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

Submitted as Degree Final Project
of Bachelor Degree in Informatics Engineering 15/10/2015

Índice general

1. 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 dataïs 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.

1.1. 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]

1.2. 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.^{a}pproach 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.

1.3. 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].

2. Objectives

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

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

2.1. 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 3.1 contains the software design description with detailed information of each section of the program and its control parameters.

2.2. Refactor pyProCT with pyCOMPSs

This objective comprises the code adaptation to make pyProCT work with the pyCOMPSs framework. The section 3.2 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 10.1

2.3. 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 3.3.

3. Methodology

3.1. 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.

3.1.1. 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 7 Documentation section.

3.1.2. 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.

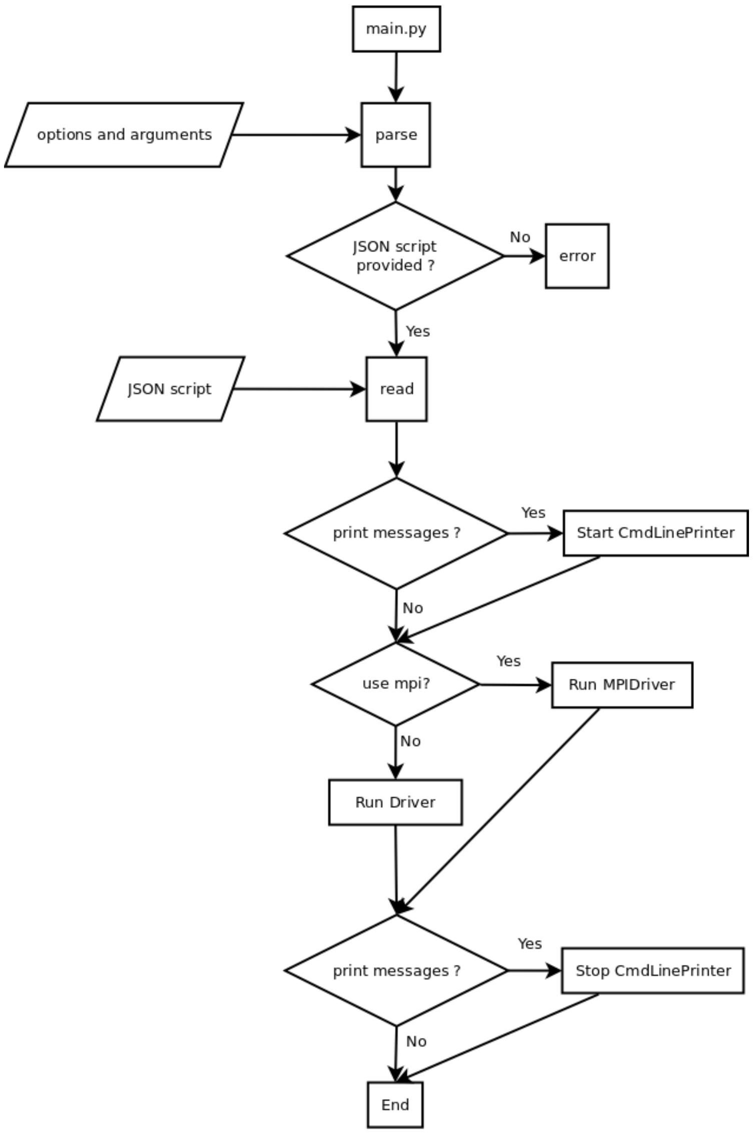


Figura 3.1: Global Section Execution Flow

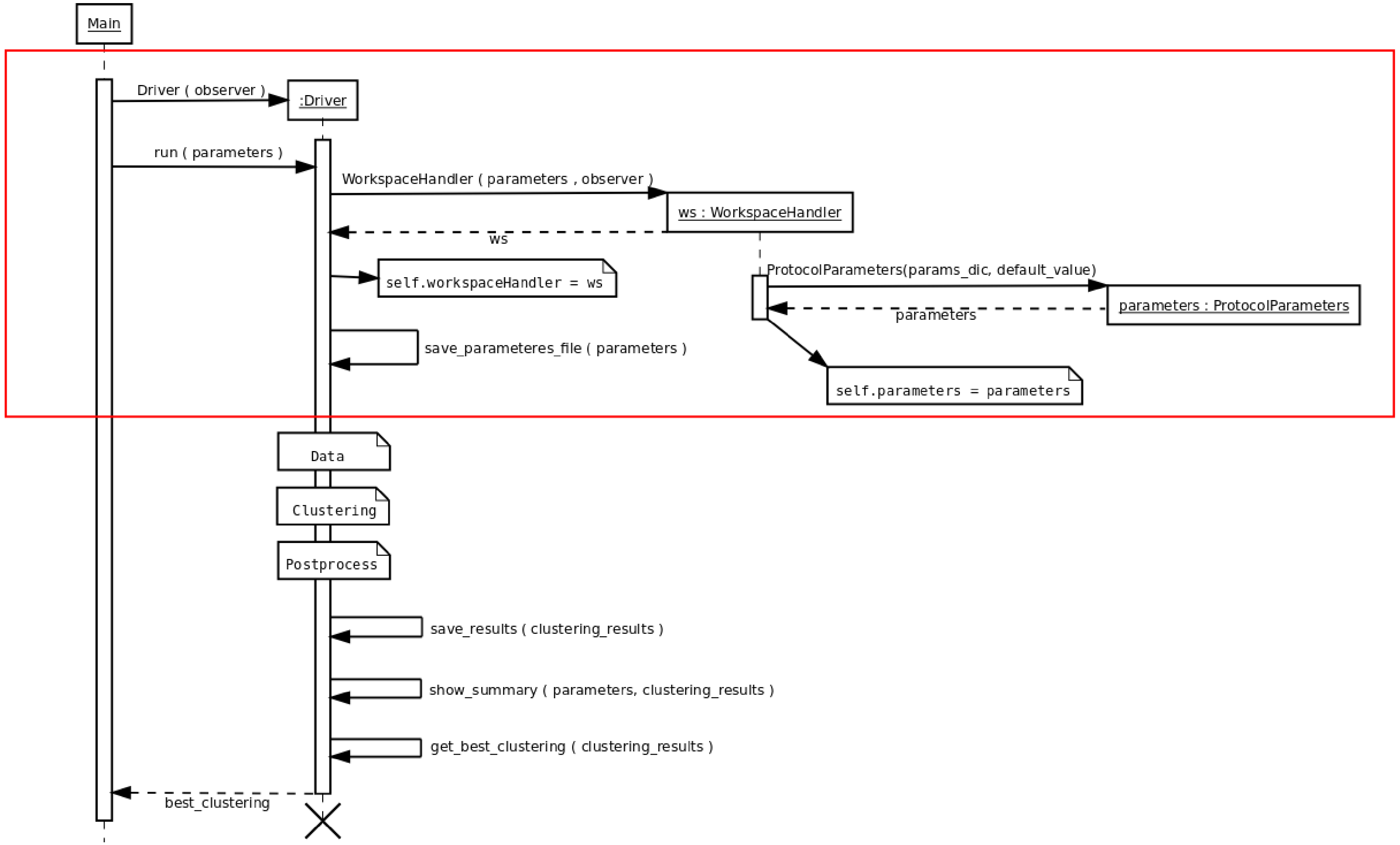

Main
Driver (observer)
:Driver
run (parameters)
WorkspaceHandler (parameters , observer)
ws : WorkspaceHandler
ws
self.workspaceHandler = ws
ProtocolParameters(params_dic, default_value)
parameters : ProtocolParameters
parameters
save_parameteres_file (parameters)
self.parameters = parameters
Data
Clustering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

Figura 3.2: 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:

