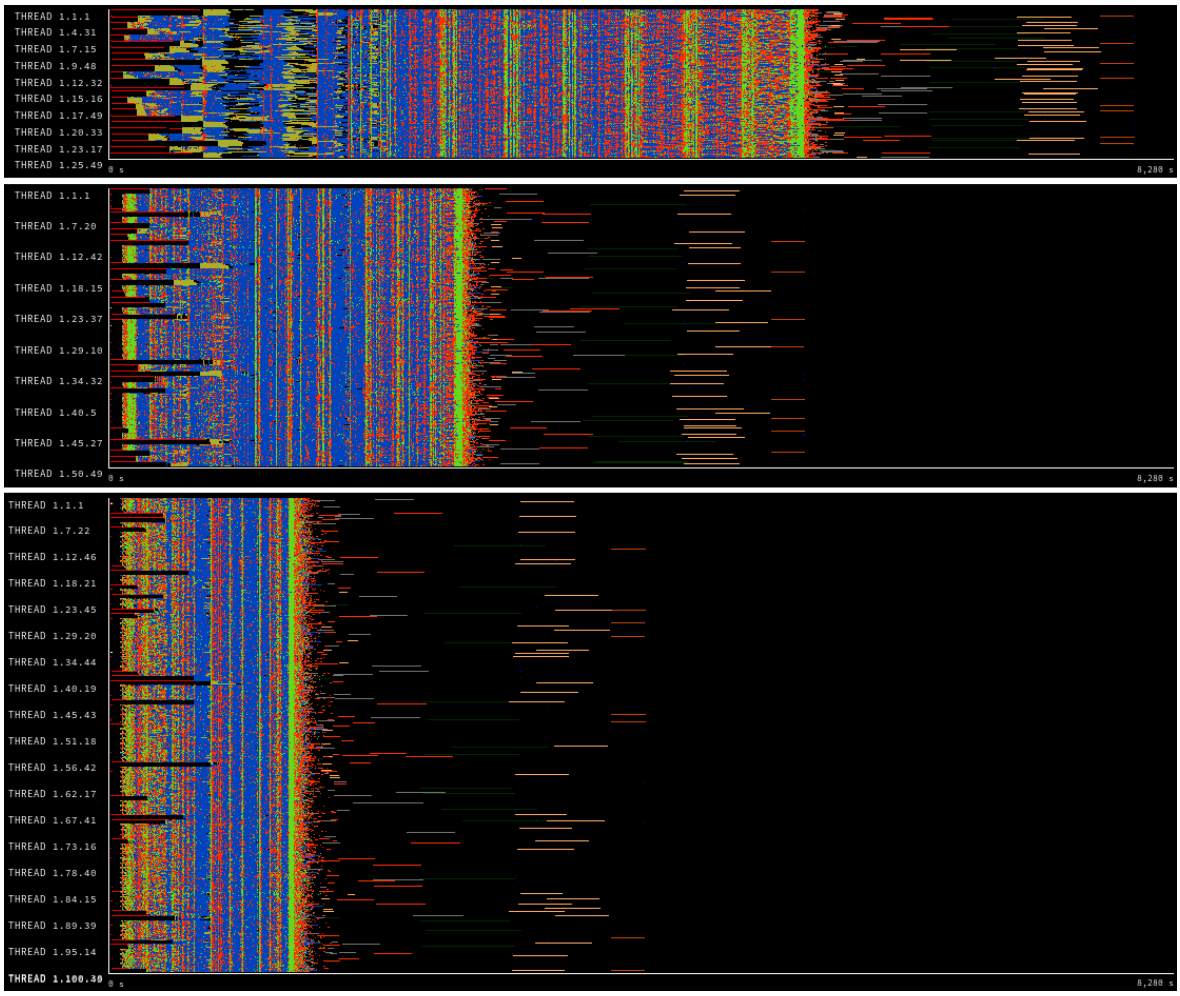



FIGURE 5.11: Execution traces of GWAS with 300 inputs in the first input level, 2 in the second and, 8 in the third one with 1200, 2400, and 4800 cores respectively (25, 50, and 100 nodes).

Table 5.2 summarises the total execution time, the parallel region's execution time, the total speed-up, the ideal speed-up, and the parallel region's speed-up of the previous executions. The speed-ups are calculated with respect to the smallest run. For the sake of simplicity, although the previous traces show that the application has some parallelism during all the execution, the ideal speed-up only considers the parallel region shown in the table.

| #Cores | Execution Time (s) | | Speed-up (u) | | |
|---|---|---|---|---|---|
| | Total | Parallel | Total | Ideal | Parallel |
| **1200** | 7995 | 5314 | 1.00 | 1.00 | 1.00 |
| **2400** | 5417 | 2658 | 1.48 | 1.50 | 1.99 |
| **4800** | 4211 | 1368 | 1.90 | 1.99 | 3.88 |

TABLE 5.2: Strong scaling analysis with 1200, 2400, and 4800 cores (25, 50, and 100 nodes respectively).

Notice that the parallel region represents 66.5% of the execution time when running with 1200 cores. Hence the application's global speed-up is limited by the workflow itself (1.99 ideal speed-up), leading to a maximum speed-up of 1.90 when running with 4800 cores.

### 5.5.2 Weak Scaling

Considering appreciations from the previous section regarding the computational complexity, the time should remain constant while increasing the number of inputs in the third level. Nevertheless, not all the tasks last the same time (the executions rely on stochastic procedures) so it is not possible to compute an entirely realistic study. This is why, even if results can suggest that we obtain super-speedups, it is more reasonable to think that COMPSs scales linearly with the number of cores and tasks.

| #Cores | #Tasks | Tasks per core | Execution Time (s) |
|---|---|---|---|
| **1200** | 279,562 | 232.97 | 4347 |
| **2400** | 483,576 | 201.49 | 4184 |
| **4800** | 893,123 | 186.07 | 4211 |

TABLE 5.3: Weak scaling analysis with 1200, 2400, and 4800 cores (25, 50, and 100 nodes respectively).

Hence, I have executed the workflow with 300 inputs in the first level, two inputs in the second level, and 25 nodes with 2, 50 nodes with 4, and 100 nodes with 8 inputs in the third level. Table 5.3 shows the number of tasks, the number of tasks per core, and the total execution time for 1200, 2400, and 4800 cores (25, 50, and 100 nodes respectively). Notice that the execution time and the number of tasks per core remain almost the same in all the executions due to the previous considerations. Figure 5.12 shows the execution traces of the executions corresponding to the weak scaling analysis. The time has been scaled in such a way that is the same for the three executions.

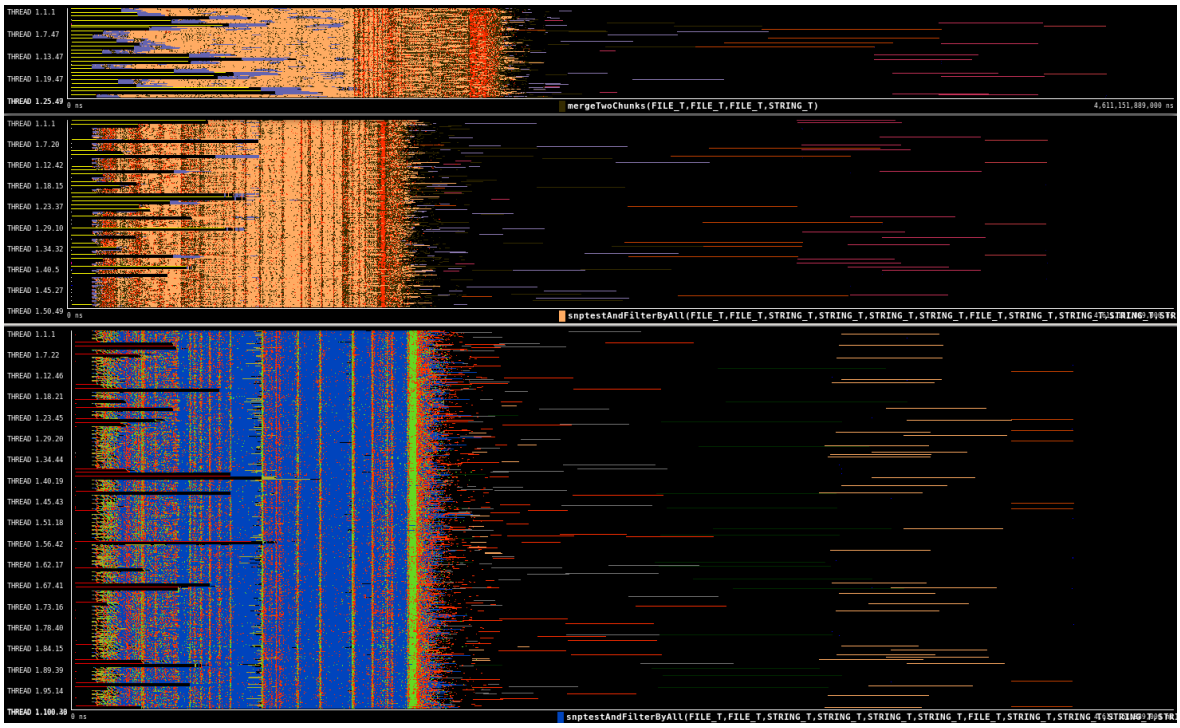



FIGURE 5.12: Execution traces corresponding to the three executions performed to state the weak scaling

## 5.6 Portability

### 5.6.1 Cloud computing

In order to evaluate the portability, I have performed a little execution with a really small dataset using the Google Cloud Platform. The free tier is limited to 8 concurrently executed CPUs, which limited a lot the size of the execution to launch. Nevertheless, the main goal was to demonstrate that the workflow could be executed in a supercomputing facility or in the cloud without modifying a single line of the workflow.

