Universidad de Cantabria

FACULTAD DE CIENCIAS

INFORME DE LAS PRÁCTICAS

PROFILER INTEGRATION IN A PYTHON-BASED WORKFLOW MANAGER

INTEGRACIÓN DE UN PROFILER EN UN GESTOR DE WORKFLOWS BASADO EN PYTHON

Grado en Ingeniería Informática
Curso 2022-2023

Agosto, 2023

PABLO GOITIA GONZÁLEZ

Acerca de las prácticas

Las prácticas se desarrollan en el entorno del Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC-CNS), dentro del programa BSC International Summer HPC Internship Programme.

- **Tutor académico**: Julio Ramón Beivide Palacio (ramon.beivide@unican.es)
- **Tutor en la empresa**: Miguel Castrillo Melguizo (miguel.castrillo@bsc.es)
- **Duración de las prácticas**: 16 de junio - 8 de agosto, 2023
- **Modalidad**: Presencial. Plaça Eusebi Güell, 1-3, 08034 Barcelona, España

---

Resumen

La misión del departamento de Earth Sciences del BSC es estudiar distintos aspectos del sistema terrestre, desde los procesos atmosféricos a corto plazo y su impacto sobre la actividad humana, hasta la evolución a largo plazo de los distintos subsistemas y su impacto sobre el clima terrestre o la salud de las personas. El departamento realiza su actividad analizando datos provenientes de distintas fuentes, tanto observacionales como de experimentos de modelización.

Llevar a cabo estos experimentos requiere de apoyo técnico a distintos niveles, incluyendo el estudio de la eficiencia computacional de los modelos, la gestión de la infraestructura y análisis de datos, y el desarrollo de herramientas que permitan realizar la experimentación de manera automática. Estos experimentos habitualmente constan de un gran número de tareas (desde cientos a decenas de miles) a ejecutar en diferentes plataformas y con múltiples dependencias que deben resolverse en tiempo de ejecución. En definitiva, no es un proceso que pueda realizarse manualmente, y ahí es donde es indispensable el trabajo del grupo de Computational Earth Sciences y su equipo de Models and Workflows, que se encarga del desarrollo de herramientas que permiten crear, gestionar y monitorizar estos experimentos de simulación.

Una de estas herramientas es Autosubmit, un gestor de workflows hecho en Python que hace las veces de interfaz con los HPC y orquesta todas las tareas y sus dependencias automáticamente. Mi objetivo es desarrollar e integrar una nueva funcionalidad de profiling en Autosubmit para que desarrolladores y usuarios puedan obtener perfiles de rendimiento precisos e información sobre el uso de memoria, para detectar posibles cuellos de botella o regresiones de eficiencia y aplicar las optimizaciones o correcciones necesarias.

Abstract

The mission of the Earth Sciences Department at BSC is to study the Earth system, from short-term atmospheric processes to the long-term evolution of the different subsystems and their impact on the Earth's climate or human activity and their health. The department carries out its activities by analyzing data from different sources, both observational and modeling experiments.

Carrying out these experiments requires technical support at different levels, including the study of the computational efficiency of the models, the management of the infrastructure and data analysis, and the development of tools that allow performing the experimentation automatically. These experiments usually consist of a large number of tasks (from hundreds to tens of thousands) to be executed on different platforms and with multiple dependencies that must be resolved at runtime. In short, it is not a process that can be performed manually, and this is where the work of the Computational Earth Sciences group and its Models and Workflows team is essential, which is responsible for developing methodologies and tools that allow the creation, management and monitoring of these simulation experiments.

One of these tools is Autosubmit, a Python-based workflow manager that interfaces between the users and the HPC's, and orchestrates all the tasks and their dependencies automatically. My goal is to develop and to integrate a profiling functionality in Autosubmit so that developers and users could obtain accurate performance profiles and memory usage information, in order to detect possible bottlenecks or efficiency regressions and apply the necessary optimizations or corrections.

Contents

Abstract 1

Introduction 3

Starting at BSC and Earth Sciences 3

Initial tasks and meeting people . . . 3

Setting up the workspace . . . 3

Running dummy experiments in Autosubmit 4 . . . 4

In a local installation of Autosubmit . . . 5

In MareNostrum 4, through the Autosubmit's virtual machine . . . 5

First merge request for training . . . 6

Project requirements 7

Functional requirements . . . 7

Non-functional requirements . . . 7

Designing the new functionality 7

Enhancement proposals . . . 9

Programming the profiler 10

The profiler's module . . . 10

The code structure . . . 10

CPU profiling . . . 12

Memory profiling: . . . 12

Putting all together . . . 13

Some problems found during the process . . . 14

Integrating the profiler in Autosubmit . . . 15

The argument parser . . . 15

Autosubmit's run_experiment function . . . 15

Enhancement proposals . . . 16

Testing code 17

Black box cases . . . 17

White box cases . . . 18

Writing documentation 20

Demonstration: Profiling a dummy workflow 21

Presentation at the CES meeting, August 4th 22

Introduction

Autosubmit[1] is a lightweight workflow manager designed to meet climate research necessities. Unlike other workflow solutions in the domain, it integrates the capabilities of an experiment manager, workflow orchestrator and monitor in a self-contained application. The experiment manager allows for defining and configuring experiments, supported by a hierarchical database that ensures reproducibility and traceability. The orchestrator is designed to run complex workflows in research and operational mode by managing their dependencies and interfacing with local and remote hosts. These multi-scale workflows can involve from a few to thousands of steps and from one to multiple platforms. It is also a robust software that can handle a variety of setbacks, automatically recovering from, for example, network or I/O errors.

It is currently used at Barcelona Supercomputing Centre (BSC) to run models (EC-Earth, MONARCH, NEMO...), operational toolchains, data-download workflows and many others. Autosubmit has run these workflows in different supercomputers in BSC, CESGA, CSC, OLCF, LMU, KIT, and others.

It has contributed to various European research projects and runs different operational systems. During the following years, it will support some of the Earth Digital Twins as the Digital Twin Ocean within the European Commission's Destination Earth[2] (DestinE) project.

AUTOSUBMIT

Starting at BSC and Earth Sciences

Initial tasks and meeting people

My arrival at BSC coincided with a meeting of the Earth Sciences Department, so I had the opportunity to introduce myself to all the attendees. The following week, while completing some initial tasks I attended some more meetings. The first one was the Autosubmit weekly meeting, where I met the team. Its purpose is to analyze reported issues and enhancements proposals. The next ones were the Models and Workflows Team and the Computational Earth Sciences Group ones, where I also had to introduce myself. At these meetings, participants explain the latest developments in their projects or make presentations on a topic they have been researching.

The first days were dedicated to completing some introductory tutorials and getting to know people and being interested in the projects they were working on. I learned a lot about the main lines of research in the Department, and I quickly became familiar with the BSC environment, tools, software, services and intranet.

I have started to get into the project I have been working on until the next few weeks by reading the documentation of Autosubmit and also of some candidate libraries to be useful for the future profiler..

After that, I managed to get access to the HPC resources that I would use later, such as Marenostrum 4 and Nord 3. I also got access to the virtual machines running Autosubmit, needed to run the experiments on the HPC's.

Setting up the workspace

Autosubmit uses GitLab to manage its source code version control, so installing and configuring Git locally is mandatory. In addition to Git, it was necessary to install a tool to create and manage virtual environments in order to be able to have multiple simultaneous Python installations in the system. This way, it is possible to have an environment for each project, where you

[1]Definition partially extracted from the Earth Sciences Dept. wiki: `https://earth.bsc.es/wiki/doku.php?id=tools:autosubmit`

[2]`https://destination-earth.eu/`

can update or downgrade Python versions at will, recover it in case of corruption or just install libraries without having to deploy them system-wide. There are many tools for this purpose, such as Conda or Mamba. I decided to use the last one by recommendation of my mentor. After a brief meeting I learned its basics and then, I created my first virtual environment for Autosubmit.

Once Git and Mamba were installed, I proceeded to create my workspace. The virtual environment's name was "autosubmit4", and it ran under Python v3.10. The libraries and the Autosubmit's project dependencies would be installed there.