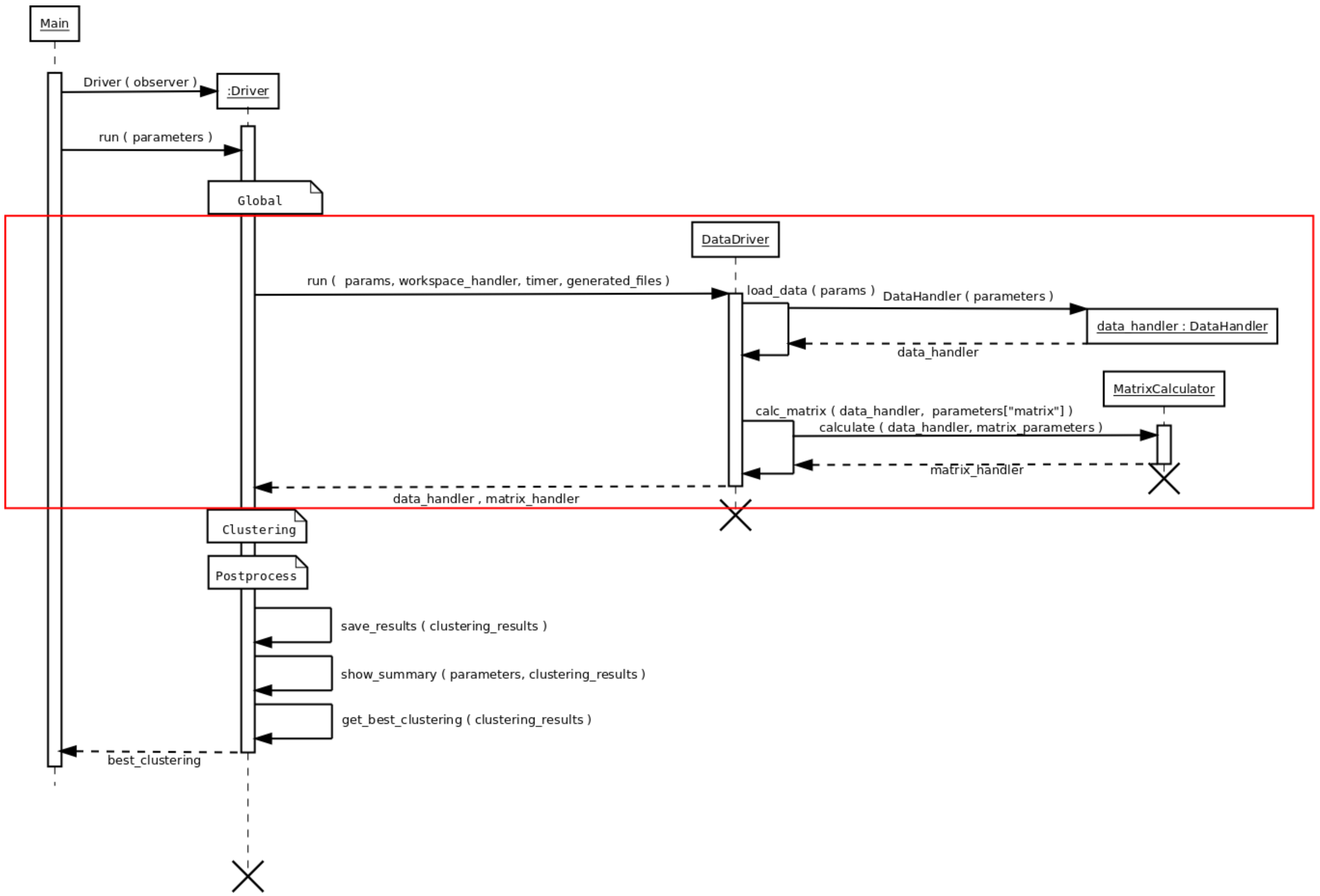

Main
Driver (observer)
:Driver
run (parameters)
Global
DataDriver
run (params, workspace_handler, timer, generated_files)
load_data (params)
DataHandler (parameters)
data_handler : DataHandler
data_handler
MatrixCalculator
calc_matrix (data_handler, parameters["matrix"])
calculate (data_handler, matrix_parameters)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

Figura 3.3: 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 3.4 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 "generateïs 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.

