Deployment of a HPC operational service

Quim Aguado Puig

Resum– La cendra volcànica en suspensió crea greus problemes de seguretat i econòmics al sector aeronàutic. Les erupcions volcàniques que hi ha hagut fins ara han demostrat que és difícil aconseguir prediccions precises dels núvols de cendra. En aquest treball s'ha dissenyat i implementat un component per controlar processos HPC (en aquest cas, simulacions de dispersió de cendra volcànica), en un sistema operacional en temps real.

Paraules clau– HPC, Sistema Operacional, Cendra volcànica, Perills atmosfèrics

Abstract– Atmospheric dispersion of volcanic ash creates important economic and safety problems for the aviation industry. Past volcanic events have shown that is difficult to access precise forecasts of volcanic ash clouds. This project shows the design and implementation of a component for controlling remote HPC jobs (in this case, volcanic ash dispersion simulations), in a real-time operational system.

Keywords– HPC, Operational Service, Volcanic Ash, Atmospheric hazards

1 Introduction

Atmospheric hazards have caused problems to the aviation sector for years,[1] one of the main ones are volcanic ash clouds at low and jet-cruise atmospheric levels. High precision forecasts of volcanic ash will help airlines to save money, give better services to their costumers, and provide safer flights. A famous case in Europe was in 2010, when the Eyjafjallajökull volcano erupted in Iceland, the airspace of most eropean countries was partially or totally closed, causing loses of more than 130M EUR per day.[2]

1.1 Context

This project have been developed in Barcelona Supercomputing Center (BSC-CNS), in the Mitiga group. Mitiga is now a spin-off of BSC, it develops and comercialize solutions to evaluate and manage volcanic hazards.[3] At the inital point there were three main components developed.

- **Operational service**: A service running 24/7, capable of automatically getting relevant data from volcanic eruptions, and managing certain events for the data.
- **Simulation model**: An implementation of a model to forecast volcanic ash clouds.[4]
- **Impact calculator**: Responsible to get the affected flight routes for a certain ash cloud.

Simulations could be launched manually. The next step in the operational was to be able to launch the simulations automatically.

The simulations are launched at Marenostrum, the BSC supercomputer, so the operational needed some component to manage the simulations that were executed remotely. That component was the part developed in this project.

1.2 Objective

The aim of this project is to design, develop and test the component of the operational service responsible to manage HPC jobs in remote clusters (in this case, it'll be used to run simulations of volcanic ash dispersion in the air).

The programming language used is Python,[5] in its version *3.x*. The main reason for using this language is that the operational service is programmed in Python, so it'll be easier to integrate.

This operational component will be called *Forecast*, and it'll be composed of multiple processes.

Forecast receive as an input the necessary data to run the simulations (meteorologycal information, volcanic eruption data...) and it returns the result of the simulation (where the volcanic ash will be in the future).

Error handling is also a really important part of *Forecast*, as the operational will be giving data to real clients that operates real flights. The internal libraries used by the operational already includes proper error handling.

• E-mail de contacte: quim.aguado@e-campus.uab.cat
• Menció realitzada: Enginyeria de Computadors
• Treball tutoritzat per: Miquel Àngel Senar Rosell (CAOS)
• Curs 2018/19

1.3 The operational service

The operational service is composed of different processes, and groups of processes organized hierarchically. The communication is done via messages and pipes, and a process can just send messages to its parents and child, so there's no horizontal process communication (Fig. 1).

Some of the more important processes are:

- **Oracle**: Main process. It's on the top of the hierarchy tree, most of information in the system is controlled by Oracle.
- **Advisory**: The process that gets information about volcanic ash.
- **Forecast**: The process that executes volcanic ash cloud forecasts, it has subprocesses that do specific tasks, like managing remote HPC resources or controlling a group of simulations for an eruption.

