EasyVVUQ-QCGPJ

Apr 21, 2021

Basics

Python API for HPC execution of EasyVVUQ

EasyVVUQ-QCGPJ (EQI) is a lightweight plugin for parallelization of EasyVVUQ with the QCG-PilotJob

It is a part of the VECMA Toolkit

The tool provides API that can be effortlessly integrated into typical EasyVVUQ workflows to enable parallel processing of demanding operations on HPC machines.

It can work also on your laptop so you can start using it whenever you want: from the beginning of your work with EasyVVUQ or once you realise that the serial execution of EasyVVUQ is no longer sufficient.

CHAPTER 1

Installation

1.1 Requirements

The software requires Python 3.6+ for usage.

Moreover, since EasyVVUQ-QCGPJ is a wrapper over EasyVVUQ and QCG-PilotJob, you need to have both these packages available in your environment. This version of the library is compatible with EasyVVUQ v0.8 and QCG-PilotJob v0.11.1. Compatibility with other versions is not confirmed and may be limited. Thus, if you want to be sure that correct versions of required packages are available, install them in the following way:

```
$ pip3 install --force-reinstall easyvvuq==0.8
$ pip3 install --force-reinstall qcg-pilotjob==0.11.1
```

1.2 Automatic installation

The software could be easily installed from the PyPi repository:

```
$ pip3 install easyvvq-qcgpj
```

1.3 Manual installation

If you prefer manual installation or you want to install specific branch of the software you can get it from the the github repository. The procedure is quite typical, e.g.:

```
$ git clone https://github.com/vecma-project/EasyVVUQ-QCGPJ.git
$ cd EasyVVUQ-QCGPJ
$ git checkout some_branch
$ pip3 install .
```

CHAPTER 2

Quick Start

The usage of EasyVVUQ with EasyVVUQ-QCGPJ is very similar to the typical usage of EasyVVUQ. In the same way as it is in a regular EasyVVUQ script, the user defines the Campaign object and configures it to use specific encoders, decoders, samplers. The identical is also the part of collating results and analysis. The difference is in the middle, in the way how the campaign is executed.

Basically, the code has to be instrumented with a few instructions required to configure EasyVVUQ-QCGPJ. This comes down to:

1. Creation of the EasyVVUQ-QCGPJ Executor object.
2. Creation of the QCG-PilotJob Manager for the use by the Executor.
3. Configuration of EasyVVUQ Tasks to be executed by the Executor (in practice to be executed by QCG-PilotJob Manager as separate processes).
4. Execution of EasyVVUQ workflow consisted of the Tasks using the Executor.
5. Finalization.

2.1 Example workflow

In order to explain the basic usage of EasyVVUQ-QCGPJ API we will use an example.

Note: For the full code of this example please look into the test case available at the EasyVVUQ-QCGPJ GitHub (https://github.com/vecma-project/EasyVVUQ-QCGPJ) in the path: `/tests/test_pce_pj_executor.py`

Here we briefly outlines the main parts of that workflow concentrating on the EasyVVUQ-QCGPJ and skipping fragments that are common with the standard execution of EasyVVUQ.

```
# ...
import easyvvuq as uq
import eqi
```

(continues on next page)

(continued from previous page)

```
from eqi import TaskRequirements
from eqi import Task, TaskType, ProcessingScheme

jobdir = os.getcwd()
tmpdir = jobdir
appdir = jobdir

TEMPLATE = "tests/cooling/cooling.template"
APPLICATION = "tests/cooling/cooling_model.py"
ENCODED_FILENAME = "cooling_in.json"


def test_cooling_pj(tmpdir):
    my_campaign = uq.Campaign(name='cooling', work_dir=tmpdir)
    # ...
    # Skipped the typical code of EasyVVUQ that initialises the campaign with␣
↪encoders, decoders etc.
    # ...
    my_campaign.draw_samples()

    ################################
    # START of EasyVVUQ-QCGPJ part #
    ################################

    # Create Executor
    qcgpjexec = Executor(my_campaign)

    # Create QCG-PilotJob-Manager with 4 cores
    # (if you want to use all available resources remove resources parameter)
    qcgpjexec.create_manager(resources='4')

    # Declare tasks, one for encoding and one for execution, providing their resource␣
↪requirements
    qcgpjexec.add_task(Task(
        TaskType.ENCODING,
        TaskRequirements(cores=1)
    ))

    qcgpjexec.add_task(Task(
        TaskType.EXECUTION,
        TaskRequirements(cores=1),
        application='python3 ' + jobdir + "/" + APPLICATION + " " + ENCODED_FILENAME
    ))

    # Execute the encoding and execution steps of the campaing using Executor
    qcgpjexec.run(processing_scheme=ProcessingScheme.SAMPLE_ORIENTED)

    # Terminate the created QCG Pilot Job manager
    qcgpjexec.terminate_manager()

    ##############################
    # END of EasyVVUQ-QCGPJ part #
    ##############################

    # The rest of typical EasyVVUQ processing
    my_campaign.collate()
```

(continues on next page)

(continued from previous page)

```
    # ...
    # Skipped code
    # ...
```

As you can see, the code required for parallel encoding and execution of the samples stored in an EasyVVUQ campaign is quite concise. The user just need to create an Executor object providing the already initialised Campaign as an argument (the set of samples should be ready for processing) and then, using the methods provided by the object, steer the execution from the relatively high level.

