Optimization of the cluster analysis tool pyProCT with pyCOMPSs

by Pol Alvarez

Supervisor: Víctor Alejandro Gil Sepúlveda

Institution of supervisor: Barcelona Supercomputing Center

Tutor: Rosa Maria Badia Sala

Department of tutor: UPC – FIB – Departament of Computer Architecture

Specialization: Computing

Defense date: 26/10/2015

Submitted as Final Project of Bachelor Degree in Informatics Engineering
at Barcelona School of Informatics of Universitat Politècnica de Catalunya - BarcelonaTech

Contents

1. Acknowledgements

I wish to express my sicere thanks to Víctor Gil Sepúlveda, supervisor of this project, for the continuos encouragement, valuable guidance and constant availability to discus issues.

Besides my advisor, I would like to thank Prof. Rosa Maria Badia Sala for her insightful comments and hard questions which incented me to widen my research from various perspectives.

I take this opportunity to also express gratitude to all of the Workflows and Distributed Computing members for their help and support. Special thanks to Francesc Lordan Gomís, Cristian Ramon-Cortés and Francisco Javier Conejero for their advices to further increase performance and help to solve issues.

I also place on record, my sense of gratitude to one and all, who directly or indirectly, have lent their hand in this venture.

2. Abstract

Nowadays the amount of available digital information is exponentially increasing. However, to use that raw data it is necessary to process it with some method. One of those methods is clustering, which is used to categorize data effectively. Thanks to it we can reduce the size of large data sets by extracting the most relevant information, usually the common features of a group or a subset of representatives. With the aforementioned dramatical increase in data size researchers have focused on improving performance as much as possible. The cost of managing and storing information has considerably dropped rendering modern supercomputers an accessible asset to speed up clustering tools. On this scenario we find pyProCT clustering tool and pyCOMPSs parallelization framework. The first is a software originally designed to work with proteins, hence the name **py**thon **Pro**tein **C**lustering **T**ool, which has evolved into a general purpose clustering analysis program. The later is a framework built at the Barcelona Supercomputing Center that facilitates the development of parallel computational workflows in Python. It is focused on providing an easy programming model able to run on supercomputers and cloud resources. This project will refactor pyProCT with pyCOMPSs framework in order to increase its performance by optimizing its executions on MareNostrum III supercomputer.

La quantitat d'informació digital disponible s'ha vist incrementada exponencialment ens els últims anys. Emperò, abans d'usar aquestes dades s'han de processar d'alguna manera. Un dels métodes utilitzats per a aquesta fi és el clustering, que es basa en categoritzar les dades efectivament. Gràcies al seu ús és possible reduir grans conjunts de dades mitjançant l'extracció de la informació més rellevant, habitualment els trets distintius d'un grup o subconjunt de elements representants. Tenint en compte el gran augment del tamany de les dades mençionat abans, les línies de recerca solen estar orientades en incrementar al màxim el rendiment. El cost de gestionar i emmagatzemar informació ha disminuït notablement últimament. Això ha fet que els superodinadors siguin accessibles a un major públic i convertint-los en un gran recurs per millorar les eines de clustering. És en aquest escenari que es presenten l'eina de clustering pyProCT i l'entorn de treball pyCOMPSs. El primer és un programa creat originalment per a treballar amb proteines que ha evolucionat fins a convertir-se una eina de clustering de propòsit general. El segon, per la seva banda, és un entorn de treball, creat al centre de supercomputació de Barcelona, que facilita el desenvolupament de d'aplicacions paral·leles en python. Està enfocat a proporcionar una interfície d'ús senzilla mentre a la vegada que la capacitat de ser executat en superodinadors i recursos en el núvol. L'objectiu d'aquest projecte és modificar pyProCT per a que usi pyCOMPSs. Esperem que això millori el seu rendiment gràcies a l'execució optimizada per a superodinadors com MareNostrum III, on es faran les proves.

3. Introduction

Nowadays the amount of available digital information is exponentially increasing. Just on 2015 we generated almost 8.000 exabytes of information. Facebook generates 105 terabytes of data each half hour, more than 48 hours of video per minute are uploaded to youtube and google has at least 1 million queries/minute. But why do we observe this massive increase? To start the cost of creating, managing and storing information has dramatically dropped: EMC Corporation estimates that on 2011 this cost has been cut to a 1/6 of what it was on 2005. But more importantly people is more connected than it has ever been; mobiles, websites and social channels are just some examples of a whole new world of data-generating people interactions.

In this scenario is where we found the hot topic of today: Big Data. So what is it?, usually the term is used referring to data sets too big or complex to be processed with traditional data applications or on-hand management tools. According to the IT giant Gartner, Inc Big Data can be characterized by the "3 Vs", velocity, volume and variety [12]:

> "Big data" is high-volume, -velocity and -variety information assets that demand cost-effective, innovative forms of information processing for enhanced insight and decision making.

However all this raw information needs to be processed and categorized in an effective way before being used. Cluster analysis methods are one of the most used tools to address this issue.

3.1 Clustering

The term **cluster analysis** (first used by Trion, 1939) refers to the task of sorting similar objects of a data set into groups (called clusters) in a way that the degree of similarity between each pair is maximal if they belong to the same cluster and minimal otherwise. Data sets can be imagined as points in a multidimensional space, where each feature of an object would represent a dimension. The CA methods need to identify, as efficiently as possible, the denser areas and group them into clusters.

Thanks to the clustering we can reduce the size of large data sets by extracting the most relevant information, usually the common features of a group or a subset of representatives. Cluster analysis (CA from now onwards) techniques thrive in the Big Data world because it's not feasible to manually label objects, there is no prior knowledge of the number and nature of the clusters and, also, their identifying traits may change over time.

It is important to note that cluster analysis it's not an specific algorithm but rather the general task to perform. Due to the fact that the similarity criteria it's subjective and can change a lot between data sets, there isn't an optimal clustering algorithm. This is the reason why there are so many clustering algorithms, each with it's advantages and inconveniences. Each algorithm uses it's own kind of cluster model that defines how the algorithm groups the items and defines the clusters. Some of the most relevant examples are:

Hierarchical Clustering Analysis (HCA)

These methods seek, as their name indicates, to build a hierarchy of clusters. These can be done by starting with all elements in one clusters and the divide them in a "top down" way. This method is called **Divisive**. Opposed to this one, we find the **Agglomerative** method, where each data point starts in a different cluster merging them as one moves up the hierarchy.

Centroid Clustering,

On these algorithms the similarity between different clusters is defined as the similarity between their centroids. **K-means** clustering is one of such methods. On it, each observation belongs to the nearest centroid which, in turn, serves as the representative or prototype of the cluster.

Distribution-based Clustering

Clusters are modelled by statistical distributions. On this category falls the well-known **expectation-maximization (EM) algorithm** which uses multivariate normal distributions.

Density Clustering,

These methods follow the intuitive notion, described earlier, of considering the observations as clouds of points in a multidimensional space and so, they identify clusters as connected dense regions in the data space. **DBSCAN** is one of such algorithms and it's one of the most common and cited in scientific literature.

It is also possible to classify clustering methods by some other properties such as:

Hard Clustering,

where each element belongs to a cluster or not.

Soft Clustering,

where each element has likelihood of belonging to a certain cluster.

Cluster analysis methods can be applied to a wide range of subjects. Basically it can be used in any context where finding groups in sets of data is useful, for example:

Image segmentation,

dividing an image into clusters or regions enhances a number of computer vision methods. Some examples are border detection or object recognition. [3]

Market analysis,

grouping enterprises [5] or consumers [15] to perform better market analysis or customize ads for each kind of consumer.

Education tracking,

grouping students to keep track of their record and apply more custom techniques to each student needs. [6]

Mathematical chemistry,

to analyse, group and find structural similarities in chemistry compounds, minerals, and any kind of material for which a chemical analysis is convenient. [7]

Most of the clustering analysis methods are not new. However with the dramatical increase in data size mentioned earlier, researchers have focused on improving their performance as much as possible. From this need arise new, but more rough, methods such as **canopy clustering** [16] which can process huge amounts of information by pre-partitioning data to then analyse smaller partitions with slower methods.

The increasing amount of information each data point contains it's also a problem for some algorithms. This information leads to high-dimensional data which, in turn, causes problems to a big part of the modern algorithms. This is known as the curse of dimensionality, which basically points out the fact that high-dimensional data often becomes sparse due to the large volume of space. It is important to note that this problem is not due to data itself but to the algorithm used. Some modern approaches try to overcome this difficulty by reducing the data-dimensionality. Methods such as **principal component analysis** [11] use just some part of it. **Subspace clustering** [1] is an example of them. It has adopted ideas from density-based algorithms.

Another way to boost clustering tools performance is high performance computing. Supercomputers provide an amount of computing power that no traditional computer can offer. They allow to extremely reduce execution times or handle applications which require humongous amounts of memory. Cloud computing further enhances the utility of HPC machines. For example, grid systems have a good synergy with the cloud: using virtualization to create OS instances adds features like self-service resource provisioning, scalability or elasticity to grids' raw computing power.[2][9]

3.2 pyProCT

The correct usage of clustering analysis methods is not easy: right algorithm selection, better parameters estimation or appropriate result analysis are just some of the problems that CA tools user faces. However, we also showed that CA is present in a lot of different subjects and is used, or would be useful, to people with limited knowledge both on algorithmic methods and statistics. On the other way around we also find that CA specialists may not be able to correctly assess the results of a clustering due to the nature of the data itself.

Python Protein Clustering Tool [10] (pyProCT from here onwards) focus is to deal with the aforementioned problems . It provides an improved clustering performance through the initial definition of an hypothesis or goal. This way, through a more "semantic" approach users can "guide" pyProCT without forcing them to deeply understand the pros and cons each method w.r.t to an specific kind of data.

First it computes the distance's matrix of each pair of elements; then it uses a number of different cluster analysis algorithms on the dataset trying to estimate the best parameters for each one, and, finally, it rates the performance of each method and parametrization with a common scoring function.

The programming language used for this tool is Python. Its use in big data applications has increased dramatically over the last years due to the big number of available scientific libraries and its easy usage. However, python still lacks easy solutions for distributed systems. Most of the available parallelization methods rely on the use of message-passing interfaces (MPI) or are best suited for embarrassingly parallel computations. Currently pyProCT's scheduler uses two of them: mpi4py [8] and python's multiprocessing module.

pyProCT could greatly benefit from HPC. Because of that, we want to refactor the software with a python framework designed for supercomputing systems: pyCOMPSs.

3.3 pyCOMPSs

PyCOMPSs [17] is a framework that facilitates the development of parallel computational workflows in Python. This framework provides a sequential programming model to achieve a parallel and optimized execution pipeline. This differs from other models and paradigms which require the developer to have a deep knowledge of the hardware executing the code such as MPI interfaces and OpenMP C pragmas amongst others. The user just needs to decorate the functions to be run as asynchronous parallel tasks.

It is based on the java framework COMPSs [14]. Its runtime deals with the data dependencies of the defined tasks and assigns them to the available resources in order to achieve the best execution pipeline. This runtime was originally created for GRID superscalar [4] from which COMPSs evolved. At the present time COMPSs team is working on a brand new release (1.3). The first stable version (1.2) added to GRID superscalar new features like cloud computing, better hardware abstraction and python and C++ APIs. The 1.3 release under development introduces new communication adaptors, which will further increase its performance, an easier usage and more options for the APIs.

COMPSs is infrastructure unaware making the code portable. Thanks to its set of pluggable connectors it can work with a wide range of infrastructures, such as clouds[13] and grids, while providing an uniform interface for the user. This increases its scalability and allows elasticity of resources.

The framework offers two interesting tools for execution analysis: the monitor and the tracing system. The monitoring offers online information of an execution such as diagrams of data dependencies, resources state details, statistics and easy access to the framework logs.The tracing tool generates trace files which allow users to analyse the performance with the graphical tool paraver [19].

4. Objectives

The goal of this project is to refactor pyProCT with pyCOMPSs programming model and framework. In order to achieve that three objectives were defined:

- **Understand pyProCT**
- **Refactor pyProCT** with pyCOMPSs
- **Validate the results**

4.1 Understand pyProCT

The first objective was to understand and analyse pyProCT. A complete description of the task's scheduling and pipeline was necessary to decide how to refactor it with pyCOMPSs. The methodology section 5.1 contains the software design description with detailed information of each section of the program and its control parameters.

4.2 Refactor pyProCT with pyCOMPSs

This objective comprises the code adaptation to make pyProCT work with the pyCOMPSs framework. The section 5.2 is composed of two parts: the initial setup required to test the code under development and the actual refactor with pyCOMPSs. The analysis tools research and a more detailed description of the old pyProCT scheduler can be found on appendix A.

4.3 Validate the results

The last objective refers to the correctness of the refactored code. We need to ensure that the new scheduler produces the same results as the old ones. To do so we used the validation software called pyProCT which will be described in section 5.3.

5. Methodology

5.1 Analysis of pyProCT

PyProCT uses **pyScheduler** controller to handle the execution of the algorithms . It features three modes: sequential, parallel, using python's multiprocessing module, and parallel using MPI. The refactor added a fourth mode to run the tool with pyCOMPSs. It is important to note that the modifications were limited to pyProCT, so COMPSs acts as a substitute of the current controller, not as a new scheduling method inside pyScheduler.

5.1.1 Algorithms

pyProCT uses the following five algorithms to find the best clustering. It also has an extra one which clusters the data randomly. This one is used for comparative purposes so it won't have more consideration that the utility it provides for other's behaviour analysis.

1. K-medoids
2. Hierarchical
3. DBSCAN
4. GROMOS
5. Spectral

For more information about the actual implementation and parameter estimation of pyProCT check dropbox documentation on G Documentation section.

5.1.2 Execution Flow

The execution flow of pyProCT can be subdivided into four main sections linked to the JSON script structure:

Global,

initialization of the software by reading parameters and options, parsing the JSON script, setting up the workspace and create the scheduler to be used.

Data,

construction of the distance matrix to be used. It offers three options: load, distance and rmsd.

Clustering,

calculation and evaluation of the clusterings.

Postprocess,

processing of the results to offer useful information about the clustering found.

On the next sections each part is going to be described, both it's execution flow and the json parameters associated with it.

Global

The global section is the responsible of initializing everything for the execution. This part is located on the main.py file and ends up calling the corresponding driver with the correct parameters. This is the main.py structure:

This main file creates and initializes the Driver class which is the one orchestrating all the execution. Then the Driver class creates the workspace handler and saves the parameters before starting the Data, Clustering and Postprocess sections. The following figure shows encased in red the part corresponding to the global section inside the Driver. The next sections will expand the boxes Data, Clustering and Postprocess.