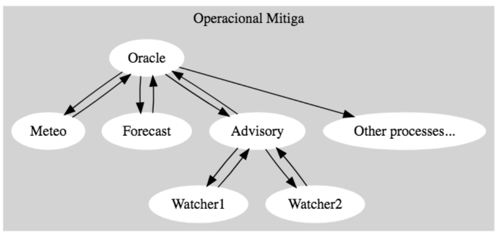

Fig. 1: Operational service

There's an internal library already developed, that allows to create processes, manage messages and exceptions in an easy way.

2 State of the Art

Nowadays, there's no standard way to programmatically control HPC jobs in remote clusters.

There's a Python library called *Fyrd* that controls *slurm* or *torque* clusters.[6] It does not completely solve the problem as it can't manage remote clusters, it can just control local instances. It works and it's easy to add new queue systems, but some parts are not object oriented, and it doesn't always use best coding practices.

For making connections to remote machines, the most secure, easy and standard way to proceed is SSH. For traffic redirection *SSH tunnels* can be used,[7] and to mount remote file systems, a FUSE file system can be used through SSH.[8]

3 Methodology

This project has been developed using the Agile methodology. There have been two main stages, design and development of the *Forecast* component and the related libraries and tools.

In the design stage, it was defined how *Forecast* must be constructed, which libraries to use, how it has to interact with the rest of the operational processes and what simulation parameters to use for the model.

In the development stage, *Forecast* was developed with the Mitiga team, it was also important to do proper tests of all processes while developing them to avoid future errors.

Additionally, weekly reports were created by each member of the team to track our progress.

4 Development

4.1 Design

The first part of the design consisted of creating the process structure for *Forecast*. Three types of processes were designed (Fig. 2):

- **Forecast**: It is the main process, is the one that communicates with the rest of the operational system and controls the rest of the processes.
- **Ensemble**: It's a superclass (there can be derived specialized processes of it). It controls the simulations of a specific event, like a volcanic eruption.
- **JobTask**: It's also a superclass. It controls a specific HPC job (in this case, a simulation run) in the remote resources.

A detailed explanation of each process type is given further in this document (See sections 4.1.1, 4.1.2 and 4.1.3).

All the design was documented using Sphinx,[9] during the programming stage, the code was also documented following the Sphinx standards, so the documentation can be autogenerated.

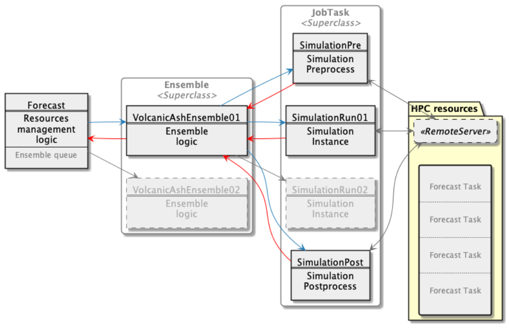

Fig. 2: *Forecast* processes high level view

There were multiple non-trivial problems to be solved, those were just exposed because they weren't essential to start the development of the *Forecast* component, and they are not supposed to be solved in this project.

- **Uncertainty**: How uncertainty is measured and propagated into the system.
- **Computational domain size**: What domain size to use for a simulation of a volcanic eruption. It has to be big enough to be able to contain the whole ash cloud, but small enough to be able to run the simulation in a decent resolution in the minimum time. In section 4.2.3 is explained how this have been implemented, and in section 7.2 a better implementation is discussed.
- **Merging and post-processing**: How to merge the output of the different simulations of an eruption.

- **Priority**: How to assign priority to the simulations, as there are some eruptions more important than others.
- **Ensemble size**: How many simulations (and with which parameters) must be launched in each ensemble.
- **Deal with new information**: How to manage when new information arrives, but a previous simulation for the same eruption have not finished.

4.1.1 Forecast process

The forecast process is responsible to launch ensembles when it's required by the operational. It also controls available HPC resources.

It manages the different ensembles running, keeping track of available and allocated resources. If it's necessary, it can kill an ensemble at any time to get the allocated resources back.

