Using the Submitter class for high throughput HPC calculations with Soprano

Simone Sturniolo

February 20, 2018

Chapter 1

The Submitter class

In Soprano, the `hpc.submitter` module provides classes and tools meant to help computational scientists to automate the process of generating, running and possibly post-processing huge batches of molecular calculations like molecular dynamics, DFT calculations and so on on high performance computing machines where access to the nodes is regulated by a queueing system. The target is to allow users to create their own pipelines for high throughput calculations, like running the same analysis tools through an entire database of chemical structures, with only a few lines of Python code. Since this is, however, still a complex task that will inevitably require to be customised and tailored to the needs of each specific case, using this functionality requires a somewhat higher level of programming confidence than the rest of the library and can look a little daunting at first. No worries - this manual will guide you through the steps necessary to deploy your own Submitter to your machine of interest to run potentially any computational task you can think of. Besides instructions on the creation of a new Submitter, this manual also will include advice on how to test it and debug it locally, in order to minimise the risk of running into bugs after deploying the system.
First, let us clarify what a Soprano Submitter instance *is* and *isn't*. A Submitter is a very general class designed to automate the process of queueing, running and postprocessing huge amounts of calculations on a computing cluster. It can be used to do this locally on the machine it is running on or to use a remote system to do the heavy lifting. It is *not*:

- a system handling parallel calculations. There is no parallelism included in a Submitter. As the name suggests, it simply submits jobs to a queueing system like GridEngine, which then itself provides them with the required number of cores. To achieve maximum compatibility, the Submitter is entirely agnostic of the underlying hardware architecture;
- a system ready for deployment as-is. As it comes shipped with Soprano, the Submitter class basically does nothing. It is meant to be used as a parent class to write your own sub-class which specifies the details of the task

of interest through a number of designated methods. This is absolutely on purpose - the objective here is maximum flexibility for potentially very demanding tasks, sacrificing immediate usability;

- a system meant to work as a server running continuously and accepting a flow of user requests. While it is possible to have a Submitter run indefinitely it is not advised to rely on it that way, as it currently lacks the robustness to deal with all the errors that could happen during that kind of usage and to restart itself in case of a crash. Rather, it should be best employed with runs of definite size, with a clear end in sight.

Now let's see a quick overview of what the typical pipeline for deploying a Submitter system is, followed by a more detailed explanation of all the steps involved.

Chapter 2

Deploying a Submitter

The process to create and launch a Submitter requires the following steps:

- Write a class inheriting from `soprano.hpc.submitter.Submitter` implementing the desired calculation through its core methods;
- Create a submission script for your HPC system using tags which are going to be replaced by specific values in each job;
- Write a Python source file creating an instance of this class (it can also be the same file) and setting up its parameters, and positioning this file in the working directory. In some cases this might require creating a `soprano.hpc.submitter.QueueInterface` as well;
- Launch the newly created submitter from the command line by calling the appropriate Soprano script.

Let's see these steps in more detail.