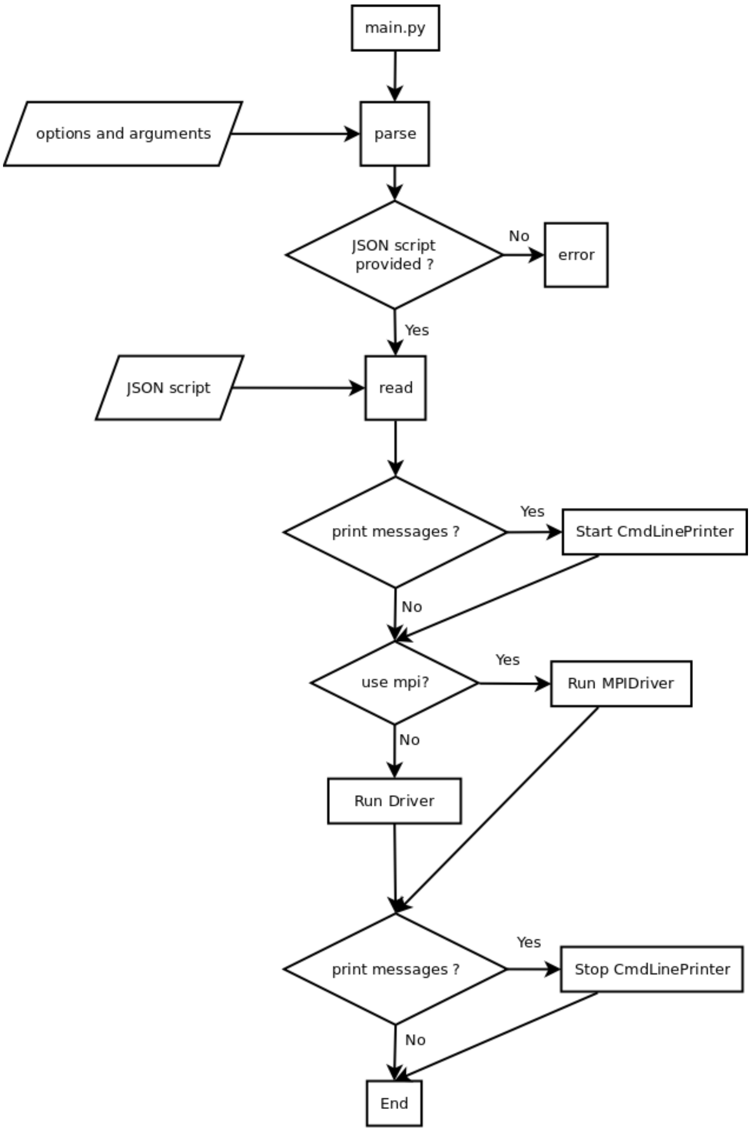


Figure 5.1: Global Section Execution Flow

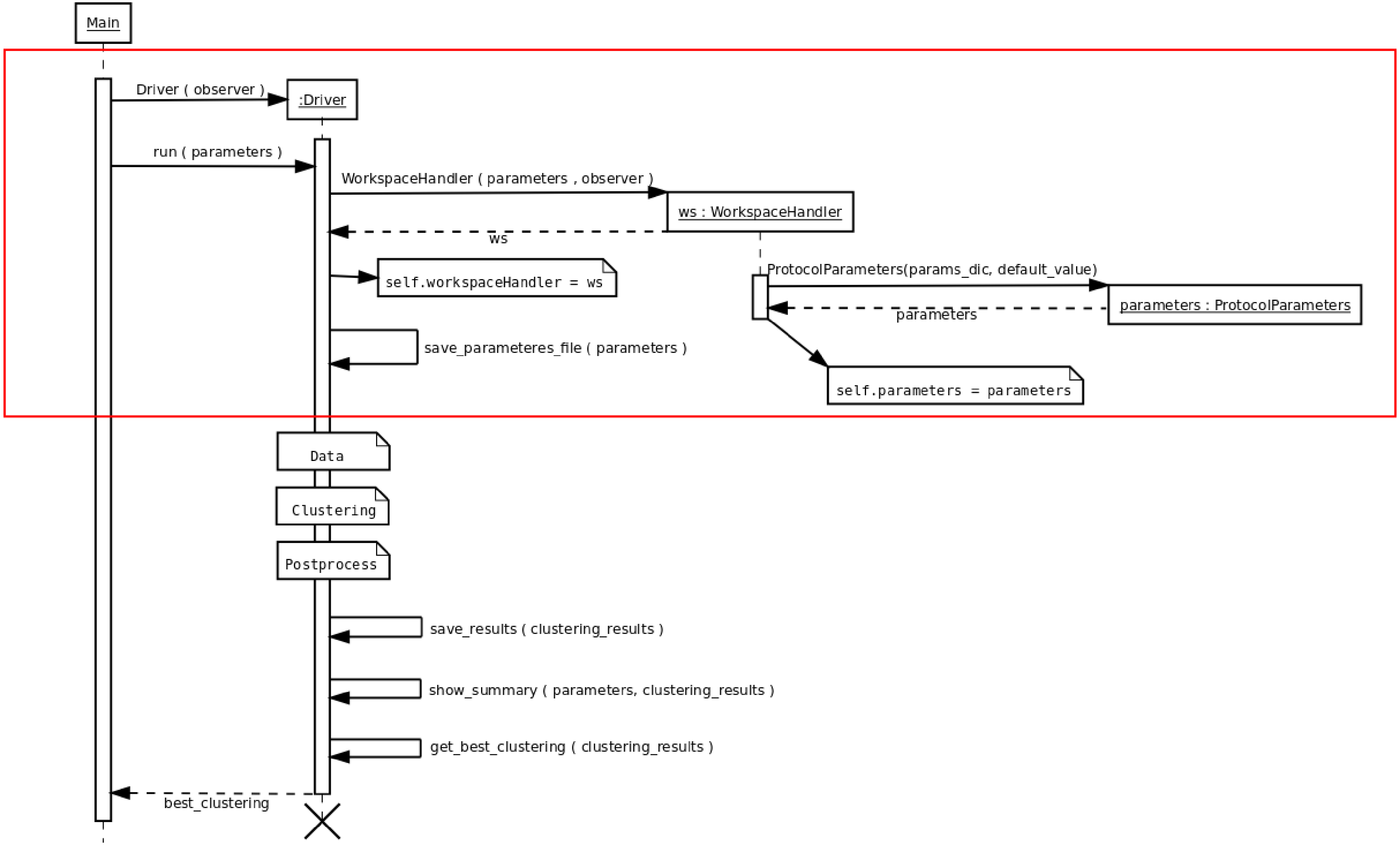

Main
Driver (observer)
:Driver
run (parameters)
WorkspaceHandler (parameters , observer)
ws : WorkspaceHandler
ws
self.workspaceHandler = ws
ProtocolParameters(params_dic, default_value)
parameters : ProtocolParameters
parameters
save_parameteres_file (parameters)
self.parameters = parameters
Data
Clustering
Postprocess
save_results (clustering_results)
show_summary (parameters, clustering_results)
get_best_clustering (clustering_results)
best_clustering

Figure 5.2: Global Section Execution Flow

The global section parameters that can be specified on the json file are divided into two groups: Control and Workspace.

- **Control**

  Scheduler type, defines the kind of scheduler to use (serial, parallel, MPI or, after the refactor, pyCOMPSs)

  Number of processes, if the parallel scheduler type is selected this option defines the number of processes to be used.

- **Workspace**

  Base, is mandatory and defines the base workspace path.

  Tmp, defines the folder to store temporal files.

  Matrix, defines the folder to store the distance matrix (if applicable).

  Clusterings, defines where cluster-related files are going to be stored, however the clusterings are stored as part of the results file.

  Results, defines where the results file should be stored.

  Parameters:

  > **Overwrite,** if true, existing folders will be removed before execution.
  >
  > **Clear after execution,** defines the folders to be removed after execution.

Data

This section defines how the distance matrix should be build. Essentially it runs the *DataDriver*'s method *run()* with the *WorkspaceHandler* initialized on the global section and the retrieved parameters. This data driver initializes and returns to the main driver the *DataHandler* and *MatrixHandler* to be used later. The first one is directly instantiated with the corresponding parameters. The second one, on the other hand, first loads the matrix calculator defined on matrix's method of the json control file. With this calculator, the data handler and the parameters, it computes and returns the desired matrix handler.

The following figure shows a simplified sequence diagram of this process:

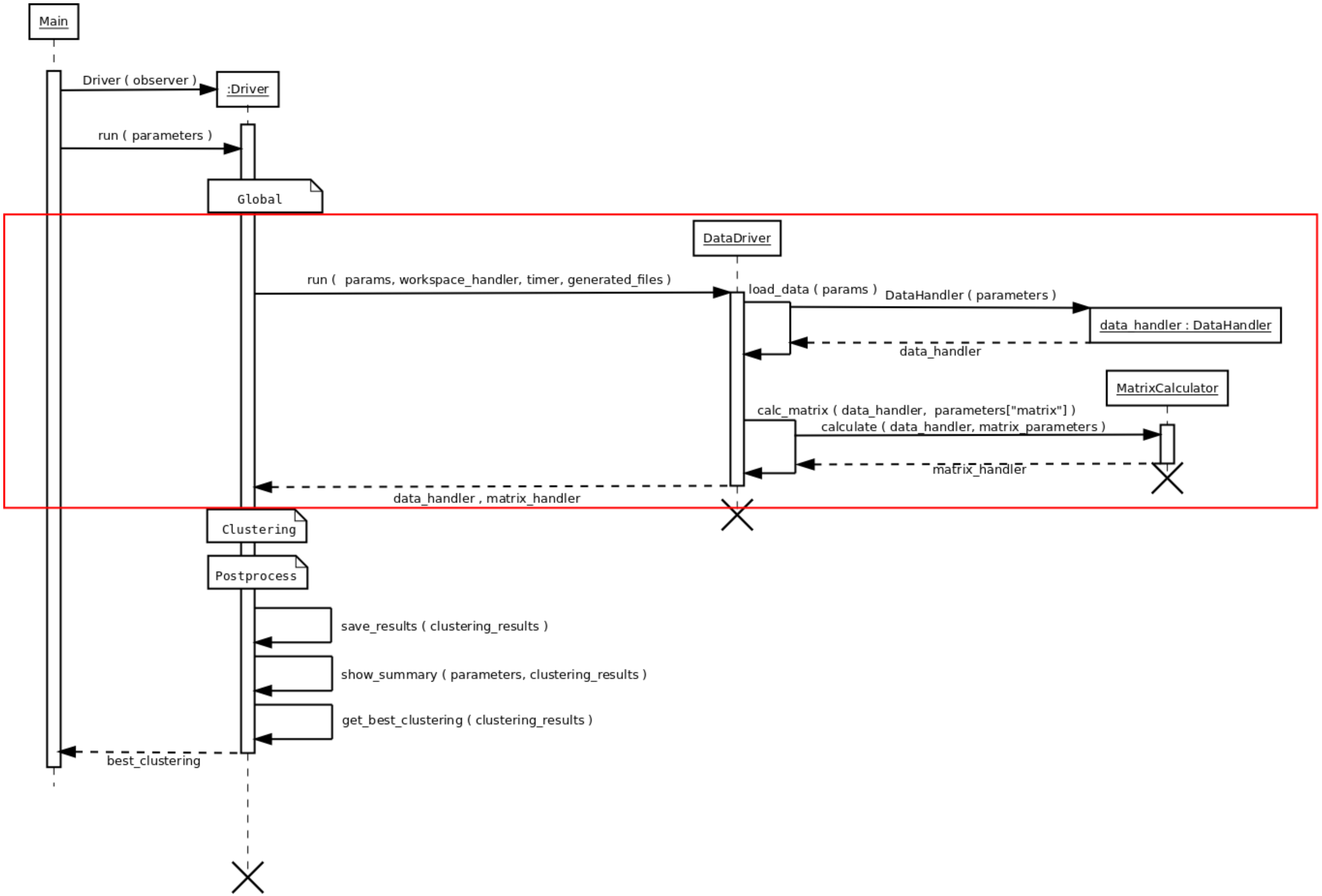

Main
Driver (observer)
:Driver
run (parameters)
Global
DataDriver
run (params, workspace_handler, timer, generated_files)
load_data (params)
DataHandler (parameters)
data_handler : DataHandler
data_handler
MatrixCalculator
calc_matrix (data_handler, parameters["matrix"])
calculate (data_handler, matrix_parameters)
matrix_handler
data_handler , matrix_handler
Clustering
Postprocess
save_results (clustering_results)
show_summary (parameters, clustering_results)
get_best_clustering (clustering_results)
best_clustering

Figure 5.3: Data Section Execution Flow

The data section parameters specify the type and origin of the data, the method used to calculate the distance matrix and some more matrix-related parameters:

Type, sets the kind of data loader to use for the dataset.

Files, defines the location of the input files.

Matrix

- **Method,** selects the method used to calculate the distance matrix.
- **Parameters,** allows to customize some parameters used to by the distance matrix calculator like:
    - **Calculator Type,** must be one of the local pyRMSD installation available ones.
    - **Fit Selection,** for distance or rmsd methods.
    - **Body Selection,** for distance method.
    - **Calculate Selection,** for rmsd method.
    - **Path,** in case we are loading the matrix.
- **Image,** setting this section will result in rendering a visual representation of the matrix.
    - **Filename,** desired path of the rendered image.
    - **Dimension,** sets the leading dimension of the matrix image [default:1000px]
- **Filename,** name of the file where the distance matrix will be saved (if applicable) inside the folder defined on workspace::matrix section.

Clustering

This section is the one performing the actual clustering exploration and evaluation of the results.

As Figure 5.4 shows, the driver calls it's *clustering_ section()* function which checks wether it needs to perform the exploration or load an existing clustering.

If the method selected is "load" then the function *from_ dic(...)* turns the data into a Clustering instance.

If "generate" is the selected method then it calls *perform_ clustering_ exploration(...)* which initializes and runs the *ClusteringProtocol* class. This one, in turn, runs and initializes the classes *ClusteringExplorer*, *ClusteringFilter*, *AnalysisRunner* and the *BestClusteringSelector*.

This clustering explorer deals with the actual exploration pipeline. It generates diverse parameter structures for each defined CA algorithm and adds them to the scheduler tasks queue, runs the scheduler and returns the resulting clustering_info structures.

The clustering filter tries to reduce the size of the clustering. To achieve this it eliminates the clusters whose parameters are outside the defined ranges on the evaluation section, removes the not selected clusters and checks that there are no repeated clusterings amongst the selected ones.

The analysis' runner is the one handling the evaluation of the selected clusterings. Similarly to *ClusteringProtocol*, it creates an scheduler instance, queues the parametrization of each clustering analysis into it and runs it. Finally, it attaches the results to the clustering structure evaluated.

The BestClusteringSelector normalizes the calculated evaluations, scores each evaluation with the defined criteria and finally returns the id with higher score and the score itself.

Finally the ClusteringProtocol returns the clustering results containing: *best_ clustering_ id*, *selected_ clusterings*, *not_ selected_ clusterings* and *all_ scores*. This results are then returned to the driver for the postprocessing section.

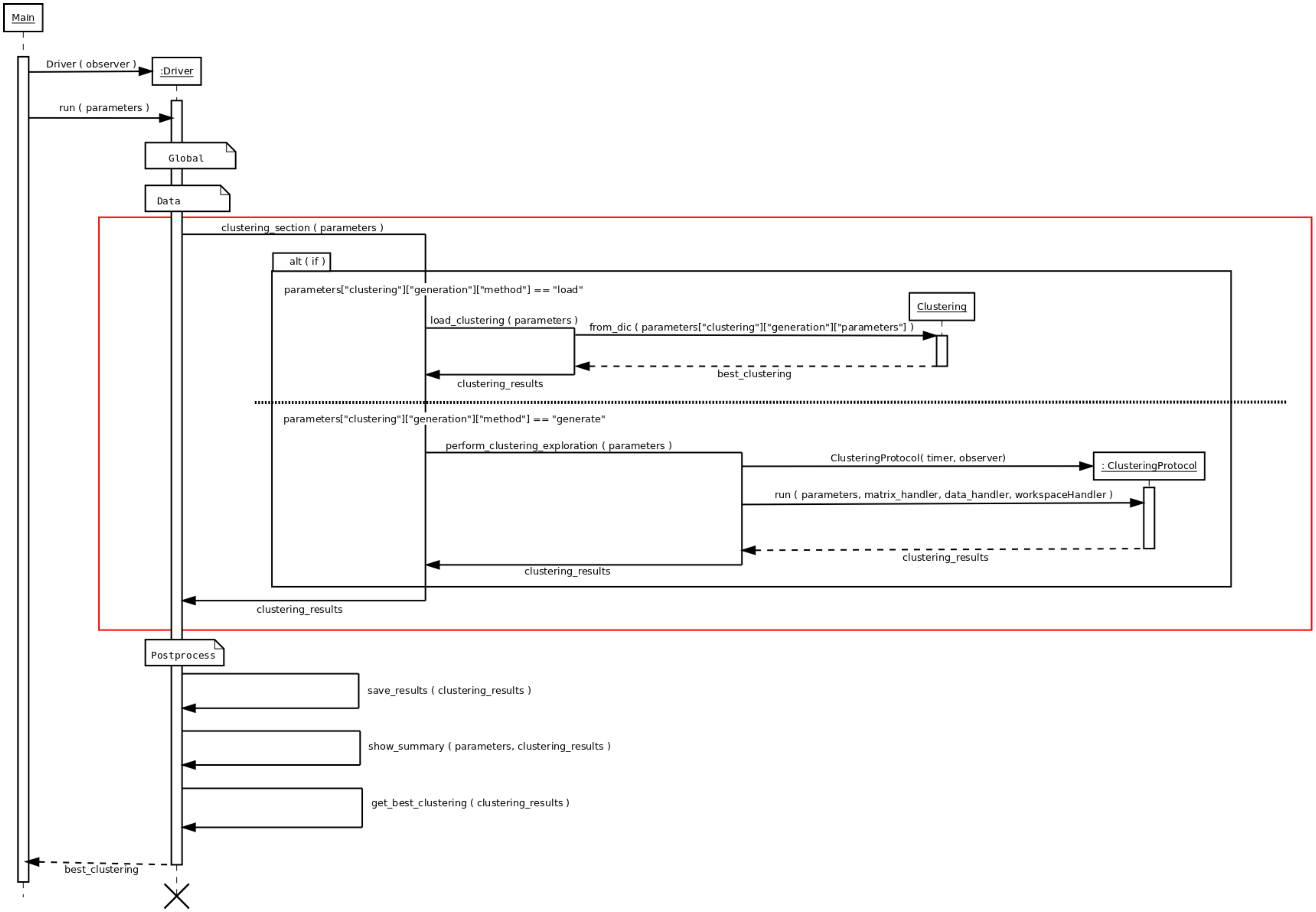

Main
Driver (observer)
:Driver
run (parameters)
Global
Data
clustering_section (parameters)
alt (if)
parameters["clustering"]["generation"]["method"] == "load"
Clustering
load_clustering (parameters)
from_dic (parameters["clustering"]["generation"]["parameters"])
best_clustering
clustering_results
parameters["clustering"]["generation"]["method"] == "generate"
perform_clustering_exploration (parameters)
ClusteringProtocol(timer, observer)
: ClusteringProtocol
run (parameters, matrix_handler, data_handler, workspaceHandler)
clustering_results
clustering_results
clustering_results
Postprocess
save_results (clustering_results)
show_summary (parameters, clustering_results)
get_best_clustering (clustering_results)
best_clustering

Figure 5.4: Clustering Section Execution Flow

The clustering section parameters that can be specified on the json file define if the clustering should be generated or loaded, which algorithms and parameters to use and, finally, the evaluation section which is where the user should define his goal or hypothesis.

Generation

Method, selects wether we want to load or calculate the best clustering info. If it's loaded it will use the json dictionary defined on clustering::generation::clusters.

Clusters, clustering data if the method is "load". Each cluster object must define: id, prototype and elements.