Below we shortly describe particular elements of this process:

1. Instantiation of the QCG Pilot Job Manager

The Executor internally uses QCG-PilotJob Manager to submit Tasks. The Pilot Job Manager instance needs to be set up for the Executor. To this end, it is possible to use one of two methods: the presented `create_manager()` or `set_manager()`. More information on this topic is presented in the section *QCG-PilotJob Manager initialisation*.

2. Declaration of tasks

The Executor with the `add_task()` method allows to define a set of Tasks that will be executed once the `run()` method is launched. A Task added with the `add_task()` method needs to be of some type. Currently EaasyVVUQ-QCGPJ supports three types of Tasks that maps to EasyVVUQ steps that should be executed within a Task: `ENCODING`, `EXECUTION` and `ENCODING_AND_EXECUTION`. These types are described in section *Task types*.

3. Execution of tasks

The Executor configured with the QCG-PilotJob Manager instance and filled with a set of appropriate Tasks is ready to perform parallel processing of encoding and execution steps for all Campaign's samples using the `run()` method. This method takes `processing_scheme` parameter to define a type of the scheme for submission and execution of Tasks by QCG-PilotJob Manager. The available schemes differ in a several aspects:

- *scope of covered EasyVVUQ steps*: encoding and execution, or just execution,
- *order of submission: step oriented or sample oriented*,
- *way of execution of tasks by QCG-PilotJob Manager*: separate tasks for all steps of a run vs a common task for all steps of a run (condensed), separate tasks for each run for a given step vs an iterative task for all runs within the step.

There is no general rule for the selection of scheme as its applicability and performance depends on many factors. For more demanding use-cases it is worth to analyse which scheme works best. More information about the schemes can be found in the section *Processing schemes*.

2.2 Launching the workfow

The way of starting the defined workflow is typical, e.g.:

```
python3 tests/test_pce_pj_executor.py
```

Common environment

Please only be sure that the environment is correct for both, master script and tasks. More information on this topic is presented in the section *Passing the execution environment to QCG-PilotJob tasks*.

Note: It is worth noting that the workflow can be started in a common way on both local computer and cluster. In case of the batch execution on clusters, the above line can be put into the job script.

2.3 Resuming the workfow

EQI is able to resume processing of tasks within QCG-PilotJob Manager if the workflow was not completed (for example when it was stopped due to crossing the walltime limit). The resume mechanism is enabled by default and it is used whenever Executor is inited with the campaign for which EQI processing was already started but not yet completed. For more information see *Resume mechanism*

CHAPTER 3

API description

3.1 EasyVVUQ-QCGPJ Executor

`Executor` is the main object responsible for steering the configuration and parallel execution of selected EasyVVUQ tasks with QCG-PilotJob. The object needs to be tied to the already prepared instance of the EasyVVUQ campaign and therefore it takes it as the mandatory `campaign` parameter for the constructor.

The second (optional) parameter of the `Executor`'s constructor is `config_file`, which can be used to initialise the environment of tasks started by QCG-PilotJob. More information on this topic is presented in the section *Passing the execution environment to QCG-PilotJob tasks*

The next parameter is `resume`. By default it is set to True, which means that EQI will try to resume not completed workflow of tasks submitted to QCG-PilotJob Manager.

> More on this topic is discussed in the section *Resume mechanism*

The last (optional) parameter is `log_level` that allows to set specific level of logging just for the EasyVVUQ-QCGPJ part of processing.

3.2 QCG-PilotJob Manager initialisation

The EasyVVUQ-QCGPJ Executor needs to be configured to use an instance of QCG-PilotJob Manager service. It is possible to do this in two ways:

- The first and simpler option is to use `create_manager()` method that creates QCG-PilotJob Manager in a basic configuration. The method takes the following optional parameters:
  - `dir` to customise a working directory of the manager (by default current directory)
  - `resources` to specify resources that should be assigned for the Pilot Job. If the parameter is not specified, the whole available resources will be assigned for the Pilot Job: it means that in case of running the Pilot Job inside a queuing system the whole allocation will be used. If the parameter is provided, its specification should be consisted with the format supported by Local mode of QCG-PilotJob manager, i.e. `[NODE_NAME]:CORES[,[NODE_NAME]:CORES]...`

  - `enable_rt_stats` to enable collection of QCG-PilotJob Manager runtime statistics
  - `wrapper_rt_stats` when `enable_rt_stats` is set to `True`, this parameter should point to the location of a QCG-PilotJob tasks wrapper program, used for collection of statistic for executed tasks.
  - `reserve_core` to specify if the manager service should run on a separate, reserved core (by default `False`, which means that the manager's core will be shared with executed tasks)
  - `log_level` to set logging level for QCG-PilotJob Manager service and client parts.
- The second and more advanced option is to use `set_manager()` method. This methods takes a single parameter, which is an instance of externally created QCG-PilotJob Manager instance. Don't try to use this method unless you have very specific needs.

  For the reference go to: QCG-PilotJob documentation.

3.3 Task types

EasyVVUQ-QCGPJ supports the following types of Tasks that may be executed by QCG PJ Manager:

- `ENCODING`: this Task is used for the encoding of a single sample.
- `EXECUTION`: this Task is used for the execution of an application for a single sample. The constructor of this Task requires the `application` parameter to be specified with the value defining a command to run the application. The `EXECUTION` Task for a given sample depends on the `ENCODING` Task for the same sample.
- `ENCODING&EXECUTION`: this Task is used for running both encoding and execution for a single sample. Similarly to the `EXECUTION` Task the constructor of this Task requires the `application` parameter to be specified with the value defining a command to run the application.

