Master in High Performance Computing

Master's Thesis

A computational ecosystem for near real-time satellite data processing

Author:
Stefano Piani

Supervisor:
Dr. Stefano Cozzini

Academic Year 2014–2015

Contents

Introduction

The main goal of this work is the development of a computational ecosystem for nearly real-time inversion of high spectral resolution infrared data coming from meteorological satellites.

The overall ecosystem has been designed and developed as nearly real-time demonstration project to consolidate the quality and completeness of the envisaged level 2 products derived from the Meteosat Third (MTG) infrared hyperspectral sounder (IRS) measurements. Within this project, IASI [2] , and CrIS [4] observations, used as proxies for MTG-IRS, are processed by the Level 2 Validation and Demonstration Processor (L2VDP). The products derived from these observations are distributed for further analysis by operational forecasters or ingested in global or regional scale data assimilation systems [1, 3]. L2VDP is a prototype processor compliant to the processing foreseen by the operational MTG-IRS Level 2 Processor Facility (L2PF) as documented in [13]. Hence the L2VDP is a so-called end-to-end processor, meaning that it is a complete chain consisting of prototypes of all the processing modules foreseen for the operational processor.

L2VDP consists of many different processing modules developed in collaborative way by many scientists and assembled under the supervision of EUMETSAT. The inversion module (High Performance Level 2 Validation and Demonstration Processor) represents the evolution of the UWPHYS-RET/MIRTO package, the OSS RTM [5] (developed at A.E.R.) is the fast radiative model, the ECMWF forecast transformation module (developed by at ECMWF) represents the a-priori component of the system, and the scene classification (developed at the University of Basilicata) and the post processing (developed by EUMETSAT) are key elements for the generation of Level 2 products tailored for data assimilation purposes.

Scientific background and relevance of the project are presented and discussed in chapter 1, where details about the instruments and satellites along with an estimate of the size of the data to be processed is given. The chapter also provides a general overview of the challenges of near real-time processing along with the implemented solutions.

The project was developed along three main lines of activities strictly inter-connected which required a significant programming effort.

The first activity featured a complete software re-engineering of one of the segment of the processing system responsible of the computation of the scientific product to be disseminated. This activity is described in details in chapter 2. This chapter also explains why the Python language was chosen for the overall development of the processor.

The second activity was to provide, within a precise time period of two months, an optimised version of the inversion module above mentioned. The overall optimisation journey is illustrated in details in chapter 3.

Finally, in the third phase a complete computational ecosystem was designed and developed around the core inversion component re-arranged and optimised. This infrastructure allows the near real-time processing of the input data. The final product has been tested on some computational nodes embedded in EUMETSAT's Technical Computational Environment (TCE), which is a demonstrational computational facility. In chapter 4 estimate of the amount of computational resources needed to operate in near real-time processing for IASI instruments is also provided.

It has to be noted that the whole work was performed under a contract which imposed strict deadlines that had to be respected and specific deliverables to be submitted. For all the three activities discussed it was possible to successfully fullfill the requirements posed by the contractor.

CHAPTER 1

The physical problem

MetOp-A (launched on 19 October 2006) and MetOp-B (launched on 17 September 2012) are two satellites in a lower polar orbit, at an altitude of 817 kilometres, with the aim of providing detailed observations of the global atmosphere, oceans and continents. The two satellites will operate in parallel for as long as MetOp-A's available capacities bring benefits to users. An analogous satellite, MetOp-C, is due to be launched in 2017. They form the space segment component of the overall EUMETSAT Polar System (EPS).

EPS has brought about a new era in the way the Earth's weather, climate and environment are observed and has significantly improved operational meteorology, particularly Numerical Weather Prediction (NWP).

The data generated by the instruments carried by Metop can be assimilated directly into NWP models to compute forecasts ranging from a few hours up to 10 days ahead. Measurements from infrared and microwave radiometers and sounders on board Metop provide NWP models with crucial information on the global atmospheric temperature and humidity structure, with a high vertical and horizontal resolution. EPS also ensures continuity in the long-term monitoring of factors known to play an important role in climate change, e.g. changing patterns in the distribution of global cloud, snow and ice cover, and ocean surface temperatures and winds.

Metop carries a set of "heritage" instruments provided by the United States and a new generation of European instruments that offer improved remote sensing capabilities to both meteorologists and climatologists. These instruments augment the accuracy of temperature humidity measurements, readings of wind speed and direction, and atmospheric ozone profiles. One of the instruments that flies exclusively on the MetOp satellites is the Infrared Atmospheric Sounding Interferometer (IASI) which has the ability to detect and accurately measure the levels and circulation patterns of gases that are known to influence the climate, such as carbon dioxide.

Currently the elaborated products of the data collected from MetOp satellites by IASI instruments include temperature and humidity profiles with a vertical accuracies of 1 degree Kelvin and 10% per 1-km layer respectively, trace gases, and the cloud cleared radiances (CCR) on a global scale and these products are publicly available[1].

These results are the final product of a long chain of elaborations that, starting from the satellite data, produces the final output. The various steps of this elaboration are subdivided by levels. Level 0 represents the raw data collected by the interferometer. In the level 1 the same data is integrated with corrections and additional metrics (like the position). In particular, the level 1 is divided in three sublevels:

- *L1A*: the radiances are geolocated and calibrated
- *L1B*: the radiances are geolocated, calibrated and resampled
- *L1C*: the radiances are geolocated, calibrated, resampled and apodised

Finally, in the level 2 we have the final output (the retrievals), with the physical profiles of the atmosphere.

It is also clear that the elaborated data has to be made available to the users in the shortest time possible: the closer in time the elaborated data are to the moment they were recorded, the more valuable they are for the user.

One of the main goal of this thesis is to supply a processing system, named L2VDP, i.e. a software chain capable of produce level 2 IASI data starting from the level 1C data, which will implement a different algorithm compared to the one already implemented in the ESP project. This processing system will then be embedded in a framework in order to perform a massive near real-time analysis of all the incoming data.

Our main starting point is the High Performance Level 2 Prototype Demonstration and Validation Processor (HPL2VDP) [7]. This software package has been designed and implemented to invert high spectral resolution infrared observations of the Earth into atmospheric state parameters: vertical profiles of temperature, water vapor concentration, ozone concentration, surface temperature and surface emissivity. The code, written in Python, was developed from a baseline package named MIRTO, a Python conversion of the MATLAB retrieval system built at the Space Science Engineering Center of the University of Wisconsin, UWPHYSRET[6]. HPL2VDP has been embedded in a more general package which allows to compute from IASI L1 Data the final outcome ready for dissemination.

Another requirement of our project is to be able to elaborate, with just minor difference on the same algorithm, also the data coming from CrIS,

[1] See for instance `http://www.ospo.noaa.gov/Products/atmosphere/soundings/iasi`

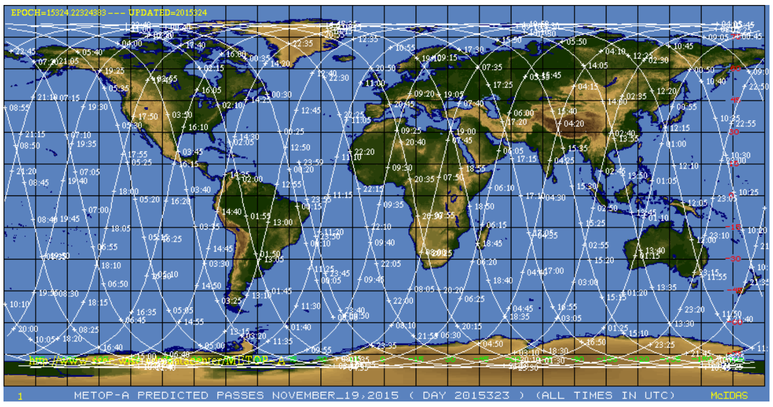

EPOCH=15324.22324383 --- UPDATED=2015324
METOP-A PREDICTED PASSES NOVEMBER_19,2015 (DAY 2015323) (ALL TIMES IN UTC)
McIDAS

Figure 1.1: The orbit of MetOp-A satellite on 19th of November 2015

another interferometer flying on the Suomi NPP satellite. While the number of channels of CrIS is smaller compared to the IASI one (and therefore the data it generates do not require an additional effort from a computational point of view), it requires some degrees of flexibility in the overall software structure to be able to handle the different data sources.

The peculiar type of the problem requires the elaboration of the IASI data to be close in time to the acquisition. Because of the amount of data collected, this is not a trivial task. Indeed, the data from the IASI instruments on board of satellites come subdivided in granula or FDU (Fundamental Data Unit). All data of a single FDU is contained inside a specific file whose name is the name of the granula. A single IASI FDU consists of 3 minutes of observations. Each FDU contains 23 scan lines, and each scan line has 120 Fields Of View (FOVs). Hence each FDU has 2760 FOVs and there are 480 FDU per 24 hours. Combining these numbers there are 1.324.800 FOV per day. If a FOV is declared cloudy, then it is impossible to extract information from it and must be therefore discarded. Assuming that on average half of the FOV will be declared cloudy and removed from further processing, then on a 24 hour time scale there will be 662.400 FOV to be processed.

The above numbers show clearly that an infrastructure should be able to operate at rough speed of at least $662400/86400 = 7.6$ FOV per second to keep the processing in real-time.

Moreover, beside the L1 IASI data file, L2VDP requires some background information extracted from the ECMWF[2] ensamble data assimilation system. A preprocessing phase is thus required to prepare and collect all the physical informations required by the algorithm (for example, the ground-type of the FOV). All these tasks should also be taken in consideration to prepare a suitable environment to elaborate the L1 IASI files.

[2]European Centre for Medium-Range Weather Forecasts

In the next chapter we will explain in more details all these phases and the overall structure of the software we developed.

CHAPTER 2

The processing system

In this chapter we present the software engineering activities which had the goal to integrate in one single package all the components needed to elaborate the L1 IASI data. The software techniques we adopted to reach this goal are also presented and discussed.

As briefly mentioned in the introduction, we used Python as a glue language among the different components. Moreover, due to the fact that HPL2VDP was also written in Python, in this thesis this programming language is not only used as a script language but also as a scientific programming tool to address HPC problems. In the conclusions we will discuss about the pros and cons of this choice.

2.1 The overall software structure

In this section we will briefly describe the steps needed to perform an analysis of the satellite data and to produce a valid retrieval for each observation. Such retrievals are computed by means of optimal estimation theory.

The overall computation can be split in two main sections: preprocessing and processing. A graphical representation of these two phases is given in picture 2.1.

In the preprocessing part (composed by several steps) we prepare all the data needed to perform the actual processing phase.

The first one is to generate an a priori information accordingly to the optimal estimation theory mentioned above: this is required for both surface emissivity, surface temperature and atmospheric component. For the *surface emissivity* of the ground we compute, for every channel of the instrument, the mean emissivity of the ground and the statistical deviation from the possible values from the mean.

For the atmospheric component, we use as *first guess* the forecast produced in the past by a theoretical model. Therefore, the preprocessor should look for a weather forecast which refers to the observation time and interpolate it on the position of the observation. The forecasts that we use come from the ECMWF (European Centre for Medium-Range Weather Forecasts) forecast model and the interpolated data is produced by ECMWF *Bgerr software*.

The last preliminary step is the *scene analysis*. If during the survey the field is cloudy then it is impossible to obtain correct results using our algorithms. For this reason it is important to detect the cloudy observations and remove them to avoid to waste computational resources.