I created a directory called "Workspace" that will be used to host all my local projects. There, I will clone the Autosubmit's repository directly from GitLab. To be able to make modifications in the remote repository, I needed to request special permissions to the IT support in the Department and then, to create a personal token to identify myself with the server.

Searching for IDE's to work, I had some doubts about choosing PyCharm or Visual Studio Code, but I decided to install the latter for simplicity. It is easy to use and it has a huge amount of extensions in case we need some extra functionality that is not integrated by default. It has important functionalities such as a debugger and it can test coverage.

At this moment, everything is ready to build Autosubmit locally. The build was performed using the `setup.py` script, located at the Autosubmit's root directory.

Running dummy experiments in Autosubmit 4

A dummy experiment in Autosubmit is a simple workflow with jobs that emulates a real model execution. It has a preset configuration and is used for training and testing purposes. This is the best way to be introduced to Autosubmit because during the creation process you can check which configuration files and parameters are involved. Throughout the development of the profiler functionality, we will test the code with them. Its structure is the following one:

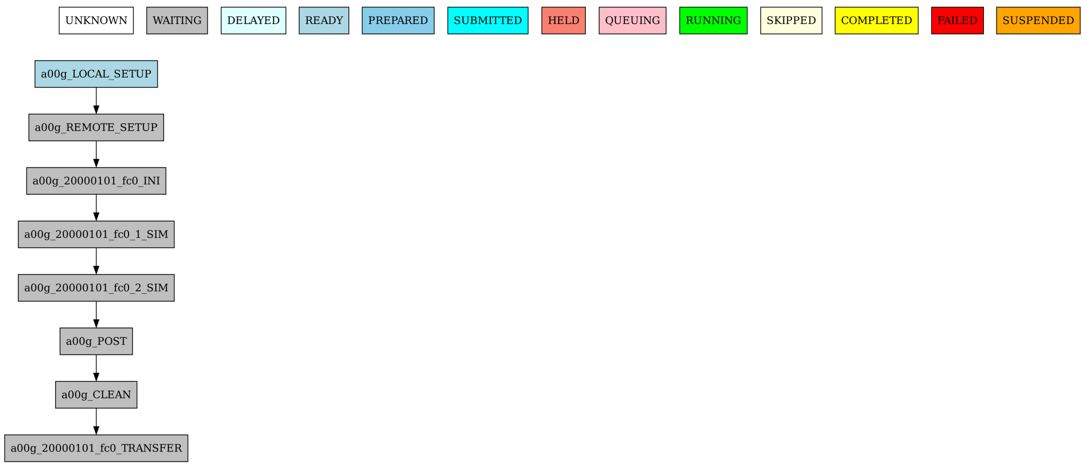


Figure 1: The diagram of a non-executed dummy workflow.

It has 8 different jobs simulating some real task, just spending CPU time.

For training purposes, I was requested to create and run dummy workflows in my local machine and in MareNostrum 4:

In a local installation of Autosubmit

Assuming that we have a version of Autosubmit already installed on the system (or in the virtual environment in our case), creating a dummy experiment is as simple as executing the following command:

```
autosubmit expid -H local -d "Pablo testing" -dm
```

What this command does?

- `autosubmit` is the command that make the main program run. Then, you must specify what you want to do. In our case, with `expid` we are creating a new experiment. This name comes from *EXPID*, or EXPeriment IDentifier. Autosubmit generates a new one for each experiment.
- The `-H` argument establishes the name of the HPC where the experiment will be executed. It could be `local` , as in the example, `marenostrum4` or other machines.
- The `-d` argument is a brief description of what our experiment does (could be anything we want).
- The `-dm` option is used to indicate that we are creating a dummy experiment, with preset configurations and scripts.

This will generate an experiment called *a000*, the first possible *EXPID* assigned.

After creating our experiment, we must create its jobs (the ones that will execute the scripts in our workflow). To do that, we will use the `create` command, passing as argument the *EXPID* of the experiment we just created. Now, we are ready to run our experiment for the first time. We can run it with:

```
autosubmit create a000
```

Once all the jobs had been created, we are ready to run our experiment for the first time with the `run` command:

```
autosubmit run a000
```

In MareNostrum 4, through the Autosubmit's virtual machine

Running a workflow in a BSC machine like MareNostrum 4 requires to follow some additional steps, but essentially the process is the same as in local.

Firstly, its recommended to keep opened one terminal on your BSC workstation, and another one SSH connected to the Autosubmit's Virtual Machine (located in the Department's internal network). The workstation is required because it has Autosubmit already installed and configured with a network shared directory. All the data of your experiment will be located at `/esarchive/autosubmit/` , together with the files of other people's experiments. So in the workstation we will create the dummy experiment, as follows:

```
autosubmit expid -H marenostrum4 -d "Pablo testing" -dm
```

This will create a new experiment, and its assigned *EXPID* in this case was *a068*. We will edit its configuration files to set the connection parameters (host url, user...). All the configuration files in Autosubmit are in YAML format. Autosubmit by default has some HPC's configured, so choosing the `marenostrum4` one and specifying the user in the platform configuration file would be enough. That file looks as follows:

```
PLATFORMS:
  MARENOSTRUM4:
    TYPE: slurm         #scheduler
    HOST: mn1.bsc.es    #ip or alias
    PROJECT: bsc32
    USER: ********      #username at the HPC, the only parameter we will change
    SCRATCH_DIR: /gpfs/scratch
    ADD_PROJECT_TO_HOST: false
    QUEUE: bsc_es       #could also be 'debug'

  #OTHER PRE-CONFIGURED PLATFORMS
```

File 1: autosubmit/a68c/conf/platforms_a068.yml

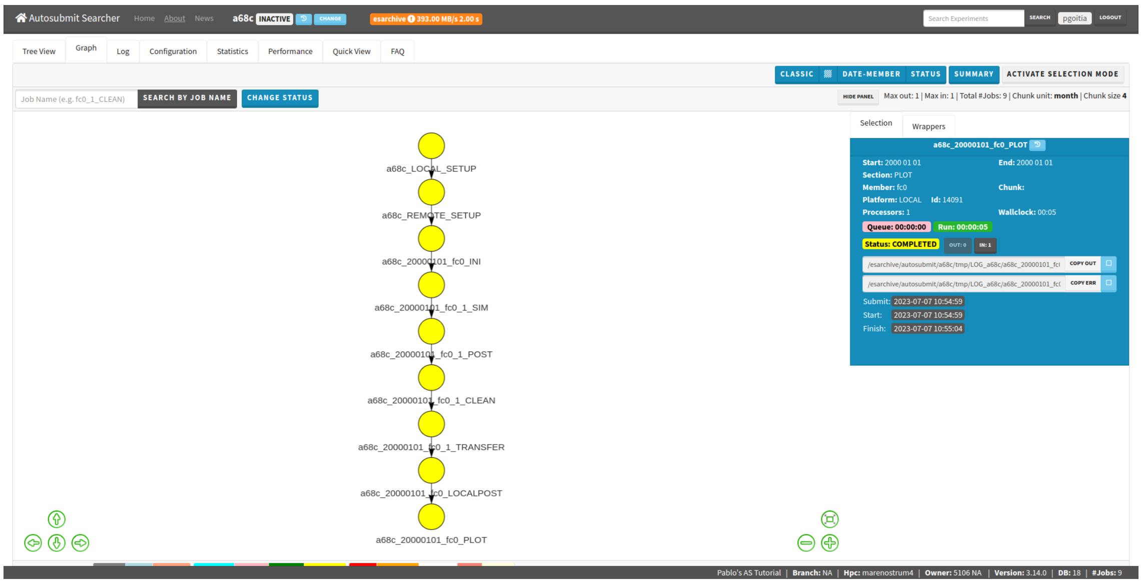
Now, we can create the jobs of the workflow, also with the `create` command:

```
autosubmit create a068
```