The addition of a Task to Executor does not condition its later use - this if the Task is actually used depends on a specific processing scheme that is selected for the execution in the `run()` method of Executor. In order to keep consistency of the environment only a single Task of a given type should be kept in the Executor.

3.4 Tasks requirements

Tasks defined for execution by the QCG-PilotJob system need to define their resource requirements. In EasyVVUQ-QCGPJ the specification of resource requirements for a Task is made directly via the Task's constructor, particularly by its second parameter - `TaskRequirements`. This object may be inited with a combination of two parameters: `nodes` and `cores`. If the only specified parameter is `cores`, the Task will run on a specified number of cores regardless of their physical location (the cores can be distributed on many nodes). If there are two parameters specified: `nodes` and `cores` the Task will use the number of cores requested by `cores` parameter on each of the nodes requested by `nodes` parameter. Therefore, in order to have good efficiency, for the multicore Tasks it is advised to specify two parameters: `nodes` and `cores` (even if there is only a need to take one node).

Both `nodes` and `cores` parameters may be of `int` type or of `Resources` type. In the case when a parameter is of an `int` type, the provided value is simply mapped to the exact number of required resources. In the case of parameters of `Resources` type, there is much more flexibility in the requirements specification, which may be obtained with the following keyword combinations:

- `exact` - the exact number of resources should be used,
- `min` - `max` - the resources number should be larger than `min` and lower than 'max,
- `min` - `split-into` - all available resources should be divided into chunks of size `split-into`, but the size of chunks can't be smaller than `min`

Example `TaskRequirements` specifications:

- Use exactly 4 cores, regardeless of their location

```
TaskRequirements(cores=4)
```

- Use 4 cores on a single node

```
TaskRequirements(nodes=1,cores=4)
```

- Use from 4 to 6 cores on each of 2 nodes

```
TaskRequirements(nodes=2,cores=Resources(min=4,max=6))
```

The algorithm used to define Task requirements in EasyVVUQ-QCGPJ is inherited from the QCG-PilotJob system. Further instruction can be found in the QCG Pilot Job documentation

3.5 Task execution models

The optional parameter of `Task` constructor is `model`. It allows to adjust the way how a task will be started by QCG-PilotJob Manager in a parallel environment. At the moment of writing this documentation, the following models are available: `threads`, `openmpi`, `intelmpi`, `srunmpi`, `default`. Since this option comes directly from QCG-PilotJob, the detailed description of the particular models is available in the QCG Pilot Job documentation

3.6 Processing schemes

EasyVVUQ-QCGPJ allows to process tasks in a few predefined schemes which differ in both the scope of covered EasyVVUQ steps as well as the order of submission and the way of processing of tasks by QCG-PilotJob.

Below we shortly describe the seven currently supported schemes, making the use of some kind of visual representation. Firstly, let's assume that we have a set of EasyVVUQ samples marked as s1, s2, ..., sN. Then:

`STEP_ORIENTED` in this scheme tasks are submitted in a priority of STEP; we want to complete encoding step for all samples and then go to the execution step for all samples. This scheme is as follows:

```
encoding(s1)->encoding(s2)->...->encoding(sN)->execution(s1)->execution(s2)->.
..->execution(sN)
```

`STEP_ORIENTED_ITERATIVE` this scheme is similar to `STEP_ORIENTED` in a sense that the tasks are submitted in a priority of STEP, but here we make use of iterative tasks of QCG-PilotJob to execute all operation within a STEP in a single iterative task (internally consisted of many iterations). This scheme can be expressed as follows:

```
encoding_iterative(s1, s2, ..., sN)->execution_iterative(s1, s2, ..., sN)
```

`SAMPLE_ORIENTED` in this scheme the tasks are submitted in a priority of SAMPLE; in other words we want to complete whole processing (encoding and execution) for a given sample as soon as possible and then go to the next sample. This scheme can be written as follows:

```
encoding(s1)->execution(s1)->encoding(s2)->execution(s2)->...
->encoding(sN)->execution(sN)
```

`SAMPLE_ORIENTED_CONDENSED` it is similar scheme to `SAMPLE_ORIENTED`, but the encoding and execution are *condensed* into a single PJ task. It could be expressed as:

```
encoding&execution(s1)->encoding&execution(s2)->...->encoding&execution(sN)
```

SAMPLE_ORIENTED_CONDENSED_ITERATIVE this type employs iterative tasks to run *condensed* encoding and execution. This is similar to `SAMPLE_ORIENTED_CONDENSED`, but here encoding&execution tasks are a part of iterative task. It could be expressed as:

```
encoding&execution_iterative(s1, s2, ..., sN)
```

EXECUTION_ONLY instructs to submit only the `EXECUTION` tasks assuming that the encoding step is executed outside QCG-PilotJob. It could be written as follows:

```
execution(s1)->execution(s2)->...->execution(sN)
```

EXECUTION_ONLY_ITERATIVE the variation of scheme to submit only the `EXECUTION` tasks, but in contrast to the `EXECUTION_ONLY` scheme, here an iterative QCG-PilotJob task is used to run all tasks. It could be written as follows:

```
execution_iterative(s1, s2,... sN)
```

The schemes use different task types that need to be added to Executor in order to allow processing:

- The `SAMPLE_ORIENTED`, `STEP_ORIENTED``and ``STEP_ORIENTED_ITERATIVE` schemes require `ENCODING` and `EXECUTION` tasks.
- The `EXECUTION_ONLY` and `EXECUTION_ONLY_ITERATIVE` schemes require `EXECUTION` task.
- The `SAMPLE_ORIENTED_CONDENSED` and `SAMPLE_ORIENTED_CONDENSED_ITERATIVE` require `ENCODING_AND_EXECUTION` task.