After all these steps, we have prepared all the information needed to elaborate the satellite observations and we are ready to enter in the core of our system: the *processor*.

This processor phase is furthermore subdivided in three parts: the *inversion* (which is inherited from HPL2VDP), the *transformation* (that was improved as discussed in section 3.2) and the *quality control*.

For the inversion, L2VDP exploit a particular implementation of the Marquardt-Levemberg [11] method.

In order to compute the values of the radiances that we expect to measure starting from a particular physical condition, we rely on a proprietary software provided by Atmospheric and Environmental Research Institute written in Fortran that is performing the Optimal Spectral Sampling. We will refer to this software as "Forward Model" because it resolves the direct problem (compute the radiances starting from the physical values) while we are interested in the inverse one (compute the physical values starting from the radiances).

The embedding is realized using the *F2Py* library, which provides a connection between Python and Fortran languages.

The inversion is a delicate phase. First of all, there is no guarantee that the convergence will be reached. Even if it is reached, it is still possible that the convergence point is not the real value we were looking for because, in principle, different physical configuration could produce the same observation from the satellite.

During the inversion, we divide the atmosphere in some layers identified by some pressure intervals.

After this phase, we have finally computed a complete profile with the values of the desired physical quantities (usually the temperature, the concentration of water vapour and the concentration of the ozone). Unfortunately, we can not completely rely on the values we have found: in principle, it is possible that we have converged in a point that is not reasonable (maybe because it is simply too far from the first guess or maybe because the values it contains are not reasonably plausible) but it could also be that we reached a local minimum. For this reason a quality control software has been implemented. Such a software try to estimate the quality of the final retrieval and mark these observations as not reliable.

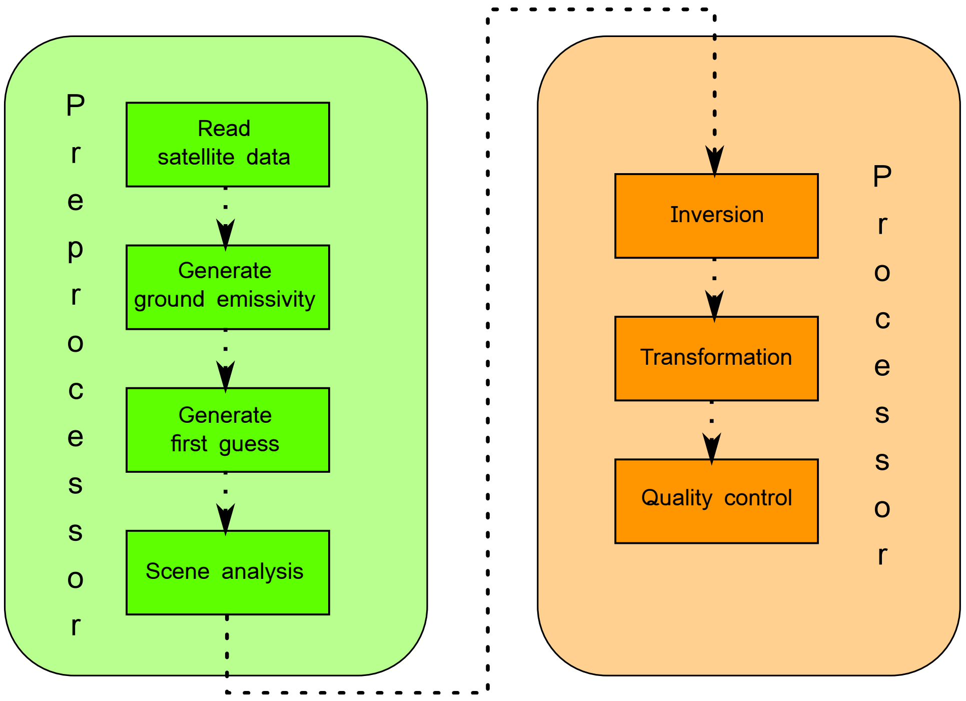


Figure 2.1: The overall structure of L2VDP

2.2 Where we start from

At the beginning of our activity, the software packages needed to perform all the actions above described were not fully integrated and no optimized procedures were available to accomplish the overall task.

As a matter of fact, the user had to invoke several different programs in order to obtain a retrieval for an observation starting from the native files and, in addition, most of the software required a lot of arguments from the command line (up to 12). Moreover, most of the times, the output of one program was not compatible to be used as input for another one and, therefore, other scripts had to necessarily be invoked to convert the files from one format to another. Some parameters were supposed to be submitted manually by the user to different programs, and it was therefore extremely easy to make mistakes by simply changing a parameter in one piece of software and forgetting to apply the same change to the others. Last but not least, a small modification in one of the programs could reflect in the need to perform many modifications as well in the source of other programs.

To set up a structure able to glue together all the software, EUMETSAT supplied us a prototype code written in Python which allowed us to identify the major steps of the computation. A substantial amount of work was therefore spent to integrate all the previous software inside the prototype and to modify it to full satisfy all the requirements.

2.3 The parallel algorithm

In this section, we want to describe the parallel algorithm that was available in the HPL2VDP implementation. Figure 2.2 gives an overview of the implementation. This is particularly relevant in the next chapter, where performance improvements of the processor are discussed. Since the operations performed by the preprocessor are strictly related to input and output, it is worth noting that there is not such a great need of parallelize also that part of the code. Although it could produce some gain in performances, the effort needed to rewrite all the software embedded in the preprocessor would make the operation not convenient.

The parallel approach of the processor, instead, is based on a master/slave paradigm where there is a main process which spawns several others processes, namely:

- *reader*, which reads all the observations and then distributes them to the computing processes
- *scribe*, which collects data from computing processes and then writes the results into a file
- *computing*, which performs the actual computation (mainly linear algebra kernels), receiving the main process data (i.e. observations) to be processed.

A parallel approach on multicore architecture is implemented using multiprocessing [24], a Python module that allows to spawn as many computing processes as the number of cores. Another level of parallelism is provided by the BLAS libraries which numpy is linked to (for example, a multithreaded version of the MKL libraries or the OpenBLAS library): all the linear algebra kernels could potentially be performed by a multithreaded algorithm provided by the library itself and completely transparent to the program. Every process is therefore able to spawn multiple threads during the execution. Of course, excessive concurrency shall be avoided, hence the product of the number of threads-per-process and processes shall not be greater than the number of the cores.

We have performed some tests to establish the right ratio between processes and threads. From these benchmarks, we discover that the best performance is achieved using just one thread and maximizing the number of processes.

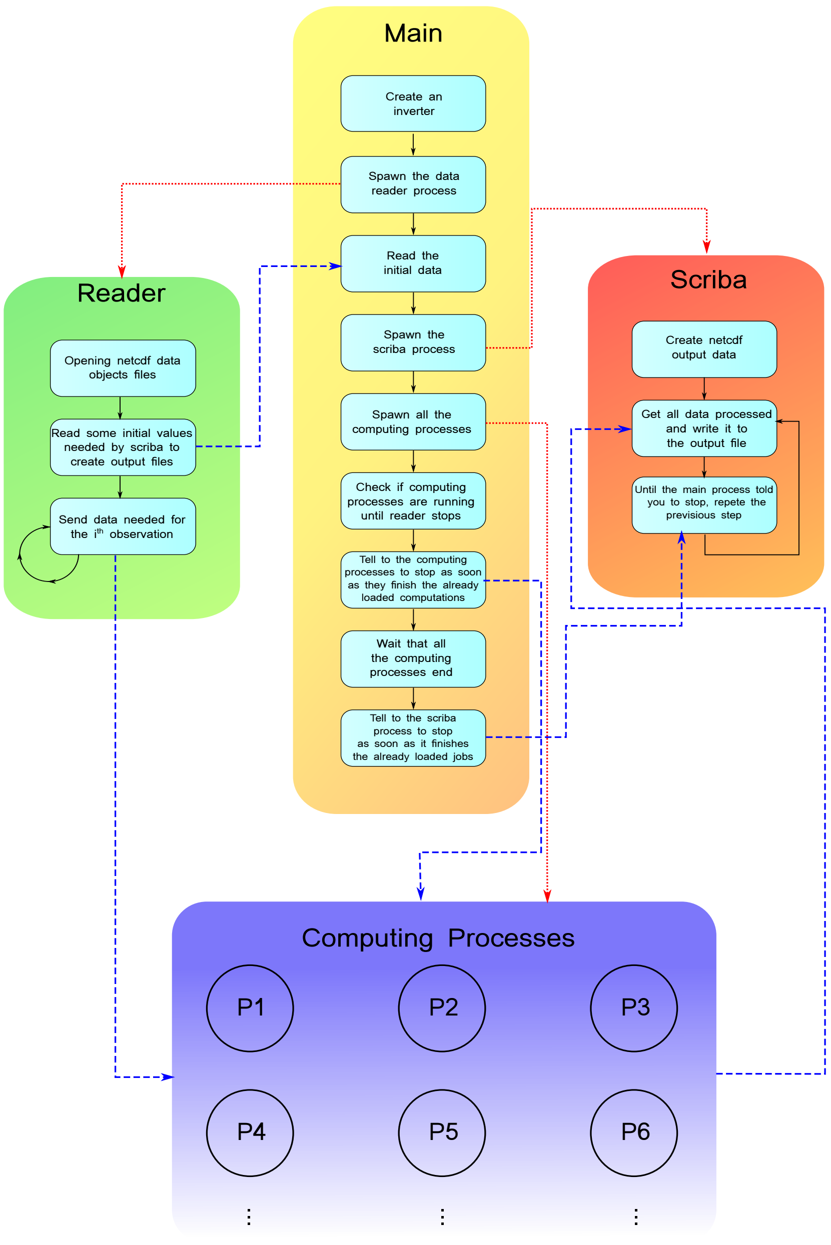

Main
Create an inverter
Spawn the data reader process
Read the initial data
Spawn the scriba process
Spawn all the computing processes
Check if computing processes are running until reader stops
Tell to the computing processes to stop as soon as they finish the already loaded computations
Wait that all the computing processes end
Tell to the scriba process to stop as soon as it finishes the already loaded jobs
Reader
Opening netcdf data objects files
Read some initial values needed by scriba to create output files
Send data needed for the ith observation
Scriba
Create netcdf output data
Get all data processed and write it to the output file
Until the main process told you to stop, repete the previsious step
Computing Processes
P1
P2
P3
P4
P5
P6

Figure 2.2: The parallel work-flow of the processor

I/O is confined in the initialization phase and then overlaps with computation (reader and scribe processes). These implementation features a Master/Slave approach where all processes take data independently from each other. Communication overhead partially overlaps with computation. We note that the Python language enforces several different limitations to the parallel implementation. One of the most noticeable is a non-straightforward approach to the shared memory exploitation, which requires the use of low level libraries like *ctypes* [15]. This could limit or hinder the overall scalability and the performance.

2.4 The final goal: an end-to-end processing system

As requirement posed by the contractor, the system we developed has to be at the same time coherent and modular. With coherent we mean that all the settings and the parameters are specified in one single file which describes the structure of the whole system. The configuration file has a structure similar to what is found in Microsoft Windows INI files and, as such, has a high human readability and it is self explanatory.