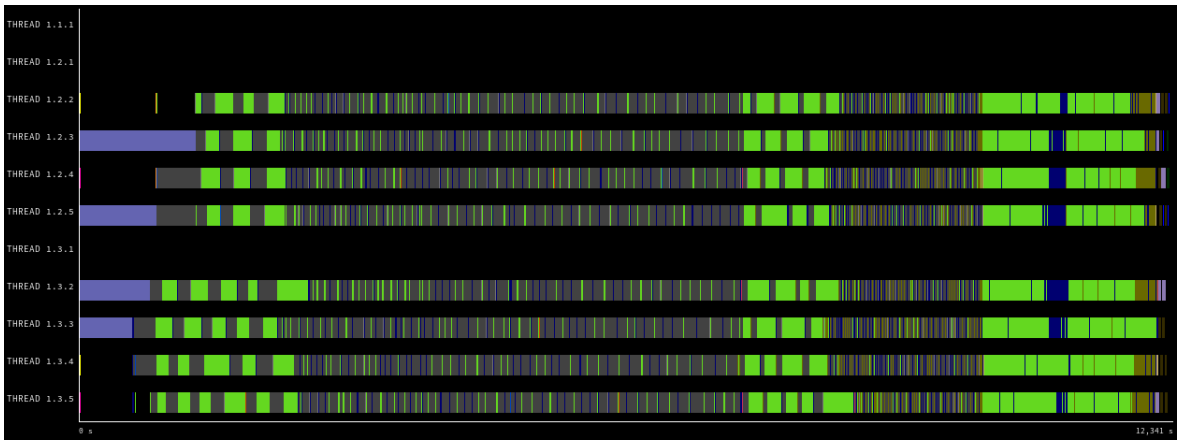



FIGURE 5.13: Execution trace of the cloud execution

I have fitted the execution to the free tier limits of the Google Cloud Platform; using 2 computing nodes with 4 vCPUs, 26 Gb of memory, Intel Xeon E5 v3 (Haswell) 2.3 GHz, and

500 Gb disk. The execution takes 12,341 s (approximately 206 minutes) and, in comparison with previous experiments, the execution requires file transfers between nodes because the cloud nodes have no shared disk configured. Some executions with buckets [92] acting as shared file system has been done. Nevertheless, the encountered performance is really poor, so this idea has been discarded.

### 5.6.2 HPC

In addition, the singularity image has been used in production executions obtaining the same performance than the bare-metal ones. Nevertheless, extrae is not working properly so no execution traces are available.

## 5.7 Scheduling and workflow improvements in the MC workflow

While in the GWAS case more executions have been done, in this case the changes in the workflow took more time and were done in parallel to the scheduling modifications. Hence, it is only possible to compare the before and after with already all the modifications in place. Figure 4.15 shows the execution without the asynchronous enhancements and the scheduling improvements but without the multithreading improvement. In total, there are 8000 samples. Figure 5.15 shows an execution with the asynchronous improvements and the last scheduler version. In total there are 51000 samples. I cannot give further details since the executions have been done by Riccardo Tosi that gently gave me images of them (I do not have the original trace file to open it with paraver). The time is not scaled. I have tried to scale the images in such a way that the duration of the tasks is similar. This can be shown because when the scheduler does not perform well it is possible to clearly see the beginning and the end of the tasks.

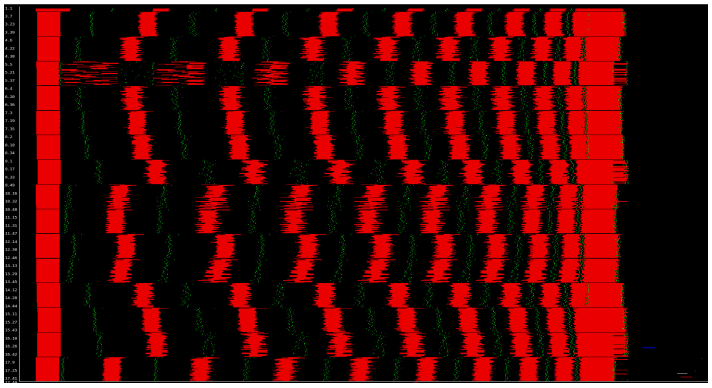


FIGURE 5.14: Execution trace of a MC execution with 8000 samples and synchronous convergence checking and 15 worker nodes

Considering the previous comments, no rigorous information con be given when analyzing this traces. Nevertheless, some qualitative information can be extracted considering that in the second execution there are two times more resources and six times more samples. For reasons to be discovered, at the beginning of the execution the scheduling behaves similarly. Nevertheless, at one point, it starts going clearly better. From the information that Riccardo has provided, the scheduling problem at the beginning is proportional to the amount of available resources and disappears after the first iteration for executions with the same amount of samples (51000) and 10, 20 and 30 available nodes.

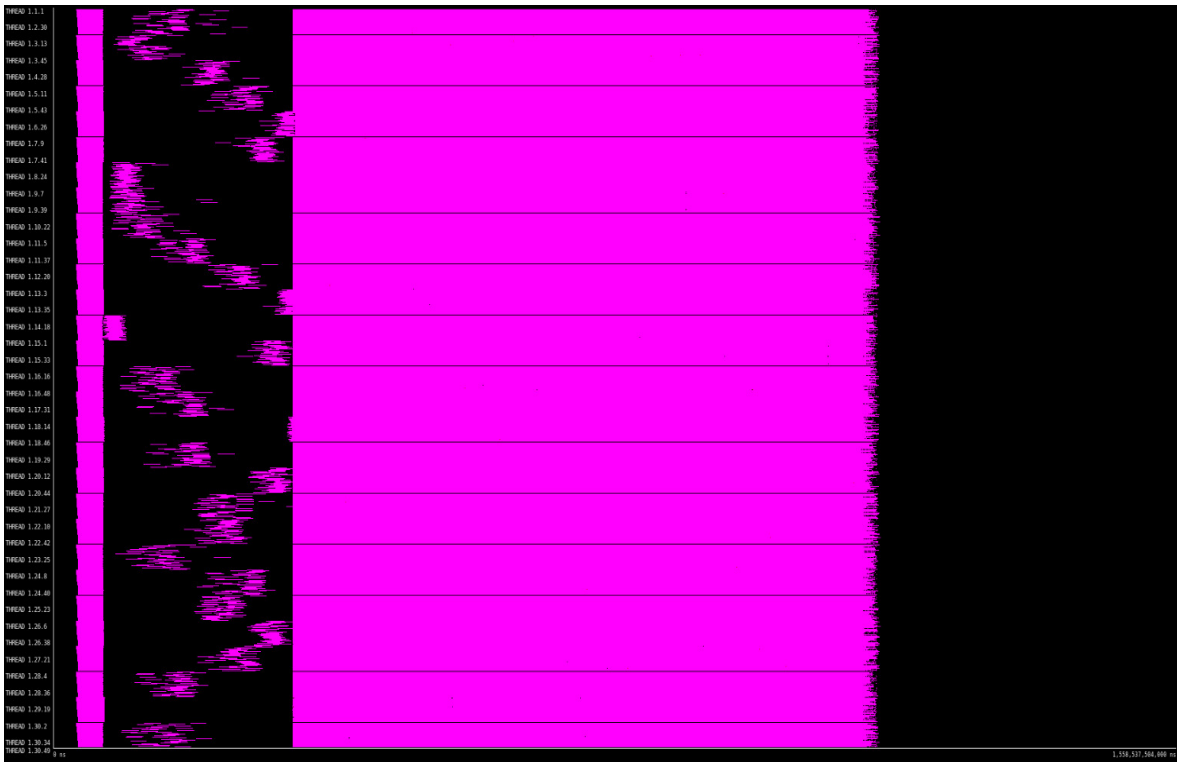


FIGURE 5.15: Execution trace of a MC execution with 51000 samples and asynchronous convergence checking and 29 worker nodes

Finally and from what has been commented, the asynchronous convergence checking is unremarkable in the execution trace so there are not performance problems related with this code improvement. In addition, the scheduler seems to work much more better.

# Chapter 6

# Conclusions and Future work

This master thesis has contributed to the good scalability to at least three different workflows with a real scientific application. In addition, it has defined deeply the changes done in order to reproduce them in other use cases.