The efficiency of the schemes may significantly differ depending on use case and resource requirements defined for execution of both the whole PilotJob and the individual task types. For many scenarios the iterative schemes could run a bit better, but there is no general rule of thumb that says so, and therefore we encourage you to test different schemes when the efficiency is priority.

3.7 Passing the execution environment to QCG-PilotJob tasks

Since every QCG-PilotJob task is started in a separate process, it needs to be properly configured to run in an environment consistent with the requirements of the parent script. On the one hand, EasyVVUQ allows to easily recover information about the campaign from the database, but some environment settings, such as information about required environment modules or virtual environment, have to be passed in a different way. To this end, EasyVVUQ-QCGPJ delivers a simple mechanism based on an idea of bash script, that is sourced by each task prior to its actual execution. The path to this file can be provided in the `EQI_CONFIG` environment variable. If this environment variable is available in the master script, it is also automatically passed to QCG-PilotJob tasks.

To the large extent the structure of the script provided in `EQI_CONFIG` is fully custom. In this script a user can load modules, set further environment variables or even do simple calculations. The content can be all things that are needed by a Task in prior of its actual execution. Very basic example of the `EQI_CONFIG` file may look as follows:

```
#!/bin/bash

module load openmpi/4.0
```

Note: The alternate option to provide the configuration file is to specify its location by the `config_file` parameter provided into the constructor of the `Executor` object.

3.8 Resume mechanism

EQI is able to resume not completed workflow of tasks submitted to QCG-PilotJob Manager (for example terminated because of the walltime crossing). By default the resume mechanism is activated automatically when Executor is inited with the campaign for which EQI processing was already started (working directory exists) but it is not yet completed. If this behaviour is not intended, the resume mechanism can be disabled with providing `resume=False` parameter to the `Executor's` constructor.

The resumed workflow will start in a working directory of the previous, not-completed execution. This is fully expected behaviour, but since the partially generated output or intermediate files can exists, they need to be carefully handled. EQI tries to help in this matter by providing mechanisms for automatic recovery of individual tasks.

How much the automatism can interfere with the resume logic depends on a use case and therefore EQI provides a few `ResumeLevels` of automatic recovery. The levels can be set in the `Task`'s constructor with the `resume_level` parameter. There are the following options available:

`DISABLED` Automatic resume is fully disabled for a task.

`BASIC` For the task types creating run directories (`ENCODING`, `ENCODING_AND_EXECUTION`), the resume checks if an unfinished task created run directory. If such directory is available, this directory is recursively removed before the start of the resumed task.

`MODERATE` This level processes all operations offered by the `BASIC` level, and adds the following features. At the beginning of a task's execution, the list of directories and files in a run directory is generated and stored. The resumed task checks for the differences and remove new files and directories in order to resurrect the initial state.

Please note that this functionality may be not sufficient for more advanced scenarios (for example if input files are updated during an execution) and those for which the overhead of the built-in mechanism is not acceptable. In such cases, the more optimal logic of resume may need to be provided on a level of the actual code of a task.

3.9 External Encoders

EasyVVUQ allows to define custom encoders for specific use cases. This works without any issues as long as we are in a single process. However, in case we want to execute the encoding in a separate processes, there is a need to instruct these processes about the encoder. This information is partially available in the Campaign itself and can be recovered, but we need to somehow instruct EasyVVUQ-QCGPJ code to import required python modules for the encoder. To this end once again we make use of environment variable - this time `ENCODER_MODULES`. The value of this variable should be the semicolon-separated list of the modules names, which are required by the custom encoder. The modules will be dynamically loaded before the encoder is recovered, what resolves the problem. In order to use `ENCODER_MODULES` variable we propose to define it in the `EQI_CONFIG`

An example configuration of `EQI_CONFIG` that includes specification of custom `ENCODER_MODULES` may look as follows (for the full test case please look in `tests/custom_encoder`):

```
#!/bin/bash

# WORKS ONLY IN BASH - SHOULD BE CHANGED (EG. TO GLOBAL PATHS) IN CASE OF OTHER␣
↪INTERPRETERS
this_dir="$( cd "$( dirname "${BASH_SOURCE[0]}" )" >/dev/null 2>&1 && pwd )"
this_file=$(basename "${BASH_SOURCE[0]}")

PYTHONPATH="${PYTHONPATH}:${this_dir}"
ENCODER_MODULES="custom_encoder"
export PYTHONPATH
```

(continues on next page)

(continued from previous page)

```
export ENCODER_MODULES

export EQI_CONFIG=$this_dir/$this_file
```

CHAPTER 4

Performance optimisation hints

There are many factors that influence on the performance of EQI. This section presents some guidance on optimisation of EQI. However, since the scenarios are very different the presented information should be considered only as hints that can help in optimisation, but they are not a ready-to-use recipes.