The main object of our program is the *chain-executor*. This object supervises every step of the processing system (including the preprocessor). To be constructed, it requires the path of a configuration file; its constructor then builds a preprocessor object and a processor object and saves a reference for them inside the chain-executor object. The constructor of the preprocessor object, in turn, create one object for each elaboration phase described in the picture 2.1. In the same way, the processor constructor creates an object for each of its own phases. When a native file, generated from a interferometer, should be elaborated the path of this file is passed to the chain-executor which calls an opportune method of the preprocessor first and of the processor afterwards. Every section is expected to produce a certain amount of data and save it in some files in a specific binary format called *FBF* (Flat Binary File) which is a quite common format in the meteorology field. The processor and preprocessor objects share the position and the name of the files between the various phases of the system. An example of the file generated and read in by the current configuration of the preprocessor of L2VDP is exposed in the picture 2.3

This structure guarantees a high degree of modularity in the system. Additional freedom is also guaranteed by the fact that every user can write and develop his or her own components and integrate them in the system. Indeed, in the configuration file there is a section where the user can specify the type of class to use for every phase of the system. The code, using some reflection techniques, will check whether a class that can be used to satisfy the user request has been implemented for that particular phase. If this is the case, then this class will be used and referenced when needed. In this way, there is no need to change any line of code to implement a new kind of

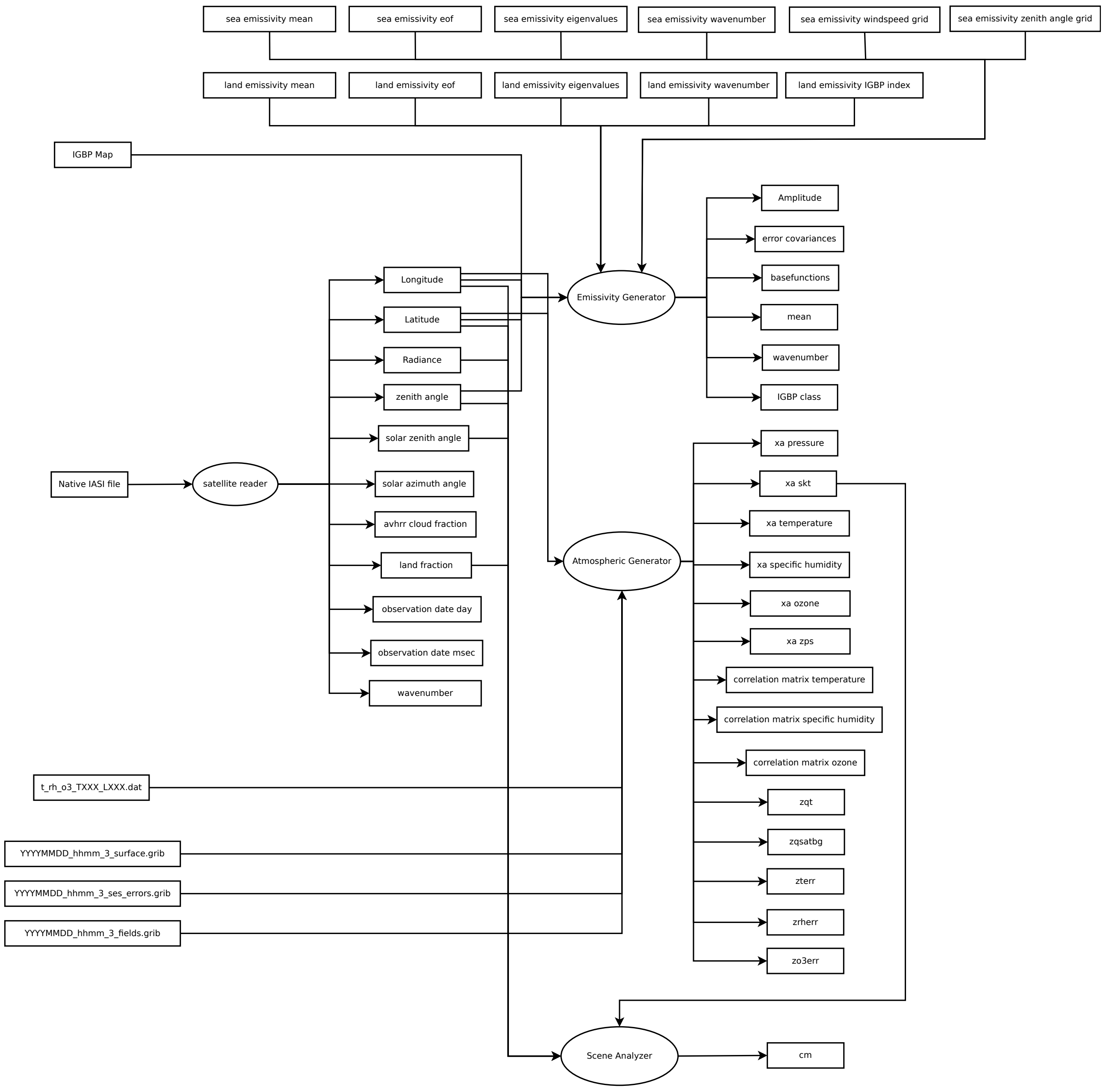


Figure 2.3: The files generated and read by the preprocessor

scene analyser, a different algorithm for the emissivity or a new reader for the ECMWF forecasts.

Currently, there are two readers for the satellite data (one for CrIS data and one for IASI). The first one is a wrapper around a C code (which is called as an external software) supplied to us by EUMETSAT while, for IASI, a dedicated library has been implemented in Python for reading the binary data. This library will be described in details in the section 2.5.

For what concerns the emissivity, an algorithm implemented in Python is available. Beside some small changes to integrate the algorithm in our software, the code was entirely written by EUMETSAT and was supplied to us in order to be integrated in the L2VDP project.

The default generation of the first guess, instead, is a wrapper around a Fortran code we received from EUMETSAT and written by the ECMWF. This software is simply called as an external utility by a Python routine.

Finally, there are three different classes that can be used as scene analyser: the first one allows to completely exclude the scene analysis and create a place-holder object. This is useful especially for debugging purposes. The second one uses some code written in Octave (which is integrated in the software using the Oct2Py library of Python) and the third class, which is also the default one, is a code written in C++ and called as an external executable. This software implements the algorithms described in [8] and has been written by the authors of the paper.

2.5 The Piasi library

The piasi reader library (Python IASI) is a library that we developed with the intent to provide users a way to manage and manipulate the content of the native binary files produced by the IASI instruments. This library has been released with the GNU LGPL license.

The IASI native files are written in a binary format which is efficient to store the data but not at all straightforward to read. Although a Fortran code able to read and extract data from the native IASI files already exists and was supplied to us by EUMETSAT, the piasi library offers several advantages. The first one is related with the Python integration: being completely written in Python, the piasi library integrates very well with our project. Thanks to this library, we could avoid to call another external software. Another benefit is in term of performance: the previous Fortran code was compelled to read the file byte by byte because of a Fortran limitation regarding binary I/O. The problem does not exist in Python and it is hence possible to achieve better performances.

The main and best advantage, though, is that the piasi library allows to manipulate the IASI files in many ways and does not just convert their format. Indeed, this library provides objects which are abstractions around the IASI native file. In this way, it is possible to write Python scripts that take decisions based on the content of the IASI files. Moreover, this opportunity can also be exploited using python in in interactive way.

To better understand this idea, we will present a small script that returns the name of the IASI files that contains observations regarding Europe inside a specific folder given as command line argument.

```
from os import listdir
from sys import argv
from numpy import any

from piasi_reader.iasi_l1c_native_file import IasiL1cNativeFile

# These are the coordinates of a square that contains the Europe
EUROPE_MIN_LAT = -40
EUROPE_MAX_LAT = 10

EUROPE_MIN_LON = 30
EUROPE_MAX_LON = 75

for filename in listdir(argv[1]):
    iasifile = IasiL1cNativeFile(filename)

    latitudes = iasifile.get_latitudes()
    longitudes = iasifile.get_longitudes()

    good_latitudes =  (latitudes > EUROPE_MIN_LAT) \
                    * (latitudes < EUROPE_MAX_LAT)
    good_longitudes = (longitudes > EUROPE_MIN_LON) \
                    * (longitudes < EUROPE_MAX_LON)

    good_observation = good_latitudes * good_longitudes

    if any(good_observation):
        print(filename)
```

Even though this script has been written for a demonstration purpose only, it shows clearly how, combining the powerful syntax of Python and the piasi library, it is possible to obtain interesting informations about the native IASI file in just a few lines of code.

2.6 Our tools

In this section we will present the modern software tools we used in the course of the software re-engineering activity performed. We consider useful to provide a short description of each of them and highlight the advantages they provide us. All the tools have been implemented inside the eXact-lab computational resources.

◇ 2.6.1 A version control system (Git)

Git is a free and open source distributed version control system designed to handle everything from small to very large projects. Git is incredibly efficient in storing the complete history of a project, allowing to create branches, moving trough different versions and undoing eventually wrong edits. Git was incredibly useful for our intents in comparing performances and results of different software versions and keeping track of the many changes we made on the code.

◇ 2.6.2 A suite for continuous integration (Jenkins)

Taking into account the complex structure of the code, we decided to use Jenkins to test the code during the development. Jenkins is a continuous integration and continuous delivery application that is able to build and test a software project continuously, making it easier for developers to integrate changes to the project. We required Jenkins to test our code using a reference set of observations: at every git commit, Jenkins was in charge of checking that the results of the new code on the reference test was the same than in the previous version. These tests were executed using different versions of Python (2.7 and 3.3) ensuring that the code was fully compatible with both the versions. In case any problem was detected, Jenkins was able to send an email to the developer and report the observed discrepancy.

◇ 2.6.3 A proficient documentation (Sphinx)

To make the software easy to use for the final users, we provided a complete documentation of its functions. For accomplish this aim, we decided to integrate the documentation with the docstrings that describe the functions and the classes in Python. In order to achieve this result, we used *Sphinx*, a tool that can manage documentation written in rst (ReStructuredText), combine it with the docstrings of the Python modules and produce several kind of outputs, like pdf or html. This allowed us to expose the documentation online through a web server (Apache) installed on purpose and dedicated to this project. Moreover, the server automatically retrieves and updates the source tree of the project and rebuild the documentation at regular intervals (every 10 minutes). This way the documentation can be updated as soon as new modifications are submitted to the source tree, with almost no effort for the programmer (beside properly writing the docstrings).

CHAPTER 3

Improving the processor performances

Here we present the optimization activities performed on the processor with the goal of improving its performance. In this chapter, with the word “processor” we refer just to the “inverter“ and to the “transformer“ subsections of the overall procedure. We focused on these two sections because these are the most intensive from the computational point of view.

The procedure we adopted to optimize the software is based on the closed-loop approach as described in [12]. The basic idea of this methodology is to establish a clear workload (baseline) for the application to be optimize and then run several times the following steps:

1. Run the the application and profile it to identify the most time consuming parts (routines and/or sections of the code)
2. Understand how to resolve the issues and implement the enhancements
3. Test the results and check whether performances improve
4. Restart the cycle to identify the next bottleneck

Clearly, the number of needed iterations is dictated by the improvement one wants to reach.

Before starting it is important to establish a baseline performance on a clear and well identified workload. This workload has to be maintained constant during the whole procedure. In our case we select a set of 358 different observations coming from a IASI granula. This sample is sufficiently big to be considered statistically significant to evaluate the performance of the code. In the next sections, all the results are reported taking in consideration that sample.

As testing machine, we used a multicore architecture made available by eXact-lab, equipped with two Intel Xeon E5-2697 v2 (12 core) working at 2.7 Ghz for a total of 24 cores and 64 GB RAM.