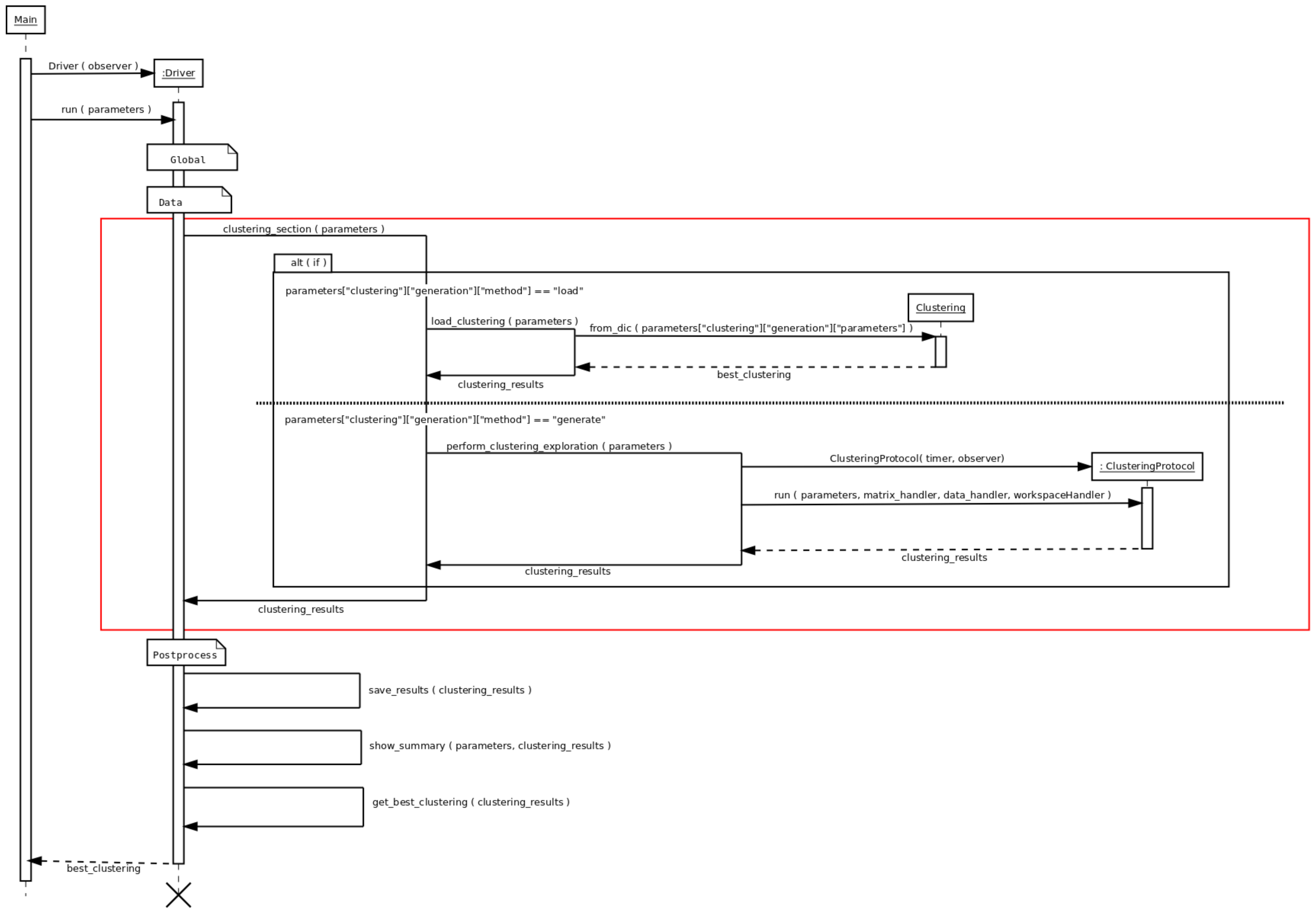

Main
Driver (observer)
:Driver
run (parameters)
Global
Data
clustering_section (parameters)
alt (if)
parameters["clustering"]["generation"]["method"] == "load"
Clustering
load_clustering (parameters)
from_dic (parameters["clustering"]["generation"]["parameters"])
best_clustering
clustering_results
parameters["clustering"]["generation"]["method"] == "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

Figura 3.4: 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)