In addition, the COMPSs scheduler has been enhanced to be able to scale out to a higher amount of resources. Despite the results presented, this improvements are not enought to scale until 200 nodes and beyond. Hence, further improvements should be put in place considering the new capabilities of the supercomputers that are being build and even those which are already in place. This could be considered as the single future work thing to do, since it is so important that under my opinion should be highly prioritized.

Finally, and since an image is worth much more than a thousand words, I would like to do a suggestion. Just look at Figure 5.2 to understand the impact of the first improvements done in the scheduler. After that, and keeping in mind that the first improvements were also present in the bad execution, I would read the section 5.4 to better understand the impact of the second batch of modifications. With just this two actions one can briefly understand the contributions of this master thesis and its impact in production environments.

# Bibliography


[1] Apache Spark. *Apache Spark*. URL: http://spark.apache.org/.

[2] Daniel Drzisga et al. "Scheduling massively parallel multigrid for multilevel Monte Carlo methods". In: *SIAM Journal on Scientific Computing* 39.5 (2017), S873–S897.

[3] Ramon Amela et al. "Executing linear algebra kernels in heterogeneous distributed infrastructures with PyCOMPSs". In: *Oil & Gas Science and Technology–Revue d'IFP Energies nouvelles* 73 (2018), p. 47.

[4] Cristian Ramon-Cortes et al. "AutoParallel: A Python module for automatic parallelization and distributed execution of affine loop nests". In: *arXiv preprint arXiv:1810.11268* (2018).

[5] European Commission. *HPC strategies and implementations*. URL: https://ec.europa.eu/programmes/horizon2020/en/h2020-section/high-performance-computing-hpc.

[6] Robert Underwood, Jason Anderson, and Amy Apon. "Measuring Network Latency Variation Impacts to High Performance Computing Application Performance". In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM. 2018, pp. 68–79.

[7] Francesc Lordan et al. "Servicess: An interoperable programming framework for the cloud". In: *Journal of grid computing* 12.1 (2014), pp. 67–91.

[8] Leonardo Dagum and Ramesh Menon. "OpenMP: An industry-standard API for shared-memory programming". In: *Computing in Science & Engineering* 1 (1998), pp. 46–55.

[9] J. Conejero et al. "Task-based programming in COMPSs to converge from HPC to big data". In: *International journal of high performance computing applications* (Apr. 2017). DOI: 10.1177/1094342017701278.

[10] Universitat Politècnica de Catalunya (UPC). *Universitat Politècnica de Catalunya (UPC)*. URL: http://www.upc.es.

[11] Barcelona Supercomputing Center (BSC). *Barcelona Supercomputing Center (BSC)*. URL: http://www.bsc.es.

[12] Barcelona Supercomputing Center (BSC). *Workflows and distributed computing group*. URL: https://www.bsc.es/discover-bsc/organisation/scientific-structure/workflows-and-distributed-computing.

[13] Barcelona Supercomputing Center (BSC). *Computational genomics group*. URL: https://www.bsc.es/discover-bsc/organisation/scientific-structure/computational-genomics.

[14] International Centre for Numerical Methods in Engineering (CIMNE). *International Centre for Numerical Methods in Engineering (CIMNE)*. URL: https://www.cimne.com/.

[15] SLURM Team. *SLURM Workload Manager - Job Array Support*. 2017. URL: https://slurm.schedmd.com/job_array.html.

[16] Sílvia Bonàs-Guarch et al. “Re-analysis of public genetic data reveals a rare X-chromosomal variant associated with type 2 diabetes”. In: *Nature communications* 9.1 (2018), p. 321.

[17] Shakuntala Baichoo et al. “Developing reproducible bioinformatics analysis workflows for heterogeneous computing environments to support African genomics”. In: *BMC bioinformatics* 19.1 (2018), p. 457.

[18] Peter Amstutz et al. *Common workflow language, v1. 0*. July 2016. DOI: 10.6084/m9.figshare.3115156.v2. URL: https://figshare.com/articles/Common_Workflow_Language_draft_3/3115156.

[19] Bongsong Kim et al. “GWASpro: a high-performance genome-wide association analysis server”. In: *Bioinformatics* (Dec. 2018), pp. 1–3. DOI: 10.1093/bioinformatics/bty989. URL: https://doi.org/10.1093/bioinformatics/bty989.

[20] M. Pisaroni, F. Nobile, and P. Leyland. “A Continuation Multi Level Monte Carlo (C-MLMC) method for uncertainty quantification in compressible inviscid aerodynamics.” In: *Computer Methods in Applied Mechanics and Engineering* 326 (2017), pp. 20 –50.

[21] Nathan Collier et al. “A continuation multilevel Monte Carlo algorithm.” In: *BIT NUMERICAL MATHEMATICS* 55.2 (2015), pp. 399 –432.

[22] Nicholas Metropolis and Stanislaw Ulam. “The monte carlo method”. In: *Journal of the American statistical association* 44.247 (1949), pp. 335–341.

[23] Michael B Giles. “Multilevel monte carlo path simulation”. In: *Operations Research* 56.3 (2008), pp. 607–617.

[24] Anubhav Jain et al. “FireWorks: A dynamic workflow system designed for high-throughput applications”. In: *Concurrency and Computation: Practice and Experience* 27.17 (2015), pp. 5037–5059.

[25] Abybhav Jain. *Introduction to FireWorks (workflow software) - FireWorks 1.8.7 documentation*. 2013. URL: https://materialsproject.github.io/fireworks/#.

[26] Katherine Wolstencroft et al. “The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud”. In: *Nucleic Acids Research* 41.W1 (May 2013), W557–W561. ISSN: 0305-1048. DOI: 10.1093/nar/gkt328. URL: https://doi.org/10.1093/nar/gkt328.

[27] Taverna Committers. *Apache Taverna*. 2014. URL: https://taverna.incubator.apache.org/.

[28] Bertram Ludäscher et al. “Scientific workflow management and the Kepler system”. In: *Concurrency and Computation: Practice and Experience* 18.10 (2006), pp. 1039–1065.

[29] UC Davis, UC Santa Barbara, and UC San Diego. *The Kepler Project*. 2004. URL: https://kepler-project.org/.

[30] Enis Afgan et al. “The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update”. In: *Nucleic Acids Res.* 44.W1 (2016), W3–W10.

[31] Galaxy Team. *Galaxy*. 2005. URL: https://usegalaxy.org/.

[32] Michael Wilde et al. “Swift: A language for distributed parallel scripting”. In: *Parallel Computing* 37.9 (2011), pp. 633 –652. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2011.05.005. URL: http://www.sciencedirect.com/science/article/pii/S0167819111000524.

[33] Swift Project Team. *The Swift Parallel Scripting Language*. 2011. URL: http://swift-lang.org/main/.

[34] Barcelona Centre for Genomic Regulation (CRG). *Nextflow: A DSL for parallel and scalable computational pipelines*. 2014. URL: https://www.nextflow.io/.

[35] *Dask: Library for dynamic task scheduling*. Dask Development Team. 2016. URL: https://dask.org.

[36] University of Chicago. *Parsl: Parallel Scripting in Python*. 2017. URL: http://parsl-project.org/.

[37] Rosa M. Badia and et al. "COMP superscalar, an interoperable programming framework". In: *SoftwareX* 3 (Dec. 2015), pp. 32–36. URL: https://doi.org/10.1016/j.softx.2015.10.004.

[38] *Extrae*. Web page at https://tools.bsc.es/extrae. (Date of last access: 19th December, 2016).

[39] Vincent Pillet and et. al. "Paraver: A Tool to Visualize and Analyze Parallel Code". In: *Transputer and occam Developments* (Apr. 1995). http://www.bsc.es/paraver - Accessed April, 2012, pp. 17–32.

[40] *Paraver: a flexible performance analysis tool*. Web page at https://tools.bsc.es/paraver. (Date of last access: 19th December, 2016).

[41] James Gosling et al. *The Java language specification*. Addison-Wesley Professional, 2000.

[42] Guido Van Rossum and Fred L Drake. *Python language reference manual*. Network Theory, 2003.

[43] Jesús Labarta et al. Enric Tejedor Rosa M. Badia. "PyCOMPSs: Parallel computational workflows in Python". In: *The International Journal of High Performance Computing Applications (IJHPCA)* 31 (2017), pp. 66–82. URL: http://dx.doi.org/10.1177/1094342015594678.

[44] Karim Djemame et al. "Towards an Energy-Aware Framework for Application Development and Execution in Heterogeneous Parallel Architectures". In: *Hardware Accelerators in Data Centers*. Springer, 2019, pp. 129–148.

[45] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0201325772.

[46] Shigeru Chiba. "Load-time Structural Reflection in Java". In: *ECOOP 2000 - Object-Oriented Programming* 1850 (May 2000), pp. 313–336. URL: http://dx.doi.org/10.1007/3-540-45102-1_16.

[47] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science and Engg.* 13.2 (Mar. 2011), pp. 22–30. DOI: 10.1109/MCSE.2011.37. URL: http://dx.doi.org/10.1109/MCSE.2011.37.

[48] Eric Jones, Travis Oliphant, and Pearu Peterson. *SciPy: Open source scientific tools for Python*. 2001–. URL: http://www.scipy.org/.