Profiling was done by means of cProfile Python library and for our baseline we identified the three most time consuming sections within the code:

- *Matrix matrix multiplications*: L2VDP performs a lot of matrix multiplications. Even if the dimensions of the matrices are quite small, the operation is performed so many times that it become extremely relevant in the overall execution.
- *Singular value decompositions*: L2VDP performs several SVD during the transformation process (2 for every convergent iteration), which are well-known to be time consuming operations.
- *Forward Model*: The Fortran part of the code. A more accurate description of this part of the code is written in the section 2.1. We were explicitly required to not modify this part of the code in any way.

A graphical representation of the time used by every section is given figure 3.1 which clearly shows that matrix matrix multiplication is the first bottleneck to tackle with.

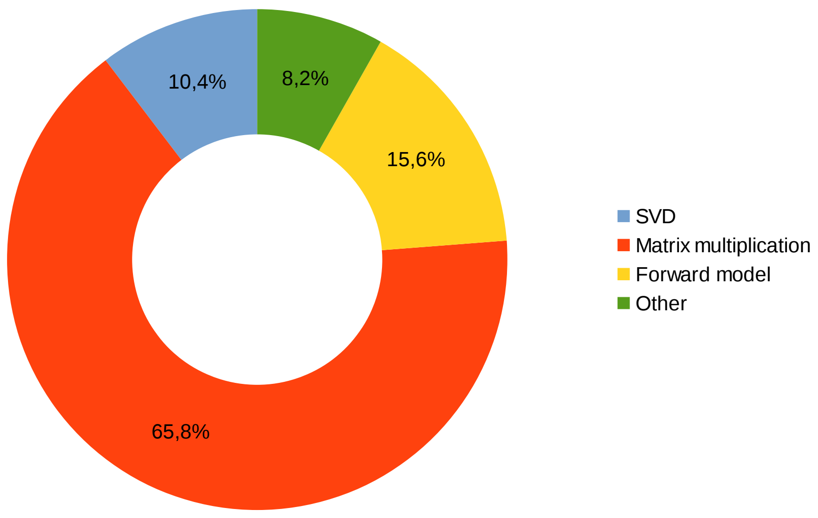


Figure 3.1: The most time-consuming operations in L2VDP

In the following subsection we will discuss in details the seven steps we performed in the optimization procedure and, for each of them,all the techniques, methods and tricks adopted will be described and the improvements evaluated.

3.1 Step 1: matrix-matrix multiplication bottleneck

The first step is focused on matrix-matrix multiplication bottleneck. The table 3.1 shows exactly the matrix multiplications performed by L2VDP and which ones are the most time-consuming. From the same table is also clear that almost all the time spent during the matrix multiplications is absorbed in just two of them.

Matrix dimension	Tot. time	Calls	Avg. time
$(3306, 3306) \times (3306, 3306) \times (3306, 417)$	1263.77	326	3.877
$(417, 3306) \times (3306, 3306)$	1057.88	2323	0.455
$(417, 3306) \times (3306, 417)$	156.45	2649	0.0591
$(3306) \times (3306, 3306) \times (3306)$	25.52	2323	0.0110
$(3306, 417) \times (417, 417)$	21.12	326	0.0647
$(417, 417) \times (417, 3306)$	20.39	326	0.0625
$(417, 3306) \times (3306, 417) \times (417)$	19.46	326	0.0597
$(417, 417) \times (417, 417) \times (417, 417)$	16.13	978	0.0165
$(417, 3306) \times (3306)$	3.07	2323	0.00132
$(1200, 5) \times (5)$	1.80	5264	0.00034
$(417, 417) \times (417)$	1.00	4618	0.00022
$(20, 417) \times (417, 417)$	0.27	326	0.00084
$(417) \times (417)$	0.07	2295	0.00003
$(20, 20) \times (20, 20) \times (20, 417) \times (417)$	0.04	326	0.00011
$(20, 20) \times (20, 417)$	0.03	326	0.00008
$(20, 20) \times (20, 20)$	0.01	326	0.00004
$(20, 417) \times (417)$	0.01	326	0.00003
$(20) \times (20)$	0.01	326	0.00002

Table 3.1: The time spent for each matrix multiplication in the baseline

In particular, the first line shows a multiplication of three matrices performed 326 times, a relatively small numbers if compared with other operations. However the average time of each execution is noticeably greater than any other matrix multiplication. It is worth to perform a brief analysis of the number of the floating point operations that are performed during that operation to better understand the reason of the time spent.
Let A be the first matrix, B the second and C the third, so that the most time consuming matrix matrix multiplication is

$$A \cdot B \cdot C$$

which was performed in the following order.

$$(A \cdot B) \cdot C$$

In a matrix multiplication of a $m \cdot n$ matrix with a $n \cdot l$ matrix is quite obvious that we perform n multiplications and $n - 1$ sums for every entry of the resulting matrix; the total number of floating point operations is therefore $m \cdot (2n - 1) \cdot l$. Consequently, the $A \cdot B$ operation requires

$$3306 \cdot 3306 \cdot (2 \cdot 3306 - 1) \approx 7.2 \cdot 10^{10}$$

This matrix multiplication produce a new $3306 \cdot 3306$ matrix and, therefore, multiply this matrix with C requires

$$3306 \cdot 417 \cdot (2 \cdot 3306 - 1) \approx 9.1 \cdot 10^{9}$$

and the overall operation requires

$$7.2 \cdot 10^{10} + 9.1 \cdot 10^{9} = 8.1 \cdot 10^{10}$$

We decided to take advantage of the associativity of the matrix product:

$$(A \cdot B) \cdot C = A \cdot (B \cdot C)$$

In this way, the number of operations required to compute $B \cdot C$ is still

$$3306 \cdot 417 \cdot (2 \cdot 3306 - 1) \approx 9.1 \cdot 10^{9}$$

but we can now multiply A with a matrix that is 3306 times 417 big and so we spend again $9.1 \cdot 10^{9}$ floating point operations. Totally this operation requires

$$9.1 \cdot 10^{9} + 9.1 \cdot 10^{9} = 1.8 \cdot 10^{10}$$

Compared to the previous one, the number of the floating point operations is reduced by a factor of 4.5.

Starting from this considerations, we implemented a simple reordering in the matrix multiplications. Our above estimate in the number of floating point operations does not take into account either the memory bandwidth or the locality of the data. However we still expect to see an improvement of at least 4 times in this section. Furthermore, we realized that part of the data we computed where not useful for the model: for the transformation, just the first 274 values (instead of 417) were needed. In this way, we reduced the columns of the third matrix from from 417 to 274. After all these changes, the time needed to execute the multiplication of the first line reported in table 3.1 decreased to 210.43 seconds with an overall gain in time of about 6 times. The overall times consumed by the matrix matrix multiplication operation is reduced to 58% of what it was before and the execution of the whole program is 1.35 times faster.

3.2 Step 2: transformation phase improvement

In the second step we mainly improved the transformation part implementing a completely new approach. This improvement was done for scientific reasons and was not intended to improve performance. This however means that some lines of the table 3.1 could be different for this version because of the different approach. For this reason, in the table 3.2 we reported all the new matrix multiplications and we considered these numbers as a new baseline.

Matrix dimension	Tot. time	Calls	Avg. time
$(417, 3306) \times (3306, 3306)$	1082.21	2323	0.466
$(3306, 3306) \times (3306, 3306) \times (3306, 274)$	196.74	307	0.641
$(417, 3306) \times (3306, 417)$	140.09	2323	0.0603
$(3306) \times (3306, 3306) \times (3306)$	30.13	2323	0.0130
$(274, 274) \times (274, 20) \times (20, 20) \times (20, 3306)$	11.11	307	0.0362
$(3306, 274) \times (274, 274)$	9.94	307	0.0324
$(274, 274) \times (274, 274) \times (274, 274)$	4.73	921	0.00514
$(417, 3306) \times (3306)$	3.62	2323	0.00156
$(1200, 5) \times (5)$	1.80	5264	0.00034
$(417, 417) \times (417)$	1.20	4618	0.00026
$(274, 3306) \times (3306, 274) \times (274)$	0.78	307	0.00254
$(20, 20) \times (20, 274) \times (274, 274)$	0.15	307	0.00050
$(20, 20) \times (20, 274) \times (274, 274) \times (274)$	0.07	307	0.00023
$(417) \times (417)$	0.07	2295	0.00003

Table 3.2: The time spent for every matrix multiplication (new baseline) after improvements of step 2

In this step also a small bug was identified and removed. The code was computing (without saving the results) the transformation phase even for some non-convergent inversions. This means that the expensive matrix multiplication optimized in step one was called uselessly several times. We fixed this problem and indeed on our workload only 307 observations converged and therefore require transformation. In table 3.2 our new baseline is reported. The number of times we perform that specific matrix operation is decreased from 326 times to 307, which is the correct number of converged inversions. We note that the bug was just affecting the execution time and not the scientific correctness of the final results.

The total execution time, however, reduced only by a factor that is less than 1%.

3.3 Step 3: matrix structure identification

The table 3.2 indicated clearly that in step 3 we need to optimize the other really time consuming matrix multiplication which is from step 2 onward the most consuming one. This is the multiplication among a $(417, 3306)$ matrix and the inverse of an input matrix A of dimension $(3306, 3306)$. Based on some physical consideration we know that the A matrix could be of three different kinds: a dense matrix, a block banded matrix or, in the best case, a diagonal one. For this reason, we implemented in the software the opportunity to exploit the structure of the matrix in a transparent way for the software. We therefore developed a wrapper around the input matrix which offers some methods like `multiply` or `inverse_multiply`. For every kind of structure (general, banded, diagonal) it is possible to create a subclass of the general wrapper and implement algorithms that exploit the properties of the matrix.

In the current implementation, the software reads the matrix from a netcdf file. In that file there is also a specific metadata field which indicates what kind of structure the input matrix has. In case the metadata reports that the input matrix is diagonal then the inverse matrix is still a diagonal matrix whose diagonal elements are just the inverse of the elements on the diagonal of the input matrix. In that case, the function `multiply` in just a function that multiplies the i-th row of a matrix by the element in the position (i, i) of the input matrix A; the `inverse_multiply`, instead, divides every row by its corresponding element on the diagonal.

More interesting is the case of the block banded matrix. Unfortunately, the banded structure is not preserved while computing the inverse. A much more useful property is to be a block matrix along the diagonal; the reason why this is useful is that this property is also present in the inverse. Therefore it is possible to implement a specific algorithm that discover the block structure of that matrix and of its inverse. To perform a matrix multiplication with the block matrix or with its inverse, we use the following algorithm: in the initialization of the software, we compute the inverse for every block of the matrix and we save them in the memory; when we have to perform a multiplication with a given matrix M, we divide M in vertical bands of the same dimension of the side of the diagonal square blocks of the block matrix. Then we multiply each band by the respective block of A if we are executing `multiply` or by the inverse of the blocks we have computed before if we are performing `inverse_multiply`. Therefore, we put the results side by side into a new matrix which is, exactly, the result of all the multiplication. The algorithm is represented in the figure 3.2.

This improvement reduces the multiplication time by almost a factor of three (from 1082.21 to 376.00) in the table 3.2; the total improvement in the overall code is of about 137%.