Algorithms,

for each desired algorithm to use of the six available (dbscan, gromos, hierarchical, kmedoids, random and spectral) defines it's parameters. All algorithms share the 'max', defining the maximim number of parametrizations for the algorithm, and the 'parameters' properties.

Kmedoids,

Seeding type, defines if the initial seeds should be randomly placed or at equidistant points.

Tries, if the initial seeds are to be randomly placed, this defines the number of repetitions done with different seeds (default: 10).

Spectral,

Sigma, defines the sigma parameter for the spectral clustering. If not set, the default is to calculate local sigmas.

Evaluation

Minimum clusters, minimum number of clusters each clustering must contain to be evaluated.

Maximum clusters, maximum number of clusters a clustering must contain to be evaluated.

Minimum cluster size, any cluster smaller than this threshold will be considered noise (thus increasing the clustering noise).

Minimum noise, clusterings with higher noise than this threshold won't be evaluated.

Query types, list of details to be reported about the clustering found.

Evaluation criteria, list of criteria objects, each criteria containing one or more evaluation objects.

Evaluation object, defines the sigma parameter for the spectral clustering. If not set, the default is to calculate local sigmas.

Name, defining the quality function.

Action, defines wether the function should be maximized (">") or minimized ("<").

Weight, defines the relative weight of this quality function (not mandatory that they add up to 1).

Postprocess

The postprocessing section's allows users to extract useful information about the clustering. This section is the only optional one amongst the four described.

The Driver first gets the best clustering, which is the only remaining information needed to call the PostprocessingDriver class. The run method of this class loads all the available action classes and, for each one defined on the postprocessing section of the json file, runs it with the clustering information provided. The extracted information needs to be visualized with pyProCT GUI in some cases and saved into pdb files on the others (refer to Postprocessing Parameters for more info)

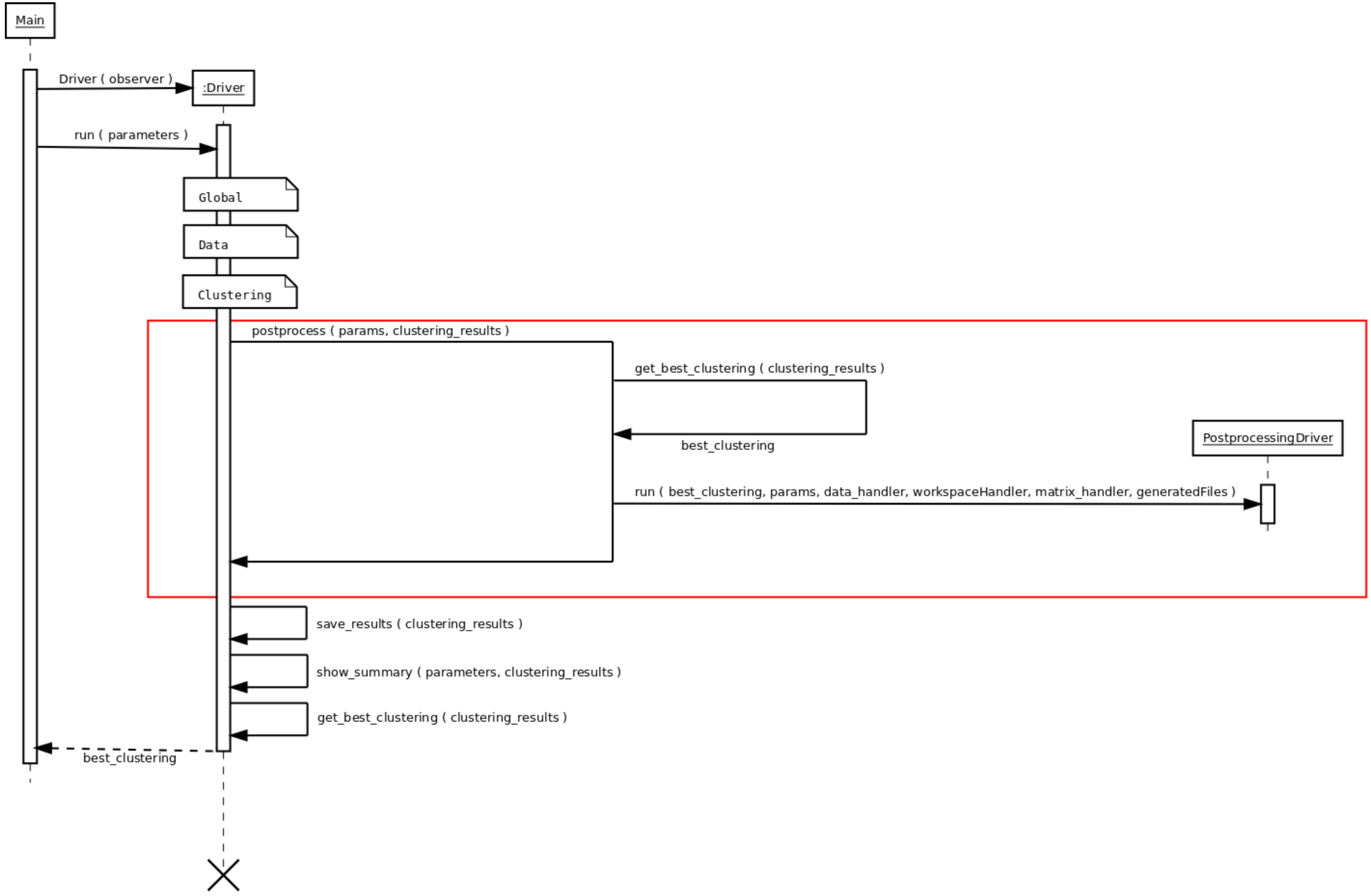

Main
:Driver
Driver (observer)
run (parameters)
Global
Data
Clustering
postprocess (params, clustering_results)
get_best_clustering (clustering_results)
best_clustering
PostprocessingDriver
run (best_clustering, params, data_handler, workspaceHandler, matrix_handler, generatedFiles)
save_results (clustering_results)
show_summary (parameters, clustering_results)
get_best_clustering (clustering_results)
best_clustering

Figure 5.5: Postprocess Section Execution Flow

These are the possible postprocessing actions to be performed:

Rmsf, pyProCT will generate the global and per-cluster rmsf data to be visualized with the GUI.

Centers and trace, pyProCT will generate the data of all geometrical centers of the calculation selection of the system (to be visualized with the GUI)

Representatives, pyProCT will save the data of the medoid of each cluster on the best clustering in a pdb file.

- **Keep remarks,** if true, stored models will be saved with the their original remarks header (default: false).
- **Keep frame number,** if set to true, the model number of any stored conformation will match the original pdb one (default: false).

Pdb clusters, pyProCT will save each cluster information in a pdb file.

- **Keep remarks,** if true, stored models will be saved with the their original remarks header (default: false).
- **Keep frame number,** if set to true, the model number of any stored conformation will be the original pdb one. Default: false.

Compression, this option will produce a compressed version of the input trajectories with less redundancy thanks to the resulting clustering.

- **File,** name of the output file withouth extension (default:compressed.pdb)
- **Final number of frames,** number of frames the compressed file must have.
- **Type,** sampling method for cluster elements (default: kmedoids).
  - **Random,** randomly samples elements for each cluster.
  - **Kmedoids,** uses k-medoids to get samples of the clusters.

Cluster stats, this will generate a human readable file with the distance among cluster centers and their diameters.

File, name of the generated file, without extension, to be stored inside results folder (default: per_cluster_stats.csv).

5.2 Refactoring with pyCOMPSs

This section will walk you through all the refactor process. It will provide a full description of the issues found, wether they were solved or not, the design decisions made and the reasons behind them, and all the information relevant for debugging, testing and further developing both pyCOMPSs and pyProCT.

5.2.1 Set up

The installation of pyProCT as described on the pyProCT repo is trivial on a local machine. On MareNostrum III pyProCT is already installed. In order to use my version under development (instead of the package installed both on MN3 or a local machine) the user just needs to point the python path to it. This is useful to switch between different working versions (for example to use pyProCT-regression validation or to meet the different instrumentation requirements of each scheduler). Later some issues will also force me to use this same method to customize some of the dependencies of pyProCT such as the pyScheduler or pyRMSD.

Choosing a good structure to set up the environment is a must for executions on MN3 in order to avoid spending a lot of time on configuration problems. This kind of issues kept popping up during the whole project and are described on appendix B.

5.2.2 pyCOMPSs

Prior to starting the refactor it was necessary to analyse which would be the best way to parallelize it. PyCOMPSs works by using python's decorators to define some functions as *COMPSs' tasks*. These are executed on previously defined resources such as a MN3 node or a cloud. For each task the framework checks whether that function's parameters depend on some previous task; if it has no dependencies then the task is assigned to a resource which runs it.

pyProcT clustering and postprocessing sections, as previously stated, are embarrassingly parallel: all algorithm's executions depend only on the distance's matrix calculation; the postprocessing actions all depend on the best clustering (that is to say: the whole clustering section). Knowing this we decided to define as task each algorithm execution and each postprocessing action.

It was decided to support the use of the other schedulers after the refactor so the overall structure of pyProCT was maintained. However, it is desirable to exploit the possibility of reduce the code complexity while achieving the maximum performance improvement. This is mentioned because to make the sequential version of pyProCT work with pyCOMPSs is enough to place the decorators on the right functions. It is true that this would also raise some issues to be addressed; the point is that the lines of code required are fewer if the goal is just to make it work. This is not the goal though. The refactor described from here onwards tries to minimize the code size while making it clearer. It also removes functionality duplication between the framework and the software. For example pyProCt has a loop which enqueues the tasks for the scheduler. COMPSs also has an internal queuing system rendering this loop unnecessary.

Bearing this in mind, differences are basically found on the Driver, Protocol and Explorer classes, which deal respectively with all the sections pipeline execution, the clustering pipeline, and the clustering exploration *per se*. A new Driver for the COMPSs scheduling was created. The main checks whether pyCOMPSs is the scheduler or not and calls one driver or the other accordingly (same method being used for MPI). From the driver onwards the key classes are substituted by the COMPSs versions.

One of the advantages of pyCOMPSs is the small amount of work required to use it. On a normal sequential program we just need to use the *@task()* decorator and the *obj = compss_ wait_ on(obj)* API call to create synchronization points for future objects; from COMPSs manual:

> If the programmer defines, as a task, a function or method that returns a value,

> that value is not generated until the task is executed. However, in order to keep the asynchrony of the task invocation, COMPSs manages future objects: a representant object is immediately returned to the main program when a task is invoked.

Internally COMPSs has queue of tasks so the step to add the tasks to the scheduler is no longer required; instead the decorated methods (which internally COMPSs enqueues to it's pending's list) are called directly. However this caused a problem related to the how the framework deals with the data.

To send the data needed by each task, that is, the method's parameters, COMPSs serializes them to files (except basic types). This means that python's pickle must be able to do the translation which is not the case for the distances matrix.

PyProCT uses pyRMSD [**?**] (which stands for python Root Mean Squared Deviation) to represent the distances matrix. It is basically a python wrapper for a C matrix structure. The goal of this implementation is to highly reduce the access time to the matrix elements.

> Python is slow [...], why: it boils down to Python being a dynamically typed, interpreted language, where values are stored not in dense buffers but in scattered objects. [18]

PyProCT overcomes this with the lean and specialized pyRMSD. The problem is that these structure is not a native python type nor it's built with a combination of them. Because of this the framework can not serialize this matrix to send it to each worker. The first idea to solve it was to dump the internal data of the matrix into a python list (which can be serialized arbitrarily) with the already implemented method *get_ data*; this list then can be passed to the class constructor *CondensedMatrix()* obtaining again the original one. This raised the question of where to perform the translation.

The matrix's cast to a python list is linked to which is the function decorated as task so first is necessary to check how the algorithms are used.

Each algorithm is implemented in a different class so we have *kMedoidsAlgorithm.py*, *spectralClusteringlAlgorithm.py* and so on. All of them however return the same kind of data: clusterings; in order to simplify all the execution pipeline and the code they all have the same structure: all the required arguments are passed to the class constructor and then they all have the *perform_ clustering* method which returns the clusterings found for that parametrization.

With this structure then it would be necessary to decorate the *perform_ clustering* method of each algorithm but only when pyCOMPSs scheduling is used. In order to avoid having code duplication (one with the decorator and one without for each algorithm) a wrapper class was implemented. It is called CompssTask and allows the execution all the algorithms with a single pyCOMPSS-decorated method.

The class is constructed with all the required information for the task execution. During the initialization the aforementioned translation of the matrix to a python list is performed. After the constructor the run method, which is the actual pyCOMPSs task executed on a worker, recreates the Condensed Matrix from the python list and executes the algorithm's clustering method.

After the execution of the algorithm it was necessary to again deassign the computed matrix because it is part of the class and, even if it's not a result, once the task is finished pyCOMPSs again tries to serialize the object and fails. Deassigning also the *self* parameter (the python class instance) improved the performance because, otherwise, pyCOMPSs transfers it back because it has been modified (during the condensed matrix assignation).

After analysing the results we found out two more simple optimizations. Because hierarchical and spectral clustering methods have long initializations (performed in a sequential fashion) the framework is locked while computing them. Due to the fact that they are the first two algorithms to be launched the workers remain idle during that time. In order minimize that time the algorithm's execution order was changed calling them at the end. This way while their initialization is performed all the other algorithms are already running and that initialization time is shadowed under the rest computations. Traces and results of this modification can be seen on section 6.4.

Next part was to parallelize the analysis tasks which give each clustering a scoring based on different parameters. The method was almost identical to the one used for the algorithm's because these functions also need the condensed matrix to work. Again a wrapper was written to pass the matrix data to each task. Results and traces of this trial are reported nonetheless on section 6.3.

5.3 Validation tool: pyProCT-regression

pyProCT-Regression is the software designed to validate the pyCOMPSs refactor code. It implements the so-called black-box validation method. The validator will take a list of tests to perform. First we need to generate the expected results with the original version or pyProCT, then we'll run the same tests with the new version and make sure that the output matches the expected results.

pyProCT results depend on the parameters defined on the control script because we can select which results to save, which format, whether we want to save the computed matrix (and it's image) and so on (see Section 5.1.2 for more info). Because of that the validator needs to be flexible, allowing to define more or less files to check. On the other hand, we have two different test scenarios: one is to validate the results of the original version against the refactor, and the other to validate the new scheduler against known results of the other schedulers.

To achieve this behaviour, Regression takes a test list as input with all the information it needs to check for each test scenario. Each test has the following attributes.

Name, a unique test name.

Description, an small description of the test.

Script, defines the input script for the pyProCT execution.

Expected results dir, is the folder containing the expected output and the files specified on files_to_check

Files to check: a list of the additional files that regression will check, together with the default ones: test.out and test.err.

If we run the tester with the "GENERATE" option it will, for each test, run the installed pyProCT with the defined control script and save the normal standard output and error as well as the "files to check" on the "expected results dir".

On the other hand, running the tester with "TEST" will also run the control script with the installed pyProCT but after that it will check that the generated output matches the content of the "expected results dir".

The last three attributes allow us to use a single expected results directory with different scripts and schedulers, or use the new version of pyProCT with the same tests but on testing mode to validate the refactor. When validating that the original schedulers work as expected other files, such as the parameters.json and the clustering folders (containing information about the generated clustering), are added to the validation process.

5.3.1 Basic tests and issues

The basic tests validate each of the four sections of pyProCT (global, data, clusering, postprocess) incrementally. However once we started generating the MPI results the first issue was: MPI (and later also pyCOMPSs) scheduler needs to be called with mpirun and runcompss.

To solve we could make all the schedulers work with the same call or adapt the tester to each scheduler (reading that information from the control script). It was decided to follow the first approach because the scheduler is set on the control data (so the main file can act as a switch performing the runcompss of mpirun if needed) and pyProCT will be easier to use. Now the main file calls the bash script with the runcompss [params] and mpirun [params]. This way all the versions work same way and the tests on Regression just need to change the control script.

With this modifications it was easier to write the tests. Before starting the refactor all the expected results were generated. However this tests did not validate against themselves because of the random initializations of some algorithms. To solve this we "set the seeds" for the algorithm's initializations removing the stochastic and non-deterministic parts.

Afterwards, for each scheduler the basic tests were run, first with the original pyProCT, then with the refactor. However the pyCOMPSs mode was not available before the refactor so its output was validated against the expected results of the MPI (could be any of the others). pyCOMPSs embeds the application output on its own so, in this case, the tester checks if the expected output is contained within the pyCOMPSs one.

Finally for each major modification of pyProCT this suite of tests was run to validate the work done.

6. Results

This section describes the results of refactoring pyProCT with pyCOMPSs. It is divided into three subsections. The first one reports the benefits of using pyCOMPSs programming model. The second subsection contains the performance analysis of the results. Finally, the last part deals with tools that the framework offers.

6.1 Tools

COMPSs tools offers a great way to inspect the code executions as well as the resource usage and parallelization technique. Figure 6.1 shows an example of the tasks graph generated by pyProCT.

This execution was limited to four algorithms (blue) and two analysis (yellow) tasks to keep it simple. We can observer that all blue tasks are executed on parallel because they have no dependencies. Red octagons are the synchronization points. Their descending structure is due to how pyCOMPSs deals internally with lists of future objects but they all belong to the same wait loop. After the clustering tasks and results retrieval (dependencies d8), the analysis tasks have the required data to be executed (d12): the generated clusterings. After them no more tasks are left and the execution is resumed (7).