[49] Intel Corporation. *Intel Math Kernel Library. Reference Manual*. Santa Clara, USA. ISBN 630813-054US. Intel Corporation, 2015.

[50] Shuai Che et al. "Rodinia: A Benchmark Suite for Heterogeneous Computing". In: *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*. IISWC '09. IEEE Computer Society, 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797. URL: http://dx.doi.org/10.1109/IISWC.2009.5306797.

[51] Karim Djemame et al. "TANGO: Transparent heterogeneous hardware Architecture deployment for eNergy Gain in Operation". In: *CoRR* abs/1603.01407 (2016). URL: http://arxiv.org/abs/1603.01407.

[52] Computational Genomics Group. *GUIDANCE*. http://cg.bsc.es/guidance/. 2016.

[53] Olivier Delaneau, Jonathan Marchini, and Jean-François Zagury. "A linear complexity phasing method for thousands of genomes". In: *Nature methods* 9.2 (2012), p. 179.

[54] Po-Ru Loh et al. "Reference-based phasing using the Haplotype Reference Consortium panel". In: *Nature genetics* 48.11 (2016), p. 1443.

[55] Bryan N Howie, Peter Donnelly, and Jonathan Marchini. "A flexible and accurate genotype imputation method for the next generation of genome-wide association studies". In: *PLoS genetics* 5.6 (2009), e1000529.

[56] Sayantan Das et al. "Next-generation genotype imputation service and methods". In: *Nature genetics* 48.10 (2016), p. 1284.

[57] Ketian Yu and Sayantan Das. *Minimac4*. URL: https://genome.sph.umich.edu/wiki/Minimac4.

[58] Jonathan Marchini and Bryan Howie. "Genotype imputation for genome-wide association studies". In: *Nature Reviews Genetics* 11.7 (2010), p. 499.

[59] Shaun Purcell et al. "PLINK: a tool set for whole-genome association and population-based linkage analyses". In: *The American Journal of Human Genetics* 81.3 (2007), pp. 559–575.

[60] Vagheesh Narasimhan et al. "BCFtools/RoH: a hidden Markov model approach for detecting autozygosity from next-generation sequencing data". In: *Bioinformatics* 32.11 (2016), pp. 1749–1751.

[61] Heng Li et al. "The sequence alignment/map format and SAMtools". In: *Bioinformatics* 25.16 (2009), pp. 2078–2079.

[62] Colin Freeman. *GTool*. URL: http://www.well.ox.ac.uk/~cfreeman/software/gwas/gtool.html.

[63] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2015. URL: https://www.R-project.org/.

[64] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux Journal* 2014.239 (2014), p. 2.

[65] Todd C. Miller. *sudo*. URL: https://www.sudo.ws/.

[66] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. "Singularity: Scientific containers for mobility of compute". In: *PloS one* 12.5 (2017), e0177459.

[67] Peter Mell, Tim Grance, et al. "The NIST definition of cloud computing". In: (2011).

[68] Google Cloud. *Google Cloud including GCP and G Suite*. 2008. URL: https://cloud.google.com/.

[69] exaQUte. *Exascale Quantification of Uncertainties for Technology and Science Simulation*. URL: https://www.exaqute.eu.

[70] Pooyan Dadvand, Riccardo Rossi, and Eugenio Oñate. "An object-oriented environment for developing finite element codes for multi-disciplinary applications". In: *Archives of computational methods in engineering* 17.3 (2010), pp. 253–297.

[71] CIMNE. *Kratos Multiphysics*. https://github.com/KratosMultiphysics/Kratos. 2019.

[72] Christian Bayer et al. "ON NONASYMPTOTIC OPTIMAL STOPPING CRITERIA IN MONTE CARLO SIMULATIONS." In: *SIAM JOURNAL ON SCIENTIFIC COMPUTING* 36.2 (2014), A869 –A885.

[73] Michele Pisaroni, Sebastian Krumscheid, and Fabio Nobile. *Quantifying uncertain system outputs via the multilevel Monte Carlo method-Part I: Central moment estimation*. Tech. rep. 2017.

[74] Michele Pisaroni, Sebastian Krumscheid, and Fabio Nobile. *Python glossary*. Tech. rep. URL: https://docs.python.org/3/glossary.html.

[75] Barcelona Supercomputing Center (BSC). *MareNostrum 4 User Guide*. URL: https://www.bsc.es/support/MareNostrum4-ug.pdf.

[76] IT4Innovations. *Salomon Hardware Overview*. URL: https://docs.it4i.cz/salomon/hardware-overview/.

[77] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[78] Charles Dapogny, Cécile Dobrzynski, and Pascal Frey. "Three-dimensional adaptive domain remeshing, implicit domain meshing, and applications to free and moving boundary problems". In: *Journal of computational physics* 262 (2014), pp. 358–378.

[79] George Karypis. "METIS and ParMETIS". In: *Encyclopedia of parallel computing* (2011), pp. 1117–1124.

[80] Michael Heroux et al. *An Overview of Trilinos*. Tech. rep. SAND2003-2927. Sandia National Laboratories, 2003.

[81] Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 2000.

[82] Intel. *C++ Intel's compuiler developer guide*. Tech. rep. 2019. URL: https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference.

[83] Barcelona Supercomputing Center (BSC). *User support group*. URL: https://www.bsc.es/discover-bsc/organisation/support-structure/user-support.

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

[85] Philip Schwan et al. "Lustre: Building a file system for 1000-node clusters". In: *Proceedings of the 2003 Linux symposium*. Vol. 2003. 2003, pp. 380–386.

[86] Oracle. *Java HashMap documentation*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html.

[87] Oracle. *Java Priority Queue documentation*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html.

[88] BSC. *COMPSs' exaQUte branch*. https://github.com/bsc-wdc/compss/tree/exaQUte. 2019.

[89] Barcelona Supercomputing Center (BSC). *MareNostrum 3 User Guide*. URL: https://www.bsc.es/support/MareNostrum3-ug.pdf.

[90] Mark S Birrittella et al. "Intel® Omni-path architecture: Enabling scalable, high performance fabrics". In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE. 2015, pp. 1–9.

[91] Institute of Political Economy and Governance (IPEG). *Institute of Political Economy and Governance (IPEG)*. URL: http://barcelona-ipeg.eu/.

[92] Google. *Google Cloud buckets documentation*. URL: https://cloud.google.com/storage/docs/json_api/v1/buckets.