The experiment is ready to be executed, but Autosubmit needs to establish password-less SSH connections in order to run and monitor workflows on remote platforms. Once we have stored the keys in each platform we must access to the VM and execute the `run` command:

```
autosubmit run a068
```

Autosubmit provides a web interface where we can check the status of the execution of all the models. The following screenshot shows the diagram of our dummy workflow, when all the jobs have finished:



Max out: 1 | Max in: 1 | Total #Jobs: 9 | Chunk unit: **month** | Chunk size **4**

a68c_20000101_fc0_PLOT

Start: 2000 01 01 **End:** 2000 01 01
Section: PLOT
Member: fc0 **Chunk:**
Platform: LOCAL **Id:** 14091
Processors: 1 **Wallclock:** 00:05
Queue: 00:00:00 **Run: 00:00:05**
Status: COMPLETED OUT: 0 IN: 1
/esarchive/autosubmit/a68c/tmp/LOG_a68c/a68c_20000101_fc
/esarchive/autosubmit/a68c/tmp/LOG_a68c/a68c_20000101_fc
Submit: 2023-07-07 10:54:59
Start: 2023-07-07 10:54:59
Finish: 2023-07-07 10:55:04

Pablo's AS Tutorial | **Branch:** NA | **Hpc:** marenostrum4 | **Owner:** 5106 NA | **Version:** 3.14.0 | **DB:** 18 | **#Jobs:** 9

Figure 2: The diagram of the executed *a068* dummy workflow, at the web GUI.

First merge request for training

As a part of the training in the Models and Workflows Team, I was required to read the Autosubmit's documentation in search of some grammatical mistakes and correct them. This was interesting to familiarize myself with the Git commands and proceedings that I used every day.

I proceeded to create a new branch, and then a merge request to merge it into the master's branch[3]. Its title was "Draft: Fix: Documentation corrections". The "Draft:" tag at the beginning is to indicate that the branch is not ready to be merged by now. The "Fix:" one is just descriptive.

[3] `https://earth.bsc.es/gitlab/es/autosubmit/-/merge_requests/349`

This was my first contribution to the Autosubmit development. It included 2 changes that were made in two separate commits, and I pushed them up individually to the remote server. Then, I was requested to manually *squash* both commits into one as a training exercise, although GitLab already offers an option to do this automatically when you make a merge request.

The final squashed commit was "Changed 'euns' at line 49 to 'runs' in the qstartguide index, and 'user' to use in userguide/config 's line 82". Then, I made the third push to the server (forcing), and deleted the draft status to formalize the merge request. My mentor approved and merged the branch, and now the changes appear in the last build of the Autosubmit's Read the Docs[4], the website that hosts the Autosubmit's documentation.

Project requirements

The requirements that the profiler must meet were specified in a GitLab issue[5] opened by my mentor some weeks before I arrived to BSC. It starts mentioning that it would be useful to have a way to run Autosubmit with a profiler, but without having to install a profiler module and modify the Autosubmit source code each time we want to profile it. The requirements for that profiler are as follows:

Functional requirements

Code	Requirement description
F001	The implementation will use the Python *cProfile* and *psutil* built-in modules for profiling.
F002	The profiler must be integrated in the **autosubmit run command**, so users can enable or disable it.
F003	The profiler must be disabled and not interfere with Autosubmit commands by default.
F004	It must produce a simple output that gives users a simple view of the performance of Autosubmit.

Non-functional requirements

Code	Requirement description
NF001	It must work on the laptop of common users, workflow developers, Autosubmit developers, and also on other servers/platforms like MareNostrum4, Nord3, or external sites like LUMI.

Designing the new functionality

After a rigorous reading of the documentation of the functions that I will use to build the profiler, I started to think how to structure my code to ensure an easy integration with the Autosubmit code in the future.

[4]`https://autosubmit.readthedocs.io/en/master/index.html`
[5]`https://earth.bsc.es/gitlab/es/autosubmit/`

Firstly, I will implement a basic profiler able to get the CPU time consumption of each function, thanks to the *cProfile* and *pstats* libraries. I will also add a memory profiling functionality using the *psutil* library, but just after having that first part.

I spent several hours reading documentation, but here is a summary of the main purpose of each of the libraries mentioned:

- The ***cProfile***[6] library implements a deterministic profiler based in C and introduces less overhead to the code execution than other similar libraries.
- The ***pstats*** library will be used for processing and handling the profiling results.
- The ***psutil***[7] library is useful when you need to get information about the processes running in the system, or when you need to limit its resources.

From here, it is important to remember that the profiler must be designed to be easily integrated into the `autosubmit run` command. This command executes the experiment starting the different tasks as their dependencies are fulfilled and repeating its execution in case of failure. This command has an associated function in the Autosubmit's main file called `run_experiment()` . This function, as many others in Autosubmit, consists of a loop surrounded by a *try/except* statement (or *try/catch*, in other languages).

After some consideration, I decided that the best way to achieve the requirements described in the GitLab issue, and above, was to make the profiler follow a sequential model, divided in 4 functions that will be called in order like in the following diagram:

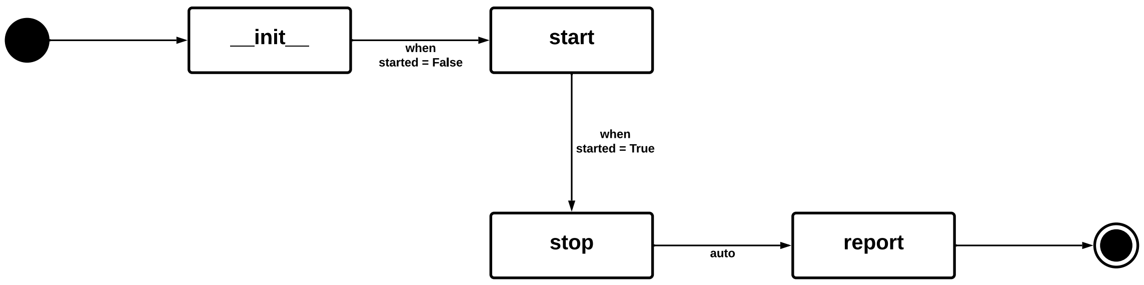


Figure 3: The definitive status diagram of our profiler.

What each function is supposed to do?

- ***init***: is the function of each Python object that is executed each time it is instantiated. It will establish the initial files, data structures and instance variables to ensure the correct work of the other functions.
- ***start***: starts the profiler imported from the *cProfile* library, takes the current memory usage, and will handle any errors if necessary.
- ***stop***: ends the profiler, so that it finishes taking measurements, takes again the current memory usage, and will do error handling if necessary. It will also generate the final report.
- ***report***: takes the profiling results and processes them to create a report that will be printed by console.

Note

The `report` function is called automatically by the stop function, but I have decided to keep it in the diagram for a better understanding.

[6] `https://docs.python.org/3/library/profile.html`
[7] `https://psutil.readthedocs.io/en/latest/`

All these functions will be located in a new module in order to facilitate code comprehension and maintainability. The `profiler.py` module will be located at the `autosubmit/profiler` directory.

Thanks to the sequential design, we could firstly instantiate a new Profiler object, and make it to start calling `start()` before the main loop of the `run\_experiment()` function. After it, and in a *finally* clause inside the *try/except* statement, we could call to the `stop()` function, that will automatically generate and save the report. This way, the profiler will always produce a report even if something inside the function raises an exception that abruptly finishes the command execution.

Enhancement proposals

Apart from the original design explained above, I had some more ideas to make the profiler more useful. In the same week I finished designing the basics of the profiler, I started to build the implementation, so in the next Autosubmit Weekly Meeting I presented the design, part of the code, and the ideas I had during the implementation. We discussed about the following ones:

- **The generated report would be stored in the Autosubmit's log files so that developers could access it later.**

  They liked this proposal, and it was finally implemented in the final version.

- **Exporting a file with the final report so that developers could post-manipulate it as they prefer.**

  The main reason for this is that the *stdout* and the log contain a preformatted report that maybe do not fit the developer needs. This file could be opened with the SnakeViz[8] library, which allows us to view the results both graphically and in a detailed way through a web GUI.