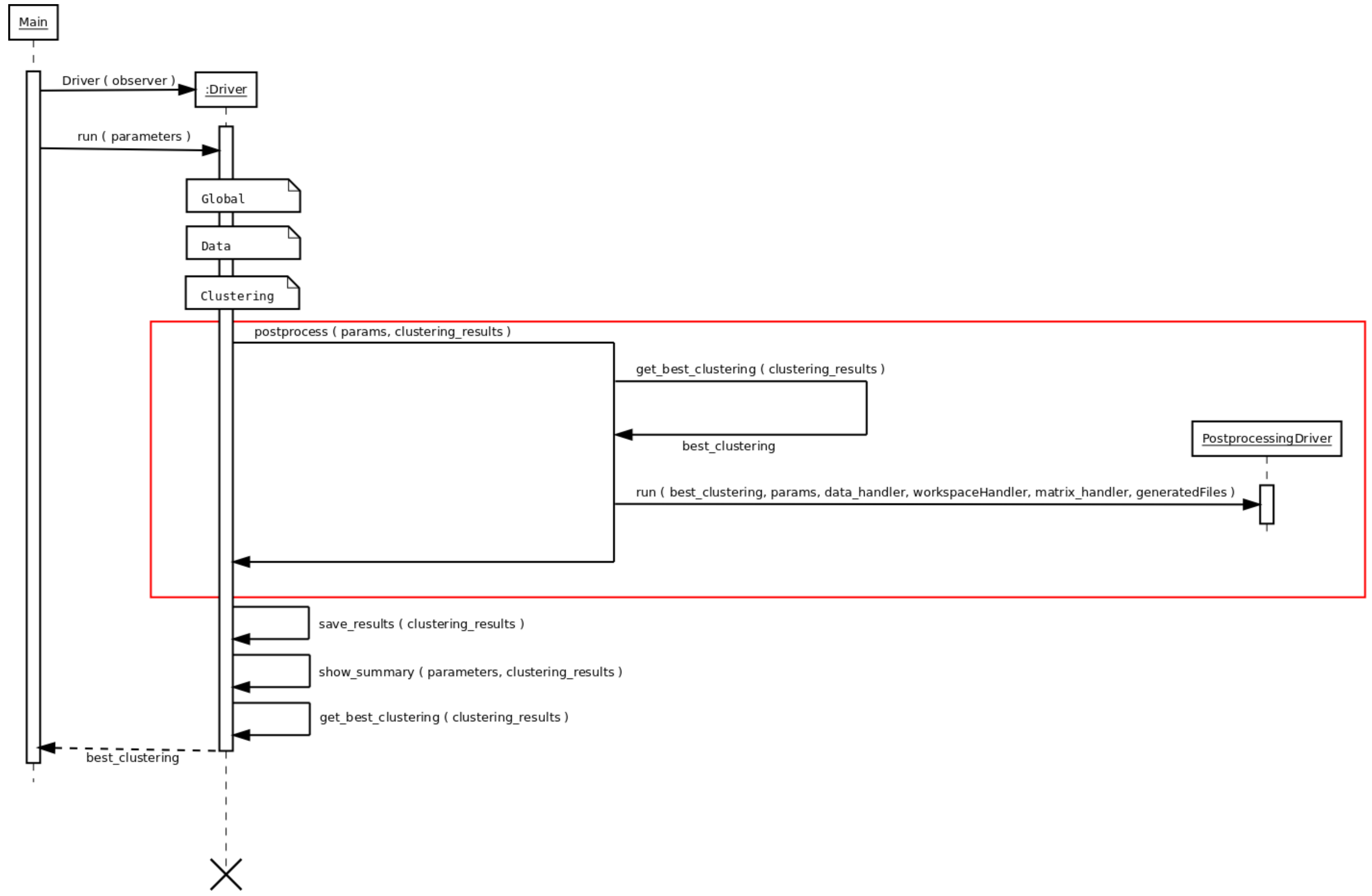

Main
Driver (observer)
:Driver
run (parameters)
Global
Data
Clustering
postprocess (params, clustering_results)
get_best_clustering (clustering_results)
best_clustering
PostprocessingDriver
run (best_clustering, params, data_handler, workspaceHandler, matrix_handler, generatedFiles)
save_results (clustering_results)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

Figura 3.5: 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).

3.2. 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.

3.2.1. 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.

Once I installed and ran a few pyProCT examples on my local machine I proceeded to MareNostrum III to do the same. Choosing a good structure to set up the environment is a must for executions on MN3.

I faced and spent a lot of time on configuration problems. This kind of issues kept popping up during the whole project. Despite that I prefered to group them here and give a brief description of the issues and how they were solved.

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 I compiled and installed it with the default release (2.6.9) with setup.py. On MN3 I had to add a custom installation path (with *–prefix=PATH* option) to setup.py because I have no permissions to write into the default installation directory. After installing it I found out that pyProCT can not be run under python 2.6.9 so I started again all the installation process with 2.7.3 once I figured what was causing the error.

The new installation lead to the following new bug:

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 I cloned the repo again and started 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).

3.2.2. pyCOMPSs

Prior to starting the refactor I analyzed which would be the best way to parallelize it. pyCOMPSs works by using python's decorators to define some functions as *COMPSs' tasks*. These tasks 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.