# Appendices

# Appendix A

# GWAS

## A.1 Dockerfile

```
FROM ubuntu:18.04

ARG DEBIAN_FRONTEND=noninteractive

ENV TERM linux

RUN echo 'debconf debconf/frontend select Noninteractive' | debconf-set-selections

RUN apt-get update && \
    apt-get purge openjdk-\* icedtea-\* icedtea6-\* && \
    apt-get remove openjdk-11-jre openjdk-11-jdk openjdk-11-jre-headless openjdk-11-jdk
    -headless && \
    apt-get purge openjdk-* && \
    apt-get install -y --no-install-recommends openjdk-8-jdk && \
    update-alternatives --install /usr/bin/java java /usr/lib/jvm/java-8-openjdk-amd64/
    jre/bin/java 9999 && \
    rm -rf /var/lib/apt/lists/*

# Install Packages
RUN apt-get update && \
    apt-get install -y --no-install-recommends apt-utils && \
    apt-get install -y --no-install-recommends \
    git  \
    vim \
    wget \
    sudo \
    openssh-server && \
    yes yes | ssh-keygen -f /root/.ssh/id_rsa -t rsa -N '' > /dev/null && \
    cat /root/.ssh/id_rsa.pub > /root/.ssh/authorized_keys && \
    git config --global core.compression 9 && \
# ================================================================================
# Dependencies for building COMPSs
# ================================================================================
# Build dependencies
    sudo apt-get install -y --no-install-recommends maven \
# Runtime dependencies
    openjdk-8-jdk graphviz xdg-utils \
# Bindings-common-dependencies
    libtool automake build-essential \
# C-binding dependencies
    libboost-all-dev libxml2-dev csh \
# Extrae dependencies
    libxml2 gfortran libpapi-dev papi-tools \
# Misc. dependencies
    openmpi-bin openmpi-doc libopenmpi-dev uuid-runtime curl bc \
# Python-binding dependencies
    python-dev python3-dev libpython2.7 python-pip python3-pip python-setuptools
    python3-setuptools && \
    pip2 install wheel && \
    pip3 install wheel && \
    pip2 install wheel numpy==1.15.4 dill guppy decorator mpi4py==1.3.1 && \
    pip3 install wheel numpy==1.15.4 dill decorator mpi4py==3.0.1 && \
# Python-redis dependencies
```

```
    pip2 install redis==2.10.6 redis-py-cluster && \
    pip3 install redis==2.10.6 redis-py-cluster && \
# pycompsslib dependencies
    pip2 install scipy==1.0.0 scikit-learn==0.19.1 pandas==0.23.1 && \
    pip3 install scipy==1.0.0 scikit-learn==0.19.1 pandas==0.23.1 && \
# AutoParallel dependencies
    apt-get install -y --no-install-recommends libgmp3-dev flex bison libbison-dev
    texinfo libffi-dev && \
    pip2 install astor sympy enum34 islpy && \
# Testing dependencies
    pip3 install enum34 tabulate && \
# Configure user environment
# ============================================================================
# System configuration
# ============================================================================
# Add environment variables
    echo "JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/" >> /etc/environment && \
    echo "MPI_HOME=/usr/lib/openmpi" >> /etc/environment && \
    echo "LD_LIBRARY_PATH=/usr/lib/openmpi/lib" >> /etc/environment && \
    mkdir /run/sshd && \
    rm -rf /var/lib/apt/lists/*

RUN rm -rf ./framework && \
    export JAVA_HOME="/usr/lib/jvm/java-8-openjdk-amd64/" && \
    export MPI_HOME="/usr/lib/openmpi" && \
    export LD_LIBRARY_PATH="/usr/lib/openmpi/lib" && \
    git clone --branch "exaQUte" https://github.com/bsc-wdc/compss.git framework && \
    cd ./framework && \
    ./submodules_get.sh && \
    ./submodules_patch.sh && \
    echo "${JAVA_HOME}" && \
    sudo /framework/builders/buildlocal -P -M /opt/COMPSs && \
    rm -rf /root/.cache && \
    cd .. && \
    rm -r ./framework


#Copy binaries into the container
RUN mkdir /TOOLS
COPY ./TOOLS/shapeit.v2.r727.linux.x64 /TOOLS/shapeit.v2.r727.linux.x64
COPY ./TOOLS/R_scripts /TOOLS/R_scripts
COPY ./TOOLS/deps.R /TOOLS/deps.R
RUN chmod 775 /TOOLS/R_scripts/*

WORKDIR /TOOLS

RUN export DEBIAN_FRONTEND=noninteractive && \
    sudo apt-get update && sudo apt-get install -y --no-install-recommends gnupg2
    software-properties-common && \
    sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys
    E298A3A825C0D65DFD57CBB651716619E084DAB9 && \
    sudo add-apt-repository 'deb https://cloud.r-project.org/bin/linux/ubuntu bionic-
    cran35/' && \
    sudo apt-get update && DEBIAN_FRONTEND=noninteractive sudo apt-get install -y --no-
    install-recommends apt-utils && \
    sudo sed -i 's/^mesg n$/tty -s \&\& mesg n/g' /root/.profile && \
    DEBIAN_FRONTEND=noninteractive sudo apt-get install -y --no-install-recommends r-
    base r-base-dev r-base-core libcurl4-openssl-dev \
    jags libpq-dev libmariadb-client-lgpl-dev && \
    rm -rf /var/lib/apt/lists/*

RUN /usr/bin/Rscript /TOOLS/deps.R

######### IN CASE WE WANT TO UPGRADE QCTOOL #########

#Install mercurial (for QCTool)
#RUN apt-get install -y mercurial

#Install qctoolNew
#RUN hg clone -r ba5eaa4 https://gavinband@bitbucket.org/gavinband/qctool qctool_2.0 &&
     \
#    cd qctool_2.0 && \
```

```
#    ./waf-1.5.18 configure && \
#    ./waf-1.5.18 && \
#    ln -s /TOOLS/build/release/qctool_v2.0.1 /usr/bin/qctool2.0

######### END OF QCTOOL INSTALACTION ################

#Install qctool
RUN wget http://www.well.ox.ac.uk/~gav/resources/archive/qctool_v1.4-linux-x86_64.tgz &
    & \
    tar zxvf qctool_v1.4-linux-x86_64.tgz && \
    rm qctool_v1.4-linux-x86_64.tgz && \
    chmod -R 755 /TOOLS/qctool_v1.4-linux-x86_64/ && \
    ln -s /TOOLS/qctool_v1.4-linux-x86_64/qctool /usr/bin/qctool1.4

#bcftools and samtools dependencies
RUN sudo apt-get update && \
    sudo apt-get install -y --no-install-recommends zlib1g-dev libbz2-dev liblzma-dev
    libncurses5-dev libncursesw5-dev && \
    rm -rf /var/lib/apt/lists/*

#Install bcftools
RUN wget https://github.com/samtools/bcftools/releases/download/1.8/bcftools-1.8.tar.
    bz2 -O bcftools.tar.bz2 && \
    tar -xjvf bcftools.tar.bz2 && \
    rm bcftools.tar.bz2 && \
    cd bcftools-1.8 && \
    make && \
    make prefix=/usr/local/bin install && \
    ln -s /usr/local/bin/bin/bcftools /usr/bin/bcftools

#Install samtools
RUN wget https://github.com/samtools/samtools/releases/download/1.5/samtools-1.5.tar.
    bz2 -O samtools.tar.bz2 && \
    tar -xjvf samtools.tar.bz2 && \
    rm samtools.tar.bz2 && \
    cd samtools-1.5 && \
    make && \
    make prefix=/usr/local/bin install && \
    ln -s /usr/local/bin/bin/samtools /usr/bin/samtools

#Plink dependencies
RUN sudo add-apt-repository universe && \
    sudo apt-get update && \
    sudo apt-get install -y --no-install-recommends libatlas-base-dev libblas-dev
    liblapack-dev libatlas-base-dev && \
    rm -rf /var/lib/apt/lists/*

#Install plink
RUN git clone https://github.com/chrchang/plink-ng.git && \
    cd plink-ng && \
    rm -r 2.0 && \
    cd 1.9 && \
    ./plink_first_compile && \
    ln -s /TOOLS/plink-ng/1.9/plink /usr/bin/plink

#Install Eagle
RUN wget https://data.broadinstitute.org/alkesgroup/Eagle/downloads/old/Eagle_v2.3.tar.
    gz && \
    tar -zxvf Eagle_v2.3.tar.gz && \
    rm Eagle_v2.3.tar.gz && \
    rm -r Eagle_v2.3/example/ && \
    ln -s /TOOLS/Eagle_v2.3/eagle /usr/bin/eagle

#Install Impute ### This step will stop working once they upgrade the program since
    only the last version is available
RUN wget https://mathgen.stats.ox.ac.uk/impute/impute_v2.3.2_x86_64_static.tgz && \
    tar -zxvf impute_v2.3.2_x86_64_static.tgz && \
    rm impute_v2.3.2_x86_64_static.tgz && \
    rm -r impute_v2.3.2_x86_64_static/Example/ && \
    ln -s /TOOLS/impute_v2.3.2_x86_64_static/impute2 /usr/bin/impute2

#Install snptest
```

```
RUN wget http://www.well.ox.ac.uk/~gav/resources/archive/snptest_v2.5_linux_x86_64_static.tgz && \
    tar -zxvf snptest_v2.5_linux_x86_64_static.tgz && \
    rm snptest_v2.5_linux_x86_64_static.tgz && \
    rm -r snptest_v2.5_linux_x86_64_static/example/ && \
    chmod -R 755 /TOOLS/snptest_v2.5_linux_x86_64_static/ && \
    ln -s /TOOLS/snptest_v2.5_linux_x86_64_static/snptest_v2.5 /usr/bin/snptest_v2.5

#Install minimac3
#RUN git clone https://github.com/Santy-8128/Minimac3.git && \
#    cd Minimac3 && \
#    make -w && \
#    sudo ln -s /TOOLS/Minimac3/bin/Minimac3 /usr/bin/minimac3
RUN wget ftp://share.sph.umich.edu/minimac3/Minimac3Executable.tar.gz && \
    tar -zxvf Minimac3Executable.tar.gz && \
    rm Minimac3Executable.tar.gz && \
    chmod -R 755 /TOOLS/Minimac3Executable/bin && \
    ln -s /TOOLS/Minimac3Executable/bin/Minimac3-omp

#Minimac4 dependencies
RUN sudo apt-get update && \
    sudo apt-get install -y --no-install-recommends cmake python-pip python-dev && \
    pip install cget

#Install minimac4
RUN git clone https://github.com/Santy-8128/Minimac4.git && \
    cd Minimac4 && \
    bash install.sh && \
    ln -s /TOOLS/Minimac4/release-build/minimac4 /usr/bin/minimac4
#    sudo ln -s /TOOLS/Minimac3/bin/Minimac3-omp /usr/bin/minimac3
#    sudo ln -s /TOOLS/Minimac3/bin/Minimac3 /usr/bin/minimac3

RUN apt-get update && \
    apt-get autoremove openjdk-11-jre openjdk-11-jdk

RUN ln -s /usr/lib/jvm/java-8-openjdk-amd64 /usr/lib/jvm/default-java

ENV LC_ALL "C"
#ENV PLINKBINARY "/TOOLS/plink_1.9/plink"
#ENV EAGLEBINARY "/TOOLS/Eagle_v2.3/eagle"
#ENV IMPUTE2BINARY "/TOOLS/impute_v2.3.2_x86_64_static/impute2"
#ENV QCTOOLBINARY "/TOOLS/qctool_v1.4-linux-x86_64/qctool
#ENV SNPTESTBINARY "/TOOLS/snptest_v2.5_linux_x86_64_static/snptest_v2.5"
#ENV MINIMACBINARY "/TOOLS/Minimac3/bin/Minimac3"
ENV RSCRIPTDIR "/TOOLS/R_scripts/"
ENV SHAPEITBINARY "/TOOLS/shapeit.v2.r727.linux.x64"
#ENV MINIMACBINARY "/TOOLS/Minimac3/bin/Minimac3"

ENV PLINKBINARY "/usr/bin/plink"
ENV QCTOOLBINARY "/usr/bin/qctool1.4"
ENV EAGLEBINARY "/usr/bin/eagle"
ENV IMPUTE2BINARY "/usr/bin/impute2"
ENV QCTOOLBINARY "/usr/bin/qctool1.4"
ENV SNPTESTBINARY "/usr/bin/snptest_v2.5"
ENV MINIMAC3BINARY "/usr/bin/minimac3"
ENV MINIMAC4BINARY "/usr/bin/minimac4"

ENV RSCRIPTBINDIR "/usr/bin/"

ENV BCFTOOLSBINARY "/usr/bin/bcftools"
#ENV QCTOOLSNEWBINARY "/gpfs/scratch/pr1ees00/pr1ees14/GCAT/SHAPEIT_IMPUTE/qctool/build/release/qctool_v2.0-rc9"
#ENV QCTOOLSNEWBINARY "/usr/bin/qctool2.0" ## THE INSTALLATION IS NOT PERFORMED BECAUSE THIS BINARY IS NOT USED IN THE CODE
ENV SAMTOOLSBINARY "/usr/bin/samtools"
```