4.1 Performance-aware usage of QCG-PilotJob

Firstly, the performance of EQI is naturally limited by the performance of QCG-PilotJob. At this level, the usage of `ITERATIVE` versions of tasks is preferred as it minimises communication overhead related to interaction with QCG-PilotJob Manager. The other element that is related to QCG-PilotJob and should be taken into account and may be beneficial for larger executions is the reservation of a dedicated core for QCG-PilotJob Manager. However, in general, it should be noted that the QCG-PilotJob performance may cause a problem only for extremely demanding scenarios. For the typical use cases, there are other aspects that possibly play more important role.

4.2 Tasks fitting in allocation

The critical element for good performance of EQI is to ensure good fitting of the tasks to the size of allocation so there are no empty slots during the execution. For example, it would be very inefficient to have 10 cores in allocation and execute only tasks requiring 6 cores. Then 4 cores would be empty all the time. Much more optimal would be to execute tasks that require 5 cores so two task could be executed in parallel. Thus the starting point in the process of fitting may be just setting of each task's size to be a divider of the allocation size. In the advanced scenarios however, this may be not sufficient. Please note that in EQI there are different types of tasks and different processing schemes that submit these tasks in specific orders, so the allocation of tasks may be more challenging to ensure good results. In consequence both the allocation size, tasks sizes and processing scheme selection are all variables that need to be determined through scenario analysis and pre-production tests. Please be also aware that optional reservation of a core for QCG-PilotJob Manager naturally reduces a number of available cores for tasks, thus it should be taken into account during the analysis.

4.3 Workflow splitting

The basic way of usage of EQI goes down to modification of few lines in a typical EasyVVUQ workflow and execution of the modified workflow on a computing cluster. This is easy, but it should be noted that the whole workflow will be executed in an allocation that can consist of many computing nodes. Likely not all steps of EasyVVUQ require HPC power or may be done in parallel (e.g. sampling and analysis are typically done serially) and therefore in some use cases it may be more optimal to split the workflow into parts executed on HPC machines and those executed locally (or in different, smaller allocation). EasyVVUQ provides the save / load mechanism for the Campaign based on a state file, that can be used to resurrect the workflow when some calculations have been already made in a different location. In particular, this optimisation should be considered when there is a large allocation and relatively long processing time of sampling / analysis

Going forward with this issue, also encoding tasks can be extracted from the processing in a large allocation. Please note however that possible inefficiency is here relatively small since encoding tasks can be executed in parallel over the whole allocation (unless the `EXECUTION_ONLY` processing scheme is not selected).

4.4 Resume mechanism settings

The resume mechanism, and particularly its level, can influence on task's performance. If the risk of interruption of a workflow can be accepted or if resume mechanism is provided by application itself, the automatic resume mechanism of EQI may be set on `BASIC` level or even switched-off completely.

4.5 Logging and output generation

When there is a huge number of tasks even relatively rare writes to disk may cause a problem. Therefore it may be beneficial to turn logging into less descriptive type or limit a number of output messages. This applies to the logging in EQI, but also to any code that is executed inside a task.

CHAPTER 5

Demonstration on efficient, parallel Execution of EasyVVUQ with QCG-PilotJob Manager on local and HPC resources (a step-by-step guide)

5.1 Preface

In this tutorial, you will get a step-by-step guidance on the usage of several VECMAtk components to perform uncertainty quantification calculations within a local and HPC execution environment. A simple numerical model that simulates the temperature of a coffee cup under the Newton's law of cooling is provided here as an example application, but the general scheme of conduct can be practiced in any application. To show that, we will also discuss the usage of VECMAtk to quantify uncertainties in a multiscale fusion application. In this tutorial you will learn about the following VECMA software components:

- EasyVVUQ - a Python3 library that aims to facilitate verification, validation and uncertainty quantification,
- QCG-PilotJob - a Pilot Job system that allows to execute many subordinate jobs in a single scheduling system allocation,
- EasyVVUQ-QCGPJ - a lightweight integration code that simplifies usage of EasyVVUQ with a QCG-PilotJob execution engine,
- QCG-Client - a command line client for execution of computing jobs on the clusters offered by QCG middleware,
- QCG-Now - a desktop, GUI client for easy execution of computing jobs on the clusters offered by QCG middleware.

5.2 Contents

5.3 Introduction

As the performance of supercomputers becomes more powerful, it also turns into a driving force for the science and engineering communities to construct computational models of higher complexities. These models can help explore sciences that were previously restricted by the computing powers of older-generation computers. However, are these complex computational models reliable? Are their calculations comparable to experimental measurements? Any simulation model, regardless of its level of complexity, becomes more robust if verified, validated, and minimized on uncertainties. Hence, uncertainty quantification becomes one of the central objectives in computational modelling. As defined in the VECMA glossary[1] , uncertainty quantification UQ is a "discipline, which seeks to estimate the uncertainty in the model input and output parameters, to analyse the sources of these uncertainties, and to reduce their quantities." However, this process can quickly become cumbersome because just a few uncertain inputs could require hundreds or even thousands of samples. Such a number of tasks cannot be performed effectively without (1) adequate computational resources, (2) a dedicated approach and (3) specialised programming solutions.