  To explain this idea I made a demonstration of how could I get a report file and I opened it with SnakeViz as an example. They also liked this initiative, and it was finally implemented.

  Since this characteristic was implemented, my personal recommendation for Autosubmit developers or users was to post-manipulate the exported file with SnakeViz. This is far easier than using *pstats*.

- **Some possible problems must be prevented, like saving multiple reports from different instances of *Profiler* without distinguishing them. Maybe each one could have a name provided by the programmer.**

  The Autosubmit team does not plan to use more than one instance of the profiler, so following the principle "You aren't gonna need it"[9] I discarded this idea.

- **The same profiler instance should be able to be started and stopped as many times as the developer needs, merging all the generated reports into one. If we don't implement this functionality we must handle the corresponding errors: the same profiler instance can't be used more than once.**

  Investigating a bit more about this issue, I made a short script[10] to check if *cProfile* already merge the reports when you execute its profiler more than one time, and it does, so implementing this idea would be relatively easy. However, we considered that it would be much simpler and natural to let the user to start and stop the profiler once to avoid mistakes. This decision leaves the status diagram as it is right now, without a transition from `stop` to `start` .

Some of these ideas were already implemented, at least partially, because I though that were useful for the profiler. This also helped me to demonstrate and defend their importance.

[8]https://jiffyclub.github.io/snakeviz/
[9]https://en.wikipedia.org/wiki/You_aren't_gonna_need_it
[10]https://drive.google.com/file/d/1RHQFSa3QGAmTEPwdHQuhI9TQiTSjmvoC

Programming the profiler

First of all, I cloned the Autosubmit repository[11] from GitLab to my workspace. Then, for simplicity, I created an alias in the *bashrc* file called `as4venv` to both activate the `autosumbit4` virtual environment and to access the project's directory in one step. This was one of the most used commands I had during my internship and I saved a lot of time thanks to it. The next step is to open a new merge request on GitLab. I linked it to the profiler's issue previously opened by my mentor, mentioned before. This merge request had been supervised by him and other Autosubmit collaborators along the process. It can be accessed through the following link:

`https://earth.bsc.es/gitlab/es/autosubmit/-/merge_requests/350`

The profiler's module

The code structure

Knowing the requirements and how the Autosubmit's main code was built, we can start programming the "core" of our project: the profiler's module. The first step was to find the most suitable place for it, which will be next to the other Autosubmit modules in the main directory:

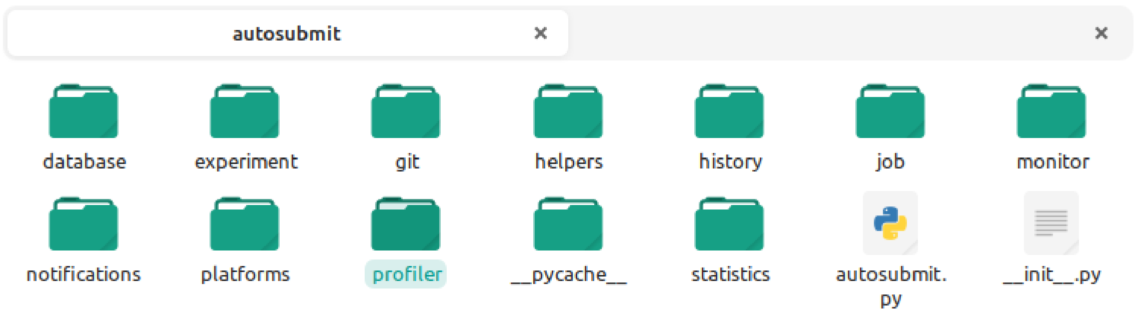


Figure 4: The Autosubmit's source code directory.

The profiler's directory will contain the `profiler.py` file, with the source code, and other files automatically generated by Python:

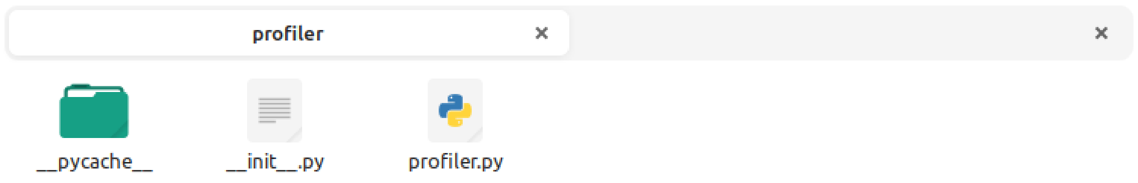


Figure 5: The profiler's directory.

The code will be composed by a class `Profiler` that will contain the functions mentioned in the design: `__init__()` , which is the equivalent of a constructor in other programming languages, `start()` and `stop()` , which will start and stop the profiling process, and `report()` , which will take the generated data and process it to generate the final report. The most basic code structure would be the following one:

```python
class Profiler:
"""Class to profile the execution of experiments."""
    def __init__(self):
        # Code of the init function

    def start(self):
        """Function to start the profiling process."""
        # Code of the start function
```

[11]`https://earth.bsc.es/gitlab/es/autosubmit/`

```
    def stop(self):
        """Function to finish the profiling process."""
        # Code of the stop function

    def report(self):
        """Function to print the final report into the stdout, log and
        ↪  filesystem."""
        # Code of the report function
```

Source Code 1: The basic structure of the "Profiler" class.

What is the "self" argument?

In Python, self represents the instance of the class. it is necessary to access the instance variables and the functions defined inside it.

During the development process I decided to make the report function private. For simplicity for the users, it will be called automatically by the stop function. In Python, private functions are differentiated adding a "_" before its name, as follows:

```
def _report(self):
    """Function to print the final report into the stdout, log and
    ↪  filesystem."""
    # Code of the report function
```

Source Code 2: Example of a private function.

The declaration of a private function in Python, however, is not strict. It is possible to call it from outside the class, but the IDE will probably warn.

Once defined the structure, we could proceed searching for a way to prevent breaking the sequential execution of the profiler, according to the state diagram defined at the Figure 3. Doing this shouldn't be more complicated than creating some error handling flags, for example, `started` and `finished` . Their values by default will be `False` and will be changed to True in the `start()` and `stop()` , respectively.

If the conditions are not met, the corresponding function will throw an exception. Autosubmit has 2 exception types for different purposes, *AutosubmitCritical* and *AutosubmitError*. We will use *AutosubmitCritical*. *AutosubmitError* exceptions are not recommended because they are thought-out for recoverable code. The exceptions in Autosubmit have an associated message and an error code. The profiler has its own entry at the error codes table[12] in the documentation webpage, the error *7074*.

If we put all together, the resulting code should be like this:

```
from log.log import AutosubmitCritical  #importing the exception type

class Profiler:
    def __init__(self):
        # Error handling flags
        self._started = False
        self._finished = False

    def start(self):
        if self._started:
```

[12] https://autosubmit.readthedocs.io/en/master/troubleshooting/error-codes.html

```
                raise AutosubmitCritical('The profiling process was already
                ↪  started.', 7074)

            self._started = True

            # more code [...]

        def stop(self):
            if not self._started or self._finished:
                raise AutosubmitCritical('Cannot stop the profiler because was
                ↪  not running.', 7074)

            # more code [...]

            self._report()
            self._finished = True

        # The report function is not affected
```

Source Code 3: The structure with conditions that don't let the user to break the sequential execution.

At this stage, it is time to start writing the functional part, also following an objective: to make the `start()` and `stop()` functions as simple as possible.

CPU profiling

As planned, we will use the *cProfile* library to get statistics about the CPU usage of all the functions executed during the profiling. The *cProfile* library provides a class called `Profile` . Its usage is simple: you must instantiate a Profile object and call to its enable and disable methods conveniently. The instantiation will be made inside the `__init__()` function, and the object will be saved as an instance attribute of our `Profiler` object, so that it could be accessed from other functions. Then, in the start function we will call the Profile's `enable()` method, and the same in the opposite way in the stop function, calling `disable()` . When calling disable, the CPU profiling stops and the statistics are saved in the Profile object. These statistics will be manipulated in the report function.