## A.2 Docker generation

```
#!/bin/bash

cp ../../src/main/R/* ./TOOLS/R_scripts/

sudo docker build -f GuidanceDockerfile -t docker_guidance .

echo "[INFO] Docker build successfully executed."
#sudo docker run docker_guidance&
#sudo docker save --output=docker_singularity.tar docker_guidance
#sudo docker ps
#sudo sudo docker exec -i -t {name or id} /bin/bash
```

## A.3 Singularity build file

```
Bootstrap: docker
From: docker://localhost:5000/docker_guidance:latest

%environment
    export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64

%setup
    mkdir -p $SINGULARITY_ROOTFS/gpfs/home/
    mkdir -p $SINGULARITY_ROOTFS/gpfs/scratch/
    mkdir -p $SINGULARITY_ROOTFS/gpfs/apps/MN4
    mkdir -p $SINGULARITY_ROOTFS/gpfs/projects/
    mkdir -p /opt/intel
    mkdir -p /scratch

# Files that are included from the host

# %files

%post

```

## A.4 Singularity generation

```
#!/bin/bash

rm -f guidance_singularity.img

#sudo docker run -d -p 5000:5000 --name registry registry:2
#sudo docker pull docker_guidance
#sudo docker image tag docker_guidance localhost:5000/docker_guidance_image
#sudo docker push localhost:5000/docker_guidance_image

#export SINGULARITY_NOHTTPS=true

## DOCKER IMAGE GENERATION ##

sudo docker ps -a | tail -n +2 | awk '{ print $1 }' | xargs -i sudo docker stop {}
sudo docker ps -a | tail -n +2 | awk '{ print $1 }' | xargs -i sudo docker rm {}

sudo docker image ls | grep "docker_guidance" | awk '{ print $3 }' | tail -n +2 | xargs
     -i sudo docker rmi -f {}

pushd docker
./build_docker.sh
popd

#sudo docker stop registry
#sudo docker ps -a | grep registry:2 | xargs -i sudo docker rm {}
sudo docker rmi -f registry:2
```

```
sudo docker run -d -p 5000:5000 --restart=always --name registry registry:2
sudo docker tag docker_guidance localhost:5000/docker_guidance
sudo docker push localhost:5000/docker_guidance

## SINGULARITY IMAGE GENERATION ##

# https://github.com/singularityware/singularity/issues/429

sudo SINGULARITY_NOHTTPS=yes singularity build guidance_singularity.img singularity/
    guidance.def
```

# Appendix B

# MC

## B.1 MLMC poster

### Scalable distributed Multilevel Monte Carlo workflow design

Riccardo Tosi (CIMNE), Marc Nuñez (CIMNE), Ramon Amela (BSC),
Rosa M. Badia (BSC), Riccardo Rossi (CIMNE-UPC), Rubén Zorrilla (CIMNE)

#### 1. Introduction

The following presents some initial results of integration of well-known algorithms developed to study Uncertainty Quantification (UQ) inside the KratosMultiphysics (Kratos) environment. The application of choice has been the resolution of the potential flow around an airfoil. The final aim is to perform Optimization Under Uncertainties (OUU) of the flow around civil structures, using embedded geometries.

#### 2. Monte Carlo and Multilevel Monte Carlo

The Monte Carlo (MC) method is the reference method in the stochastic analysis of multiphysics problems with uncertainties in the data parameters. The idea is to repeatedly generate the random input and to solve numerically the associated deterministic problem, in order to produce a statistical analysis.

- Problem under consideration considered as a black-box.
- MC estimator of the expectation of a Quantity of Interest (QoI):

$$\mathbb{E}^{\mathrm{MC}}[QoI] := \sum_{i=1}^{N} QoI(w^{(i)}) \ .$$

- Convergence to the exact statistics as the number of samples $N \to \infty$.
- Convergence rate of the mean square error $\sim O(N^{\frac{1}{2}})$:

$$\mathrm{mse}^2_{\mathrm{MC}} := \mathbb{E}[(\mathbb{E}^{\mathrm{MC}}[QoI_M] - \mathbb{E}[QoI])^2] \ .$$

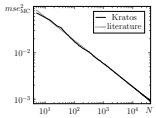



**Figure 1:** MC mean square error.

- Too high computational cost for complex problems ⇒ development of Multilevel Monte Carlo (MLMC) algorithms.

The MLMC main features are:

- Simultaneous computation of MC $QoI_{M_l}$ samples on successive refinement levels.
- MLMC estimator of the expectation of a QoI:

$$\mathbb{E}^{\mathrm{MLMC}}[QoI_M] := \sum_{l=0}^{L} \mathbb{E}[QoI_{M_l}(w^{(i,l)}) - QoI_{M_{l-1}}(w^{(i,l)})] \ .$$

- Combination of a large number of cheap and low accuracy $QoI_{M_l}$ samples with few expensive high accuracy samples.

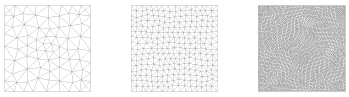


**Figure 2:** Hierarchy of computational grids, showing increasing accuracy levels.

#### 3. HPC implementation

Both the MC and the MLMC algorithms are parallelizable, since their working principle is to solve repeatedly the same problem of interest, each time with a different random input. This led to the integration among Kratos and PyCOMPSs, the python library of COMPSs. The current version is able to run several thousands of samples at once.

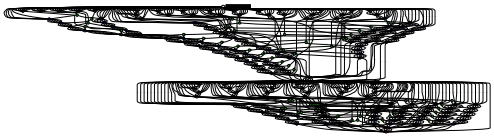


**Figure 3:** Graph connections of MLMC algorithm dependencies, running with PyCOMPSs.

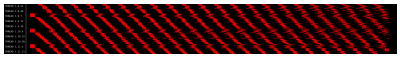


**Figure 4:** Section of the execution trace of a MC execution with $N = 16000$.

#### 4. Embedded solver

The use of embedded geometries presents some clear advantages if compared to typical body-fitted meshes.

- Dealing with incomplete geometries as input files (geometries with gaps, holes or overlaps).
- Solving problems with large boundary movement (typical in optimization).
- Accounting for complex geometries, such as volume-less bodies.

In the following figures, a simple example of the flow around an ellipse with an angle of attack of 5° is showcased, as well as a comparison to reference results.

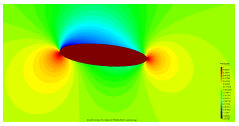


**Figure 5:** Example of the pressure distribution.

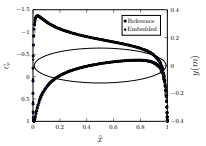



**Figure 6:** Embedded solver comparison to reference (XFoil).

#### 5. Adaptative remeshing

Adaptative remeshing enhances the accuracy of the embedded solver, thanks to the better definition of the level-set representing the input geometry. The following figures showcase the remesh of a NACA 0012 airfoil, using the MMG library.

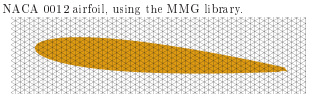


**Figure 7:** NACA 0012 airfoil embedded in an initial background mesh. Observe that the element size of the background mesh is not enough to account for the sharp geometry of the trailing edge.

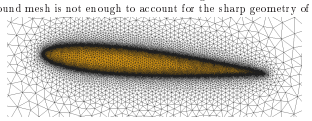


**Figure 8:** NACA 0012 airfoil, after remsehing the background mesh in terms of the inital geometry.

#### 6. Computation of sensitivities

To perform an optimization analysis with an aerodynamic solver, the computation of the gradient of an objective function is needed. This gradient is the sensitivity of the geometry with respect to the objective function. In this case, the objective functions will be aerodyanimc forces or characteristics, whose gradient with respect to the geometry is unknowkn. Thus, adjoint techinques are used to compute the gradient, which are validated with the finite differences method.