It's not in the scope of this project to develop the *Forecast* process, other people in the Mitiga group are responsible for this task.

4.1.2 Ensemble process

Ensemble is a process superclass, specific subclasses can be created for different simulation models.

The Ensemble is responsible to launch all the necessary tasks and ensure that they are run in the correct order, maintaining dependency tracking. A typical task graph of an ensemble process can be seen in Fig. 3.

Usually, ensemble tasks will be:

- **Pre-process**: Get the input data, and modify it to be meaningful for the model.
- **One or multiple runs**: Create one or multiple tasks to run the simulation with different input values.
- **Post-process**: Merge the outputs of the different runs in a single one.

For simplicity, simulations can't be paused, just stopped. Simulations don't take a long time to run (in the order of minutes), so it's not necessary to implement a complex system to be able to pause and continue simulations because the gain would be negligible.

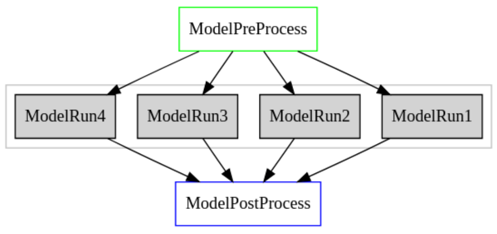

Fig. 3: Typical tasks DAG of an Ensemble process

4.1.3 JobTask process

JobTask is a superclass too. Each subclass will have the actual code that will be launched in the HPC resources.

It can take input parameters, but the code executed will be the same for every instance of a subclass.

It's the only process of the operational service that can run code in the HPC resources. The process is created, launch the code, gets the results, send them to the ensemble and dies.

To control remote jobs in a queue system like *slurm*, it was decided to use the Python library *Fyrd*,[6] as it's the only production ready solution right now. See section 7.1 for discussion about possible future alternatives.

Fyrd will need a lot of code changes and new features to be ready for the JobTask class. It lacks a lot of functionality needed, like getting metrics from the queue system or controlling remote clusters over the network.

Most of the development time was spent in the *JobTask* class and related libraries like *Fyrd*.

4.2 Programming

There were two main development stages, *Fyrd* development (4.2.1), and *JobTask* development (4.2.2). They are related to other components of the operational and external libraries, so there was minor development of those too.

After developing the superclass *JobTask*, subclasses for the pre-process, the run, and the post-process of the model were programmed too.

4.2.1 Fyrd

Fyrd is a Python library to submit jobs to clusters. It has a lot of features like dependency tracking, pandas dataframe submission, different execution profiles or a local job manager.

Most of those features are not needed for this project, but some others are really useful. Especially the ones related to script creation and submission to remote machines. This features consist in creating the necessary scripts for taking a Python function, create a pickle file for it, and execute it in some queue system (like *slurm*).

Executing python functions in a local queue system is very easy and straightforward, as shown in source code 1.

In the previous example, *Fyrd* waits synchronously for the result, but it's also easy to implement an asynchronous version of the previous code, checking from time to time if the results are ready.

Source Code 1: Simple fyrd example

```python
#!/usr/bin/env python3
import fyrd
from time import sleep

@fyrd.jobify(mem='10MB', time='00:00:30')
def test_function(a, b, c):
    # Some high-tech time-consuming algorithm
    sleep(5)
    return a + b + c

result = test_function(1, 2, 3)
```

```
# Wait for the result and print it
print(result.get())
```

The main problem was that Fyrd is not able to control clusters that are in remote hosts, a lot of work was invested to achieve this, among other functionality needed. The result was a fork of Fyrd that can control remote clusters over the network, get metrics for jobs and use different working paths in localhost and the remote host.[10]

To be able to use Python objects remotely, Pyro4 is the best library.[11] It's safe, well tested, production ready and easy to use. It's used to control the remote queue system using a Python object that can be accessed through the network.