Thanks to this tool we can assure that theoretical planning is correct and optimal: the graph clearly shows that the execution is embarrassingly parallel. The monitor generates the graph on-the-fly so the user can check the application work flow while it is running helping to identify bottlenecks and other execution-time issues.

The monitor also offers an execution and tasks information tab. There, for each task type, we can check which is its jobs IDs (in order to identify it on the traces), the host where it was executed, the job status, the number of execution and its average time.

Figure 6.2 shows the information associated to the previous execution graph. We see that the most expensive computation are the clustering ones. Not only their execution time is almost twice the analysis' one but the executed count is also the double.

The tracing system has also been an invaluable asset. It allows to generate traces just by setting a flag on the execution. On the other hand, for MPI and parallel a lot of time was spent in order to get traces. Next results' section 6.3 shows plenty of them and how they were key to some of the optimizations performed.

6.2 Programming Model

PyCOMPSs framework is supposed to require little extra code to be used so first a comparison between the difference of size of each version will be made. Figure 6.1 shows the comparison between the different classes required. It is based on the number of characters and lines because python conventions encourage the usage of line breaks so the results could be misleading (some functions have one parameter per line).

Class Name	Original	pyCOMPSs
Driver	169 / 8180	165 / 7911
ClusteringProtocol	78 / 3703	73 / 3490
PostProcessingDriver	32 / 1531	56 / 2700
ClusteringExplorer	195 / 8470	197 / 9226

Table 6.1: Size comparison of duplicated classes

We see that the refactor did not add too much space. The driver and protocol classes are in fact shorter. This is caused because of the removal of the task-adding loop and the pyScheduler initialization. Postprocessing driver is longer due to the fact that on the original version this section is not parallel. The other ones are quite even.

PyCOMPSs framework just uses python decorators and API calls so, why do we observe a size increase in some classes? This is due to the fact that our data can not automatically serialized by python's pickle. Almost all the extra size is linked to the work needed to handle the matrix data. However, other than adding this little size overhead, the code is much clean and easy to read.

Another important aspect is the execution process and tools offered by the framework. It is here were COMPSs truly shines.

The level of hardware abstraction provided by the framework is really good. To execute pyProCT in a local environment one just needs to provide the language (python on this case), classpath, executable and parameters. If the user desires to customize the framework offers two kind of hardware configurations. On the one hand we find the *resources.xml.* This file allows to define the workers to be used. This includes supercomputer nodes, cloud services, remote images and more. On the other hand we have the *project.xml* which selects which of the defined resources are to be actually used and some runtime parameters.

With this simple two files we can define a wide range of available resources to be used and then select which ones we want to use for an specific execution. Thanks to this we can use a lot of different resources without worrying about the internals and communications. COMPSs' already implements all the connectors required to use them so we just need to give a description of them, select which to use and decorate our code.

For MareNostrum III executions this process is even easier. The development team has created scripts to submit jobs to the supercomputer. By default, it requires the same parameters as a normal execution. However, it has a wide range of easy-to-use options with a clear description. With all the parameters available, such as the network and file system to be used or the number of nodes, it automatically creates the *resources.xml* and *project.xml* that best suit our needs.

This is tools are really useful. Most of the time working of this project has been spent on trying to execute the program on the supercomputer. One needs to understand how a submission queue system works, which parameters need to be specified, how the class paths are read and used, which way the supercomputer access files, which kind of problems may arise from mutex access to the datasets and so on. This is a quite daunting task for someone without advanced knowledge on the subject or with no one to ask help to.

As stated, COMPSs required number of parameters are far less. Understanding some of the aforementioned things will help the user to better use the framework but they are not really mandatory because the COMPSs manuals are good and clear. Following the examples is enough to execute your own programs.

6.3 Performance

This section reports the results of the refactor related with the performance.

Throughout this section some traces will be presented to describe the numerical results. However, the times reported on the traces can not be compared. This is due to the fact that each parallelization is instrumented with a different method (see Appendix A.4). Because of that we can not trust the reported times; pyCOMPSs and multiprocessing instrumentation are far slower than the MPI version because they need to create a process to emit each of the events.

When reporting execution times the x-axis will frequently contain number of threads/processes used. However, when queueing pyCOMPSs executions we specify the desired amount of nodes being the minimum 2 (one is used for the runtime/master which handles all the framework). For this reason, the executions will start at 32 processes/threads which corresponds to that 2 nodes (each node uses 16 threads). Instead, for multiprocessing the execution will be limited to 16 because it can not run on more than one node.

Table 6.2 shows the datasets used on the analysis:

ID	Type	Size (Mb)	Format
1	Protein ensemble	78 Mb	PDB
2	Pprotein ensemble	675 Mb	PDB
3	Protein ensemble	4.3 Gb	PDB
4	Numeric array	72 Kb	TXT
5	Numeric array	142 Kb	TXT

Table 6.2: Datasets used for performance analysis

6.4 Scheduling order

This first section describes the improvement achieved by ordering the execution of the algorithms.

Figure 6.3 and 6.4 show the differences of scheduling the hierarchical and spectral algorithms first or last. On violet we have the clustering tasks and the analysis ones are on yellow. We can see that the initialization of the hierarchical and spectral is performed while the workers are executing the other algorithms. The bright red stripes on the master thread (1.1.1) correspond to the action of queueing a task. On the first one the time between the start and the stripes is the hierarchical initialization. The time between that stripes and first task is the transfer of data. Threads remain idle during that time. On the second example once queued the transfer starts so while performing the costly initialization pyCOMPSs is already transferring the data for the tasks.

From here onwards the version with the ordered execution will be always used.

Next we will see an in-depth analysis of a trace for each version in order to understand the results on bigger datasets.

Figure 6.5 shows the trace for multiprocessing execution with 16 threads. The code is intrumented following pyProCT structure: initialization and parameters handling, data, clustering and postprocess. First blue stripe is the time from the start till the data section. The white one is the matrix handling and calculation. After that all the red section is the actual clustering and analysis. Last strip of first thread is the postprocess. The purple sections correspond to the algorithms and analysis tasks. Note that for multiprocessing the first thread also performs task but they are not shown because we preferred to mark the clustering section. Note that execution is sequential until the program reaches the clustering section where it uses the parallel scheduler. We can also observer that all tasks are equally distributed amongst all the available threads.

Next, figure 6.6 shows the same execution but with the MPI scheduler. First is important to see that the all the data section (in purple) and thus the matrix calculation is performed on all the threads. MPI scheduler handles the available pool of threads just when they reach the actual scheduler. That means that all the threads are doing the same work till that point and so it's redundant work. Despite that, the postprocessing section (in red) does take into the available pool and uses just one thread to avoid work duplication.

White section is the clustering section. Inside, the green parts are the ones adding the tasks to the scheduler and the blues the actual algorithms execution. The blue stripes outside the white part are the ones corresponding to the analysis tasks (which have the same name and thus the same color).Even if all the threads are performing the task-adding loop they are added only once because the task's name must be unique. Please note that blue stripes are composed of a lot of littler ones (the number of tasks is the same on both executions)

We can see that on MPI the matrix calculation is quite longer than the multiprocessing (with respect to its own total time as we can not compare across traces). This is because having 16 threads busy makes them all go slower because, being in the same node, they share some resources (like cache) and all of them are accessing the same input file. On the loop that adds the tasks we see again the same behaviour: duplicated work that wastes resources and increases execution time. Multiprocessing is almost all the time in parallel mode and that the sequential parts are small. MPI, on the other hand, despite running in parallel almost all the time it just does useful work in the blue stripes. However, multiprocessing is limited to 16 threads so this is it's performance peak and MPI can use more resources.

Figure 6.4 shows the resulting trace of executing the dataset 5 with the pyCOMPSs refactor. Even if it is not the same dataset on the other traces, for comparison purposes this is not an issue as we are just observing how tasks are distributed not comparing the actual execution times. We see that pyCOMPSs version is almost all the time computing tasks in a embarrassingly parallel mode.

The dark red section is the time in which the main execution is waiting for the tasks results. During the clustering section the main is always running as opposed to figure 6.3 in which the main is locked during the last tasks. On the analysis part the loop that calls the tasks is fast so, once performed, the main waits for the clustering scoring results in order to choose the best and start the postprocessing.

Figure **??** shows that COMPSs is the slowest parallelization for the dataset 1 and the multiprocessing is the fastest. That is caused because this method is only able to work on a single node so the initialization and overhead of data and resources handling is indeed much smaller. For this first example we have used the best result found for each version, that is: 16 threads for multiprocessing, 12 threads for MPI and 2 nodes for pyCOMPSs.

Figure 6.7 shows the executions times (normalized to 1) of the three versions for dataset 1, 2, 3 and 4. MPI and multiprocessing crash with dataset 3 if we use too much threads. That causes the performance reduction for multiprocessing on datset 3 because it only is able to run with maximum 8 threads. We observe that for datasets 3 and 4, which are the most demanding, COMPSs outperforms MPI a lot and improves multiprocessing. Table 6.3 shows the actual numerical results.

Version	Dataset 1	Dataset 2	Dataset 3	Dataset 4
COMPSs	24 s	88 s	660 s	1473 s
MPI	12 s	58 s	3100 s	2483 s
Multiprocessing	6 s	74 s	992 s	1670 s

Table 6.3: Best execution times of each versions with the optimal resources available where we can see that pyCOMPSs works best with the more demanding datasets (3 and 4)

Table 6.4 shows the execution times for dataset 1 given by the number of threads. Multiprocessing can use a maximum of 16 threads and COMPSs requires at least 32 so the other boxes are empty for them. We see that the dataset is too small to benefit from the parallelization because adding resources does not improve the performance. Clearly multiprocessing is the best. MPI's performance decreases with the number of threads, this is probably caused by the replication of work on all the threads.

Version	4	8	16	32	48	64
COMPSs	-	-	-	24.3 s	23.2 s	24.4 s
MPI	9.6 s	10.2 s	12.1 s	13.02 s	13.2 s	14.4
Multiprocessing	6.1 s	6.0 s	6.1 s	-	-	-

Table 6.4: Execution times given by number of threads where can be observed that COMPSs is not effective for the smallest dataset (1)

Dataset 2 execution times can be seen on table 6.5. Here COMPSs already outperforms both other methods. Still, pyProCT is not sscalating with the resources for the refactored code. On MPI we see that we reach the best performance at 32 threads. Multiprocessing scalates really well but can not use more than 16 threads.

Version	4	8	16	32	48	64
COMPSs	-	-	-	58.9 s	57.3 s	59.1 s
MPI	144.2 s	97.6 s	86.4 s	80.6	85.3 s	89.3
Multiprocessing	133.2 s	105.6 s	67.1 s	-	-	-

Table 6.5: Execution times of for medium-sized dataset 2 given by number of threads.

Dataset 3 is the biggest protein ensemble dataset and its results are reported on Table 6.6. The new empty boxes on MPI and multiprocessing represent the inability of the program to finish correctly. The size of the matrix is probably the cause of the crashes. Spectral algorithm also computes another matrix adding more memory load. In MPI having the aforementioned work replication the matrix rapidly grows consuming all the available memory and thus crashing. In multiprocessing the effect is the same although not having that duplication allows it to work well till 8 threads. On the other hand, COMPSs runs on multiple nodes and has more available. Also the computed matrix and the custom matrices required by some algorithms are not on the same node.

Version	4	8	16	32	48	64
COMPSs	-	-	-	667.2 s	688.3 s	687.9 s
MPI	3801.6 s	-	-	-	-	-
Multiprocessing	1447.1 s	992.6 s	-	-	-	-

Table 6.6: Execution times of biggest input, dataset 3, given by number of threads. For this dataset pyCOMPSs already bests the other versions for, at least, 5 minutes.

Last results are the ones from the numerical dataset 4 reported on table 6.7. This dataset is numeric instead of a protein ensemble. This means that the matrix computation is faster but the clustering and scoring is much more demanding. Again, some of the MPI and multiprocessing crashed. We can observe the same traits that we have been seeing during all the analysis. MPI is the slowest on large datasets and crashes on more than 16 threads rendering useless its ability to run over different nodes on distributed systems as MareNostrum 3. Multiprocessing again scalates really well till its limit. COMPSs outperforms both on big datasets and does not crash due to its ability to distribute memory load over different nodes. However we have also seen that COMPSs execution times do not benefit from the resources' increase. Next we will have a look at why that happens.

Version	4	8	16	32	48	64
COMPSs	-	-	-	1603.2 s	1506.3 s	1473.2 s
MPI	2581.1 s	2543.9	2483.7 s	-	-	-
Multiprocessing	2500.1 s	1858.6 s	1689.0 s	-	-	-

Table 6.7: Execution times of dataset 4 given by number of threads. This numeric dataset allows multiprocessing and MPI to use more threads than the previous figure before crashing. Despite that pyCOMPSs bests them for a minimum amount of 3 minutes and a half.

Figure 6.8 shows why COMPSs execution times do not decrease with the resouces' increase. Both figures show the section corresponding to the algorithm's execution of a trace. The upper ones uses 6 nodes and the lower 2. The time scale is the same on both. Green flags mark the start of each tasks (because being so close one would not be able to see it). We see that the problem is that the longer algorithms are the bottleneck of the clustering section. For two nodes, they shadow the execution time of all the smaller algorithms. When executing it on six nodes all small tasks are executed in an embarrassingly parallel fashion. However, the time is not decreased as the execution is locked waiting for the most time-consuming ones.

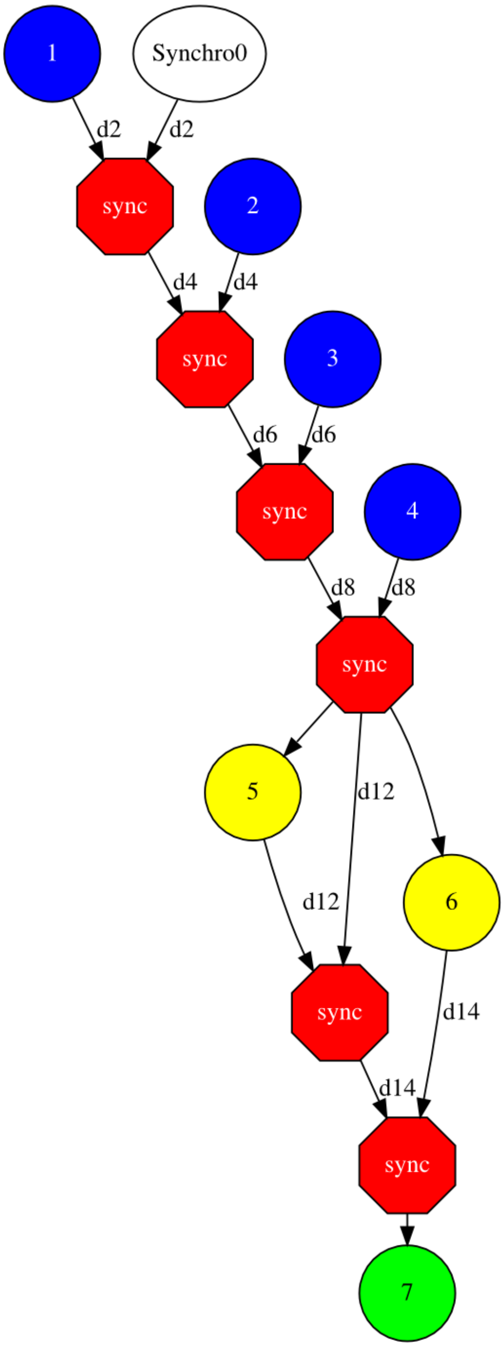


Figure 6.1: COMPSs monitor data dependencies graph for an execution of dataset 5 with 4 threads

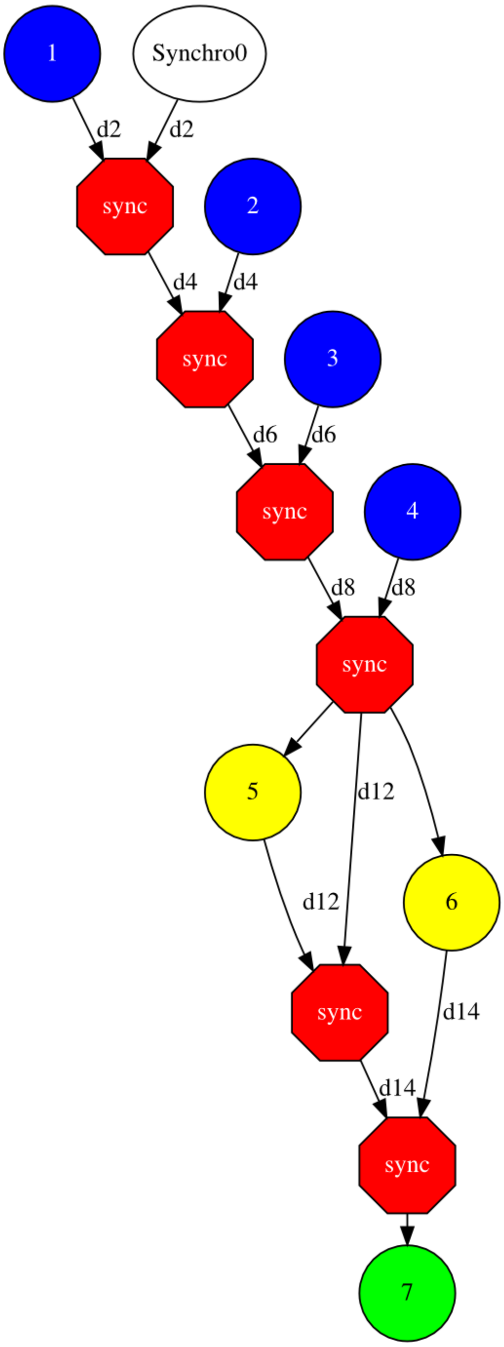


Figure 6.2: COMPSs monitor tasks information for an execution of dataset 5 with 4 threads