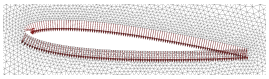


**Figure 9:** Visual representation of the sensitivities of each geometry parameter, i.e. each node that defines the airfoil geometry, computed using adjoint techniques.

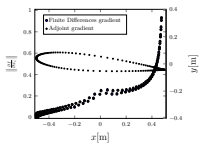



**Figure 10:** Comparison between the adjoint analysis and the finite difference method. The relative error obtained is 0.17%.

#### 7. Future Developments

- Extend embedded solver to 3D.
- Computation of sensitivities for embedded geometries.
- Optimization of MC and MLMC parallelization.
- Application of MLMC to more challenging physical problems.

#### 8. References

[1] Pooyan Dadvand, Riccardo Rossi, and Eugenio Oñate. An object-oriented environment for developing finite element codes for multi-disciplinary applications. *Archives of computational methods in engineering*, 17(3):253–297, 2010.

[2] Ramon Amela, Cristian Ramon-Cortes, Jorge Ejarque, Javier Conejero, and Rosa M Badia. Executing linear algebra kernels in heterogeneous distributed infrastructures with pycompss. *Oil & Gas Science and Technology–Revue d'IFP Energies nouvelles*, 73:47, 2018.

[3] M Davari, Riccardo Rossi, Pooyan Dadvand, Inigo Lopez, and Roland Wüchner. A cut finite element method for the solution of the full-potential equation with an embedded wake. *Computational Mechanics*, 08 2018.

[4] MMG - Surface and volume remeshers. https://github.com/MmgTools/mmg. Accessed: 21.03.2019.

#### Contacts

rtosi@cimne.upc.edu
mnunez@cimne.upc.edu
ramon.amela@bsc.es
rosa.m.badia@bsc.es
rrossi@cimne.upc.edu
rzorrilla@cimne.upc.edu

# Appendix C

# Scheduling improvements

## C.1 Scheduler auxiliar structures

```
    // Tree set is an ordered set!!
    protected HashMap<ResourceScheduler<?>, TreeSet<ObjectValue<AllocatableAction>>>
    unassignedReadyActions;
    protected final HashSet<ResourceScheduler<?>> availableWorkers;
    protected final HashMap<ResourceScheduler<?>, Future<?>> resourceTokens;
    protected int amountOfWorkers;
    ThreadPoolExecutor schedulerExecutor;
```

## C.2 Scheduler main function

```
    protected <T extends WorkerResourceDescription> void tryToLaunchFreeActions(List<
    AllocatableAction> dataFreeActions,
            List<AllocatableAction> resourceFreeActions, List<AllocatableAction>
    blockedCandidates,
            ResourceScheduler<T> resource) {
        if (DEBUG) {
            LOGGER.debug("[ReadyScheduler] Try to launch free actions on resource " +
    resource.getName() + " with "
                    + this.unassignedReadyActions.get(resource).size() + " candidates
    in this worker");
        }

        // Actions that have been freeded by the action that just finished
        for (AllocatableAction freeAction : dataFreeActions) {
            if (DEBUG) {
                LOGGER.debug(
                        "[ReadyScheduler] Introducing action " + freeAction + " into
    the scheduler from data free");
            }
            addActionToSchedulerStructures(freeAction);
        }
        dataFreeActions = new LinkedList<AllocatableAction>();

        // Resource free actions should always be empty in this scheduler
        for (AllocatableAction freeAction : resourceFreeActions) {
            if (DEBUG) {
                LOGGER.debug(
                        "[ReadyScheduler] Introducing action " + freeAction + " into
    the scheduler from resource free");
            }
            addActionToSchedulerStructures(freeAction);
        }
        resourceFreeActions = new LinkedList<AllocatableAction>();

        // Only in case there are actions that have entered the scheduler without
    having
        // available resources -> They were in the blocked list
        for (AllocatableAction freeAction : blockedCandidates) {
```

```
32              if (DEBUG) {
                    LOGGER.debug("[ReadyScheduler] Introducing action " + freeAction + "
    into the scheduler from blocked");
34              }
                addActionToSchedulerStructures(freeAction);
36          }
            blockedCandidates = new LinkedList<AllocatableAction>();
38
            Future<?> lastToken = this.resourceTokens.get(resource);
40          if (lastToken != null) {
                try {
42                  lastToken.get();
                } catch (InterruptedException | ExecutionException e) {
44                  e.printStackTrace();
                    LOGGER.fatal("Unexpected thread interruption");
46                  ErrorManager.fatal("Unexpected thread interruption");
                }
48          }
            this.resourceTokens.put(resource, null);
50
            Iterator<ObjectValue<AllocatableAction>> executableActionsIterator = this.
    unassignedReadyActions.get(resource)
52                  .iterator();
            HashSet<ObjectValue<AllocatableAction>> objectValueToErase = new HashSet<
    ObjectValue<AllocatableAction>>();
54          while (executableActionsIterator.hasNext() && !this.availableWorkers.isEmpty())
     {
                ObjectValue<AllocatableAction> obj = executableActionsIterator.next();
56              AllocatableAction freeAction = obj.getObject();
                try {
58                  if (Tracer.isActivated()) {
                        Tracer.emitEvent(Tracer.Event.TRY_TO_SCHEDULE.getId(), Tracer.Event
    .TRY_TO_SCHEDULE.getType());
60                  }
                    freeAction.tryToSchedule(obj.getScore(), this.availableWorkers);
62                  if (Tracer.isActivated()) {
                        Tracer.emitEvent(Tracer.EVENT_END, Tracer.Event.TRY_TO_SCHEDULE.
    getType());
64                  }
                    ResourceScheduler<? extends WorkerResourceDescription> assignedResource
     = freeAction
66                          .getAssignedResource();
                    tryToLaunch(freeAction);
68                  if (!assignedResource.canRunSomething()) {
                        this.availableWorkers.remove(assignedResource);
70                  }
                    objectValueToErase.add(obj);
72              } catch (BlockedActionException e) {
                    if (Tracer.isActivated()) {
74                      Tracer.emitEvent(Tracer.EVENT_END, Tracer.Event.TRY_TO_SCHEDULE.
    getType());
                    }
76                  objectValueToErase.add(obj);
                    addToBlocked(freeAction);
78                  if (DEBUG) {
                        LOGGER.debug("[ReadyScheduler] Action " + freeAction + " added to
    blocked actions");
80                  }
                } catch (UnassignedActionException e) {
82                  System.out.println("Cannot schedule action " + freeAction);
                    if (Tracer.isActivated()) {
84                      Tracer.emitEvent(Tracer.EVENT_END, Tracer.Event.TRY_TO_SCHEDULE.
    getType());
                    }
86                  if (DEBUG) {
                        LOGGER.debug("[ReadyScheduler] Action " + freeAction
88                              + " could not be assigned to any of the available resources
    ");
                    }
90                  // Nothing to be done here since the action was already in the
    scheduler
```

```
                // structures. If there is an exception, the freeAction will not be
    added
92              // to the objectValueToErase list.
                // Hence, this is not an ignored Exception but an expected behavior.
94          }
        }
96
        for (ObjectValue<AllocatableAction> obj : objectValueToErase) {
98          AllocatableAction action = obj.getObject();
            removeActionFromSchedulerStructures(action);
100     }
    }
```