To remotely execute Python functions, a pickle file is generated[12] and saved in a remote file system mounted in the host. The remote host will unpack the function from the file and execute it with the correct parameters, the result will also be saved in a pickle file.

Source Code 2: Modified fyrd example

```
#!/usr/bin/env python3
import fyrd
from time import sleep

def test_function(a, b, c):
    # Some stuff
    sleep(5)
    return a + b + c

# Create a job for the previous
# function.
# The job will be executed at
# the queue system of server
# indicated by the uri
job = fyrd.Job(
    test_function,
    args=[1,2,3],
    qtype='slurm',
    uri='PYRO:OurObject@localhost:9090'
    localpath='/tmp/path_of_localhost',
    runpath='/tmp/path_of_remote_host'
    )

print(job.get())
```

The current implementation is far from ideal, but it works well for now until there's a better alternative (see section 7.1).

4.2.2 JobTask

The JobTask superclass uses Fyrd to launch a job in a remote cluster. It has a virtual method that must be overwritten by any subclass. This is the method that will be submitted and executed in the remote cluster.

As this method is packed and sent over the network, it has some limitations. Raising custom exceptions, or passing custom objects as parameters can make the serializer fail[i].

[i]See detailed list at: `https://dill.readthedocs.io/en/latest/index.html#major-features`

During the development of the JobTask class, Fyrd was also changed, as there were some needed parts that were not present, like metrics for jobs.

Creating a new JobTask subclass is simple, as shown in source code 3.

Source Code 3: JobTask subclass example

```
class MyClusterJob(JobTask):
    # Override the method that will be
    # submitted and executed in the
    # remote cluster.
    @staticmethod
    def submit(param1=0,
               param2=0,
               param3=0):
        result = (param1
                  + param2
                  + param3)
        return dict(result=result)
```

One of the problems found in the implementation is that a function and its context have to be submitted to the remote cluster, but `submit` is a method. The steps done to convert the method and send the necessary context are the following:

- **Avoid self**: All methods expects the *self* object to be passed as a first parameter. The `@staticmethod` decorator is used to avoid this behaviour.
- Other important elements, like relevant imports, are already solved and pickled by the serializer (*dill*[13]) when they're used in the `submit` method.

4.2.3 Pre-process JobTask

A simple implementation of the pre-process for the model was programmed. It does the following steps:

- **Computational area**: Computes the area that will be simulated, temporally it's just a fixed size square near the volcano. It has to be big enough to fit all the ash cloud. A better implementation for getting this area is discussed in section 7.2.
- **Compute z layers**: Based on the ash column height and the volcano type, it computes how many vertical layers (z) have to be simulated.
- **Generate the input template**: With the data passed by the *Ensemble* and computed by the pre-process, it creates an input template for the model to run.
- **Process the meteo**: It processes the meteorology, in order to have it on the same scale than the other data used by the model.

4.2.4 Run JobTask

This is the actual process that runs the model. It's more simple than the pre-process, as most of the data processing work is already done.

- **Fill the input template**: The pre-process job generates a template, but this have to be filled with specific elements for each simulation. Those are the elements changed in different simulations of a single *Ensemble*.
- **Generate additional files**: Some other files are needed by the model to run properly.
- **Run the model**: Now that all needed information is prepared, the model is run.

The results will be saved in a file of the local file system of the HPC resource. This file system is mounted in the operational machine to access the results in an easy manner.

4.2.5 Post-process JobTask

Only one simulation per *Ensemble* is launched at this moment, so the post-process don't have to do any transformation to data.

In the future, when multiple simulations are launched for each *Ensemble*, it'll be necessary to implement a merging strategy to combine all the simulation data.

5 Results

The combination of the work done in this project, and the work done by other people of the Mitiga team, allowed to automatically launch and get the results of volcanic ash simulations.

A lot of additional functionality was added to the *Fyrd* library, some bugs were also solved.

6 Conclusions

Despite the difficulty of implementing this functionality, modifying existing libraries and integrating them with a working operational service, the project has been finished successfully.