In light of the aforementioned increase in availability of computing power, there is also an increase in operating cost of large data centers. Therefore, more emphasis must be placed on developing the appropriate mechanisms and solutions that enable effective execution of calculations and yet follow the administrative policies of the resource providers. Therefore, to address the requirements of UQ analysis and technological concerns we have integrated EasyVVUQ with QCG-PilotJob Manager in the VECMAtk to offer users a complete solution for performing highly intensive UQ studies on the HPC resources of peta- and in the future exa-scales. This solution allows users to submit the entire workflow as a single job into a HPC cluster and thus avoids the limitations and restrictions imposed by the administrative policies of resource providers. Inside the resource allocation created for a single job, QCG-PilotJob Manager deals with the execution of a potentially very high number of subjobs in an automatic, flexible and efficient way. Although QCG-PilotJob Manager is designed to support execution of complex computing tasks on HPC clusters, it can also be used on a local computer, allowing users to conveniently test their execution scenarios prior to the actual production runs using the same programming and execution environment.

The tutorial is structured as follows: first, we provide a description to a simple numerical model that serves as an example application in the tutorial, then we provide instruction on how to install the EasyVVUQ-QCGPJ component of the VECMAtk and other essential software tools. The tutorial materials download information is also included. Next, we provide a glimpse into the structure of EasyVVUQ-QCGPJ workflow, followed by instructions into the configuration on environment-specific settings. Then, we showcase 4 different approaches (local, SLURM, QCG-Client, and QCG-Now) you can choose from to execute EasyVVUQ on the sample application, all under the management of the QCG-PilotJob. For any reader who is interested in learning more about UQ applied to a multiscale workflow, a section describing the fusion model is positioned at the end of the tutorial.

[1] https://wiki.vecma.eu/glossary

Notice 1: The tutorial contains some steps related to the execution of EasyVVUQ / QCG-PilotJob task via queuing system and/or QCG access tools. To follow these steps you must have an account with a computing cluster controlled by SLURM and if you want to use QCG tools it has to be part of the QCG infrastructure. In order to get access to Eagle cluster at Poznan Supercomputing and Networking Center, which is available with Slurm and QCG, please drop an e-mail with a short motivation to VECMA infrastructure's leader - Tomasz Piontek: piontek_at_man.poznan.pl.

5.4 Application model for the tutorial

To give readers a sense of how EasyVVUQ-QCGPJ works, we provide a simple cooling coffee cup model as a test application throughout the entire tutorial. This allows users to quickly grasp the concept behind the model so they can put their attention towards the functionality of EasyVVUQ with QCG-PilotJob, and how the toolkit assists users with the process of UQ on their numerical model. In reality, many types of numerical codes can also benefit from EasyVVUQ with QCG-PilotJob. Multiscale fusion modeling, for example, uses the same software to apply UQ. To learn more about the multiscale fusion model and how the toolkit helps in quantifying uncertanties, please refer to the last section of the tutorial.

The sample physics model in this tutorial is inspired by the "cooling coffee cup model" from[2]. A cup of coffee is placed inside some environment of temperature *Tenv*. Consequently, the cup of coffee experiences heat loss and its temperature *T* varies in time *t*, as described mathematically by the Newton's law of cooling:

dT(t)/dt=-K(T(t) - Tenv),

where *K* is a constant that describes the system. The python script cooling_model.py, which is provided as part of the tutorial materials, takes the initial coffee temperature *T0*, *K* and *Tenv* and solve the above equation to find *T*. Since there are uncertainties to the inputs *K* and *Tenv*, the goal is to take the uncertain inputs into consideration when obtaining the probability distribution of the measured value *T*. Please note that, from this point forward, all quantities will be mentioned without explicit units.

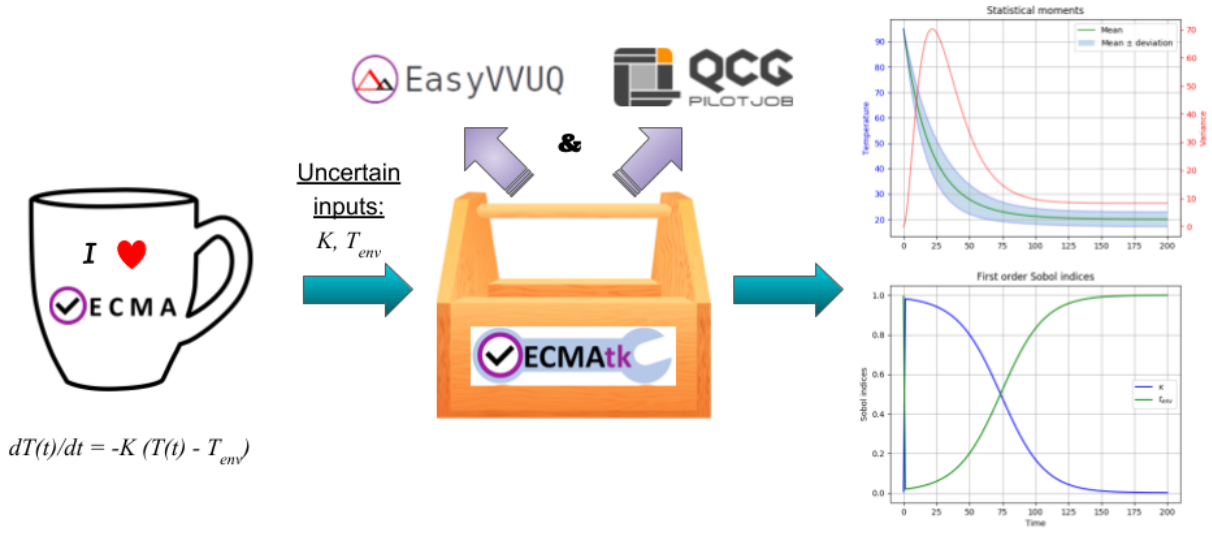
We begin the UQ calculations to the model by defining lower and upper threshold values to a uniform distribution for both uncertain inputs:

0.025 <= K <= 0.075, and

15.0 <= Tenv <= 25.0.