I wanted to maintain the possibility to use the other schedulers after the refactor so I kept the overall structure of pyProCT. However, I also wanted to exploit the possibility of reduce the code complexity while achieving the maximum performance improvement. I mention this because 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; my point is that the lines of code required are few 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, make 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*. I simply created a new Driver for the COMPSs scheduling. 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 I called directly the decorated methods (which internally COMPSs enqueues to it's pending's list). However this caused a problem related to the how the framework deals with the data.

COMPSs is a framework which allows to define a lot of different resources. The communication layer needs a high level of abstraction because workers (resources able to execute tasks) use different protocols (e.g. SSH or NIO). To send the data needed by each task, that is, the method's parameters, COMPSs serializes them to Java objects (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 to represent the forementioned 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]

To overcome this, Víctor wrote the lean and specialized pyRMSD (which stands for python Root Mean Squared Deviation). 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 approach raised two issues.

The first issue was where to perform the translations to a python list, which is linked to which is the function decorated as task. Before presenting the solution and the next issue it is worth to make a point about how the algorithms are implemented and called.

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) I implemented a wrapper class, called CompssTask, for the algorithm's execution with a single pyCOMPSS-decorated method.

The class is constructed with all the required information for the task execution. During the initialization I also do the forementioned translation of the matrix to a python list. 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.

3.3. 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, wether we want to save the computed matrix (and it's image) and so on (see Section 3.1.2 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.^{o}ption 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.^{o}n the .^{ex}pected 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 .^{ex}pected results dir".

The last three attributes allow us to use a single expected results dir 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 I also add other files to check such as the parameters.json and the clustering folders (containing information about the generated clustering).

3.3.1. Basic tests and issues

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

To solve it I could make all the schedulers work with the same call or adapt the tester to each scheduler (reading that information from the control script). I decided on the first 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.

Once done this modifications it was easier to write the tests. Before starting the refactor I generated all the expected results. I tested this results with the same code that generated them to make sure the tester worked but some of them failed because of the second, and expected, issue: the random initializations of some algorithms. To solve this I did the same that on tic-tac-toe, albeit a bit more complex: "set the seeds"for the algorithm's initializations removing the stochastic and non-deterministic parts.

Afterwards, for each scheduler I ran the basic tests, first with the original pyProCT, then with the refactor. However the pyCOMPSs mode was not available before the refactor so I validated it's output 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 I ran this suite of tests to validate the work done.

4. Results

This section describes the results of refactoring pyProCT with pyCOMPSs. I 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.

4.1. Programming Model

pyCOMPSs framework is supposed to require little extra code to be used so first I will compare difference of size between each version. Figure 4.1 shows the comparison between the different classes required for each parallelization. It's 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

Cuadro 4.1: Size comparison of duplicated classes

Python is a language designed to be easy to read with strong coding conventions so I did not try to minimize the code. I coded in a normal fashion trying to make things clear. Bearing this in mind 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 cleaner 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 an script to specifically 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 work 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.

4.2. Performance

COMPSs is designed to harness all the power of supercomputers and use all the available computing resources. The whole initialization of such a framework is not a trivial matter and because of that I expect it to be slow with respect to MPI or OpenMP. However, being designed to run programs with humongous datasets and computation times, this initial overhead is negligible. I expect to see that MPI and multiprocessing schedulers are faster than pyCOMPSs on small datasets and slower on larger ones.

4.3. Tools

5. Conclusion

6. 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.

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.

7. 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

Índice de figuras

Índice de cuadros

10. Appendices

10.1. Performance Tools Research

10.1.1. Motivation

After the familiarization with pyProCT it is time to explore its scheduler and to find the tools necessary to analyze its performance. I wrote an small sequential testing program 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 I will parallelize the program with the scheduler. Last two sections will describe the analysis process: first the instrumentation for generating traces and then the visualization tool. During the process I will try to find possible problems that may arise due to the parallelization, like the reproducibility on random algorithms or results' validation, which could complicate the posterior work on pyProCT.

10.1.2. 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 10.1 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.

10.1.3. pyScheduler refactor

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

10.1.4. 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 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, I set *play, montecarlo* and *exploration_handler* methods to be automatically instrumented and added a two events: 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

[1] Find extrae documentation on the performance tools section of 7 .Documenation
[2] Find paraver documentation on the performance tools section of 7 .Documenation
[3] +

able to extract correct traces as extrae was not able to detect the parallelization method. When I tried the MPI version it didn't work either 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 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 me some ideas to try but to no avail. I linked 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 but it didn't work. I left the issue open and went forward. Fortunately, after working on the new tracing system for pyCOMPSs at the BSC, I managed to develop a workaround tracing system albeit quite more rudimentary and inconvenient to use. Extrae has a command line usage of which I used two basic commands:

extrae-cmd init node slots

extrae-cmd emit slot event_value event_type,

I created a Python class wrapper for this two commands. This way I reused it to instrument the pyProCT later. This class deals with the extrae paths, 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 I dit raised a segmentation fault; knowing that emitting an event with an out-of-range thread ID raises a segmentation fault I figured out that the initialization was not correct. I tried a number of different methods. Because this extrae usage is not the recomended nor normal approach there is no documentation for it nor examples for it. After several days working on it I resolved to meet with the extrae team. Working with them we found out that the segmentation fault was caused by a bug on the release I was using. Kindly they fixed it and made a custom package for me to use. With it I finally was able to achieve a basic instrumentation for the parallel (with python-multiprocessing) version. This is limited to emit events and can not produce the advanced visualizations and results achieved on the serial and MPI version.