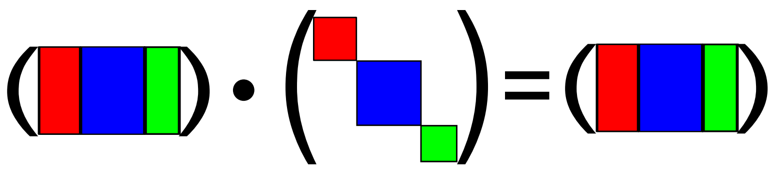


Figure 3.2: The block algorithm we used to improve matrix multiplication. The parts with the same color are multiplied together

3.4 Step 4: exploiting diagonal properties

Step 4 deals again with the matrix multiplication we optimised in step 1 by applying associative property. This operation is still consuming 196.74 seconds (see the second row in table 3.2) and it is worth trying to further improve the performance. An inspection of the code shows that the second matrix is diagonal (because it is obtained from a SVD) and the first one must have the same block structure of the block matrix with discussed about in the section 3.3. So we rewrote the matrix multiplication with the diagonal matrix just as a loop that multiplies rows, and then we reimplemented the same block algorithm implemented in step 3. The overall result is that the time spent by the matrix multiplication goes from 196.74 seconds to 40.13 seconds with a 5x gain. Unfortunately, such a large improvement in this section does not impact similarly on the overall code performance due to the fact that matrix multiplication is no more so dominant compared to the others. In any case, with this changes, the code runs 1.06 times faster than the previous version.

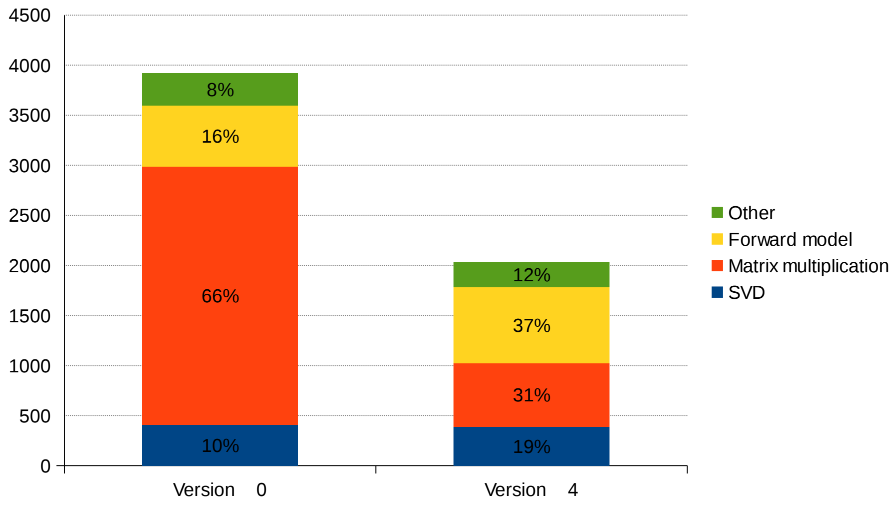


Figure 3.3: Comparison of the time distribution between the original code and the code of step 4

At this point, it is worth looking at how the computation weight moved in the program after all the changes. In picture 3.3, we compare the distribution of computational times of the different sections at the baseline and after four rounds of optimisation. At this stage the forward model is the most time consuming section, matrix multiplication is following and SVD is consuming almost the 20% of all the time. Due to the fact that the Forward model must not be optimised, the target of our next optimisation step is the SVD section.

3.5 Step 5: optimization of SVD algorithm

In order to help us describing how the SVD routines have been optimised within the code we recall here the basic mathematical concept about the Singular Value Decomposition (SVD). Given a (real) matrix M, a singular value decomposition for M are three matrix U, D and V such that

- U is a orthogonal matrix ($U^{-1} = U^T$)
- V is a orthogonal matrix ($V^{-1} = V^T$)
- D is a diagonal matrix with non-negative elements on the diagonal
- $U \cdot D \cdot V = M$

The diagonal entries of D are known as the singular values of M. From now on, we will assume that they are ordered, which means that

$$D_{11} \geq D_{22} \geq \ldots \geq D_{nn}$$

The rows of U are called "right-singular vectors" of M, while the columns of V are called "left-singular vectors" of M.
A truncated (or partial) SVD of order k of M are three matrix $\tilde{U}$, $\tilde{D}$, $\tilde{V}$ such that

- $\tilde{U}$ is made of the first k rows of U
- $\tilde{D}$ is a diagonal matrix with the first k singular values of M as diagonal entries
- $\tilde{V}$ is made of the first k columns of V

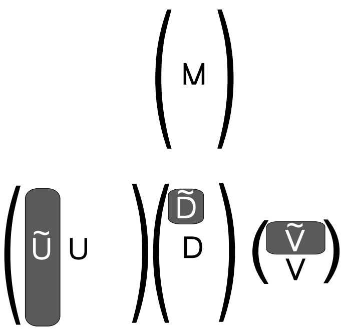


Figure 3.4: The dimension of the SVD of the matrix M

In the code, we perform a singular value decomposition twice for each convergent observation. In particular, the first one we perform is on a square matrix of dimension $(274, 274)$ and the second one is on a matrix of dimension

$(3306, 274)$. The time needed to perform the first SVD is negligible if compared to the time required to perform the second one because of the different dimensions of the matrix. Table 3.3 shows how much time is required for both the operations after the fourth step of the optimisation. .

Matrix dimension	Tot. time	N. of times	Avg. time
$(3306, 274)$	378.25	307	1.232
$(274, 274)$	7.41	307	0.024

Table 3.3: The SVD operation performed in the code

Focusing on the most time consuming SVD operation and analysing carefully the code we note that a complete SVD operation was performed but just the first 20 singular values were used later-on by the code. A possible optimisation strategy is therefore to perform a truncated SVD instead of a complete one by means of some existing and hopefully optimised scientific library. We note however that the code needs both the first 20 left-singular vectors and the first 20 right-singular vectors.

We tested several different libraries to perform the truncated SVD in Python and, at the end, the best library we found was the gensim library which uses a stochastic algorithm as exposed in [10].

Unfortunately, gensim library only returns the left-singular vectors. It is however possible to compute the first 20 right-singular vectors by means of a few more matrix multiplications as described here below.

From the relation

$$U \cdot D \cdot V = M$$

and because of the fact that, being U orthogonal, it is not singular, we have

$$D \cdot V = U^{-1} \cdot M$$

Therefore

$$D \cdot V = U^T \cdot M$$

Let A be the matrix $D \cdot V$ of dimension $(3306, 274)$ (the same of M). For every index i and j, considering that D is diagonal, we have that

$$A_{ij} = D_{ii} \cdot V_{ij}$$

if i is less than 274 (0 otherwise). It is also true that

$$A_{ij} = \sum_{l=1}^{3306} U_{li} M_{lj}$$

Let now consider a truncated decomposition of order k and let suppose that k is so small that

$$D_{ii} \neq 0$$

for every $i \leq k$. For the same indices, the previous equation is still true because it is all contained inside the truncated SVD:

$$\tilde{D}_{ii} \cdot \tilde{V}_{ij} = \sum_{l=1}^{3306} \tilde{U}_{li} M_{lj}$$

and therefore

$$\tilde{V}_{ij} = \frac{\sum_{l=1}^{3306} \tilde{U}_{li} M_{lj}}{\tilde{D}_{ii}}$$

The previous relationship can be rewritten in a matrix format in the following way

$$\tilde{V} = \tilde{U}^T \cdot M \cdot \tilde{D}^{-1}$$

We note that $\tilde{D}$ is a diagonal matrix and therefore computing $\tilde{D}^{-1}$ is not a demanding operation.

We implemented in the code the usage of the gemsim library and the additional matrix operations needed to compute the right singular vectors. Table 3.4 shows the time spent in computing Single value decompositions after the improvement.

Matrix dimension	Tot. time	N. of times	Avg. time
$(3306, 274)$	77.77	307	0.253
$(274, 274)$	7.41	307	0.024

Table 3.4: The SVD operation performed by the code after the improvement

The gain in the SVD section is in the order of five times, and this makes the overall code 1.25 faster than previous version.

3.6 Step 6: SuperLU library

In this step we went back to the matrix multiplication sections and we focused again on the multiplication by the inverse that is shown in the first line of the table 3.2 and that we already improved during step 3. Indeed, we exploited the block structure but we did not exploit the fact that the matrix was banded. Taking into account this property, instead of multiply by the inverse we compute an LU decomposition of the matrix and solve the linear system. This will be particular efficient if we use a special library like SuperLU for Python that can take advantage of the sparse structure (banded matrix) of the original matrix. In this way we can perform the decomposition before starting the computation and then use it to compute the multiplication by the inverse. Of course, we still exploit the block structure of the matrix. This means that we compute one LU factorization for every block of the sparse matrix. The only problem we had to deal with is the fact that the library is only able to find a vector x such that

$$A \cdot x = b$$

where A is the observation error matrix. Working column by column, the library is also able to solve the problem

$$A \cdot X = B$$

where X and B are now matrices. This means that X is $A^{-1}B$ and so we were able to compute the product of the inverse of A with another matrix B without actually compute the inverse pf A. Unfortunately, for our problem we need to compute $B \cdot A^{-1}$ and so we need to solve

$$X \cdot A = B$$

which the library is unable to handle. In order to solve the problem, we exploited the fact that $(AB)^T = B^T A^T$ in the following way. First we compute B^T from B and then we solve the problem

$$A \cdot X^T = B^T$$

to find X^T. In this way, $X^T = A^{-1}B^T$. Because of the fact that $A = A^T$ we have

$$X^T = {A^{-1}}^T B^T = \left(BA^{-1}\right)^T$$

and so $X = A^{-1}B$ which was exactly what we wanted to obtain.

This approach requires two transpositions (one for B and one for X). While the transpositions are usually computationally heavy, due to the fact that these matrices are considerably smaller than A the time spent computing transpositions is not so relevant in the overall computation of X.

We therefore implemented this new approach based on the superLU library. The time required to perform the particular matrix multiplication is reduced by a half. The overall weight of matrix-matrix multiplications is finally reduced of about 18%.

3.7 An overall view

Picture 3.5 shows the improvements along the optimisation steps performed. As a final result of this optimisation effort, the code is more than three times faster than it was in the first version. We also reached the overall goal set at the beginning: reduced the dominant computational load of the linear algebra sections to less than one half of the total time. At the end of our work now the dominant section is the forward model we were asked not to modify in any way. The final time distribution is clearly reported in the

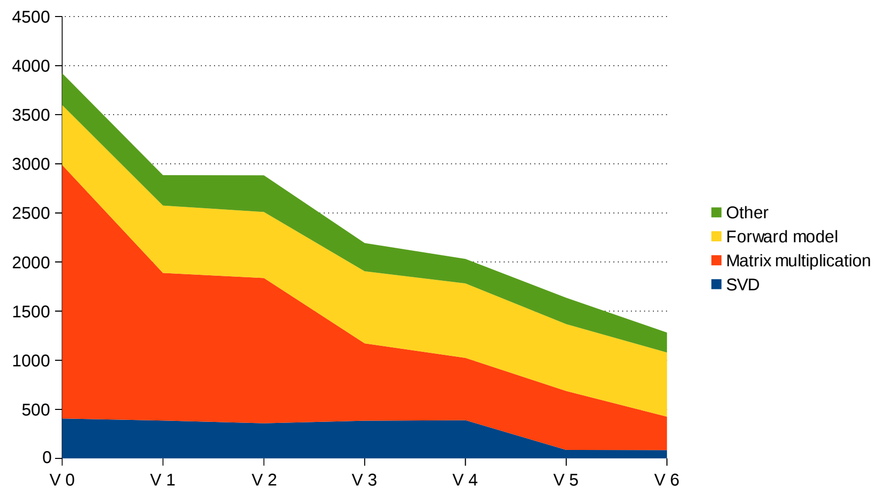


Figure 3.5: The time spent by the different sections trough the different steps

plot 3.6. The overall optimisation procedure was very successful: the code sections we optimised are now running 5.3 times faster. More in details the matrix multiplication section has been improved by a factor of 7.2 while for SVD the improvement is 4.6. We underline that the this activity was conducted in less than two months period time (May/June 2015) with a clear deadline indicated by the customer.