The initial coffee temperature *T0* is set to be *95.0*, and the calculation runs from *t=0* to *t=200*. At the end of the simulation, we defined two extra parameters *Te* and *Ti*, with *Te* identically equal to *T* and *Ti* identically equal to *-T*. We select the Polynomial Chaos Expansion[3] PCE method with 1st order polynomial, which would result in (1 + 2)2 or 9 sample runs. A python script is provided in the tutorial material "test_cooling_pj.py", showcasing how EasyVVUQ-QCGPJ takes the input parameters and handle all sample calculations in an efficient manner, and provides statistical analysis to the outputs *T(t)* (i.e. mean, standard deviation, variance, Sobol indices[4]). Here is a schematic depicting the entire UQ procedure described above.

[2] https://uncertainpy.readthedocs.io/en/latest/examples/coffee_cup.html
[3] https://en.wikipedia.org/wiki/Polynomial_chaos
[4] https://en.wikipedia.org/wiki/Variance-based_sensitivity_analysis



UQ of the cooling coffee cup model: the EasyVVUQ-QCGPJ of the VECMAtk takes the uncertain inputs and produces statistical analysis to *T(t)*. The plots on the right are the calculated average temperature, standard deviation, and variance (top plot); and the first order Sobol indices for the uncertain input parameters *K* and *Tenv* (bottom plot).

The rest of the tutorial will guide you through the toolkit installation and execution of this model. Before "running test_cooling_pj.py", please be sure to check all parameters and make changes accordingly.

5.5 Installation of EasyVVUQ-QCGPJ

1. If you are going to work remotely on a cluster, please login into access node and start an interactive SLURM job (we are doing it on Eagle cluster, which is a part of the VECMA testbed).

```
$ ssh user@eagle.man.poznan.pl
$ srun -n 1 --time=2:00:00 --partition=plgrid --pty /bin/bash
```

2. Be sure that **Python 3.6+** and **pip 18.0.1+** are installed and available in your environment. In case of Eagle cluster use the module for the newest version of the python.

```
$ python3 -V Python
3.6.6
$ module load python/3.7.3
$ python3 -V Python 3.7.3
```

3. Add `~/.local/bin` to your `$PATH` environment variable (if it is not yet already there) and make it permanent by updating the `.bashrc` file.

```
$ export PATH=/home/plgrid/user/.local/bin:$PATH
$ echo 'PATH=/home/plgrid/user/.local/bin:$PATH' >> .bashrc
```

4. Check if *virtualenv* is installed on your system and if not install it.

```
$ virtualenv --version
bash: virtualenv: command not found
$ pip3 install --user virtualenv
Collecting virtualenv
  Downloading https://files.pythonhosted.org/packages/ca/ee/
↪8375c01412abe6ff462ec80970e6bb1c4308724d4366d7519627c98691ab/virtualenv-16.6.0-
↪py2.py3-none-any.whl (2.0MB)
    100% || 2.0MB 2.0MB/s
Installing collected packages: virtualenv
  The script virtualenv is installed in '/home/plgrid/user/.local/bin' which is␣
↪not on PATH.
  Consider adding this directory to PATH or, if you prefer to suppress this␣
↪warning, use --no-warn-script-location.
Successfully installed virtualenv-16.6.0
$ virtualenv --version
16.6.0
```

5. Create *virtualenv* for the EasyVVUQ with QCG-PilotJob support:

```
$ virtualenv ~/.virtualenvs/easyvvuq-qcgpj
Using base prefix '/opt/exp_soft/local/generic/python/3.7.3'
New python executable in /home/plgrid/user/.virtualenvs/easyvvuq-qcgpj/bin/
↪python3.7
Also creating executable in /home/plgrid/user/.virtualenvs/easyvvuq-qcgpj/bin/
↪python
Installing setuptools, pip, wheel...
done.
```

6. Activate this virtualenv:

```
$ . ~/.virtualenvs/easyvvuq-qcgpj/bin/activate
(easyvvuq-qcgpj) user@e0192:~$
```

7. Install *EasyVVUQ*, *QCG-PilotJob* and the *EasyVVUQ-QCGPJ* packages using pip3
   (Note: if you are not able to use pip in your environment you can always install all required packages manually as they are publicly available, e.g. by cloning repositories for missing packages and invoking python3 `setup.py install for each one` - take a look for the requirements here: https://github.com/vecma-project/EasyVVUQ-QCGPJ/blob/master/setup.py)

```
(easyvvuq-qcgpj)$ pip3 install easyvvuq
(easyvvuq-qcgpj)$ pip3 install qcg-pilotjob
(easyvvuq-qcgpj)$ pip3 install easyvvuq-qcgpj
```

5.6 Getting the tutorial materials

1. Create directory for the tutorial

```
$ mkdir tutorial
```

2. The materials used in this tutorial are available in GitHub EasyVVUQ-QCGPJ repository. Get them with the following commands:

```
$ cd ~/tutorial
$ git clone https://github.com/vecma-project/EasyVVUQ-QCGPJ.git
$ cp EasyVVUQ-QCGPJ/tutorials/cooling_cup .
```

After invoking these commands all the tutorial files should be available in the `~/tutorials/cooling_cup` folder

5.7 Execution of EasyVVUQ with QCG-PilotJob

In this tutorial we describe 4 ways to execute EasyVVUQ with QCG-PilotJob:

1. Local execution
2. With SLURM
3. With QCG-Client
4. With QCG-Now

Each method has its own advantages and disadvantages. The local execution can be easily performed on a laptop and instantly provide an overview to users. The execution using SLURM, similar to the execution with QCG-Client, may be useful for those who are using queuing system on a daily manner. The execution with QCG-Now could be an interesting option for those who prefer GUI and the automatized access to resources.