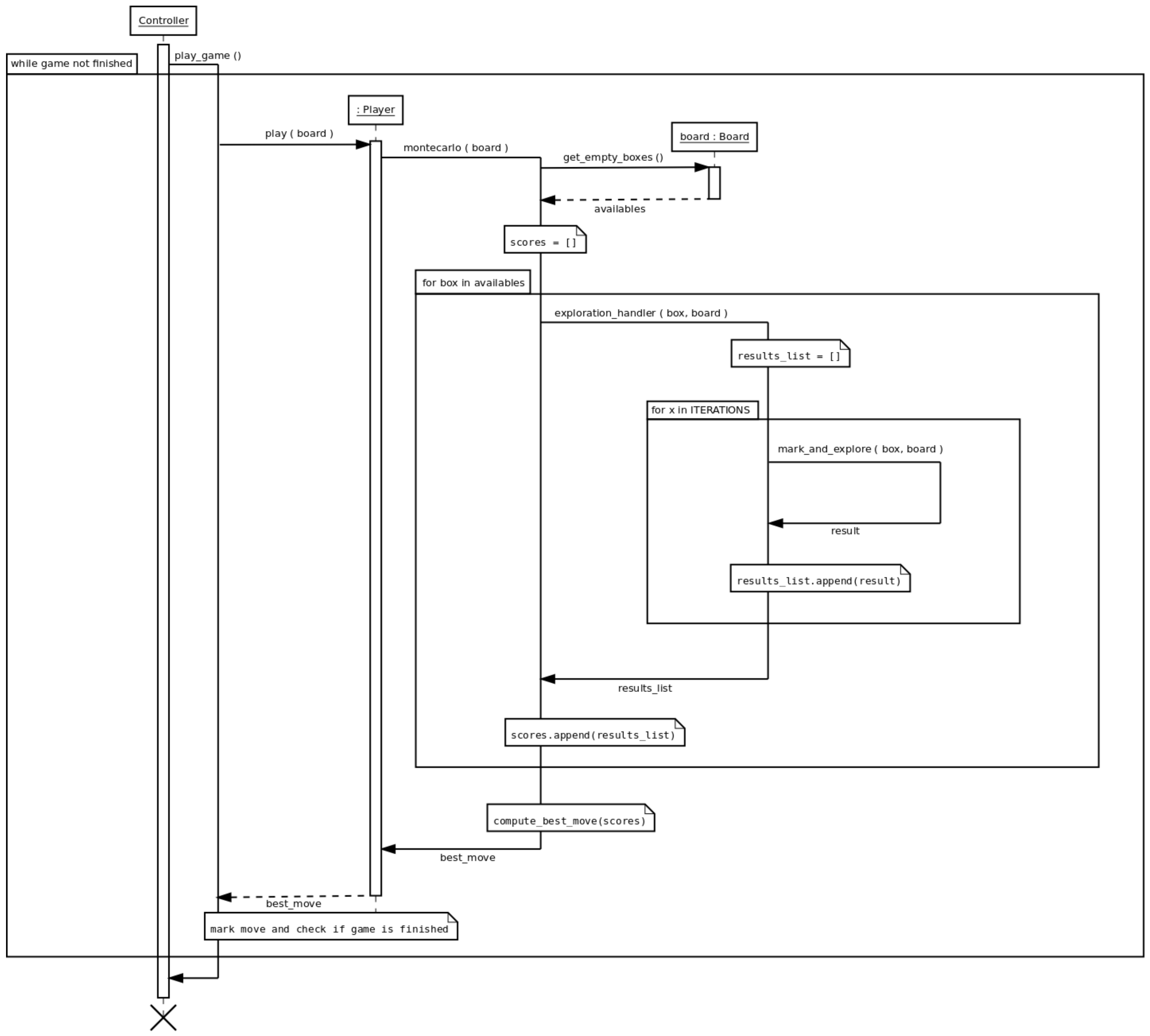


Figura 10.1: Tic-tac-toe Execution Flow

10.1.5. Visualizing with Paraver

Once generated the trace file the next step is to analyze 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 I configured 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 10.2 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 10.3), 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 10.4 we can see that the most computation-intensive areas associated with the exploration_handler function (in blue on Figure 10.2) have a good efficiency.