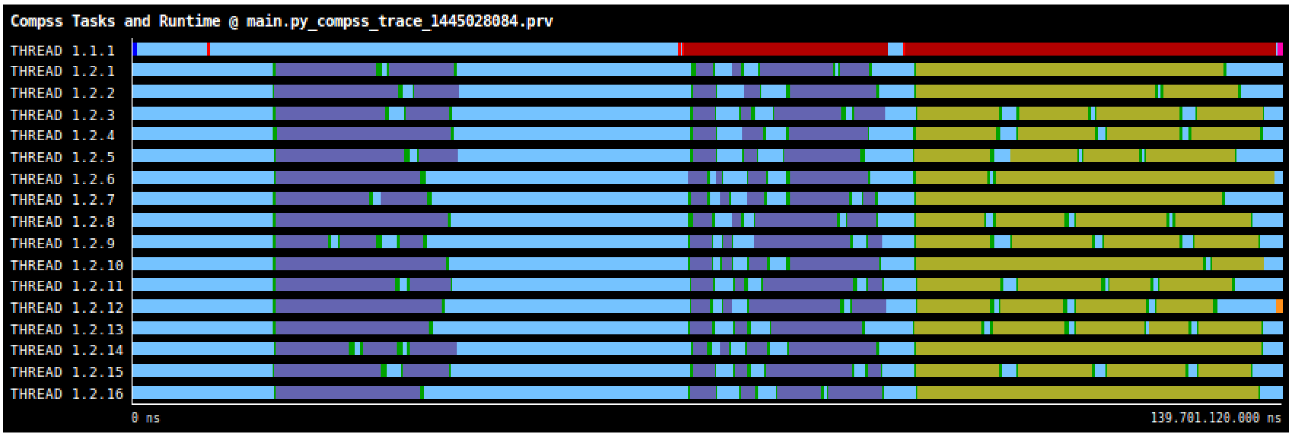


Figure 6.3: Trace for dataset 5 without ordering algorithm's execution

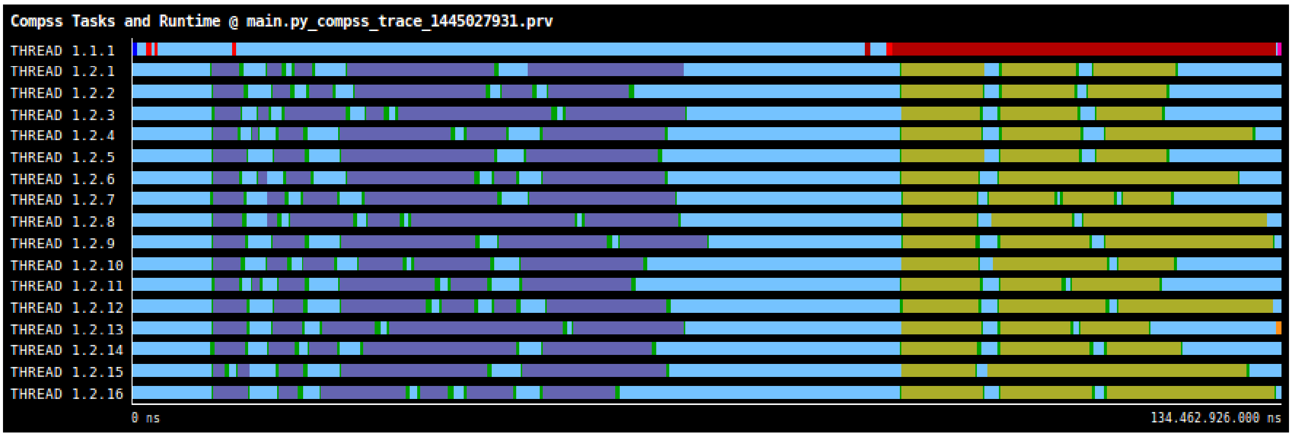


Figure 6.4: Trace for dataset 5 showing the resulting performance improvement derived of ordering the algorithm's execution

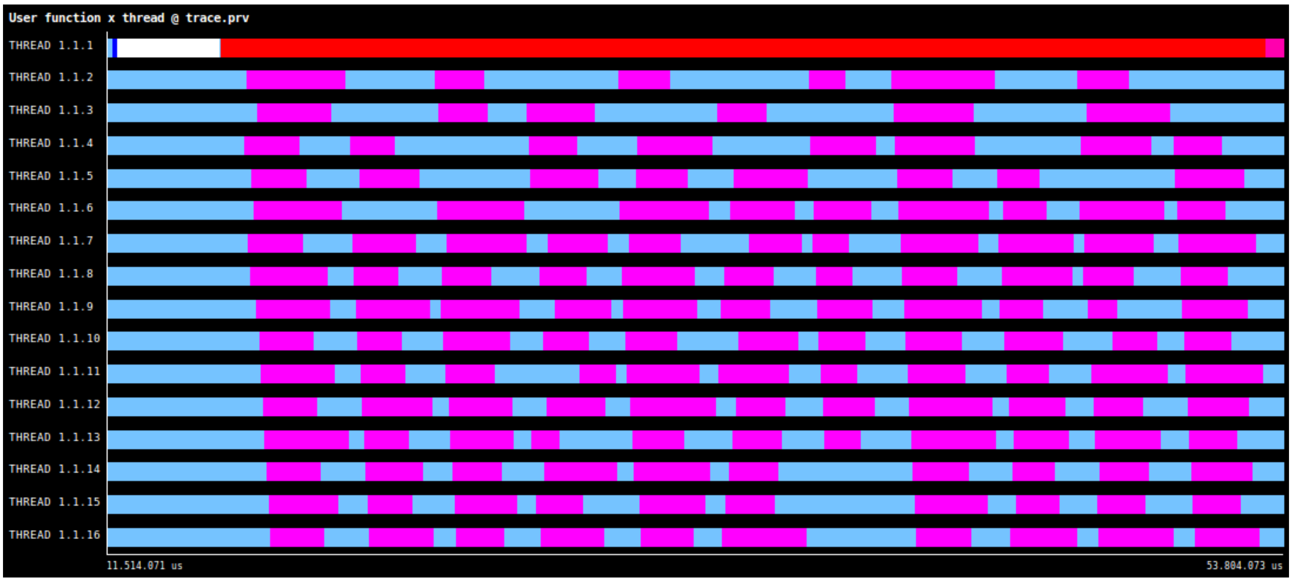


Figure 6.5: Multiprocessing scheduler with dataset 1 and 16 threads

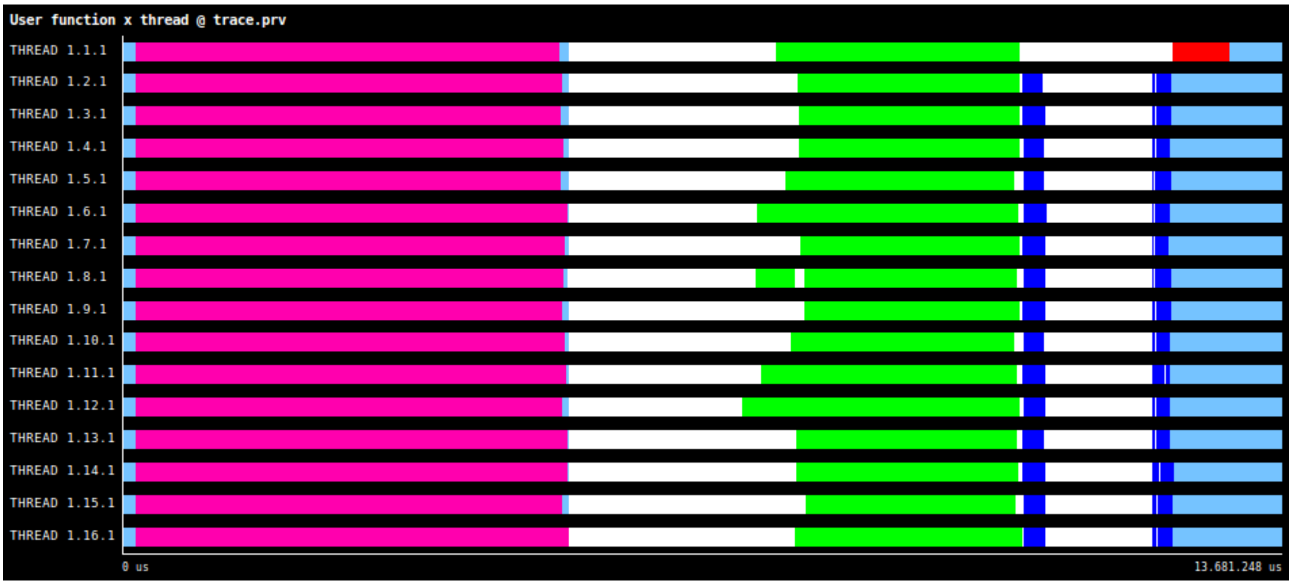


Figure 6.6: MPI scheduler with dataset 1 and 16 threads

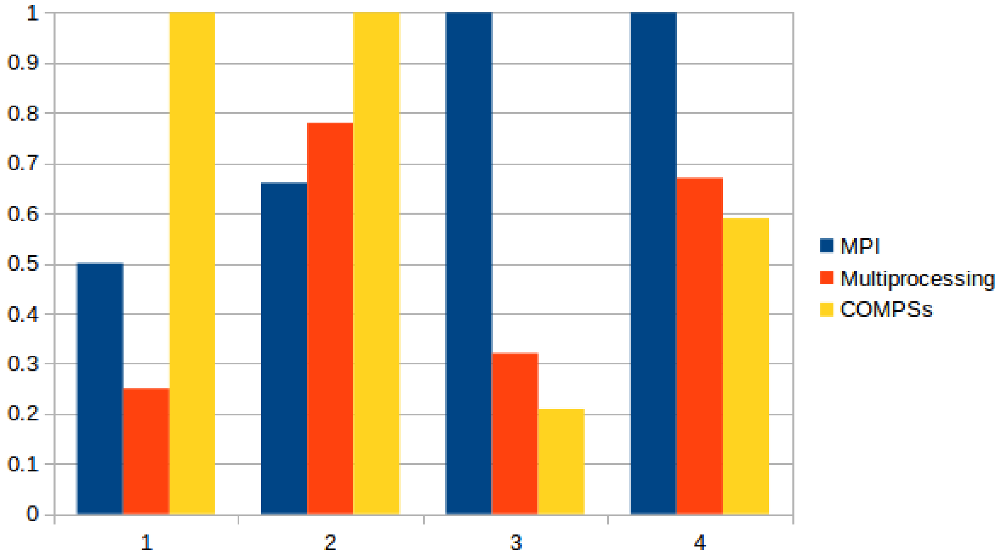


Figure 6.7: Relative execution times for three versions and datasets 1, 2, 3 and 4

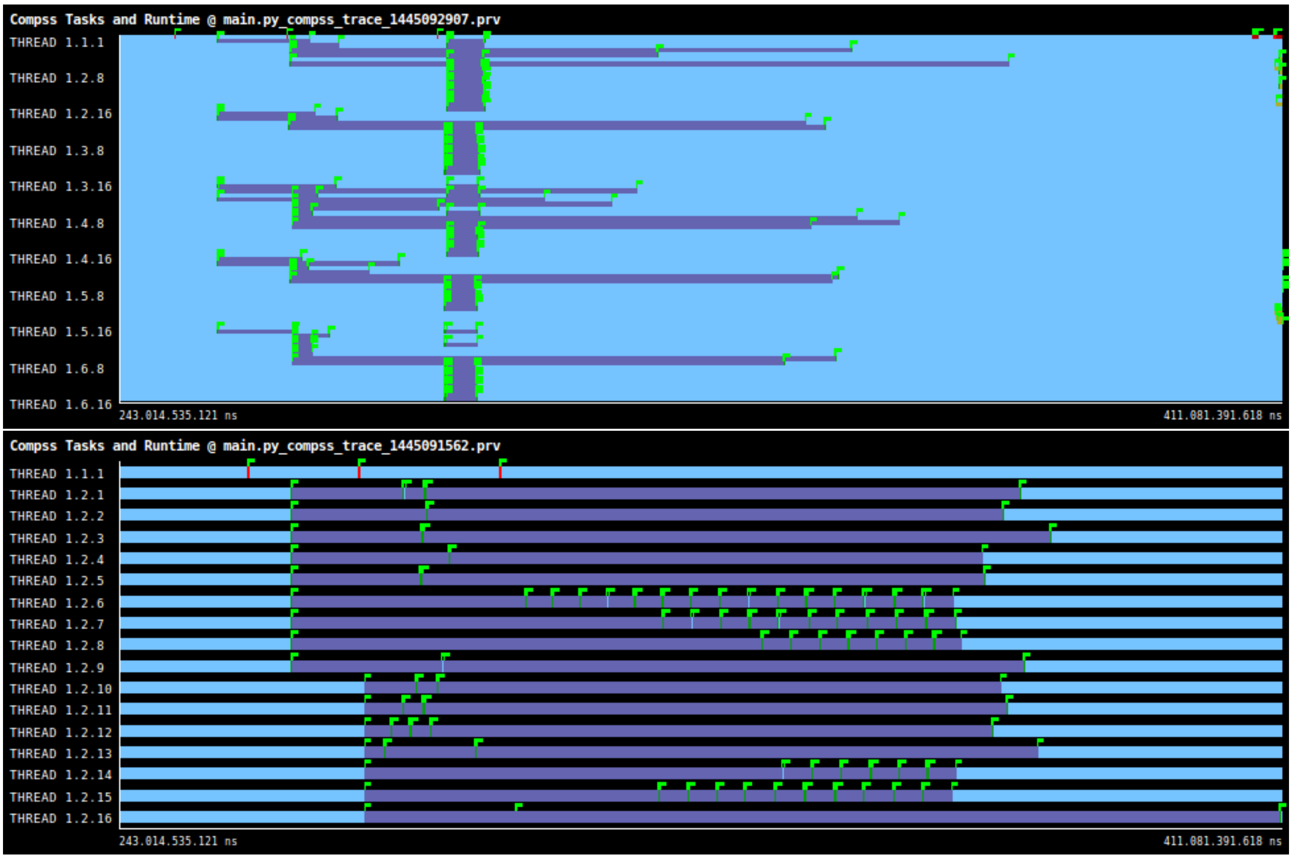


Figure 6.8: Section of traces for execution of dataset 4 with 6 nodes (upper) and 2 nodes (lower) where we can see why pyCOMPSs performance does not increase with the number of nodes assigned.

7. Conclusion

We have been able to parallelize pyProCT with the pyCOMPSs framework. On large datasets, we have seen that the performance is improved. On smaller inputs it is not worth to use it due to all the complex initialization process performed by the framework. Because of that we deem a good decision to have added pyCOMPSs to pyProCT's schedulers, instead of substituting them, leaving to the user the decision of which fits best its datasets. Despite that, the performance could be further increased by defining as tasks the algorithm's and analysis' classes themselves instead of using the wrapper method. This would require to duplicate all code for the algorithms and analysis in order to support the other schedulers.

Thanks to running over multiple nodes now the program can overcome memory problems. However performance is limited by the slowest algorithms no matter how many resources are assigned to it. PyCOMPSs is now offering the ability to define the required resources for a task. This could allow to parallelize that slower algorithms inside the tasks thus reducing that bottleneck.

This work has revealed some of pyProCT and pyCOMPSs issues. The clustering program is heavily limited by the slowest algorithms and the computation of the initial matrix. Future improvements should be focused on addressing this two sections of the pipeline. On the other hand, pyCOMPSs framework transfer of the matrix data could be improved. Currently they are working with memory-level transfer methods which could greatly enhance this transfers. The inability to serialize some classes' instances is also something to be improved because it requires work-arounds (such as the wrappers) that add too complexity to the simple and clear programming model and usage of pyCOMPSs.

To sum up, pyProCT is now faster and offers all pyCOMPSs tools to analyse code and executions. The way it was refactored makes that if more algorithms are added the performance will be further increased. Aso pyCOMPSs is actively under development so new useful features and performance improvements will be presented and pyProCT will benefit from them without requiring modifications, or maybe small ones, at all. The work done has also posed new research lines for both the framework and the clustering tool through the application and analysis of both to real world cases.

8. Glossary

This section describes the technical terms used in this document.

.bashrc, is a shell script that Bash runs whenever it is started interactively (when login into MN3 for example).

Cloud computing is a model for enabling ubiquitous, convenient, on-demand access to a shared pool of configurable computing resources. Cloud computing and storage solutions provide users and enterprises with various capabilities to store and process their data in third-party data centers.

Cluster analysis(CA), it the task of grouping a given set of objects in way that items on the same group or cluster are more similar (in some sense) to each other than to those in other groups.