Inside the report function, we will take advantage of the *pstats* library to adjust the report to our needs. *pstats* will take the statistics in the `Profile` object and convert them into a comprehensible string format. By default, the order of the functions in the report is random, but what the Autosubmit developer wants is to quickly locate possible bottlenecks or functions that are delaying more time than the expected, so we will change this order criteria to "cumulative time", i.e. the total time consumed by all calls to a single function. This way, the functions that had been using more CPU will be listed at the top of the report.

Memory profiling:

As I proposed in the design, the *psutil* library will be used to collect memory usage details. *psutil* is really useful to get information about running processes in the system. In our case, it would be enough getting the total amount of memory used by the function or the fragment of code we want to profile. To get more detailed memory information, it is better to use external tools dedicated exclusively to this purpose. I also searched alternatives to this library, for example, memory-profiler[13], but it didn't quite fit what I was looking for.

Then, to get the total memory consumption, we can take the current memory consumed in both start and stop functions, and the result will be the difference between these 2 measures. To get the current memory I created a private function outside the Profiler's class, which obtains

[13]https://pypi.org/project/memory-profiler/

the *PID* used to instantiate a `Process` object provided by the own *psutil* function, and then it calls the `memory_info()` [14] method of that process. That method will return multiple memory consumption data, but the most important are the *rss* and the *vms*. *rss* means "Resident Set Size", the non-swapped physical memory a process has used, and *vms* means "Virtual Memory Size", the total amount of virtual memory used by the process. We will return back to the caller function the first one.

In the report, we will take the 2 memory values previously collected to calculate its difference. Now we have the total memory consumed, but in bytes. I wrote a simple loop to convert it to its most suitable unit. For example, if we get 1048576 bytes, the loop will convert it to 1 MiB. This is more useful for the developer.

Putting all together

A resumed code of what we have been doing in the last sections would be as follows:

```
import cProfile
import pstats
import psutil

from log.log import AutosubmitCritical

class Profiler:
    def __init__(self):
        self._profiler = \textit{cProfile}.Profile()

        # Memory profiling variables
        self._mem_init = 0
        self._mem_final = 0

        # Error handling flags
        self._started = False

    def start(self):
        # Here was the error handling section

        self._profiler.enable()
        self._mem_init = _get_current_memory()
        self._started = True

    def stop(self):
        # Here was the error handling section

        self._profiler.disable()
        self._mem_final = _get_current_memory()
        self._report()

    def _report(self):
        # Here we generate the CPU usage report

        # Generate memory profiling results
        mem_total = self._mem_final - self._mem_init  # memory in Bytes

        # Here is the loop to reduce the value to its most suitable unit
```

Source Code 4: A very simplified vision of the code after getting all the profiling statistics.

At this moment, the most important part of the profiler's module is done. What continues is merging the results into a single report and saving it somewhere. Merging the report is an

[14] `https://psutil.readthedocs.io/en/latest/#psutil.Process.memory_info`

straightforward task, it just require to join the CPU results string with the memory value, and adding some headers to differentiate each section. Printing the report to console and storing it in the log could be done in a single step, because the Autosubmit's logger automatically prints to the *stdout* channel what you log. This step is quite easy and it only requires to import the Autosubmit's `Log` library. However, storing the results into the file system is not so simple: we must be careful to respect the format implemented by other functionalities in Autosubmit to be consistent and not confuse the users.

The profiler will export 2 different files: a plain text `.txt` with the same content as the report previously shown to the user and a `.prof` binary file which contains the statistics offered by our `Profile` object without the post-processing with *pstats*. This last file could be interpreted and manipulated by external tools such as SnakeViz, as I mentioned before.

The name of the generated files will follow the format below:

- For the stats (binary): [*EXPID*]_profile_[date]-[time].prof
- For the report (plain text): [*EXPID*]_profile_[date]-[time].txt

An example of filename could be: `a000\_profile\_20230804-124833.prof`

The files will be saved into the directory `[EXPID]/tmp/profile` of the related experiment, next to other generated files by Autosubmit. To build the path we need to request the *EXPID* to the programmer. We could request it while instantiating the Profiler object, as follows:

```
def __init__(self, expid):
    self._expid = expid
    # More variables [...]
```

Source Code 5: Where to request the *EXPID* to build the output files path

The report function thanks to the Path library and some Autosubmit constants will build the absolute path to the export directory.

Now, the profiler module is completely functional, and is ready to be integrated in the `autosubmit run` command.

Some problems found during the process

- **GitLab's pipeline fails when I submit the newest changes.**

  Each time you push your last local changes to GitLab, your code passes through a "pipeline" which compiles and tests the code and the documentation. The failure, this time, occurred because the *psutil* library was not installed by default. I solved this dependency problem by adding '*psutil*' to the install_requires list inside the Autosubmit's setup script. It also needed to be added to the requirements.txt file in the AS' root directory. Hereinafter, the Autosubmit's setup will install *psutil* in the local machine if it's not installed yet.

- ***pstats* output is not ordered as indicated. It follows a "call" order, not accumulated time as specified.**

  It happened because of the order of the calls that manipulate the statistics. The function sort_stats() must appear next to strip_dirs() one.

- **Duplicated commits because of a "git pull" made from my personal computer at home.**

  Solved with `$git rebase` . The process took a long time.

Integrating the profiler in Autosubmit

Having the profiler ready to be integrated in the `autosubmit run` command, I took a look in the Autosubmit's main source file to learn how it works and to get some first ideas of the code that I will need to modify. I inserted some prints and re-installed the code with the `pip install -e [AS path]` command.

The installation was successful and the results were the expected ones, so I could continue writing the actual code:

The argument parser

Autosubmit uses a parser library (argparse[15]) to recognize its commands and to save the parameters. I located the section where the "run" subparser was, and with the help of other existing arguments, I added a new Boolean flag for the profiler. It will be activated when the user writes the flag `[-p, --profile]` . Its value by default is `False` , as described in the requirements section.

The code snippet below contains the creation of the "run" subparser and the "profile" argument:

```
# Run
subparser = subparsers.add_parser('run',
            description="runs specified experiment")
# Other arguments [...]
subparser.add_argument('-p', '--profile', action='store_true', default=False,
            required=False, help='Prints performance parameters of the
            execution of this command.')
```

Source Code 6: Where to request the *EXPID* to build the output files path

Autosubmit's run_experiment function

Once the `autosubmit run` command could recognize the profile flag, it was necessary to add it to the `run\_experiment()` function declaration, and passing it as a parameter in the corresponding calling. To access the flag we just need to use `args.profile` . Naturally, I also added the new parameter to the function docstring, following the reStructuredText docstring format[16].

Then, I wrote a small section of code to test if the work was done well or not. If the user sets the `--profile` flag (or `-p`), a message will be printed by console. I removed that code in subsequent commits.

Now, it is time to correctly place the `Profiler` instantiation and its function calls. As planned in the design phase, if the flag is `True` , the `Profiler` object will be created at the beginning of the function, followed by the call to the start function. Remember that the main loop of the function is wrapped by a *try/except* statement, so the stop function will be called from the finally clause to ensure that the profiler always shows its results even if something during the Autosubmit's execution fails.

Here is the resulting code of applying the idea above:

```
def run_experiment(expid, notransitive=False, start_time=None,
 ↪  start_after=None, run_only_members=None, profile=False):
    """
    Runs and experiment (submitting all the jobs properly and repeating its
↪   execution in case of failure).
    :param expid: the experiment id
    :param notransitive: if True, the transitive closure of the graph is not
↪   computed
    :param start_time: the time at which the experiment should start
```

[15]https://docs.python.org/es/3/library/argparse.html
[16]https://realpython.com/documenting-python-code/#restructuredtext-example

```
    :param start_after: the expid after which the experiment should start
    :param run_only_members: the members to run
    :param profile: if True, the function will be profiled
    :return: None

    """
    # Start profiling if the flag has been used
    if profile:
        profiler = Profiler(expid)
        profiler.start()

    try:
        #########################
        # AUTOSUBMIT - MAIN LOOP
        #########################
        while job_list.get_active():
            # Here autosubmit executes the jobs
    except BaseException as e:
        raise AutosubmitCritical("This seems like a bug in the code, please
        ↪  contact AS developers", 7070, str(e))
    finally:
        if profile:
            profiler.stop()
```

Source Code 7: Simplified view of the function with the profiling functionality already implemented.