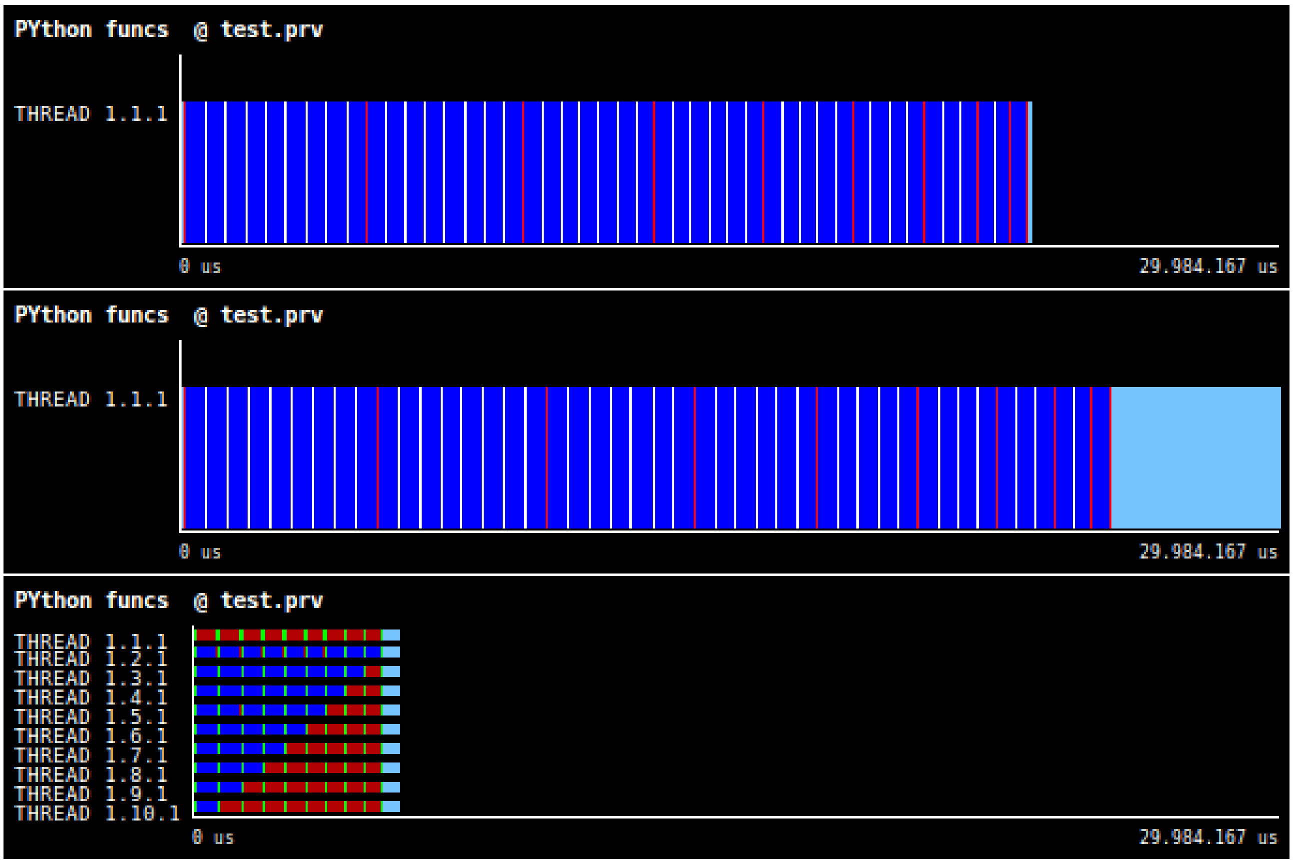


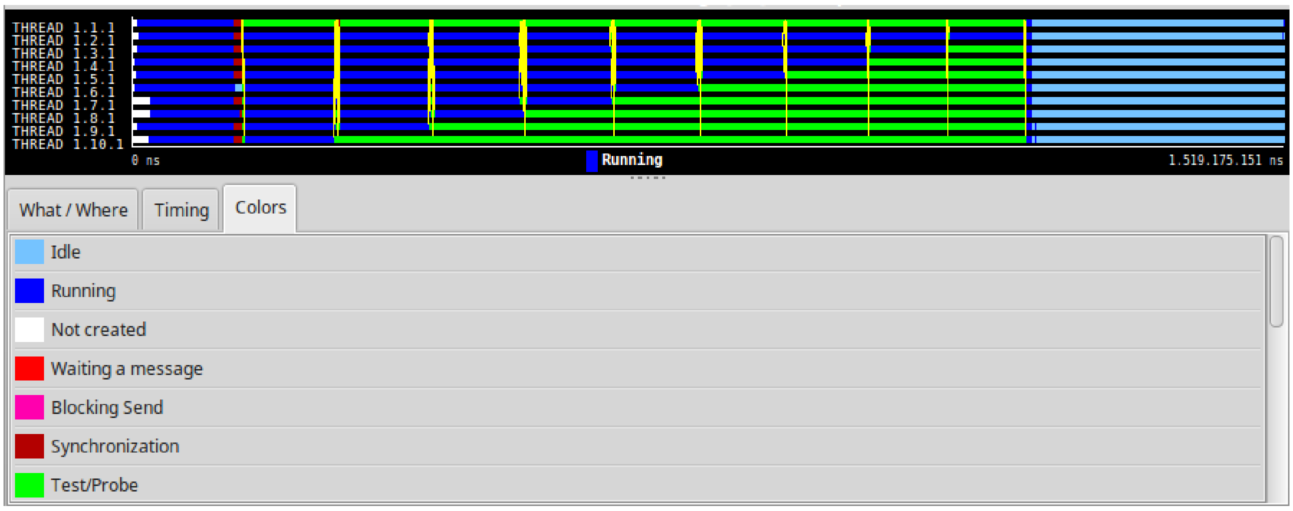
Figura 10.2: Tic-tac-toe 500 iterations executions

10.2. 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.