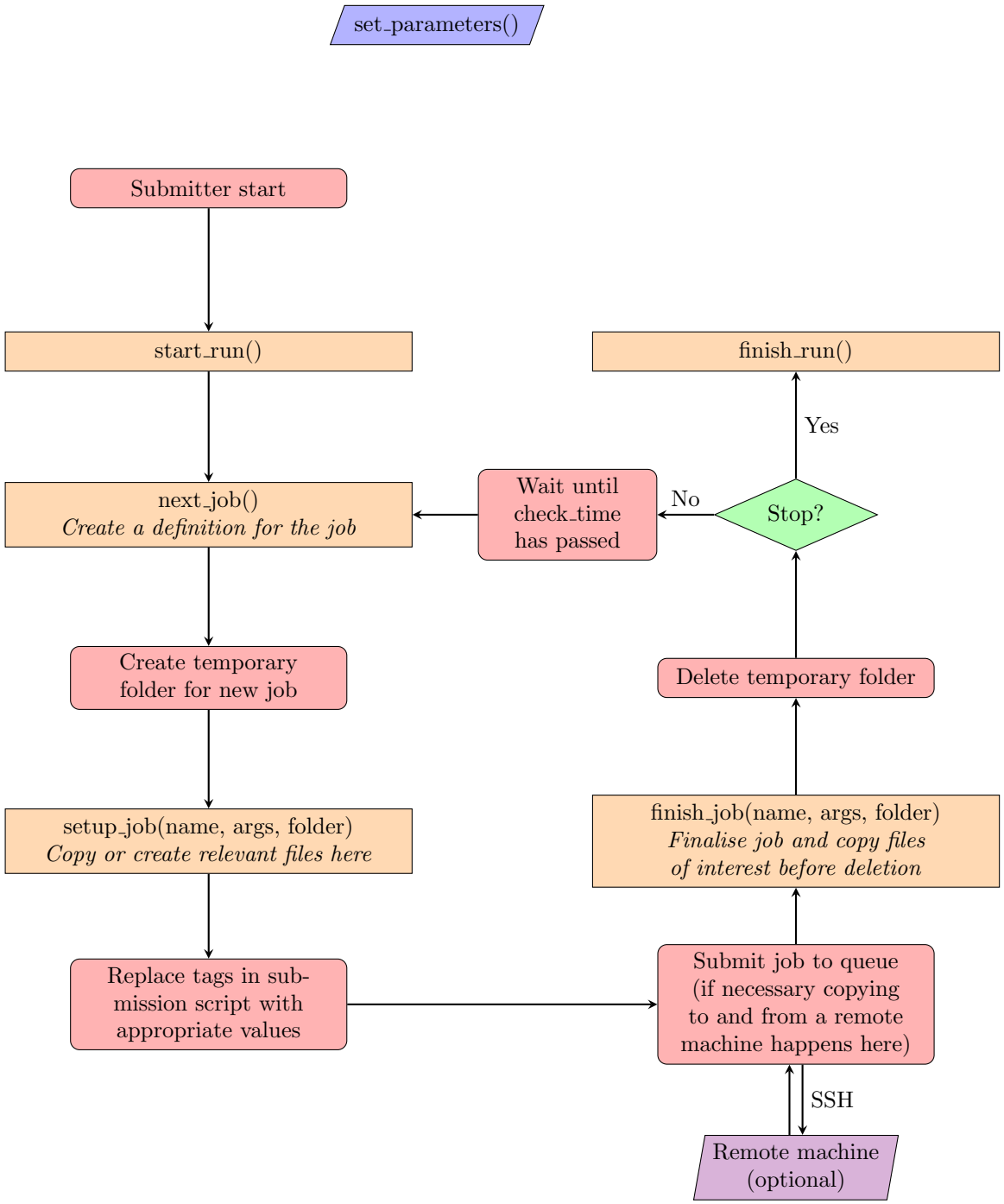
2.1 Writing a Submitter class

The basic template to inherit from the Submitter class can be as simple as this:

```
from soprano.hpc.submitter import Submitter

class MySubmitter(Submitter):
        pass
```

However, such a class would be of no use since it does not implement any particular behaviour. The Submitter class originally has a number of methods. Some of them are preceded by an underscore and are meant to not be touched, as they handle the core behaviour of any Submitter. Others are meant to be overridden when inheriting from the base class and control custom behaviour - the specifics of the problem at hand. The core methods determine the Submitter workflow, which can be described as follows:



- `set_parameters(**kwargs)` - Must be executed by hand by the user after creating the Submitter instance. Can be overridden and have any interface.

Since the `__init__` method should *not* be altered, this method provides a way for the user to feed class-specific parameters to any derived Submitter types.

- `start_run()` - Executed at the beginning of a run. Here any tasks that are preparatory to initialising the entire run should be accomplished. By default does nothing.

- `next_job()` - Executed whenever the Submitter has a free slot to start a new job. Should return a definition for said job in the form of a dictionary containing a `name` (in the form of a string) and some `args` in the form of a dictionary: by default returns `{'name': 'default_job', 'args': {}}`.

- `setup_job(name, args, folder)` - Executed immediately after `next_job()` and the creation of a temporary folder to contain its files. Must perform any operations required to set up the input for the upcoming calculations. For example, `next_job` could return a Python object defining an atomic structure on which the calculation ought to be performed; `setup_job` is where the input files to which said structure is written should be created and saved in the path defined by `folder`.

- `finish_job(name, args, folder)` - Executed once the job has been run and the output files are in `folder` (if the job has been run on a remote machine, at this point the files will have been downloaded back to the local file system). At this point any post-processing should be done and any output files of relevance should be *copied to a persistent folder*. This is especially important as after this step the temporary folder will be deleted.

- `finish_run()` - Executed at the end of the run, takes care of clean up. If any files need to be closed or resources deallocated this is the place to do it. By default does nothing.

- `save_state()` - Only executed when the Submitter is initialised with the argument `continuation=True`. This allows the Submitter to be stopped (or in case of accidents, crash) and store its current state in order for it to be restarted manually at a later time. By default the submitter only stores the list of the currently running jobs so that it can get back to monitoring them; however, here the user can define any additional data they want to be stored as well, as long as it can be 'pickled' [1]. This data must be returned as members inside a dictionary. This method *must* return a dictionary, even if empty. The file is saved as `<name>.pkl`, where `<name>` is the 'name' argument passed to the Submitter's constructor.

- `load_state(loaded_data)` - Only executed when the Submitter is initialised with the argument `continuation=True` and a previously saved state file is found. Receives as argument a dictionary containing the saved data; this

[1] For more information on the process of serialisation, or 'pickling', in Python, see: https://docs.python.org/2.7/library/pickle.html

should be in the same format as the one returned by `save_state`. In general it should perform the same operations in reverse, restoring the Submitter to its original state.