3.8 The diagonal case

In the section 3.3 we described that a particular input matrix could be banded or diagonal. So far, we always described the banded case, as it is the most common one. In particular circumstances, though, the matrix can be diagonal. In this case, the software can exploit some dedicated algorithms increasing even more the speed. Unfortunately, it is difficult to compare the software performance for diagonal and banded matrix, as the underlying physic is different. Furthermore the diagonal case requires more iterations to reach a convergent point. What we can compare is instead the weight distribution of the section in the two codes. From the graphic 3.7 it is clear that in the diagonal case, the Forward Model is even more predominant than in the banded case.

3.9 OpenBLAS vs MKL

In this last subsection we discuss the role of different highly optimised library that can be used by numpy. All the above results has been performed

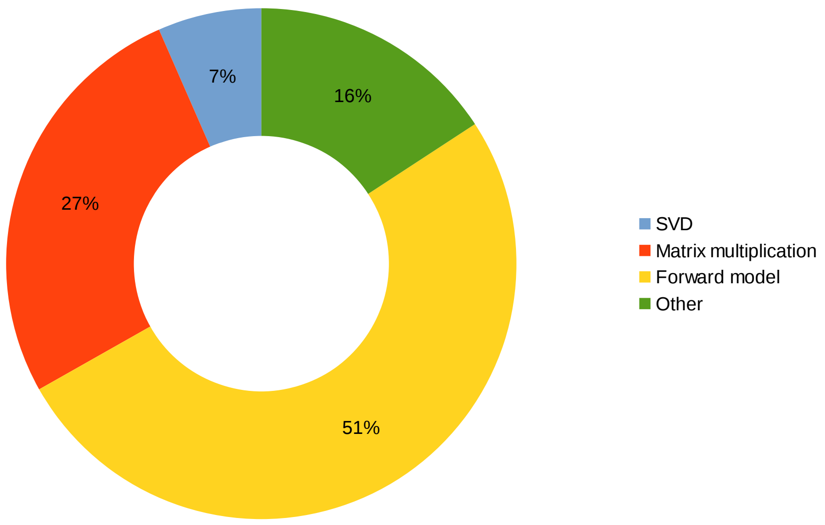


Figure 3.6: The time distribution of the final version

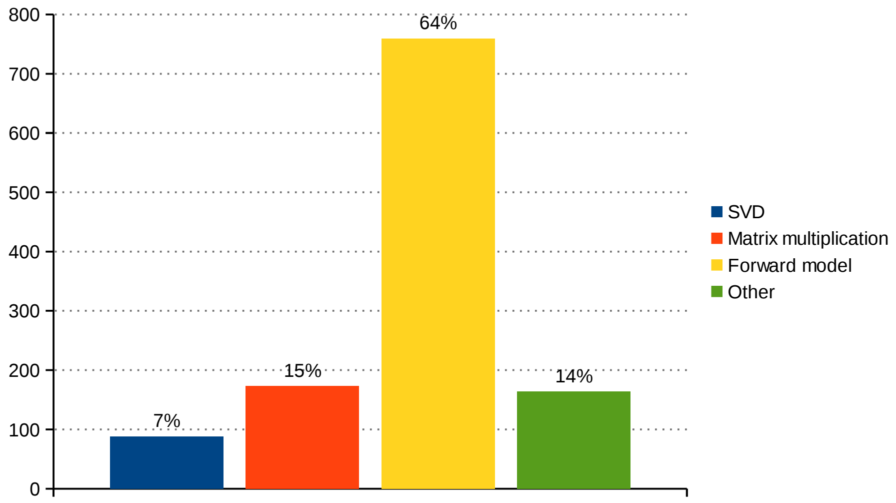


Figure 3.7: The time distribution in the diagonal case

by a numpy library exploiting MKL intel library [23]. It is however worth to show the difference in time using another highly optimised library and freely available: OpenBlas [25]. Picture 3.8 shows the difference in time among the difference steps of the optimisation procedures.

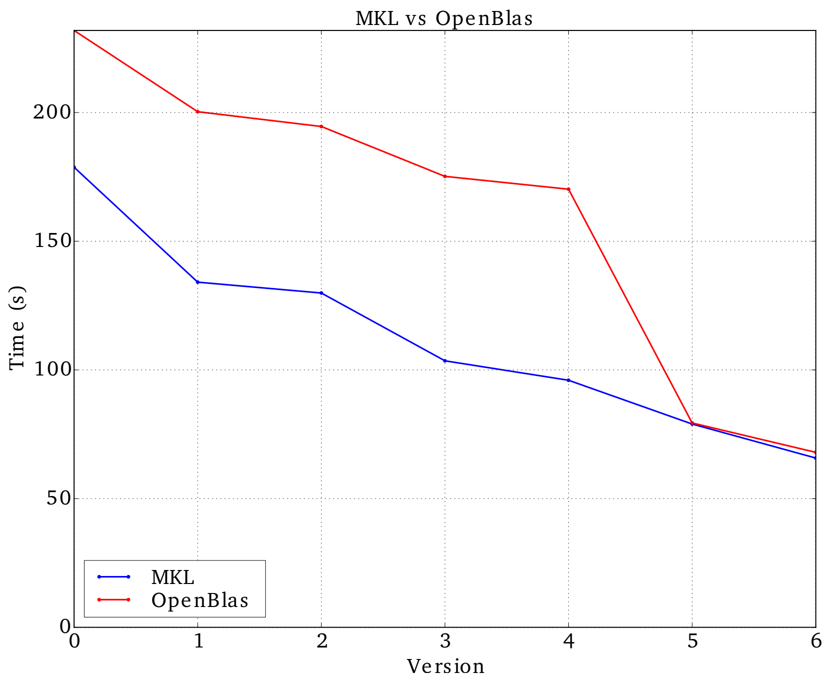


Figure 3.8: The time required using MKL or using OpenBLAS

It is interesting to note how, till step 4, there is a huge gap in the performance among the two libraries. Such gap disappears from step 5 onwards. The explanation for this behaviour is related to the fact that at the beginning of the code development we relied on the MKL libraries mainly for two operations: the matrix multiplication and the SVD. All our optimisation steps try to decrease the impact of these linear algebra operations computed by means of standard algorithms. In particular in step 5 we used a specific library (gensim) to perform the SVD that not even rely on MKL. With respect to matrix multiplication, our optimisation procedures which involve block techniques have the result of performing matrix-matrix multiplications on smaller matrices. This approach favours the usage of OpenBLAS library due to the fact that the gap in performance in the matrix matrix multiplication between OpenBLAS and MKL is proportional to the size of the matrix. We can therefore conclude that, for the current version of the code, there is no good reason to prefer MKL to OpenBLAS.

CHAPTER 4

The computational ecosystem

The goal of this chapter is to illustrate the design of an infrastructure able to process all the data coming from the various satellites in nearly real-time. We refer to such infrastructure with the name "computational ecosystem" to underline the complexity of the overall architecture. As a matter of fact, near real-time processing means that we should be able to process an FDU in less than three minutes without any interruption. To tackle such a challenge it is clear that the L2VDP software should be executed on more than one computational node. The computational system here described takes care to complete this task. An important requirement is to keep the possibility to run L2VDP as a standalone software in a single machine; therefore the ecosystem should be completely transparent to the L2VDP package.

4.1 The overall design

◇ 4.1.1 Features and requirements

The framework has been designed with the following features in mind:

- *Independence*: the components of the framework that are in charge of elaborating data from an instrument should not interfere with the others. In this way, a failure in one instrument is not relevant for the others;

- *Isolation*: the components of the framework should be isolated from the other software of the environment where the framework is running and should not interfere with it;

- *Maintainability*: in case of issues, it should be easy to isolate the problem. Moreover, it should be possible to replace any single component and bring it back to the original working state;
- *Elasticity*: the framework should be able to handle some small changes in the underling environment (like adding or removing a node) with just some minor effort and without interrupting the execution;
- *Easy to deploy*: the framework should require a minimum amount of requirements to be installed. The installation procedure should be similar in every environment and should be as much automated as possible.

Beside these above features, it has to be noted that it was known from the beginning that the deployment on EUMETSAT TCE had to be done as user (i.e. without having root privileges).

◇ 4.1.2 Docker

All the above requirements motivated us to implement the ecosystem by using Docker [17], an open platform for distributed applications which offers several advantages.

To better understand why Docker is a efficient solution, it is worthy to spend a few words about how Docker works. The main idea is to completely isolate a selected process from the system where this process is running. To achieve this result, Docker takes advantage of a technology called namespaces. In this way, Docker provides isolation from the network, from the other processes, from the filesystem and even from the Unix Timesharing System. This means, for example, that the isolated process has not access to the disk beside a specific dedicated part. Inside the docker, the process can spawn others processes that will share with it the environment and that will still be isolated from the processes that are not running inside that docker.

Being completely isolated, a process inside a docker has no access to the libraries and utilities that are installed on the machine. For that reason, it is common to replicate the installation of a minimal system in the dedicated space that the process has access to. Such a minimal collection of libraries and tools that the isolated process will find on its disk is called *image*. A this point it could be tempting to think a docker as some sort of virtual machine: we prepare an image with an operating system and start it such a way that it is isolated from the system. In fact, the analogy ends here. While a virtual machine is a process that simulates other processes, in docker the isolated process is really running on the bare metal. When a process that is running inside a virtual machine attempts a system call, it submits a request to the kernel of the virtual machines. The supervisor, which is in charge of run the virtual machines, will therefore translate the request for the real machine and, if necessary, the request will be forwarded to the real kernel machine. Instead, a process that runs inside a docker can submit a system call directly to the kernel machine. Because of the absence of a supervisor, the docker processes have almost no overhead compared to the native processes and this is one of the main reason why we prefer docker instead of a virtual machine.

Another advantage is related with the time needed to start our ecosystem. If we had used virtual machines, we would not have been able to avoid the time required to start them. With docker, this is not even an issue. Indeed, while the docker images usually contains an entire operating system, this operating system is there just in case the isolated processes need one of its files; no kernel should be loaded, no services should be initialised and so on. While this dramatically reduce the time required to start the overall ecosystem, it also means that we remove all the redundant system service that a virtual operating system needs. In docker, we run just the process that we really need.

What instead is shared with a virtual machine approach is a completely deterministic environment. Though we depend on the kernel of the machine where docker is running, we do not relay on any library or software in the userspace. For this reason, our system it is easy to deploy and does not require a lot of dependencies to be installed.

To orchestrate all the docker infrastructure, we used Dockerops [18]. We rely on this software for starting and stopping the dockers in a consistent way and managing the network links between different dockers.

◇ 4.1.3 Slurm

Once the dockers have been deployed, we have to face the problem of handling the distribution of the workload. Being the algorithm embarrassingly parallel and considering that the computational effort to compute one single granula is quite relevant, a master/slave architecture can respond efficiently to the needs of out project. Indeed, the elaboration of each granula on a slave does not imply any communication with the master (beside the initial one to start the computation and assign the granula) and, therefore, it is mostly unlikely that the master would not be able to manage smoothly the whole system because of the excessive workload (at least for a rather small number of nodes). Of course, to keep the elaboration of the data generated by each instrument independent to the others, it will be necessary to deploy three different masters, each one with its own set of slaves.