I have used Paraver and Extrae, 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



🟦	Idle
🟦	Running
⬜	Not created
🟥	Waiting a message
🟪	Blocking Send
🟥	Synchronization
🟩	Test/Probe

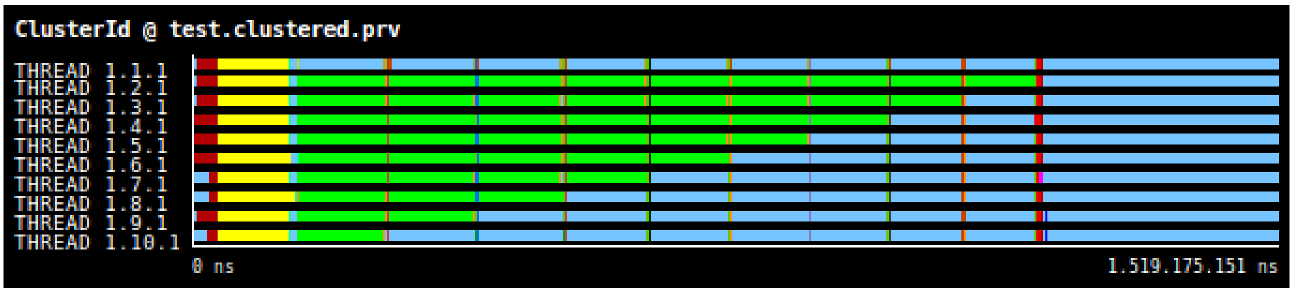
Figura 10.3: Tic-tac-toe MPI information for a 500 iteration execution

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 I used a laptop with the text editor Sublime Text 3. 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 10.3.2 Hardware Resources and 10.3.3 Software Resources sections).



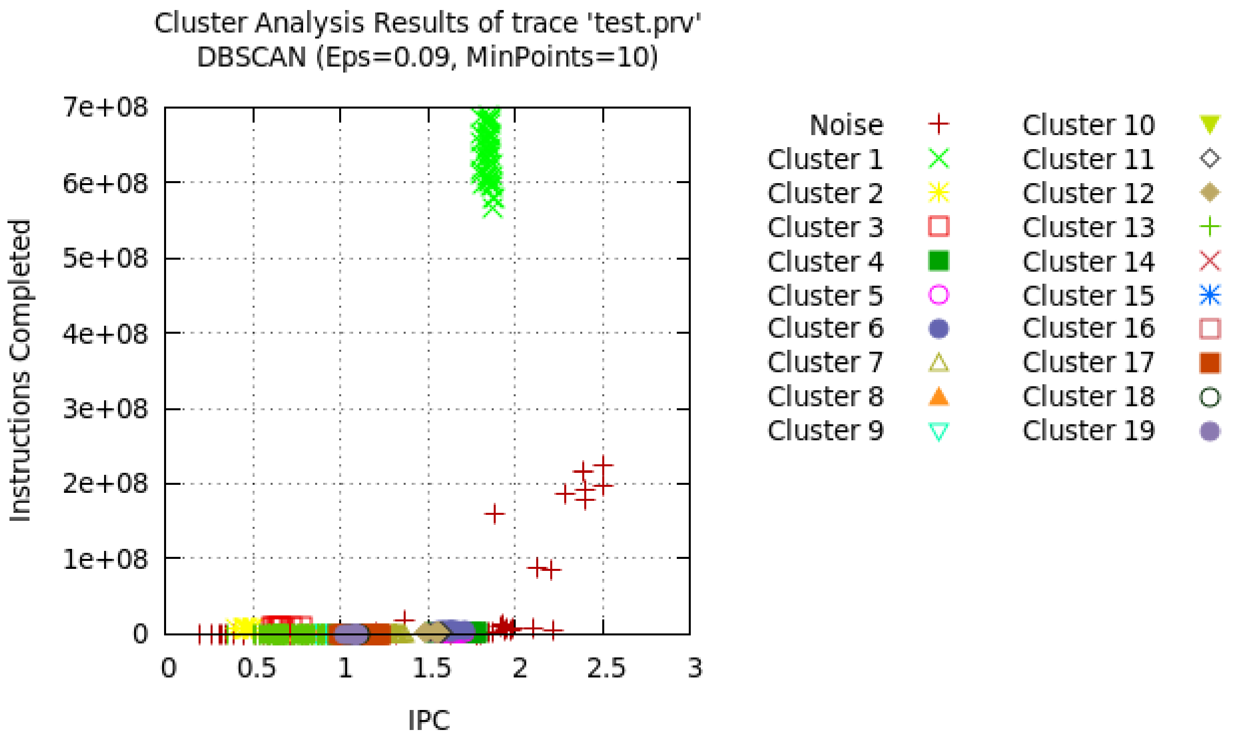


Figura 10.4: Tic-tac-toe timeline and clustering for 500 iteration MPI execution

10.2.1. 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, in case it affects the testing process, I will try to eliminate the stochastic issues with random seeds and manual and fixed 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 **??** 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 it, on the one hand, I went to a seminar about that tools. On the other hand I contacted the tools team to get their help when needed. This support is taken into account as an exterior consultant.

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

I 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 I can not finish my work if the release is not stable enough to run pyProCT.

10.3. 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.

10.3.1. 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 **??** 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, I will start writing the report before having all the results. This way I will have almost everything ready before having the results.

The defined work period has 176 workdays from 12th February to 13th October (12d + 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 €

Cuadro 10.1: Human Resources Budget

10.3.2. 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[4] €
Laptop	1.200,00 €	3 years	150,09[5] €
Total	22.701.200,00 €		150,09 €

Cuadro 10.2: Hardware Resources Budget

10.3.3. 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.

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

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

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[6]	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 €€

Cuadro 10.3: Software Resources Budget

10.3.4. 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).

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

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[7] €
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 €

Cuadro 10.4: Total Budget

10.4. Problems

When trying to generate traces with extrae (for MPI and sequential version) I got an Prody error. 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 I had 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.

Another issue was raised by some datasets. Depending on the computer and data when I try to recreate the condensed matrix on the pyCOMPSs task I get an incompatible format error. 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 I found that 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

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

issue.

Bibliografía

[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.