2.2 Writing a tagged submission script

When submitting a job to a queueing system, one needs to do so in the form of a script, which usually contains the desired commands as well as a number of comments functioning as parameters for the queueing system itself. In the simplest case, this script could be limited to an executable (the program we need to run) and the name of its input file. When creating jobs with a Submitter, this script will be ran from within the folder in which the input files have been created with `setup_job`; however, it is reasonable to imagine that these might have different file names or parameters from job to job. To account for that, when one creates a submitter, a script is passed to in which any word that is supposed to change is replaced by a tag surrounded by angle brackets. The main purpose for this is to use the argument `name`; the same can be done with any element appearing in the `args` dictionary returned as part of the output of `next_job`. So for example

```
echo "<sentence>" > <name>.txt
```

for a job defined as

```
{'name': 'hello',
 'args': {'sentence':
          'Hello␣World!'}
}
```

would produce a file named `hello.txt` containing the sentence "Hello World!".

2.3 Creating a Submitter instance

Given that one has created and specified a derived Submitter class and a properly tagged submission script, it is then necessary to create a file that defines an instance of the Submitter itself. Assuming that we already imported the definition of the new class, the basic template for such a file will be

```
from soprano.hpc.submitter import QueueInterface

# Other interfaces supported are LSF and PBS,
# plus you can define your own.
myQ = QueueInterface.GridEngine()
# Here we load the script
myScr = open('tagged_script.sh').read()

mySub = MySubmitter('test_submitter', myQ, myScr)
mySub.set_parameters(...) # Any parameters needed...
```

The constructor for a Submitter takes a number of optional parameters beyond the three obligatory ones (name, queue interface and script). They can all be found in the documentation but for convenience here's a rundown:

- `max_jobs (Optional[int])` - maximum number of jobs to submit at a given time. When this number of job has been submitted to the queue, the Submitter stops and waits until one of them is complete. Default is 4.
- `check_time (Optional[float])` - time in seconds between consecutive checks for the queue status and attempts to submit new jobs. Default is 10.
- `max_time (Optional[float])` - time in seconds the Submitter will run for before shutting down. If set to zero the thread won't stop until killed with Submitter.stop.
- `temp_folder (Optional[str])` - where to store the temporary folders for the calculations. By default it's the current folder.
- `remote_workdir (Optional[str])` - if present, uses a directory on a remote machine by logging in via SSH. Must be in the format `<host>:<path/to/directory>` (without the angle brackets, replace host name and directory suitably; the directory must already exist). Host must be defined in the user's `~/.ssh/config file` - check the docs for RemoteTarget for more information. It is possible to omit the colon and directory, that will use the home directory of the given folder; that is HEAVILY DISCOURAGED though. Best practice would be to create an empty directory on the remote machine and use that, to avoid accidental overwriting/deleting of important files.
- `remote_getfiles (Optional[list(str)])` - list of files to be downloaded from the remote copy of the job's temporary directory. By default, all of them. Can be a list using specific names or wildcards. Filenames can also use the placeholder name to signify the job name, as well as any other element from the arguments.
- `ssh_timeout (Optional[float])` - connection timeout in seconds. Default is 1.
- `continuation (Optional[bool])` - if True, when the Submitter is stopped it will not terminate the current jobs; rather, it will store the list of ongoing jobs in a pickle file. If the submitter is ran again from the same folder then it will "pick up from where it left" and try recovering those jobs, then restart. If one wishes for additional values to be saved and restored, the `save_state` and `load_state` methods need to be defined.

2.4 Launching a Submitter

Finally, once everything in place, it is possible to launch the Submitter. This is done by running the command `soprano_submitter.py`. The syntax is described as follows:

```
soprano_submitter.py [-h] [-n N] [-nohup] action submitter_file
```

where `submitter_file` is the name of the file in which the *instance* of the Submitter was created (as described in 2.3) and `action` can be one of `start`, `stop` or `list`. The previous two are pretty self-explanatory; the third lists which Submitters, among the ones defined in the given file, are running, and how long they have been. The additional options work as follows:

- `-h` - display the help message
- `-n N` - launch the submitter of name N. This must be used when multiple Submitters are instanced in `submitter_file`. The name must be the name of the variable (so for example in the example in 2.3, that would be `mySub`)
- `-nohup` - put the Submitter thread in 'no hangup' mode, meaning that it will keep running even if one closes the terminal session from which it was launched. This is mainly for the case in which a Submitter was launched on a machine to which we connect remotely via `ssh`.

When a Submitter is started it runs in the background of a shell session until `max_time` seconds have passed or it is explicitly stopped. When stopped, if `continuation=True` wasn't set, it kills all still ongoing jobs and deletes all the temporary folders; otherwise it leaves everything running and stores a pickled file with the pattern `<submitter_name>.pkl` in the current folder from which it can pick back up. Deleting the file in question is equivalent to resetting the run. During and at the end of a run, a log file called `<submitter_name>.log` is available containing messages detailing the workings of the Submitter. When defining a new Submitter class, one can use the method `self.log('Message')` inside it to write to this log file additional lines.