Enhancement proposals

The following ones, are just ideas that I had while working on the code. Each one required discussion in different Autosubmit Weekly Meetings:

- **Avoid the user to execute the profiler module directly from the command line.**

  This just require a couple more lines on the code, but it isn't necessary at all, so we decided not to implement it.

- **Create a custom decorator for the Autosubmit's profiler. This should make it easier to use in other functions.**

  A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure[17].

  With a decorator, profiling a function would be as simple as writing `@profile` above its heading:

  ```
  @profile                        #this is de decorator
  def run_experiment(...):        #this is the function heading
      #the function code
  ```

  Source Code 8: Example of use of the decorator in the *run_experiment* function.

  This is the most complex proposal that I've been working on. It would be quite useful for easily integrating the profiler into the Autosubmit commands. However, building a decorator has its difficulties: you need a way to indicate when to profile or not depending on the status of the flag, and it cannot be passed as a parameter because the only way to do it is saving the flag as a global variable, and this is bad practice in Python for security reasons. It also has one more handicap: where to save the profiling results. Until now, the answer seemed to be obvious, we could remain saving them into the experiment's `tmp` path, but not

[17]Extracted from `https://www.datacamp.com/tutorial/decorators-python`

every command in Autosubmit uses an *EXPID*, so if the profiler is working on that kind of functions it may need to export the results to a generic place, and this is not consistent at all.

Despite that, I programmed the decorator solving these problems, but we decided not to include it in the last reviews because the logic I introduced was too difficult to be rewritten to include it in an upcoming project that the Autosubmit developers have pending, which consists of refactoring the entire Autosubmit code to make it even more modular.

After removing it, I opened an issue on the Autosubmit's GitLab to briefly explain other programmers how to integrate the decorator again: `https://earth.bsc.es/gitlab/es/autosubmit/-/issues/1094`

- **Adding type hints as defined at PEP 484[18].**

  This idea had been proposed by my mentor. Python by its nature assigns types to variables dynamically. Adding type hints, we would be forcing a specific type to each one, as other languages such as C, Java... do. Implementing this proposal have been a first step for a future project about compiling the Python modules to C extensions to make Autosubmit more efficient[19].

  A brief example:

```
def _generate_title(title):
"""
Generates a title banner with the specified text.

:param title: The title that will be shown in the banner.
:type title: str
:return: The banner with the specified title.
:rtype: str
"""
```

  Source Code 9: The _generate_title() auxiliary function with its docstring.

  If we add type hints to the function above, the result would be the following one:

```
def _generate_title(title:str) -> str:
```

  Source Code 10: The _generate_title() auxiliary function after adding type hints.

Testing code

To ensure code quality and reliability I was requested to build unit tests and try to get as high coverage percentage as possible. All of them had been designed following black/white box methodologies.

The tests have been programmed in a different file called `test_profiler.py` , located next to the other Autosubmit tests. They works thanks to the *unittest*[20] library, and to execute them I used the *nosetests*[21] tool.

Black box cases

Black box defines coverage criteria for status machine based classes, with status and transition coverage. This criteria is ideal for our case. Status coverage means that all the possible status are reached, and transition coverage means that all the possible transitions between status are contemplated. We will also cover those cases in which you try to go to an unreachable status (from another one that doesn't have any connection).

[18]https://peps.python.org/pep-0484/
[19]https://earth.bsc.es/gitlab/es/autosubmit/-/issues/876
[20]https://docs.python.org/3/library/unittest.html
[21]https://pypi.org/project/nose/

Then, the tests will contain the following cases:

- **Status machine coverage**

  It will execute the whole profiling process. I.e, it will call `__init__()` , `start()` and `stop()` in that order. Remember that we are not taking into account `report()` because it is internally called from `stop()` .

- **Transition coverage**

  According to the status diagram in the Figure 3, the possible transitions are: from `__init__()` to `start()` and from `start()` to `stop()` , that will automatically call to report. These cases, essentially, are exactly the same as those in the last item, so we will preserve these instead of the others. Repeating cases would not make sense.

- **Wrong transitions coverage**

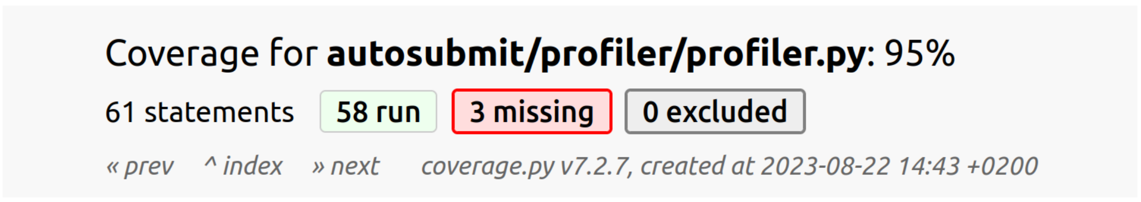
  According to the diagram, wrong transitions are from `__init__()` to `stop()` , from `start()` to `start()` and from `stop()` to `stop()` .

How the actual tests are:

```python
class TestProfiler(TestCase):
    def setUp(self):
        self.profiler = Profiler("a000")

    def test_transitions(self):
        # __init__ -> start
        self.profiler.start()

        # start -> stop
        self.profiler.stop()

    def test_transitions_fail_cases(self):
        # __init__ -> stop
        self.assertRaises(AutosubmitCritical, self.profiler.stop)

        # start -> start
        self.profiler.start()
        self.assertRaises(AutosubmitCritical, self.profiler.start)

        # stop -> stop
        self.profiler.stop()
        self.assertRaises(AutosubmitCritical, self.profiler.stop)
```

Source Code 11: Black box test cases.

Only with black box tests we got a high percentage of coverage of the code, so that a lot of white box cases could be omitted later. The coverage is **95%** for profiler.py.



Coverage for **autosubmit/profiler/profiler.py**: 95%

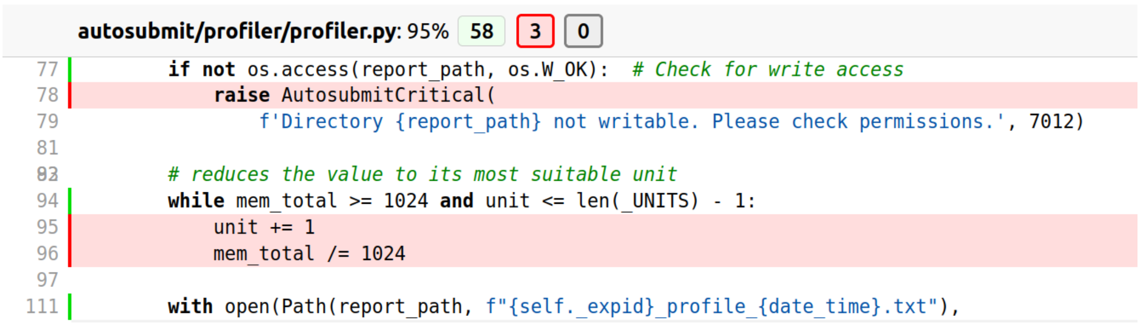
61 statements 58 run 3 missing 0 excluded

« prev ^ index » next coverage.py v7.2.7, created at 2023-08-22 14:43 +0200

Figure 6: Screenshot of the coverage report just running the black box test cases.