## C.3 Scheduler auxiliar calls

```
1   private void addActionToResource(
            Map.Entry<ResourceScheduler<?>, TreeSet<ObjectValue<AllocatableAction>>>
    currentEntry,
3           AllocatableAction action) {
        ResourceScheduler<?> resource = currentEntry.getKey();
5       TreeSet<ObjectValue<AllocatableAction>> actionList = (TreeSet<ObjectValue<
    AllocatableAction>>) currentEntry
                .getValue();
7       Score fullScore = action.schedulingScore(resource, generateActionScore(action))
    ;
        if (fullScore != null) {
9           ObjectValue<AllocatableAction> obj = new ObjectValue<>(action, fullScore);
            actionList.add(obj);
11      }
    }
13
    private void removeActionFromResource(
15          Map.Entry<ResourceScheduler<?>, TreeSet<ObjectValue<AllocatableAction>>>
    currentEntry,
            AllocatableAction action) {
17      currentEntry.getKey();
        TreeSet<ObjectValue<AllocatableAction>> actionList = currentEntry.getValue();
19      Score fullScore = action.schedulingScore(currentEntry.getKey(),
    generateActionScore(action));
        if (fullScore != null) {
21          ObjectValue<AllocatableAction> obj = new ObjectValue<>(action, fullScore);
            actionList.remove(obj);
23      }
    }
25
    private Runnable createAddRunnable(
27          final Map.Entry<ResourceScheduler<?>, TreeSet<ObjectValue<AllocatableAction
    >>> currentEntry,
            final AllocatableAction action, final Future<?> token) {
29      Runnable addRunnable = new Runnable() {
            public void run() {
31              if (token != null) {
                    try {
33                      token.get();
                    } catch (InterruptedException | ExecutionException e) {
35                      e.printStackTrace();
                        LOGGER.fatal("Unexpected thread interruption");
37                      ErrorManager.fatal("Unexpected thread interruption");
                    }
39              }
                addActionToResource(currentEntry, action);
41          }
        };
43      return addRunnable;
    }
45
    private Runnable createRemoveRunnable(
```

```
47              final Map.Entry<ResourceScheduler<?>, TreeSet<ObjectValue<AllocatableAction
    >>> currentEntry,
            final AllocatableAction action, final Future<?> token) {
49          Runnable removeRunnable = new Runnable() {
            public void run() {
51              if (token != null) {
                    try {
53                      token.get();
                    } catch (InterruptedException | ExecutionException e) {
55                      e.printStackTrace();
                        LOGGER.fatal("Unexpected thread interruption");
57                      ErrorManager.fatal("Unexpected thread interruption");
                    }
59              }
                removeActionFromResource(currentEntry, action);
61          }
        };
63      return removeRunnable;
    }
65
    private void addActionToSchedulerStructures(AllocatableAction action) {
67      if (!this.unassignedReadyActions.isEmpty()) {
            if (DEBUG) {
69              LOGGER.debug("[ReadyScheduler] Add action to scheduler structures " +
    action);
            }
71          Iterator<Map.Entry<ResourceScheduler<?>, TreeSet<ObjectValue<
    AllocatableAction>>>> iter = unassignedReadyActions
                    .entrySet().iterator();
73          Map.Entry<ResourceScheduler<?>, TreeSet<ObjectValue<AllocatableAction>>>
    currentEntry = iter.next();
            TreeSet<ObjectValue<AllocatableAction>> actionList = (TreeSet<ObjectValue<
    AllocatableAction>>) currentEntry
75                  .getValue();

77          ResourceScheduler<?> resource = currentEntry.getKey();
            Score actionScore = generateActionScore(action);
79          Score fullScore = action.schedulingScore(resource, actionScore);
            ObjectValue<AllocatableAction> obj = new ObjectValue<>(action, fullScore);
81          if (!actionList.add(obj)) {
                return;
83          }
            while (iter.hasNext()) {
85              currentEntry = iter.next();
                resource = currentEntry.getKey();
87              Future<?> lastToken = this.resourceTokens.get(resource);
                this.resourceTokens.put(resource,
89                      schedulerExecutor.submit(createAddRunnable(currentEntry, action
    , lastToken)));
            }
91      } else {
            if (DEBUG) {
93              LOGGER.debug(
                        "[ReadyScheduler] Cannot add action " + action + " because
    there are not available resources");
95          }
            addToBlocked(action);
97      }
    }
99
    private void removeActionFromSchedulerStructures(AllocatableAction action) {
101     if (!this.unassignedReadyActions.isEmpty()) {
            if (DEBUG) {
103             LOGGER.debug("[ReadyScheduler] Remove action from scheduler structures
    " + action);
            }
105         Iterator<Map.Entry<ResourceScheduler<?>, TreeSet<ObjectValue<
    AllocatableAction>>>> iter = unassignedReadyActions
                    .entrySet().iterator();
107         Map.Entry<ResourceScheduler<?>, TreeSet<ObjectValue<AllocatableAction>>>
    currentEntry = iter.next();
            ResourceScheduler<?> resource = currentEntry.getKey();
```

```
109             TreeSet<ObjectValue<AllocatableAction>> actionList = currentEntry.getValue
    ();
                Score actionScore = generateActionScore(action);
111             Score fullScore = action.schedulingScore(resource, actionScore);
                ObjectValue<AllocatableAction> obj = new ObjectValue<>(action, fullScore);
113             if (!actionList.remove(obj)) {
                    return;
115             }
                while (iter.hasNext()) {
117                 currentEntry = iter.next();
                    resource = currentEntry.getKey();
119                 Future<?> lastToken = this.resourceTokens.get(resource);
                    this.resourceTokens.put(resource,
121                         schedulerExecutor.submit(createRemoveRunnable(currentEntry,
    action, lastToken)));
                }
123         }
    }
125

127     protected <T extends WorkerResourceDescription> void tryToLaunchFreeActions(List<
    AllocatableAction> dataFreeActions,
                List<AllocatableAction> resourceFreeActions, List<AllocatableAction>
    blockedCandidates,
129             ResourceScheduler<T> resource) {
            if (DEBUG) {
131             LOGGER.debug("[ReadyScheduler] Try to launch free actions on resource " +
    resource.getName() + " with "
                        + this.unassignedReadyActions.get(resource).size() + " candidates
    in this worker");
133         }

135         // Actions that have been freeded by the action that just finished
            for (AllocatableAction freeAction : dataFreeActions) {
137             if (DEBUG) {
                    LOGGER.debug(
139                         "[ReadyScheduler] Introducing action " + freeAction + " into
    the scheduler from data free");
                }
141             addActionToSchedulerStructures(freeAction);
            }
143         dataFreeActions = new LinkedList<AllocatableAction>();

145         // Resource free actions should always be empty in this scheduler
            for (AllocatableAction freeAction : resourceFreeActions) {
147             if (DEBUG) {
                    LOGGER.debug(
149                         "[ReadyScheduler] Introducing action " + freeAction + " into
    the scheduler from resource free");
                }
151             addActionToSchedulerStructures(freeAction);
            }
153         resourceFreeActions = new LinkedList<AllocatableAction>();

155         // Only in case there are actions that have entered the scheduler without
    having
            // available resources -> They were in the blocked list
157         for (AllocatableAction freeAction : blockedCandidates) {
                if (DEBUG) {
159                 LOGGER.debug("[ReadyScheduler] Introducing action " + freeAction + "
    into the scheduler from blocked");
                }
161             addActionToSchedulerStructures(freeAction);
            }
163         blockedCandidates = new LinkedList<AllocatableAction>();

165         Future<?> lastToken = this.resourceTokens.get(resource);
            if (lastToken != null) {
167             try {
                    lastToken.get();
169             } catch (InterruptedException | ExecutionException e) {
                    e.printStackTrace();
```

```
171                 LOGGER.fatal("Unexpected thread interruption");
                    ErrorManager.fatal("Unexpected thread interruption");
173             }
            }
175         this.resourceTokens.put(resource, null);

177         Iterator<ObjectValue<AllocatableAction>> executableActionsIterator = this.
    unassignedReadyActions.get(resource)
                .iterator();
179         HashSet<ObjectValue<AllocatableAction>> objectValueToErase = new HashSet<
    ObjectValue<AllocatableAction>>();
            while (executableActionsIterator.hasNext() && !this.availableWorkers.isEmpty())
     {
181             ObjectValue<AllocatableAction> obj = executableActionsIterator.next();
                AllocatableAction freeAction = obj.getObject();
183             try {
                    if (Tracer.isActivated()) {
185                     Tracer.emitEvent(Tracer.Event.TRY_TO_SCHEDULE.getId(), Tracer.Event
    .TRY_TO_SCHEDULE.getType());
                    }
187                 freeAction.tryToSchedule(obj.getScore(), this.availableWorkers);
                    if (Tracer.isActivated()) {
189                     Tracer.emitEvent(Tracer.EVENT_END, Tracer.Event.TRY_TO_SCHEDULE.
    getType());
                    }
191                 ResourceScheduler<? extends WorkerResourceDescription> assignedResource
     = freeAction
                            .getAssignedResource();
193                 tryToLaunch(freeAction);
                    if (!assignedResource.canRunSomething()) {
195                     this.availableWorkers.remove(assignedResource);
                    }
197                 objectValueToErase.add(obj);
                } catch (BlockedActionException e) {
199                 if (Tracer.isActivated()) {
                        Tracer.emitEvent(Tracer.EVENT_END, Tracer.Event.TRY_TO_SCHEDULE.
    getType());
201                 }
                    objectValueToErase.add(obj);
203                 addToBlocked(freeAction);
                    if (DEBUG) {
205                     LOGGER.debug("[ReadyScheduler] Action " + freeAction + " added to
    blocked actions");
                    }
207             } catch (UnassignedActionException e) {
                    System.out.println("Cannot schedule action " + freeAction);
209                 if (Tracer.isActivated()) {
                        Tracer.emitEvent(Tracer.EVENT_END, Tracer.Event.TRY_TO_SCHEDULE.
    getType());
211                 }
                    if (DEBUG) {
213                     LOGGER.debug("[ReadyScheduler] Action " + freeAction
                                + " could not be assigned to any of the available resources
    ");
215                 }
                    // Nothing to be done here since the action was already in the
    scheduler
217                 // structures. If there is an exception, the freeAction will not be
    added
                    // to the objectValueToErase list.
219                 // Hence, this is not an ignored Exception but an expected behavior.
            }
221     }

223     for (ObjectValue<AllocatableAction> obj : objectValueToErase) {
            AllocatableAction action = obj.getObject();
225         removeActionFromSchedulerStructures(action);
        }
227 }
```