COMPSs, is a programming model which aims to ease the development of applications for distributed infrastructures. It features a runtime system that exploits the inherent parallelism of applications at execution time.

Cython, programming language is a superset of Python with a foreign function interface for invoking C/C++ routines and the ability to declare the static type of subroutine parameters and results, local variables, and class attributes.

Decorator (python), is a function that takes another function and extends the behavior of the latter function without explicitly modifying it.

Elasticity , in cloud computing, is defined as the degree to which a system (or a particular cloud layer) autonomously adapts its capacity to workload over time.

Framework, is often a layered structure indicating what kind of programs can or should be built and how they would interrelate. Some include actual programs, specify API's, or offer programming tools for using the them.

High Performance Computing (HPC) is the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer or workstation in order to solve large problems in science, engineering, or business.

Pickle, is the python standard mechanism for object serialization; pickling is the common term among Python programmers for serialization (unpickling for deserializing).

Pragma, directives offer a way for each compiler to offer machine- and operating system-specific features while retaining overall compatibility with the C and C++ languages.

pyCOMPSs, python application programming interface (API) for COMPSs.

MareNostrum III (MN3), is a supercomputer based on Intel SandyBridge processors, iDataPlex Compute Racks, a Linux Operating System and an Infiniband interconnection located at Barcelona.

MN3 Modules Environment, is a package (http://modules.sourceforge.net/) which provides a dynamic modification of a user's environment via modulefiles. Each modulefile contains the information needed to configure the shell for an application or a compilation. Modules can be loaded and unloaded dynamically, in a clean fashion

MPI, is a standardized and portable message-passing system designed to function on a wide variety of parallel computers.

Mutex is a synchronization method that can be used to protect shared data from being simultaneously accessed by multiple threads.

Non-blocking I/O (NIO or "New I/O"), is a collection of Java programming language APIs that offer features for intensive I/O operations.

Open Multi-Processing (OpenMP), is an API that supports multi-platform shared memory multiprocessing programming. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior

Pip, is a package management system used to install and manage software packages written in Python.

ProDy, is a free and open-source Python package for protein structural dynamics analysis. It is designed as a flexible and responsive API suitable for interactive usage and application development.

Programming model or paradigm is a fundamental style of computer programming, serving as a way of building the structure and elements of computer programs.

Root-mean-square deviation (RMSD), is the measure of the average distance between the atoms (usually the backbone atoms) of superimposed proteins.

Scalability is the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged in order to accommodate that growth.

setup.py, is a python file, which usually tells you that the module/package you are about to install have been packaged and distributed with Distutils, which is the standard for distributing Python Modules. Allows to easily compile and install with *python setup.py build && python setup.py install.*

Secure Shell (SSH), is a cryptographic (encrypted) network protocol to allow remote login and other network services to operate securely over an insecure network.

Unicode, is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems.]

UCS-2 & UCS-4, are Unicode encodings which encode each code point to exactly one unit of, respectively, 16 and 32 bits.

Wrapper, function (or class) is a subroutine in a software library or a computer program whose main purpose is to call a second subroutine or a system call with little or no additional computation.

List of Figures

List of Tables

Appendices

The following appendices contain information about the old scheduler of pyProCT, the analysis tools used, the work methodology and temporal planning, an estimated budget an sustainability issues and, finally, the list of all manuals and documentation used.

First chapter relies on a simple tic-tac-toe game to explain the scheduler and use analysis tools. This approach helped to understand how to use them in a more limited context where it's easier to catch bugs or possible problems prior to their use on pyProCT. The program is first described on the following chapter to put the reader in context. Afterwards that small program is refactored with pyScheduler. Next section uses the modified code to tests extrae and paraver analysis tools on its different versions (sequential, MPI and python's multiprocessing).

Appendix B. describes all the problems that are not directly related to the pyCOMPSs refactor. It is composed of issues arisen by python versions, encodings and configurations.

Next two appendices are related to how the project was planned and developed. In them the reader will find the work methodology used (Appendix C) and the temporal planning (Appendix D).

On Appendix E the estimated cost of the project is given. It is divided into human, hardware and software resources. Next chapter, (appendix F) gives an opinion about the social, economic and environmental impact of the work done.

Last appendix, G, chapter contains all the documentation used. Here the reader will find the tools' manuals, readmes, tutorials and all the information sources which do not belong to the bibliography.

A. Scheduler and Performance Tools

A.1 Motivation

This section explores pyProCT's scheduler and the tools necessary to analyse its performance. An small sequential testing program was written in order to test it all on a much smaller scale. The next sections' decriptions rely on it to explain the scheduler and tools. More precisely, first section will describe the program to put the reader in context. On the second section the program will be parallelized with the scheduler. Last two sections will describe the analysis process: first the instrumentation for generating traces and then the visualization tool.

A.2 Test program description

The developed program was an command-line implementation of the popular game Tic-tac-toe. It suited quite well the project needs because it featured good parallelization options, stochastic elements, a clear turn structure (easing up the first analysis trials), variable-size parametrized tasks and automatized execution (having two AI playing).

Figure A.1 shows a simplified version of the program's execution flow. All the logic handling which player's turn is, how the board is marked and some other parts have been omitted because they are not relevant.

For each turn, while the game is not finished, the program calls the *Player* class' *play()* function. The algorithm implementing the AI moves is a montecarlo-like method. It randomly simulates a big number of games for each available cell and then choses the cell with the best score.

To do this it first gets all the free cells, then for each available one it calls the exploration_handler method. This method in turn calls mark_an_explore a number of times (this number is defined by the ITERATIONS parameter which is a command-line argument) with a copy of the game board. Basically, mark_and_explore first the cell passed as parameter and then fills randomly the copied board, till the game is finished (either by a player winning or a draw), returning the cell and the id of the winning player or a 0 if there is a draw. The list of winnig id's is then returned to the montecarlo function. Afterwards we initialize the score value for each available cell with infinity. Then for each tuple containing a cell and the winner of that simulation we increment the score of the cell if the player has won or decrease it if the player has lost (the algorithm could modify the score for draw results). The actual increment/decrement values can be tuned but it's not relevant for testing purposes.

A.3 PyScheduler

Once the sequential program was ready, next step was to decide how to refactor it with pyScheduler (pyProCT's scheduler). The decision was to consider each *exploration_handler()* function as a task to see how their size affects the performance of the scheduler. The ITERATIONS parameter allows to change the number of iterations performed inside *exploration_handler()* so this was deemed the best option.

The selected scheduler has three scheduling types, one sequential and two parallel:

Serial, which executes the task sequentially.

ProcessParallelScheduler, which uses python's multiprocessing module.

MPIParallelScheduler, which uses mpi4py for the parallelization.

The usage of the scheduler is simple. For each task we have to call the *add_task* method providing the following information:

Task name: a unique task name.

Dependencies: a list of this task dependencies (which must be a list of other tasks names).

Description: a description of the task.

Target function: the name of the function to be executed.

Function kwargs: the list of the keyword arguments which need to be passed to the target function.

Once the task list is completed we just need to call the *run()* method of the scheduler. The method will return a list with the execution results of each task.

For the tic-tac-toe these are the values for the queued tasks:

Task name: ExplorationXY (being X, Y the coordinates of the cell to be explored).

Dependencies: [ ] (the empty list because there are no dependencies among different explorations).

Description: Montecarlo exploration.

Target function: self.exploration_handler (as we are inside Player class namespace we must add the self).

Function kwargs: "x": x, "y": y, "board": board (being X, Y the coordinates of the cell to be explored, and board the current state of the game).

A.4 Instrumenting with Extrae

Next step was to start using the analysis tools. The decision was to use the **Extrae + Paraver** combination. Extrae [1] is the package used for instrumenting the code; Paraver [2] is the tool used to visualize the traces generated by Extrae. These tools have been both developed at the

[1] Find extrae documentation on the performance tools section of G .Documenation

[2] Find paraver documentation on the performance tools section of G .Documenation

BSC to be used together. We chose them because of the Extrae support to python, the offered assistance and proximity of the tools' experts and the fact that they are both installed and configured on MareNostrum III, which is our target execution platform.

Extrae offers two different ways to instrument the code: automatically instrument functions (providing a list of functions to the extrae XML configuration file) or including the extrae module (import pyextrae) and emit specific events inside the code with *pyextrae.eventandcounters(type, value)*.

The basic usage of the first, which does not require any changes on the code, instruments the entry and exit points of the functions. More complex behaviours are also available but for these tests the basic one is enough.

The second one just needs to add the mentioned function call wherever we are interested to emit an event.

To start, *play, montecarlo* and *exploration_ handler* methods were set to be automatically instrumented and two more events were manually added: one before the scheduler intialization and task addition and one just after the scheduler *run()*. On the sequential version [3] From now, "sequential" will refer to the schedulerless version, "serial" to the one with the sequential scheduler, "parallel" for the one using ProcessParallel and "mpi" for the mpi4py one this corresponds to before and after looping through the available cells (calling *exploration_ handler* on each iteration).

Support for python is only available from version 3.0 onwards and it's not fully tested so we faced some problems. For the sequential and serial versions everything went well, the traces were correct and they could be visualized with Paraver. For the parallel version we were not able to extract correct traces as extrae was not able to detect the parallelization method. When the MPI version was tried it didn't work for two reasons. On one hand, adding the user functions' automatic instrumentation made the whole execution to end with a segmentation fault without generation the traces. On the other hand, using just the event emit method, the visualization

[3] +

showed just one thread.

After meeting with BSC people we managed to solve the issue. The problem was that extrae used a sequential-tracing library, so it could not detect the mpi multiple threads. To solve it we substituted this sequential library with an mpi-tracing one. After some work on their part the first issue was also solved by changing some values on *Extrae_ define_ event_ type*.

For the parallel implementation with multiprocessing this approach wasn't supported. They gave some ideas to try but to no avail. Linking extrae with a number of different libraries, such as the pthreads one, to see it they were able to hook themselves to the python multiprocessing threads didn't work. The issue was left open. Fortunately, a workaround was found: the command line usage of extrae was used. It has two basic commands:

extrae-cmd init node slots

extrae-cmd emit slot event_ value event_ type,

A python class wrapper was created for this two commands. This way it's possible to reuse it to instrument pyProCT later. This class deals with the extrae paths and concurrency as well as providing a easier interface to call from python. To use it first it is necessary to initialize each used node with an ID and the number of processes/threads it will contain. Then for each event we want to emit we specify it's ID (which must be positive and smaller than the number of threads we set for that node) and the value and type of the event.

The first trials raised a segmentation fault; knowing that emitting an event with an out-of-range thread ID raises a segmentation fault hinted that the initialization was not correct. Because this extrae usage is not the recomended nor normal approach there is no documentation for it nor examples for it. After several attempts we met with the extrae team. Working with them we found out that the segmentation fault was caused by a bug on the used version. Kindly they fixed it and made a custom package for this project's use. With it, basic instrumentation for the parallel (with python-multiprocessing) version was finally achieved. This is limited to emit events and can not produce the advanced visualizations and results achieved on the serial and MPI version.

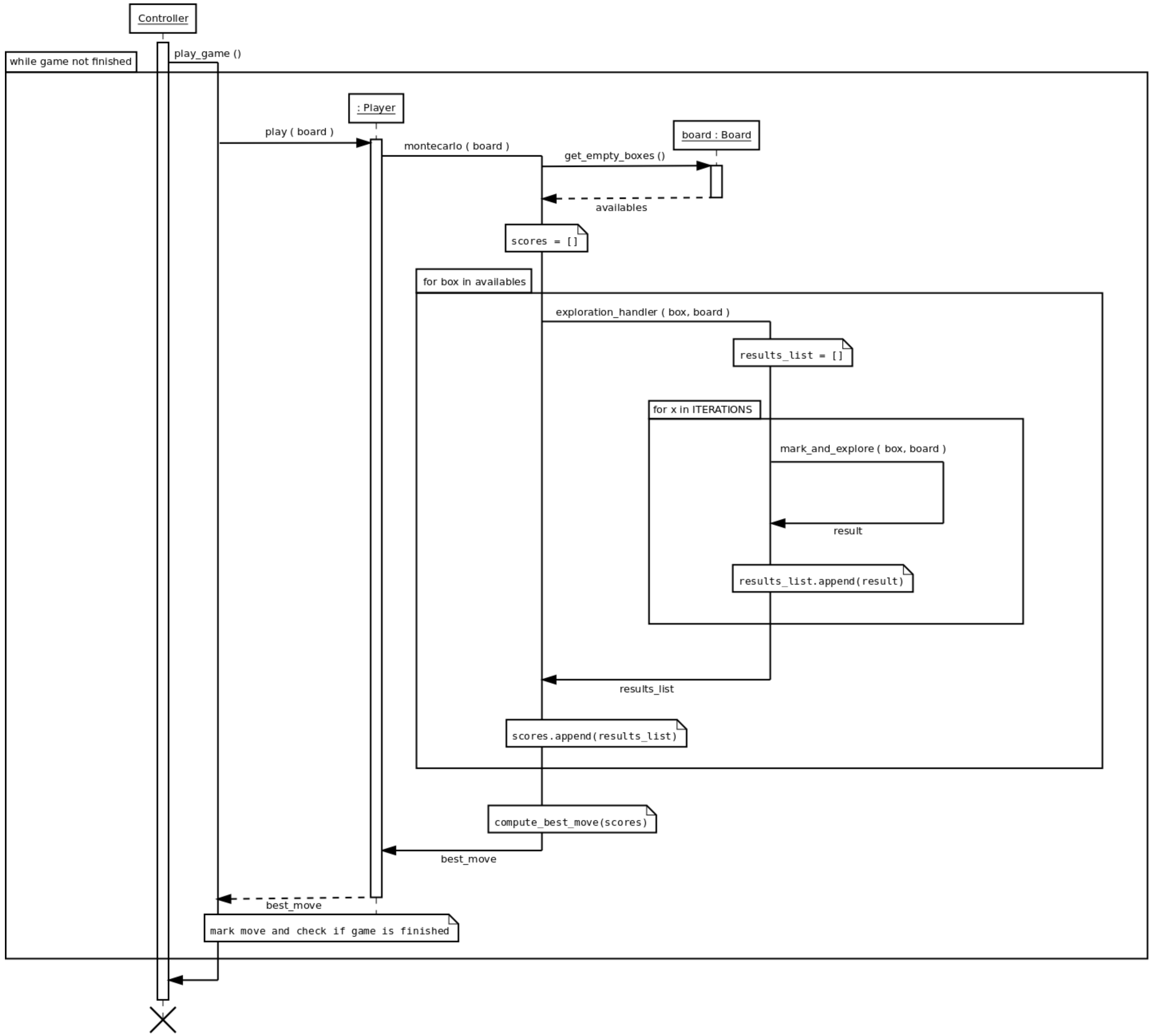


Figure A.1: Tic-tac-toe Execution Flow

A.5 Visualizing with Paraver

Once generated the trace file the next step is to analyse them with the visualization tool Paraver.

First thing was to create a configuration file that would show the instrumented user functions (montecarlo, play and exploration_handler on this program). To do so one needs to configure the event filter to show only events of the type 60000100 (which is the type assigned to instrumentated user functions), and the semantic options to show the last event value (that is the values identifying the functions) in a stacked composition. Thanks to the ability of copy/paste time info from a graphic we can quickly compare different traces.

Figure A.2 shows the visualization of three executions of the tic-tac-toe, all of them with 500 iterations. The first is an schedulerless version, the second with the serial scheduler and the last one with an MPI scheduler. At the time of writing the parallel/multithreading scheduler can't be instrumented, as this is a demo section of the Paraver capabilities we have considered that leaving the parallel version out will not harm the purpose of this section. Dark blue corresponds with exploration_handler function, white with montecarlo, red with play and light blue the time outside these three functions. The MPI version has more labels but for the current section the details aren't important. We can see on the figure that the serial scheduler has an important overhead, making it slower than the schedulerless version, but the MPI scheduler is quite faster.

Paraver has a wide range of configurations to visualize MPI information (see Figure A.3), useful execution time or IPC. We can inspect data in timeline or tabular form but also with the aid of other tools such as the Clustering. With this tool we can cluster the results to obtain, for example, a graphic relating the executed instructions and the IPC. Thanks to this we can bypass analysis problems related with the time where, sometimes, an increase or unbalance in the workload depends on the IPC rather than the number of instructions. On Figure A.4 we can see that the most computation-intensive areas associated with the exploration_handler function (in blue on Figure A.2) have a good efficiency.