In order to allow the master of distributing the granula computations for the slaves, it would have been extremely inefficient to implement by ourself a small queue system because of the time required to write, debug and test such a complex set-up. Our key idea is to leverage on a robust queue system package to distribute the computations on multiple nodes. We identified slurm [26] as the right tool capable of handling multiple jobs and offering moreover detailed logging information about the status of the resources. The computational nodes can also be added without too much burden and therefore elasticity requirement is also satisfied.

◇ 4.1.4 The overall ecosystem

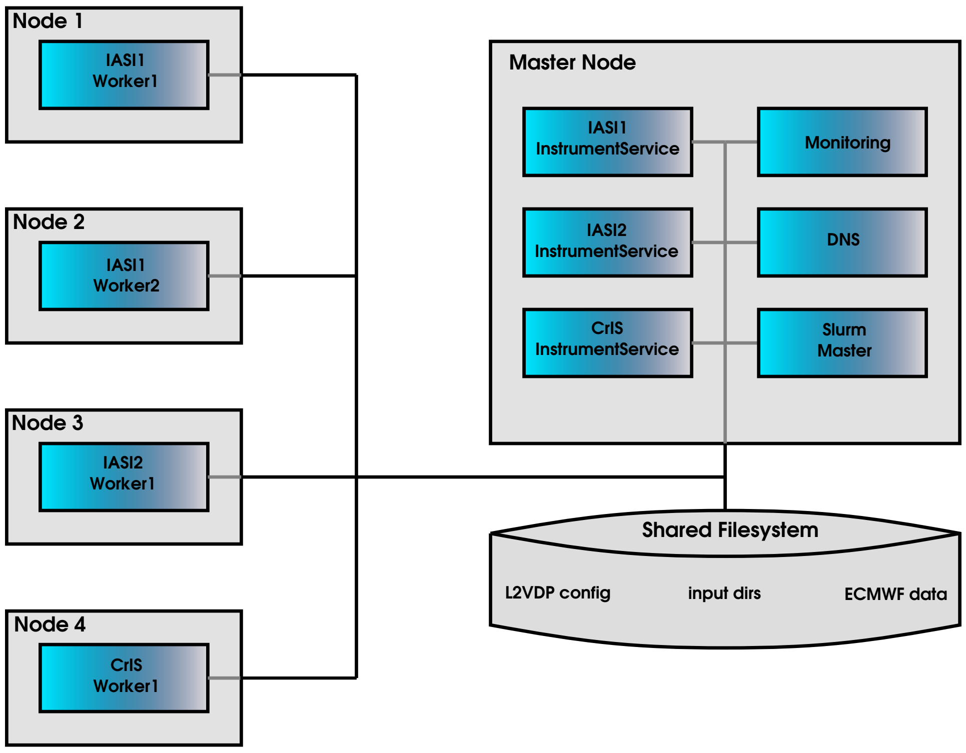


Figure 4.1: A possible architecture for the L2VDP framework

Figure 4.1 gives a graphical view how the ecosystem L2VPD is designed and then deployed on a cluster. The whole infrastructure is composed by several services (each of one managed by a different docker). On one node (identified as the master) there are 6 different services (dockers) that are not computationally intensive. The central core of the infrastructure is the slurm-master service which manages all the intensive computational tasks (i.e. various instances of the L2VDP processor described before, each of one acting on subsequent FDU arriving) and dispatch them on worker services. Interaction among such service is done by means of slurm clients which are installed on the dockers where L2VDP shall run (the workers). Three different services (one for each of three different instruments) are taking care of processing the flux of data coming by the instruments and send computational requests to the slurm-master. Every times the instrument-services want to submit a job in their partition of the cluster, they send the request to the slurm-master which will delegate the job to one of the nodes as soon as a free one is found. A monitoring services is in charge of collecting logs from the other dockers and to analyse them to generate some statistics about the usage of the nodes and the executions. Finally an internal DNS service is also provided to allow the dockers to recognize themselves and their companions as “standalone” machine with their own IP addresses and names. This, as well, facilitates both the deployment and the installation.

Slurm queue system is configured with three different partitions (again one for each of the three instruments) and each worker docker is assigned to one of them. We remark here that each worker docker is hosted on its own separate node of the cluster.

The master control docker provides a simple (in this present release) control panel which provides a managing interface for the whole the framework allowing the users to start and stop the instrument services.

As clearly indicated by the picture 4.1, we expect to have a shared filesystem among the nodes of the cluster. Such a shared file system allows an unique location for all the files needed by L2VDP package running on the worker dockers: the emissivity files, the ECMWF files, the satellite native files and so on. This constraint could also be removed, but at the moment a shared filesystem offers a robust and simple solution.

◇ 4.1.5 The logging system

Such a complex architecture requires a sophisticated logging system to provide users with a complete understanding of what is happening inside all the components. We therefore developed a logging mechanism where every component of the infrastructure logs (in a very specific file) a set of information. Logs file are written (using the Python logging library) in a way that makes them easy to be parsed with some regular expressions. In particular, every line of the logs reports the name of the instrument that the granula is related to, the current date in UTC, the name of the Python module that generated that line and the severity level of that line. If the line is related to a particular granula (which is true for all the logs beside the ones that were generated by the instrument services), also the name of that granula is reported. We also tag (by means of a random tag) any single run of the instrument service to be able to recognize, for every granula, which instrument service submitted the job that elaborated it. This tag is reported in every line of the instrument service log. Moreover the tag is also sent to every worker trough the job: indeed, the script that slurm submits sets a particular environment variable with the content of the tag. When the worker starts, it checks for this variable and, if present, it reports it in every line of the logs.

As a side effect, however, log entries are extremely long and difficult to read for a human being. It is however simple to overcome this limitation by writing a small script able to translate the long and cumbersome message in a human readable format for instance reporting only selected information related to particular granula and/or filtering only specific errors. So, with a minimum effort, it is possible to achieve a great readability compared to the standard logs. One of the main advantage in writing the logs file this way will be exposed in the section 4.4.

In the following we will discuss more in detail all the services.

4.2 The instrument service

The instrument service is the starting point of all the process that, starting from a native file, produce the desired output. This docker is a machine with slurm installed (to be able to submit the job) that runs a script called `instrument_service.py`. The instrument service monitors a directory (usually the one where the input data from the satellite will be saved) and it can run in two different modes: the operative one and the reprocessing one. The former is the standard way while the latter is used in case some data have to be processed again not in real time.

◇ 4.2.1 operative mode

In operative mode, the `instrument_service.py` runs as a demon, continuously checking if new FDU files are stored in a specific directory (that could be specified in the overall configuration file). Once a new file appears, the instrument service checks whether the file is a correct FDU file: such a check is performed examining its filename. If this is the case, the instrument service then checks if any theoretical data are available to be used as "first guess".

Sometimes, it is possible that some data could not be elaborated simply because there is no forecast close enough to the date of the FDU we are processing. The instrument service, in that case, will discard that granula and will not submit any job. If a suitable forecast is found, instead, then the instrument service asks the slurm system whether there are free resources to run the calculation: in case of negative answer, no job is submitted and the FDU is simply discarded. The motivation behind this choice is that, in operative mode, we do not want to create a queue because this will generate a delay in the delivery of the final results of all the system. It is therefore more advisable to lose some granula but keep the delivery of the data in near real-time. In case of positive answer from slurm (i.e. free resources are available) the job is submitted to the system and no other actions will be performed by the instrument service on that granula.

◇ 4.2.2 The reprocessing mode

The reprocessing mode is an alternative way of working for the instrument service.

It can be selected in the configuration file and allows a user to reprocess her or his data, for instance to use a different forecast or to test some new physical parameters in the model with respect to the first execution. In this configuration, the instrument service will not act as a demon but will simply submit a job for every file available in a selected directory when the program starts. Moreover, in this modality the jobs will be always submitted even if there are no free nodes available. Finally, the instrument service is also more strict requiring a forecast close to the data acquisition time (in the operative mode, it would discard a granula if there is no forecast in a range of 24 hours, in reprocessing mode the time is reduced to 6 hours). Indeed, we assume that, if a FDU is not requested to be elaborated in real-time, there is no reason for which the forecast could be missing, hence a 6 hours time is the maximum difference in time between two subsequent ECMWF forecasts.

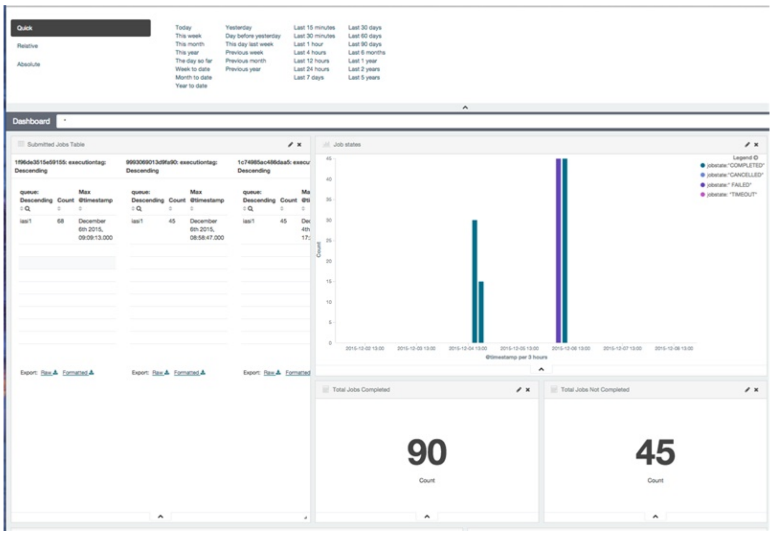
4.3 The worker

This service is hosted within the docker of the L2VDP processor. The installed slurm client is able to receive the job submitted by the slurm-master. The execution of the L2VDP processor is actually performed by a bash script generated by the instrument service. The command line to launch the processor is exactly the same as in case of a standalone run. This method is simple but, at the same time, also very efficient because it does not require to develop a particular interface for the worker.

4.4 The monitoring system

This service provides the users with a set of tools that allow a complete and detailed monitoring of the ecosystem. Moreover a users can also perform complex queries and extracts many different kind of information. The monitoring system automatically collects all the logs from the various services using *Fluentd* [20], an open source data collector. Fluentd injects all the logs into an Elasticsearch server installed in the monitoring docker. *Elasticsearch* [19] is an open source text search engine based on *Lucene.* To perform these operations, Fluentd applies some regexp on the log lines to identify the fields we discussed in section 4.1.5.

The last step is the visualization of the data. For this purpose we installed *Kibana* [22], a tool well integrated with Elasticsearch that allows to produce graphics based on the data we extracted from the logs. Kibana exposes all these results through a web interface, giving to the users a nice and comfortable way to explore what happens inside the system. This solution works on two levels: for the occasional user, Kibana exposes some common graphics with some general information. An advanced user, instead, can take advantage of all the power of Elasticsearch to build new graphics with her/his own queries. Our Kibana installation has a customized dashboard which shows all the important parameters of the system. A snapshot of this dashboard is given in figure 4.2.