White box cases

We will attempt to reach 100% in `profiler.py` with white box test cases. To simplify the process and to not redefine already existing test cases, we will directly take a look to the coverage report to see which lines are not covered by the black box cases. In the figure below, not covered lines are highlighted in red:



autosubmit/profiler/profiler.py: 95% 58 3 0

```
 77        if not os.access(report_path, os.W_OK):  # Check for write access
 78            raise AutosubmitCritical(
 79                f'Directory {report_path} not writable. Please check permissions.', 7012)
 81
 82        # reduces the value to its most suitable unit
 94        while mem_total >= 1024 and unit <= len(_UNITS) - 1:
 95            unit += 1
 96            mem_total /= 1024
 97
111        with open(Path(report_path, f"{self._expid}_profile_{date_time}.txt"),
```

Figure 7: Screenshot of the coverage report just running the black box test cases.

There are only 3 not covered lines (just the 5% of the code). The first one, in the line 78 is because the profiler checks if the directory in which the files will be exported is writable, and if not, an exception will be raised. This exception will never be raised until we force it. Doing this is not as trivial as adding an assert clause with the result expected. I needed to use a mock to accomplish this task. A mock is a “false” object that returns what we want in order to successfully test the code. We will create a mock that replaces the os.access call, making it to return `False` always. Mocking is not easy to understand, but searching for some examples and reading documentation I could get used to it relatively quickly. The resulting white box test for this part is the following one:

```python
@mock.patch("os.access")
def test_writing_permission_check_fails(self, mock_response):
    mock_response.return_value = False

    self.profiler.start()
    self.assertRaises(AutosubmitCritical, self.profiler.stop)
```

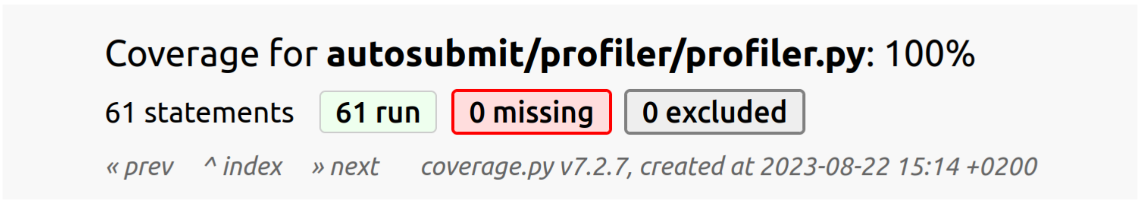
Source Code 12: Test case to cover the writing permissions condition.

The lines 95 and 96 are from the loop that convert the memory consumption unit to its most suitable. This part is never accessed because our simple tests almost don’t waste memory, so the converter is no needed. Covering these lines is as simple as running a profiler allocating memory after starting and before stopping it. There are many ways to do this. The solution below is simple and efficient:

```python
def test_memory_profiling_loop(self):
    self.profiler.start()
    bytearray(1024*1024)
    self.profiler.stop()
```

Source Code 13: Test case to cover the unit converter loop.

With these 2 cases we have got the expected 100% of coverage, and the execution of the tests doesn’t throws any kind of error nor warning, so we finished with the testing part.



Coverage for **autosubmit/profiler/profiler.py**: 100%

61 statements 61 run 0 missing 0 excluded

« prev ^ index » next coverage.py v7.2.7, created at 2023-08-22 15:14 +0200

Figure 8: Screenshot of the coverage report running both black and white box test cases.

Writing documentation

The Autosubmit documentation is written in reStructuredText format[22]. When you push changes in the documentation to the GitLab repository, they are automatically built and uploaded to the Autosubmit's Read the Docs website.

The documentation source directory looks as follows:

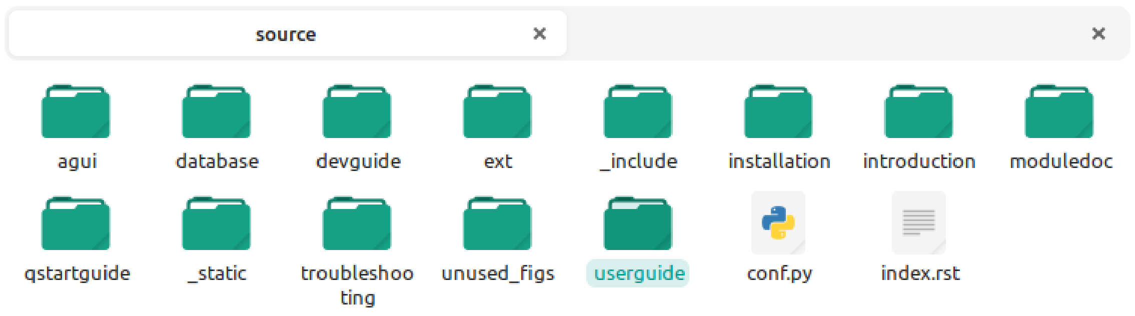


Figure 9: Documentation source directory.

The important directory for us is `userguide` (highlighted). That folder contains the documentation related to the `autosubmit run` command, and many others:

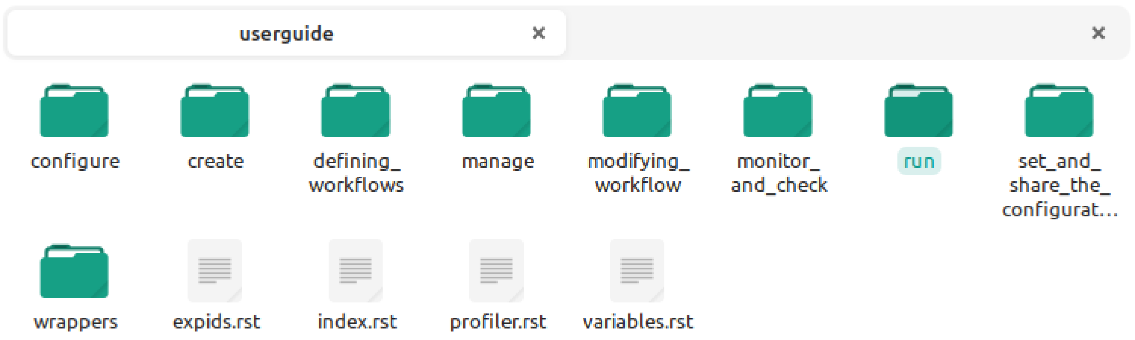


Figure 10: userguide directory inside the documentation source code.

There is a folder called `run` , where is the `index.rst` . I modified that file to include a brief section user-oriented explaining the basics of the profiler: how to activate it and interpret the results. I also created a new orphan file inside the same folder called `profiler.rst` , with more detailed documentation that will be useful for other Autosubmit developers. An orphan file in reStructuredText is a page that is not expected to be listed in a menu and just can be accessed from a internal link in another page. In fact, in the beginning, I had some compilation problems because of the orphan tag, but later I realized that it wasn't in the correct place of the header.

To represent the status diagram, I decided to use the GraphViz[23] library. This tool lets you code graphics yourself (is not a graphical tool), and is so powerful and widely used by any kind of applications. If fact, it is used to compose the diagrams generated by Autosubmit, for example the Figure 1. Programming a diagram from zero is not a trivial task and requires a lot of imagination, but is the best way to ensure that future developers could update it. I integrated the code in the documentation and it compiles successfully (and looks great). The diagram is available at the extended documentation.