I've gained a lot of experience about the whole software process, from design to testing, on real products that will be used by important clients.

A lot of time was used to design *Forecast*, it was important as there were several people involved, a common base and design were needed before everyone started to develop its parts. I've learned to do class diagrams, specifications writing, and how teams are managed correctly, giving me some valuable experience for creating serious software.

Some parts of the development were not as hard as expected, I already did an internship in BSC with the Mitiga team, so I was familiar with the codebase and could start coding faster. Other parts were a lot harder, especially *Fyrd*, as it needed a lot of modifications, and it uses some special, not really friendly and maintainable, Python features[ii].

It has been interesting to work with scientific models like Fall3D,[4] managing it in the remote HPC resources and even reprogramming parts of its data pre-processing parts in Python. Using the Marenostrum supercomputer to run the model, and working with people that write and maintain HPC code was also a really interesting and valuable experience for me.

[ii]Global variables and relative dynamic imports are used. OOP is not used in all its parts.

Additionally, I've learned about vulcanology, the airline sector, and atmospheric models. This knowledge, even if it's not directly related to my studies, helped me to know better how some parts of this world works.

Most of the code written during this project is not available, and some information has been not specified on purpose, due to the commercial nature of the whole operational service.

7 Future steps

In the future, I will continue working with the Mitiga team to finish and improve the operational service. There are some parts of this project that probably will be changed in the future. One of those is trying to find a *Fyrd* replacement (section 7.1). The pre-processing of the ash clouds dispersion model can be improved too (section 7.2).

7.1 Carcosa

Fyrd is far from ideal for this project. It has a lot of functionality that's not needed and lacks a lot of functionality that's needed.

To try to solve this problem I developed a library, called *Carcosa*, designed from scratch for our needs.[14]

The main goals are:

- **Well designed**: It's designed thinking on remote clusters, not just local ones. It uses object-oriented programming and follows the best coding practices.
- **Simple**: It has just the minimum functionality needed. It does not have fancy features like *Fyrd* (dependency tracking, local job queue server...). This makes its programming and maintenance simpler, and it's less probable to introduce bugs.
- **Modern**: It uses modern features of Python, like type annotations, that allows static analysis of the code to check for bugs[iii].
- **Well tested**: Tests are fundamental to avoid bugs and undesired behavior, every part of *Carcosa* will be tested.
- **Well documented**: Sphinx[9] documentation with detailed API reference, examples and design decisions.

It's still in early stages of development and not ready for production yet.

This library was developed outside Mitiga and BSC, and have no associatoin with them.

7.2 Model pre-process

One of the main problems in the pre-process of the model is knowing the computational area. We have no knowledge in advance of where the ash cloud will move.

Right now, as a temporal solution, a fixed sized box around the volcano is used, but this is far from ideal. The box has to be big enough to ensure that the ash cloud won't go out of the computational domain, but this is just a best

[iii]https://docs.python.org/3/library/typing.html

effort strategy, as we don't know for sure that the ash won't go out of the box until the simulation is finished.

The result of this strategy is that we have really big computational domains, that can slow down the simulation run. This is not critical at this moment as simulation times are small, and the operational is not launching a lot of simulations. In the future, having a good strategy to choose the computational domain can save a lot of money in HPC resources, and alse save time to give data to clients faster.