1f96de3515e59155: executiontag: Descending		
queue: Descending	Count	Max @timestamp
iasi1	68	December 6th 2015, 09:09:13.000

9993069013d9fa90: executiontag: Descending		
queue: Descending	Count	Max @timestamp
iasi1	45	December 6th 2015, 08:58:47.000

1c74985ac486daa5: execu		
queue: Descending	Count	Ma @ti
iasi1	45	Dec 4th 17:

Figure 4.2: The Kibana framework

4.5 Benchmarking the infrastructure

The infrastructure described above in details has been deployed on some virtual computational nodes of the EUMETSAT Technical Computing Environment (TCE). All central nodes have been deployed on tcsx142, a rather old multicore machine but well-equipped in term of RAM (72 GB). This was deemed more than enough to manage for the not so intensive computational task assigned to the central services. Workers are instead deployed on two different kinds of nodes:

1. 6 nodes with 8 cores each
2. 3 nodes with 24 cores each

The two set of workers have been assigned to the two different instruments (iasiA and iasiB) creating thus two different pools of resources that we benchmarked to assess the amount of computing resources needed to process in near real-time the data. Several benchmarking runs have been conducted, simulating the Near Real Time processing. We setup a script that copies every three minutes a IASI-A and a IASI-B file from their standard location on TCE on some local directories monitored by an instrument service on the scratch filesystem and we let both system runs for exactly 45 hours. Table 4.1 shows some of the statistics Kibana provided us.

observables	IASI-A	IASI-B
granula arrived	877	873
granula job submitted	876	836
granula job discarded	1	37
average processing computing time for granula	311	186
max processing computing time for granula	1702	976
min processing computing time for granula	31	37
average pre-processing computing time for granula	102	102
max pre-processing computing time for granula	1333	144
min pre-processing computing time for granula	83	85

Table 4.1: Performance of IASI-A vs IASI-B system

From the table it is evident that the overall efficiency of the IASI-A subsystem is much better than IASI-B. This is due to the fact that, at the global speed of the end-to-end processor three nodes are not enough to cope with the pace of the incoming file, even if these nodes are three time larger in term of computing power. The first conclusion is that, for the time being it much more convenient to have a larger number of less powerful nodes than the opposite situation.

Finally, let us conclude showing how both the configuration where able to elaborate the data in near real-time. Indeed, the amount of data discarded because of lack of resources was about the 4% of the total for IASI-B and less than 2‰ for the IASI-A. Moreover, in all our test the system has proved to be stable and capable to run for different days even without any human supervision. We can therefore claim that our architecture fulfils all the aims we set at the beginning of the development.

CHAPTER 5

Conclusions

In this chapter we want to briefly summarize the work performed in this thesis and the main results achieved. We present also a final analysis of the current status of the ecosystem we built with the aim to identify and propose future directions of development and improvements.

The main achievement of the work was the successful deployment of a complex computational environment where near real-time data processing can be performed. To reach the above result several intermediate phases of work have been completed. The first phase deals with a software re-engineering activity to merge into one unique procedure the different and variegate tools needed to process the row data. As final result we obtained a Python package with the following features:

- A complete integration of all the different phases of processing has been performed using modern software engineering tools. The whole processing package is now easy to use and a IASI raw data file can be processed by just one single command line.
- All the configuration parameters are grouped in a single readable file and all the output is well organised in netcdf format and in well defined directory paths.
- A modular approach has been chosen to make easy in the future adding or changing any component, allowing to easily adapt the software to the future needs.

We can easily state that usability and productivity of the software is highly improved, even if several weaknesses are still present as we will discuss later.

The second phase dealt with an optimisation procedure on the two most time consuming sections of the processing software: the inversion and the integration. An excellent result has been achieved in the limited amount of time at disposal to complete such complex task. The optimised section of the code now is three time faster than before and just considering the sections of code where optimisation procedure and tricks were implemented (i.e. excluding the Forward Model section that was explicitly requested not to be optimised) the gain obtained is more than five times. This was achieved thanks to a detailed profiling of all the matrix-matrix operations performed followed by a deep analysis of the structure of the matrices involved. This approach allowed us to implement some dedicated algorithms that exploit the properties (sparse, banded/diagonal) of those matrices. Some smart algorithms were also applied to compute SVD operations to reduce its computational cost. Again this was possible once it was clearly understood how the results of this operation are used in the rest of the code. We remark here that our improvement rely mainly on better and more performing algorithm that suit better the data structures within the code. No attempts were done in other directions like for instance implementing some algorithms that exploit the computational power of the accelerators, like GPU and/or intel PHI. Another opportunity is to use software libraries like cython[16] to compile the code and therefore gain some extra optimisation.

In the third phase we design and developed the overall ecosystem that wrapping around the processing package allows to process in near real-time a large amount of data. The ecosystem, largely benchmarked, has been proved to be stable and efficient. One successful idea was to rely on a workload manager like Slurm. This allows us to build in an easy and efficient way a system which inherits the stability and the efficiency of Slurm. Moreover, Docker has allowed us to build an ecosystem which is fully deterministic, easy to deploy and elastic, without having to deal with the overhead of the virtual machines. The ecosystem is so versatile that it can even deploy and run on a single machine, if this is needed for testing purpose (for example, during the development) granting a great level of abstraction between the virtual ecosystem and the real machines without computational cost.

As anticipated above there are still some weakness that should be addressed in the future. We discuss all of them now with the intent to suggest future lines of development.

- Being forced to work within severe deadlines, some sub-optimal technical choices was done mainly to keep developing time short enough. As specific examples we mention here the decision to simply embed the bgerr software in the preprocessor which is a complete serial code.

- Another severe limitation comes by the fact that embedding a lot of software, written in different languages, make the code difficult to understand and to debug. If an embedded software raises an exception, it is impossible for the main software to intervene and, therefore, this decrease the robustness of the system.

- Related to the number of embedded software, there is the problem of the excessive I/O: the embedded software needs to read its input data from the file system: so we are forced to write every input on the disk and to read the output from the same position. While this does not slow down the execution of the preprocessor in a sensible way, it creates a lot of useless files that waste disk space and that requires a periodical cleaning.

All the above points can be addressed in future phase of developments.

Acknowledgement

The activities here presented were supported by the European Organization for the Exploitation of Meteorological Satellites (EUMETSAT) under contract number EUM/CO/14/4600001349/SAT. Within such contract the computational activities were performed by eXact-lab srl, a start-up company which I would like to express my sincere gratitude to for having sponsored my participation to the master but also for all the opportunities of professional growth they have gave me. In particular, I would like to thank my supervisor Dr. Stefano Cozzini for his guidance and motivation.

Beside my supervisor, my sincere thanks also goes to dr. Paolo Antonelli for his patience, knowledge and availability and for all the help he gave me during this project.

I thank my master mates for all the stimulating discussions, for the sleepless nights we were working together before deadlines chatting on Skype and for all the fun we have had in this year. In particular, I would like to thank Moreno for his invaluable help and for all the knowledge he shared with me during this master.

I would also thank my family and my parents for supporting me throughout writing this thesis and my life in general.

Last but not the least, I would like to thank Irene for so many reasons that no thesis could ever contain all of them but, especially, for being there for me during this year.

Bibliography

[1] P. Antonelli, T. Cherubini, T. Auligne, L. Bernardini, S. Businger, and F. Marzano. *The potential of meteosat third generation (mtg) infrared sounder (irs) level 2 product assimilation in a very short range numerical weather fore- cast model* Final report, EUMETSAT, 2015

[2] D. Blumstein, G. Chalon, T. Carlier, C. Buil, P. Hebert, T. Maciaszek, G. Ponce, T. Phulpin, B. Tournier, D. Simeoni, P. Astruc, A. Clauss, G. Kayal, and R. Jegou. *Iasi instrument: Technical overview and measured performances.* Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, 5543, 2004.

[3] S. de Haan, G.-J. Marseille, and P. De Valk. *The potential of meteosat third generation (mtg) infrared sounder (irs) level 2 product assimilation in a very short range numerical weather forecast model.* Final report, EUMETSAT, 2015.

[4] Yong Han, Henry Revercomb, Mike Cromp, Degui Gu, David Johnson, Daniel Mooney, Deron Scott, Larrabee Strow, Gail Bingham, Lori Borg, Yong Chen, Daniel DeSlover, Mark Esplin, Denise Hagan, Xin Jin, Robert Knuteson, Howard Motteler, Joe Predina, Lawrence Suwinski, Joe Taylor, David Tobin, Denis Tremblay, Chunming Wang, Lihong Wang, Likun Wang, and Vladimir Zavyalov. *Suomi npp cris measurements, sensor data record algorithm, calibra- tion and validation activities, and record data quality* Journal of Geophysical Research: Atmospheres, 118(22), 2013JD020344;

[5] Jean-Luc Moncet, Gennady Uymin, Alan E. Lipton, and Hilary E. Snell. *Infrared radiance modeling by optimal spectral sampling* Journal of the Atmospheric Sciences, 65(12), 2014/02/04 2008;

[6] P. Antonelli, R. Knuteson, H. Revercomb, R. Garcia, E. Borbas, S. Bedka, D. Tobin, J. Taylor, W. Smith, *UWPHYSRET an SSEC inversion package for high resolution infrared data based on LBLRTM*;

[7] P.Antonelli, *Study of a high performance IRS level2 validation and demonstration prototype: Final Report* , 2014;

[8] M. G. Blasi, C. Serio, G. Masiello, S. Venafra, G. Liuzzi, *SEVIRI Cloud mask by Cumulative Discriminant Analysis*, Journal of Physics: Conference Series, Volume 633, conference 1;

[9] D. Coppens, R. Meyer, D. Klaes, F. Montagner, *IASI Level 1: Product Guide*, http://www.eumetsat.int/website/wcm/idc/idcplg?IdcService=GET_FILE&dDocName=pdf_iasi_level_1_prod_guide&RevisionSelectionMethod=LatestReleased&Rendition=Web;

[10] N. Halko, P. G. Martinsson, J. A. Tropp, *Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions*, SIAM Review, Vol. 53, No. 2, May 2011;

[11] C. D. Rodgers, *Inverse Methods for Atmospheric Sounding: Theory and Practice*, World Scientific, 2000;

[12] A. Supalov, A. Semin, M. Klemm, C. Dahnken, *Optimising HPC Applications with Intel Cluster Tools*, Apress Open, 2014;

[13] EUMETSAT *Algorithm Theoretical Basis Document for Level 2 Processing of the MTG Infra-Red Sounder Data*, Technical Report, EUMETSAT, 2014.
http://www.eumetsat.int/website/wcm/idc/idcplg?IdcService=GET_FILE&dDocName=pdf_atbd_pro_infrared_sounder&RevisionSelectionMethod=LatestReleased&Rendition=Web;

[14] *CProfile*, https://docs.python.org/2/library/profile;

[15] *Ctypes*, http://docs.python.org/2/library/ctypes;

[16] *Cython*, http://cython.org;

[17] *Docker*, www.docker.com;

[18] *DockerOps*, https://github.com/sarusso/DockerOps;

[19] *Elasticsearch*, https://www.elastic.co/products/elasticsearch;

[20] *Fluentd*, http://www.fluentd.org;

[21] *IASI Level 2: Product Guide* ,
http://www.eumetsat.int/website/wcm/idc/idcplg?IdcService=
GET_FILE&dDocName=PDF_IASI_LEVEL_2_PROD_GUIDE&
RevisionSelectionMethod=LatestReleased&Rendition=Web;

[22] *Kibana*, https://www.elastic.co/products/kibana;

[23] *Intel Math Kernel Library (MKL)*,
https://software.intel.com/en-us/intel-mkl;

[24] *Multiprocessing*, https://docs.python.org/2/library/multiprocessing;

[25] *OpenBLAS*, http://www.openblas.net;

[26] *Simple Linux Utility for Resource Management (SLURM)*,
http://slurm.schedmd.com;