The profiler documentation could be accessed through this link:
https://autosubmit.readthedocs.io/en/master/userguide/run/index.html

The extended documentation is also available at this page:
https://autosubmit.readthedocs.io/en/master/userguide/profiler.html

[22]`https://docutils.sourceforge.io/rst.html`
[23]`https://graphviz.org/`

Demonstration: Profiling a dummy workflow

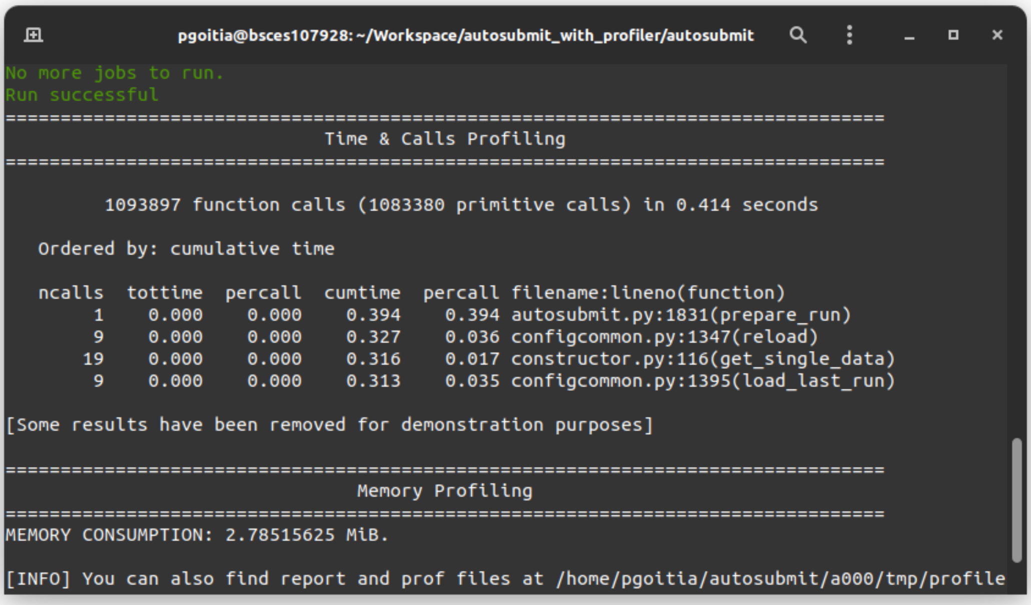
The tool is finally ready to be used, so here is a short demo of how the profiler works:

1. The first step is to create an experiment. It could be dummy. In fact, we will test the profiler with the same experiment that I created during my training when I arrived to BSC, the *a000*.

2. Execute the run command with one of the profile flags (`--profile` or `-p` , it doesn't matter at all):

```
autosubmit run a000 --profile
```

3. After the execution of the experiment, the final report is printed by console and saved in plain text at the `tmp` directory of the experiment, as well as the binary file:



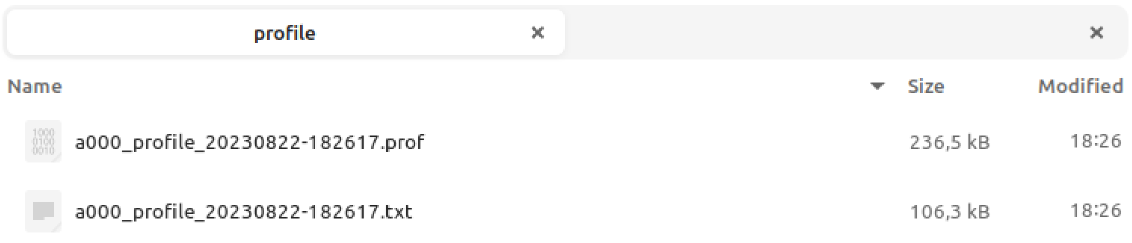
pgoitia@bsces107928:~/Workspace/autosubmit_with_profiler/autosubmit

```
No more jobs to run.
Run successful
================================================================================
                           Time & Calls Profiling
================================================================================

         1093897 function calls (1083380 primitive calls) in 0.414 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.394    0.394 autosubmit.py:1831(prepare_run)
        9    0.000    0.000    0.327    0.036 configcommon.py:1347(reload)
       19    0.000    0.000    0.316    0.017 constructor.py:116(get_single_data)
        9    0.000    0.000    0.313    0.035 configcommon.py:1395(load_last_run)

[Some results have been removed for demonstration purposes]

================================================================================
                              Memory Profiling
================================================================================
MEMORY CONSUMPTION: 2.78515625 MiB.

[INFO] You can also find report and prof files at /home/pgoitia/autosubmit/a000/tmp/profile
```

Figure 11: Simplified profiling results.

The CPU profiling report is more complete than the shown in the screenshot above. Some of the entries were omitted just to leave space to the memory profiling results in the window.

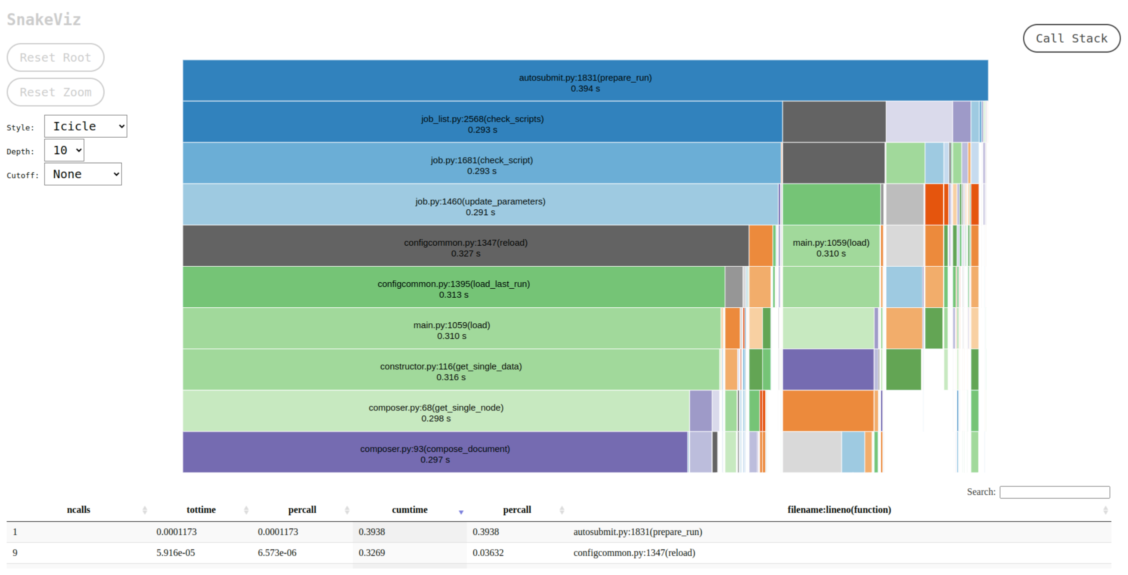
The exported files are in the experiment's `tmp/profile` path as expected:



Name	Size	Modified
a000_profile_20230822-182617.prof	236,5 kB	18:26
a000_profile_20230822-182617.txt	106,3 kB	18:26

Figure 12: The exported files.

4. There are many tools available if someone need to open and manipulate the .prof file. I always recommended to use SnakeViz, because it allows you to view the results in a graphical way and to reorder all the entries according to your needs:


SnakeViz
Reset Root
Reset Zoom
Style: Icicle
Depth: 10
Cutoff: None
Call Stack
autosubmit.py:1831(prepare_run) 0.394 s
job_list.py:2568(check_scripts) 0.293 s
job.py:1681(check_script) 0.293 s
job.py:1460(update_parameters) 0.291 s
configcommon.py:1347(reload) 0.327 s
main.py:1059(load) 0.310 s
configcommon.py:1395(load_last_run) 0.313 s
main.py:1059(load) 0.310 s
constructor.py:116(get_single_data) 0.316 s
composer.py:68(get_single_node) 0.298 s
composer.py:93(compose_document) 0.297 s
Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.0001173	0.0001173	0.3938	0.3938	autosubmit.py:1831(prepare_run)
9	5.916e-05	6.573e-06	0.3269	0.03632	configcommon.py:1347(reload)

Figure 13: The SnakeViz web interface showing the same profiling results.

Presentation at the CES meeting, August 4th

When my internship main project was finished, I presented it to the CES meeting attendees on August 4. I had a 15∼20 minute slot before the end of the meeting, in which I gave a very brief account of the development process that I have detailed in this report, focusing more on how to use the new functionality, how it can be used and how to extend it for the future. It also includes a small section about how to integrate the profiler with other functions, but warning of possible incompatibilities.

The slides I used are available to the whole Earth Science Department for anyone who may need them. The document can be accessed through the following public link:

https://earth.bsc.es/wiki/lib/exe/fetch.php?media=working_groups:ces:profilerpresentation.pdf

On august 8th, the merge request was approved and the branch was successfully merged. From now, all the Autosubmit builds from version 4 onward will integrate the new profiler.