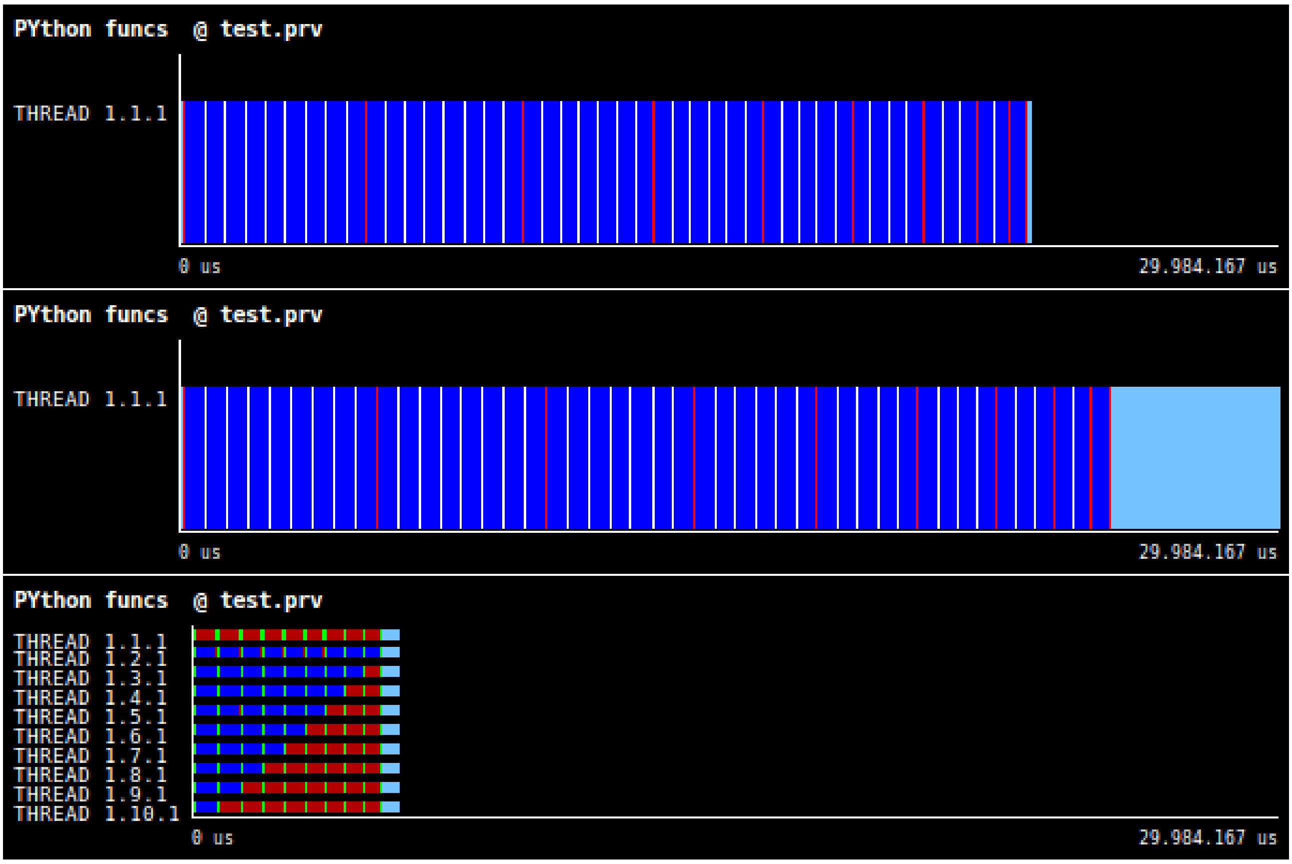


Figure A.2: Tic-tac-toe 500 iterations executions

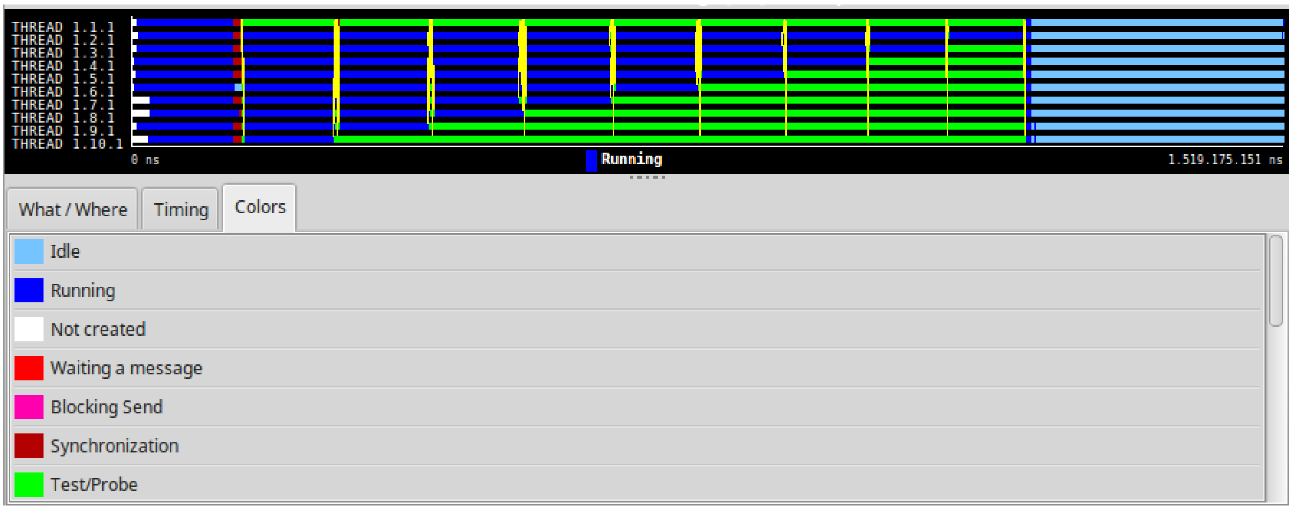


Figure A.3: Tic-tac-toe MPI information for a 500 iteration execution

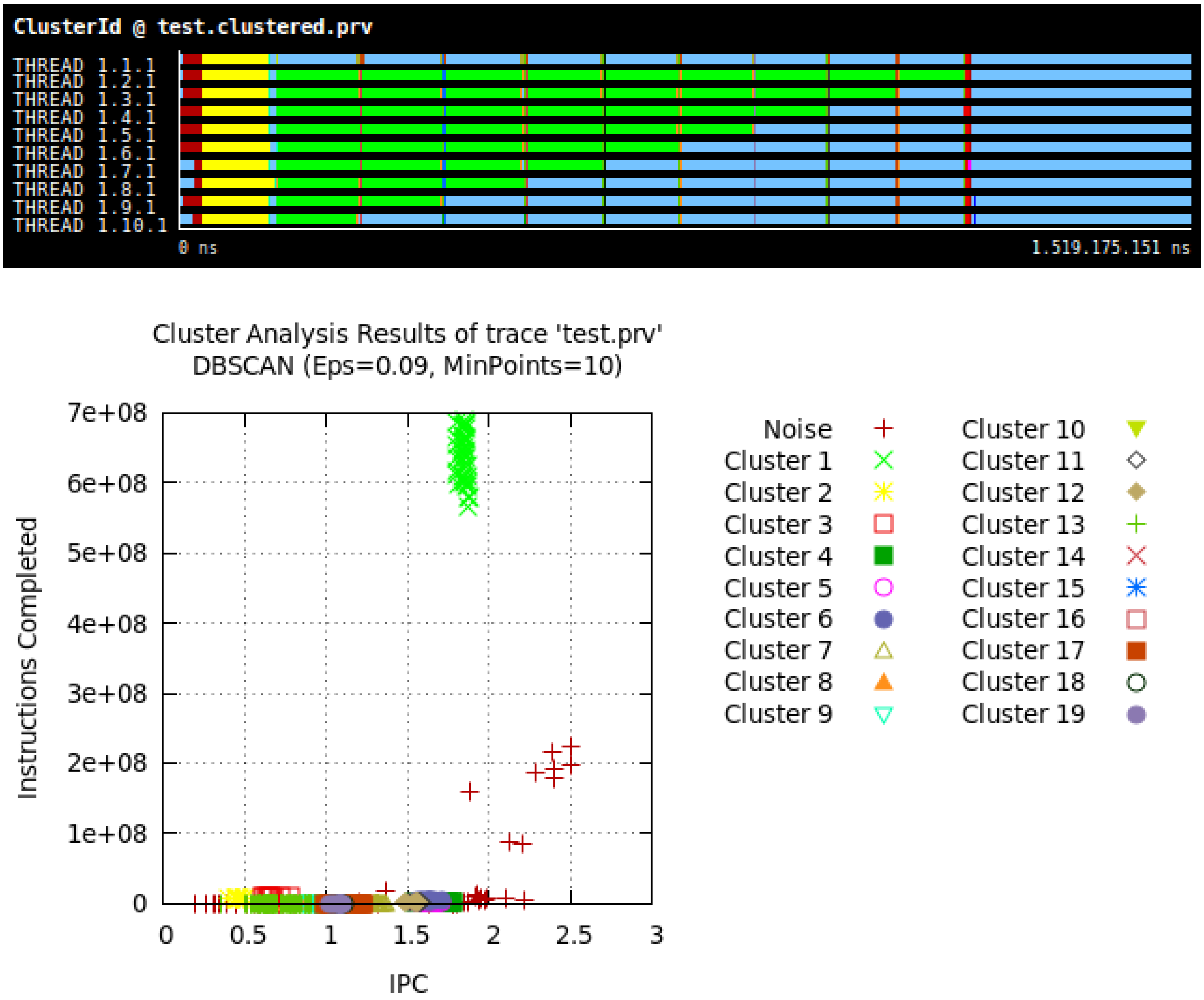


Figure A.4: Tic-tac-toe timeline and clustering for 500 iteration MPI execution

B. Problems

B.1 Python versions

This first issue arose when trying to compile and link the development version of pyProCT. It is related with MN3's modules environment. By default, on login, MN3 has 2.6.9 python, however this version is not available to be loaded through the modules; it's only available when no other python has been loaded by the .bashrc file nor manually with *module load PYTHON*.

pyProCT depends on python 2.7.3 which can be loaded with the modules. At first it was compiled and installed with the default release (2.6.9) with setup.py. On MN3 one needs to add a custom installation path (with *–prefix=PATH* option) to setup.py because of the lack of permissions to write into the default installation directory. After installing, we found out that pyProCT can not be run under python 2.6.9 so the installation process was started again with 2.7.3 version.

The new installation arose the next issue

B.2 Unicode encoding

The message error of this bug was:

undefined symbol: PyUnicodeUCS4_DecodeUTF8

This is caused when trying to use software build with UCS4 on a UCS2 python version. On MN3 each installation uses a different one.

- Python 2.7.3 → UCS2

- Python 2.6.9 → UCS4

Python is not a compiled language, so this compilation problem actually comes from the Cython modules integrated into pyProCT. This meant that the new installation (which used the same folder as source) was not recompiling the Cython modules even after issuing a clean command so the repo was cloned in order to start again from scratch. This time everything ran smoothly. As a curiosity, if pyProCT is build with python2.6.9 it can be used with python2.7.3 (although rising some compatibility warnings).

B.3 Prody and extrae

When trying to generate traces with extrae (for MPI and sequential version) a Prody error was found. When trying to resize any kind of structure Prody detects that there is more than one reference to that structure (introduced by the instrumentation) and fails to do the resize. To avoid this one needs to manually modify the Prody package and, for each resize, add the parameter *refcheck=False*. This error is raised in order to avoid integrity problems when an object has more than one reference; however we know that the instrumentation will not modify nor actively use those structures so we can safely disable the reference's check.

B.4 Size of floats

This issue was raised by some datasets. Depending on the computer and data, when the software tries to recreate the condensed matrix on the pyCOMPSs task an incompatible format error is raised. This happens when numpy stores the matrix data (in list format) as floats with 32 bits. The matrix constructor however requires that data to be in floats with 64 bits. To overcome this, numpy arrays have a method, *data_view('float64')*, to select which type of elements should be returned and thus allowing me to always format them as 64-bit floats and solve the issue.

C. Work Methodology

There are many ways to try to optimize pyProCT, some of them complex enough to be a full project. Due to the limited time and the loop structure of the development, we decided to use an Scrum based methodology. We set time-variable cycles at the end of which the work was evaluated.

Paraver and Extrae have been used for traces' analysis; pyProCT-regression, to validate the new implementation. These tools, as well as pyCOMPSs and pyProCT, are still on development and not fully tested. This scenario suited best an Scrum methodology. Having cycles meant that it was easier to evaluate if some trials led to a dead-end, were best implemented on another way or, simply, were not feasible because they were unsupported.

Each cycle was bound to an specific modification. To begin each cycle we analysed first the state of the project and how previous work affected the code to define the next goal and the expected results. Once decided the work plan, we proceeded to it's implementation.

Once finished we checked if the goals were achieved. The duration of each cycle was variable because the complexity of each optimization can vary a lot. This helped to keep track of the work and decide if a particular modification was taking too long or could not be implemented. It also reflected the possibility that an optimization did not improve the overall performance, case in which the results were analysed and reported nonetheless prior to planning the next work to be done.

This methodology takes into account how each modification affects the next one allowing a better planning. We deemed this approach better than performing a full initial analysis and deciding at once all the optimizations to be implemented.

To work on this project, a laptop with the text editor Sublime Text 3 was used. The computer had installed all the required software to run the code on the Mare Nostrum machine (through ssh), fork and manage the code versions with git, run the tests and instrument the code (for further details see both E.2 Hardware Resources and E.3 Software Resources sections).

C.1 Limitations and Risks

The reproducibility problem, defined as the impossibility to repeat an exact execution of the algorithm because of some stochastic parts, such as random initial parameters estimation for example, could difficult the validation and testing part. This could lead to a number of problems. First, the inability to use the black-box validation if two executions with the same data set lead to different results. This clearly affects all the parts of the process involving some kind of randomness. To control this the stochastic issues will be removed by using random seeds for parameter estimation.

Another issue could be the time. To mitigate this problem the initial set-up phase before the SCRUM iterations has been added (see subsection D.2.4 on Tasks Description). The goal of this is to automate the analysis, execution and all the other time-consuming tasks not related to the actual development of the optimizations.

More problems such as the inability to correctly enqueue jobs to Mare Nostrum III will be addressed by counting on the BSC team and the project director. The usage of extrae and paraver tools could also be difficult. To overcome there are seminars to which one could attend. On the other hand the tools team was contacted to get their help when needed.

At the present time COMPSs team is working on a brand new release (1.3). It has important changes with respect to the last one (1.2). The goal of this project to use, test and evaluate COMPSs so

It was decided that using the development version (1.3) would be more useful for the team and its performance is better than the old one. However, this new one is neither finished nor fully documented and tested. This is probably the major source of problems for this project.

Finally, the decision to work with the development version is the biggest risk of all because the work can not be finished if the release is not stable enough to run pyProCT.

D. Temporal Planning

D.1 Task List

This section lists all the tasks to be performed.

1. Project Management
    1.1. Project's Scope - 9h
    1.2. Temporal Planning - 6h
    1.3. Budget and sustainability - 3h
    1.4. First Presentation - 6h
    1.5. Context and Bibliography - 15h
    1.6. Degree's specialization specification - 10h
    1.7. Final document and presentation - 20h
2. Software design description - 40h
3. Analysis tools' research - 40h
4. Common set up for all SCRUM cycles - 40h
5. SCRUM iterations - 960h control cycles
    5.1. Initial analysis and optimization decision
    5.2. Optimization development
    5.3. Implementation measurements

6. Global performance analysis - 40h

7. Project Writing and Defence Preparation - 80h

D.2 Tasks Description

D.2.1 Project Management

This task is the responsible for the whole project planning and specification and covers all the deliverables of the GEP course.

Resources list

TexStudio,
desktop application used for report writing

GanttProject,
desktop application for Gantt chart creation

UPC Atenea,
online platform for deliverables submission

D.2.2 Software design description

The project aims to optimize an already existing project. This means that the first thing to do before starting to work is to explore, execute and, in general, familiarize myself with the original code. This will lead to the design description of pyProCT software.

Resources list

Git and Github,
to use the code we need a Github account to fork the original project repository. Once

forked we need a git-able OS, in this case Linux Mint 17, to clone it.

MareNostrum III account,

for executing the code on the supercomputer and correctly assess it's performance as well as execution limitations for instrumentation purposes.

SSH-able OS

to establish secure shell connections to the MareNostrum III computer for the program execution.

Paraver and other analysis tools,

to ease the understanding process with execution diagrams. Also, on this first stage, we will start looking for the best available tools for the posterior analysis stage.

D.2.3 Analysis tools' research

Once the whole program execution and mechanics, the next step is to decide which analysis tools are going to be used. Paraver is going to be amongst them. This task needs to be done after getting familiar with the code because otherwise we could end up trying to use tools which are not compatible with the python version and packages used and the remote execution pipeline.

It is important to note that this is mainly a research stage. This means that no consistent code modifications are going to be performed, just the necessary ones to ensure that the tools work well with the code.

As a big part of the project is going to be analysis, we set up first this research stage and an implementation/instrumentation one in order to correctly address the importance of this matter.

Resources list

See Code Familiarization resources list D.2.2.

D.2.4 Common set up for all SCRUM cycles

Once familiarized with the code and the analysis tools, the next step will be to instrument the code for it's preliminary analysis. The instrumentation should be deep enough to test all the possible implementations to be done, regardless they are performed on the matrix distances calculation, task level or scheduler level.

On this stage we will also do a preliminar work aimed to automate and speed up as much as possible the code analysis and execution. On one hand, this will speed up the SCRUM iterations by automating the analysis and execution with tools like bash scripts, allowing us to focus on the actual development and analysis and avoid wasting time on repetitive task as graphs' generation or the remote execution. On the other hand, it will also help to avoid human errors on the execution parameters, analysis settings and environment configuration.

Resources list

The resources required for this stage are linked to the analysis tools decisions so they can not be listed until the first stage is finished.

However common resources for automation are required (i.e. bash scripts). The Linux Mint environment used provides this basic functionalities.

D.2.5 SCRUM iterations

As explained on the Scope of the Project document, the work flow will follow the analysis-implementation-analysis structure. Each cycle is going to have, at least, one control meeting for each three weeks of work. This way we will keep track of the optimization development and solve possible problems.