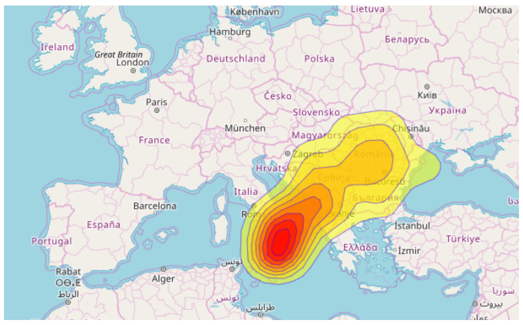

Fig. 4: Volcanic ash clouds of hypothetical Etna eruption 1

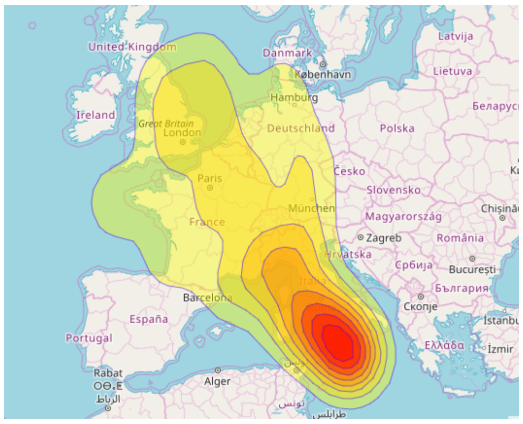

Fig. 5: Volcanic ash clouds of hypothetical Etna eruption 2

Consider ash clouds for Etna eruptions from figure 4 and 5. They obviously need different computational areas, for example, in the eruption of Fig. 4, British or French airspace don't need to be simulated. In the eruption of Fig. 5, Ukraine and Romania don't need to be included in the simulation.

Choosing a good and small computational area allows to run the model faster, or running a simulation with more precision in the same period of time.

To choose a good computational area it's needed to know in which direction, and how far the ash clouds will move. To know this information, a simulation must be run, and a computational area must be chosen.

To break this loop, a low-resolution simulation can be run with an extremely big computational area (a continent, or even the world). With the results of this simulation, the direction and extension of the ash cloud can be estimated, and a good computational domain chosen for the real simulations.

8 Aknowledgements

This project wouldn't have been possible without the help and support of the responsible in BSC, Mauricio Hanzich, and the responsible in UAB, Miquel Àngel Senar.

I also want to thank the entire Mitiga for helping me every time I needed, and for trusting me as a full member of the team.

Thanks to other coworkers at BSC that also helped me, and made nicer lunch times.

Last but not least, thanks to my family, my close friends and especially my partner for all their support during this project.

References

[1] "Volcanic ash–danger to aircraft in the north pacific." `https://pubs.usgs.gov/fs/fs030-97/`. Accessed: 2018-12-04.

[2] "Ash cloud costing airlines £130m a day." `https://www.theguardian.com/business/2010/apr/16/iceland-volcano-airline-industry-iata`. Accessed: 2018-12-04.

[3] "Mitiga solutions." `http://www.mitigasolutions.com/`. Accessed: 2019-01-17.

[4] A. Folch, A. Costa, and G. Macedonio, "Fall3d-7.1," 01 2016.

[5] "Python." `https://www.python.org/`. Accessed: 2019-01-20.

[6] "Fyrd: A pythonic way to submit jobs to any cluster with dependency tracking." `https://github.com/MikeDacre/fyrd`. Accessed: 2019-01-20.

[7] "The secure shell (ssh) connection protocol." `https://www.ietf.org/rfc/rfc4254.txt`. Accessed: 2019-01-20.

[8] "A network filesystem client to connect to ssh servers." `https://github.com/libfuse/sshfs`. Accessed: 2019-01-20.

[9] "Sphinx: A tool that makes it easy to create intelligent and beautiful documentation." `http://www.sphinx-doc.org/en/master/`. Accessed: 2019-01-20.

[10] "Custom fyrd branch." `https://github.com/raul-delacruz/fyrd/tree/pyro_slurm`. Accessed: 2019-01-20.

[11] "Pyro4: Python remote objects." `https://pythonhosted.org/Pyro4/`. Accessed: 2019-01-20.

[12] "Pickle: Python object serialization." `https://docs.python.org/3/library/pickle.html`. Accessed: 2019-01-20.

[13] "Dill: serialize all of python." `https://dill.readthedocs.io/`. Accessed: 2019-01-20.

[14] "Carcosa: Library to programmatically control remote clusters using python.." `https://github.com/quim0/carcosa`. Accessed: 2019-01-20.