The first and major implementation is going to be the COMPSs refactoring. Being the most important optimization, all iterations are going to be focused on it till it's completely ended (including analysis). If there is time for other optimizations its schedules and constraints are going to be decided at the start of each cycle.

The work to be done on each phase of the iteration has already been specified on the Project's Scope document and requires no further description.

Resources list

The common requirements for each cycle are all the ones mentioned on the previous sections. Some optimizations may require new tools and resources so at the start of each iteration, on the preliminary analysis, a resource analysis and specification is going to be performed.

D.2.6 Global performance analysis

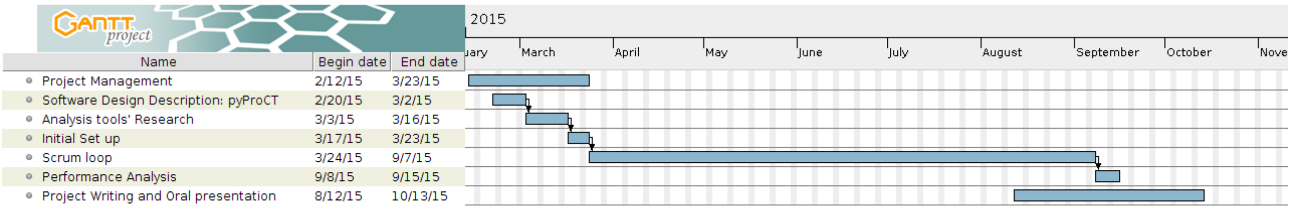
After finishing all the implementations decided, a global analysis is going to be performed. The aim of it will be to show not the changes introduced by each optimization but how these all behave and interact between them. Also the global speed up obtained, the final execution pipeline and conclusion fall into this section.

Resources list

This section requires no more resources than the needed to correctly reach this stage of the project.

D.3 Gantt and PERT charts

This is the resulting Gantt and PERT charts for the described tasks.



Name	Begin date	End date
Project Management	2/12/15	3/23/15
Software Design Description: pyProCT	2/20/15	3/2/15
Analysis tools' Research	3/3/15	3/16/15
Initial Set up	3/17/15	3/23/15
Scrum loop	3/24/15	9/7/15
Performance Analysis	9/8/15	9/15/15
Project Writing and Oral presentation	8/12/15	10/13/15

Figure D.1: Gantt Chart

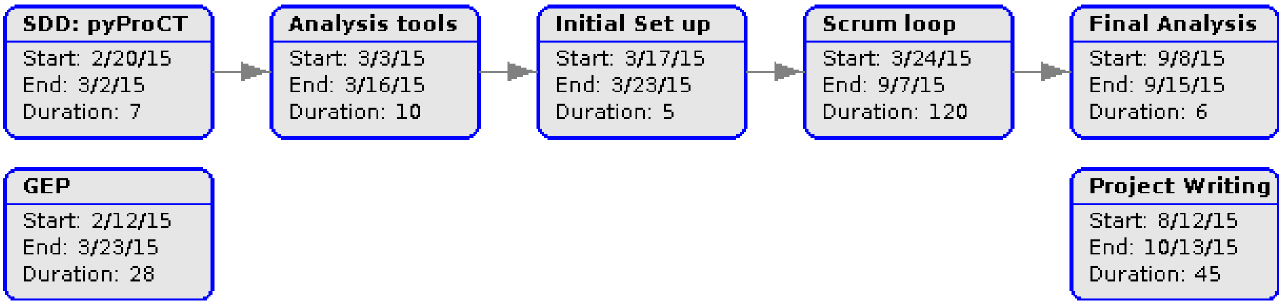


Figure D.2: PERT Chart

E. Budget

This section describes the required budget for the pyProCT optimization project. It contains a detailed description of the material and human costs. It is divided into: human, hardware and software resources; instead of specifying the costs per tasks, we have decided to use this structure because there are not remarkable differences between the resources used for each task so, grouping them this way, the document will be clearer, will avoid too many subsections and, on the Temporal Planning, the resources needed for each task have already been specified.

E.1 Human Resources

The project was completed by developer and a supervisor. The first worked with an eight-long workday from Monday to Friday. The second's task was to supervise and assess the working process, give advice and help to solve issues. The estimation of the human cost is tied to the work time represented, in this case, by the Gantt Chart and the task description provided on the D.3 Gantt and PERT charts section of the temporal planning.

First is important to note that the Project Management task overlaps with other tasks. However the duration of this section is tied to the programmed schedule of the GEP course, not to the amount of work required to finish it. Thanks to that, we will consider that from 12th February to 12th March the workday is going to be equally distributed amongst the overlapping tasks, which are less work-intensive because they are merely familiarization and research tasks. Similarly, because the results are tied to the development of COMPSs new release, it the report will be started before having all the results.

The defined work period has 176 workdays from 12th February to 13th October (12d + 22d + 22d + 22d + 22d + 22d + 5d monthly breakdown) amounting to approximately a total of 1400 hours. For the supervisor we estimate 200 hours distributed between meetings, project setup, problem's resolution and correction of this document.

Role	Price per hour	Time	Cost
Project Developer	10,00 €	1400h	14.000,00 €
Project Supervisor	30,00 €	100h	6.000,00 €
Total	-	-	20.000,00 €

Table E.1: Human Resources Budget

E.2 Hardware Resources

The hardware resources for this software project are going to be the development device, a laptop, and the testing one, the Mare Nostrum III. It's assumed that the computer used for development has an internet connection and electrical connection. These costs are covered on the total budget together with unexpected costs. However to reduce the budget one possibility would be to consider using the university facilities. The university provides to it's students and developers a free network and plugs which is more than enough in this case.

Product	Price	Useful life	Amortisation
Mare Nostrum III	22.700.000,00 €	3 years	0[1] €
Laptop	1.200,00 €	3 years	150,09[2] €
Total	22.701.200,00 €		150,09 €

Table E.2: Hardware Resources Budget

[1] MareNostrum III is a public infrastructure so users need not to pay to use it

[2] Given by: Cost / Useful life * Time used on project (664h)

E.3 Software Resources

pyProCT is an open source software hosted on a public github repository which can be used without restriction subject to the condition of citing the following article:

pyProCT: Automated Cluster Analysis for Structural Bioinformatics J. Chem. Theory Comput., 2014, 10 (8), pp 3236?3243 DOI: 10.1021/ct500306s
As our aim is to improve this software we want to keep it as it is. This means, on one hand, that all the features and optimizations added to it will also be free and public, using no third-party paying software. On the other hand, being it a public software we have decided that the development will allow reproducible research, meaning that all the tools used for analysis are also going to be free and available to anyone trying to reproduce the analysis and optimizations of this project.

Product	Price	Useful life	Amortisation
Linux Mint 17.1	0,00 €	-	0,00 €
Extrae	0,00 €	-	0,00 €
Paraver	0,00 €	-	0,00 €
Git	0,00 €	-	0,00 €
Github account[3]	0,00 €	-	0,00 €
Texstudio	0,00 €	-	0,00 €
GanttProject	0,00 €	-	0,00 €
Dia2code (UML drawing)	0,00 €	-	0,00 €
Atenea UPC	0,00 €	-	0,00 €
Other tools	0,00 €	-	0,00 €
Total	0,00 €€		0,00 €€

Table E.3: Software Resources Budget

[3] The repository is public so no premium account is required

E.4 Total Budget

Adding up all the cost described on the previous section we get total cost of the project, to which we need to add the VAT, which is 21 % in Spain. We do not expect big problems or incidents because, as we stated, we aim to use only free software so any modification or change on the task's planning will mainly just add office rental costs (taking into account that the office rental also includes the electricity and internet costs).

To control unexpected events we will add to the Total Cost an amount of money to confront them. These would cover various problems such as: an electricity or internet cost rise, more required office time (rising the rental costs and network/electricity) or, in case of not having enough time, the hiring of supporting help (other developers).

Resource	Price	Useful life	Amortisation
Hardware	22.701.200,00 €		150,09 €
Software	0,00 €		0,00 €
Developers	20.000,00 €	-	20.000,00 €
Office rental	5.000,00 €	-	5.000,00 €
Unexpected costs	3.000,00 €	-	3.000,00[4] €
Subtotal	22.729.200,00 €	-	38.011,70 €
VAT (21 %)	4.773.132,00 €	-	7.982.46 €
Total	27.502.332,00 €	-	45.994.16 €

Table E.4: Total Budget

[4] Given by: Cost / Useful life * Time used on project (664h)

F. Sustainability

F.1 Economic

An economic assessment has already been described on the Budget Section. The resources used for the development have been kept to a minimum; trying to use all the free software and resources provided by the university, in which a project like this could be developed; being developed by a single person, implying just one salary. The time used, however, could be reduced by having a developer and an analyst. This way on the SCRUB iterations while the developer is working a new feature the analyst could be working on the results of the previous one and so on. A part from that it's difficult to reduce more the amount of work because we have already considered a full-length workday without holidays.

On the major optimization, the COMPSs refactor, we are utilising an already developed framework which eases a lot the amount of work required for it. Thanks to the collaboration with the COMPSs developing team we achieve, as said, a faster implementation, good support, because the framework is currently used, and also we help a good framework as COMPSs to gain more notoriousness and relevance on research projects.

This project will have an 8 in the economical viability area because the cost can hardly be reduced. However, performing a more in-depth ot the risks of the project could help to reduce the budget for unexpected problems.

F.2 Social

On the social dimension we find that the optimization of this software will allow more research teams or enterprises to use it. This is important because on large datasets the amount of time and processing power can be overwhelming for small teams. Even if an optimization as this can not directly change a whole country, it could help universities and developers to waste less time of Supercomputing centers which are quite expensive and so improve their performance and resource disposal and use. As said, these project will not produce better results for the end user but will help the HPC providers.

The optimization covers a necessity as this project is done on BSC demand. So they will benefit from it, harming no one else on the process.

On the social area it will also receive an 8 because it will help to improve and further develop the COMPSs framework on the industry, providing more data, cases of use and information to the BSC/COMPSs development team.

F.3 Environmental

The environmental-related resources of this project are, primarily, two: the Mare Nostrum III supercomputer and the development laptop.

Both resources use electric energy. Theoretically this project will diminish the energy used by the supercomputer, which is quite high, but, in fact, the supercomputer is always running so the impact will be almost negligible, from a power-consumption point of view. However, on smaller scale devices this could in fact reduce a little the amount of energy spent, not worsening it in any case.

The COMPSs framework is used to help the development so we are reusing previously done work, giving the original project more scope and giving it more usage, helping to make the most out of the framework development.

This a software project so no other resources than the electricity used to run it is required. No direct waste is going to be produced by it's use. As it is an open source project aimed to be used or improved the whole project can be reused, both for new projects aiming to reduce even further the power and time consumption and for teams requiring a generic clustering analysis method.

On the resources analysis the project will be awarded with a 9 because the only thing that it's not environmental friendly is the electricity consumed by the Mare Nostrum III and the laptop to run the program and, compared to the average consumption per person nowadays, it's not a big deal.

G. List of References

This is a list of the documentation accessed to develop this project. It contains manuals and readmes of the tools, packages and software used.

- **pyProCT**

  Github Readme,

  https://github.com/victor-gil-sepulveda/pyProCT

  Dropbox Supporting Information,

  https://dl.dropboxusercontent.com/u/58918851/pyProCT-SupportingInformation.pdf

- **pyScheduler**

  Github Readme,

  https://github.com/victor-gil-sepulveda/pyScheduler

  Python Package Index (pypi),

  https://pypi.python.org/pypi/pyScheduler

- **MareNotrum III**

  User's guide,

  http://www.bsc.es/support/MareNostrum3-ug.pdf

- **COMPSs**

  User Guide,

  http://compss.bsc.es/releases/compss/latest/docs/compss-manual.pdf?tracked=true

  Tutorials,

  http://compss.bsc.es/releases/tutorials/?tracked=true

IDE User Guide,

http://compss.bsc.es/releases/ide/doc/1.2/COMPSs_IDE_user_guide_v1.2.pdf?tracked=true

Installation,

http://compss.bsc.es/releases/compss/latest/docs/installation-guide.html?tracked=true

Release Notes,

http://compss.bsc.es/releases/compss/latest/docs/RELEASE_NOTES?tracked=true

- **Performance Tools**

Extrae User Guide,

http://www.bsc.es/sites/default/files/public/computer_science/performance_tools/extrae-3.1.0-user-guide.pdf

ClusteringSuite intro,

http://www.bsc.es/ssl/apps/performanceTools/files/docs/T2_Clustering.pdf

ClusteringSuite manual,

http://www.bsc.es/sites/default/files/public/computer_science/performance_tools/clusteringsuit

Paraver introduction,

http://www.bsc.es/sites/default/files/public/computer_science/performance_tools/w1_introtool

Paraver internals and details,

http://www.bsc.es/ssl/apps/performanceTools/files/docs/W2_Paraver_details.pdf

Instrumentation,

http://www.bsc.es/ssl/apps/performanceTools/files/docs/2A_Instrumentation.pdf

Tools scalability,

http://www.bsc.es/ssl/apps/performanceTools/files/docs/T1_Scalability.pdf

Bibliography

[1] Amir Adler, Michael Elad, and Yacov Hel-Or. Linear-Time Subspace Clustering via Bipartite Graph Modeling. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–1, 2015.

[2] Michael Armbrust, Ion Stoica, Matei Zaharia, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, and Ariel Rabkin. A view of cloud computing. *Communications of the ACM*, 53(4):50, April 2010.

[3] Mohamed Walid Ayech and Djemel Ziou. Segmentation of Terahertz imaging using k-means clustering based on ranked set sampling. *Expert Systems with Applications*, 42(6):2959–2974, April 2015.

[4] R. M. Badia, J. Labarta, R. Sirvent, J. M. Perez, J. M. Cela, and R. Grima. Programming Grid Applications with GRID superscalar. *Journal of Grid Computing*, 1(2):151–170, 2003.

[5] Ana Maria Burca and Ghiorghe Batrînca. Application of cluster and discriminant analysis on romanian insurance market. In *Vision 2020: Sustainable Growth, Economic Development, and Global Competitiveness - Proceedings of the 23rd International Business Information Management Association Conference, IBIMA 2014*, volume 1, pages 817–824. International Business Information Management Association, IBIMA, 2014.

[6] Julia Y. K. Chan and Christopher F. Bauer. Identifying At-Risk Students in General Chemistry via Cluster Analysis of Affective Characteristics. *Journal of Chemical Education*, 91(9):1417–1425, September 2014.

[7] Joaquín A. Cortés, José Luis Palma, and Marjorie Wilson. Deciphering magma mixing: The application of cluster analysis to the mineral chemistry of crystal populations. *Journal of Volcanology and Geothermal Research*, 165(3-4):163–188, September 2007.

[8] Lisandro Dalcín, Rodrigo Paz, Mario Storti, and Jorge D'Elía. MPI for Python: Performance improvements and MPI-2 extensions. *Journal of Parallel and Distributed Computing*, 68(5):655–662, May 2008.

[9] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *2008 Grid Computing Environments Workshop*, pages 1–10. IEEE, November 2008.

[10] Víctor A. Gil and Víctor Guallar. pyProCT: Automated Cluster Analysis for Structural Bioinformatics. *Journal of Chemical Theory and Computation*, 10(8):3236–3243, August 2014.

[11] L. Kupski and E. Badiale-Furlong. Principal components analysis: An innovative approach to establish interferences in ochratoxin A detection. *Food Chemistry*, 177:354–360, June 2015.

[12] Douglas Laney. *The Importance of 'Big Data': A Definition.* Gartner.

[13] Daniele Lezzi, Roger Rafanell, Abel Carrión, Ignacio Blanquer Espert, Vicente Hernández, and Rosa M Badia. Enabling e-science applications on the cloud with compss. In *Euro-Par 2011: Parallel Processing Workshops*, pages 25–34. Springer, 2012.

[14] Francesc Lordan, Enric Tejedor, Jorge Ejarque, Roger Rafanell, Javier Álvarez, Fabrizio Marozzo, Daniele Lezzi, Raül Sirvent, Domenico Talia, and Rosa M. Badia. ServiceSs: An Interoperable Programming Framework for the Cloud. *Journal of Grid Computing*, 12(1):67–91, 2013.

[15] Henriette Müller and Ulrich Hamm. Stability of market segmentation with cluster analysis – A methodological approach. *Food Quality and Preference*, 34:70–78, June 2014.

[16] S. Nayak, C. Panda, Z. Xalxo, and H. S. Behera. *Computational Intelligence in Data Mining - Volume 2*, volume 32 of *Smart Innovation, Systems and Technologies*. Springer India, New Delhi, 2015.

[17] E. Tejedor, Y. Becerra, G. Alomar, A. Queralt, R. M. Badia, J. Torres, T. Cortes, and J. Labarta. PyCOMPSs: Parallel computational workflows in Python. *International Journal of High Performance Computing Applications*, August 2015.

[18] Jake Vanderplas. Why Python is Slow: Looking Under the Hood, May 2014.

[19] Departament D'arquitectura De Computadors Vincent Pillet, Vincent Pillet, Jesús Labarta, Toni Cortes, Toni Cortes, Sergi Girona, Sergi Girona. PARAVER: A Tool to Visualize and Analyze Parallel Code. *In WoTUG-18*, 1991.