In the rest of this tutorial, the overall structure of the EasyVVUQ-QCGPJ workflow is discussed before the step-by-step instructions are presented for each method of execution. The eventual choice of method should be based on the user's preferences and requirements.

5.7.1 EasyVVUQ-QCGPJ workflow

The approach we took to integrate EasyVVUQ with QCG-PilotJob Manager is considerably non-intrusive. The changes we introduced to the EasyVVUQ workflow itself are small and mainly concentrated at the encoding and application execution steps, thus the overhead needed to plug-in QCG-PilotJob into the basic workflow is negligible. The integration code provides a generic mechanism that could easily be adapted by different application teams to quantify uncertainties of their codes. In this section we briefly describe the main parts of a workflow used in the tutorial. For the extensive reference to how EasyVVUQ-QCGPJ works, please go to the other documentation material available at: https://easyvvuq-qcgpj.readthedocs.io

The workflow constructed for uncertainty quantification of a cooling coffee cup is available in:

`~/tutorials/cooling_cup/app/test_cooling_pj.py`

Considerably simplified, it looks as follows:

```
def test_cooling_pj():

    # Set up a fresh campaign called "cooling"
    my_campaign = uq.Campaign(name='cooling', work_dir=tmpdir)

    # ...
    # Skipped code that initialises the campaign, sets up the application
    # and generates samples for the use-case.
    # ...

    # Create EasyVVUQ-QCGPJ Executor that will process the execution
    qcgpjexec = Executor(my_campaign)

    # Create QCG-PilotJob Manager with 4 cores (if you want to use all available␣
↪resources remove the resources parameter)
```

(continues on next page)

(continued from previous page)

```
    # Refer to the documentation for customisation options.
    qcgpjexec.create_manager(resources='4')

    # Define ENCODING task that will be used for execution of encodings using␣
↪encoders specified by EasyVVUQ.
    # The presented specification of 'TaskRequirements' assumes the execution of each␣
↪of the tasks on 1 core.
    qcgpjexec.add_task(Task(
        TaskType.ENCODING,
        TaskRequirements(cores=1)
    ))

    # Define EXECUTION task that will be used for the actual execution of application.
    # The presented specification of 'TaskRequirements' assumes the execution of each␣
↪of the tasks on 2 cores,
    # but for more demanding, parallel applications the resources requirements may be␣
↪extended to many cores or
    # even many nodes.
    # Each task will execute the command provided in the 'application' parameter.
    qcgpjexec.add_task(Task(
        TaskType.EXECUTION,
        TaskRequirements(cores=2),
        application='python3 ' + APPLICATION + " " + ENCODED_FILENAME
    ))

    # Execute encodings and executions for all generated samples
    qcgpjexec.run(processing_scheme=ProcessingScheme.SAMPLE_ORIENTED)

    # Terminate QCG-PilotJob Manager
    qcgpjexec.terminate_manager()

    # The rest of typical EasyVVUQ processing (collation, analysis)
```

We can distinguish the following key elements from this script:

- Typical initialisation of a Campaign and generation of samples.
- Instantiation of EasyVVUQ-QCGPJ Executor.
- Set up of the QCG-PilotJob Manager instance using the Executor's `create_manager` method.
- Definition of tasks for Encoding and Execution steps of EasyVVUQ that will be executed as QCG-PilotJob tasks. Each definition of task includes the specification of resource requirements that the task consume.
- Parallel processing of the encodings and executions with QCG-PilotJob using a predefined scheme of processing (`ProcessingScheme`).
- Termination of QCG-PilotJob Manager using the Executor's `terminate_manager` method.
- The collation and analysis made in a typical way, unperturbed from the EasyVVUQ script.

What is worth stressing is the fact that both the presented workflow and EasyVVUQ-QCGPJ's API are generic enough such that the majority of applications can either use the presented code directly, or make small adjustments according to the specific needs of use cases. For example, we can imagine that for some applications all encoding steps have to be executed before the first execution step begins. In that case, the only required modification is to change the value of `ProcessingScheme` from `ProcessingScheme.SAMPLE_ORIENTED` to `ProcessingScheme.STEP_ORIENTED`.

5.7.2 Common configuration before execution

1. Please check and update if needed the content of environment configuration file located in: `~/tutorials/cooling_cup/app/eqi_conf.sh`. This file is used to configure system-specific settings for the developed workflow. Once you open this file, make sure the appropriate environment modules are loaded and *virtualenv* is activated. Please also check if the settings related to the environment variables, particularly `COOLING_APP` and `SCRATCH` shouldn't be adapted to the currently used environment. If this is the case modify them appropriately.
2. Source the configuration file. Once sourced, it should activate virtualenv:

```
$ . ~/tutorials/cooling_cup/app/eqi_config.sh
(easyvvuq-qcgpj)$
```

5.7.3 Local execution

1. Be sure that you have sourced the `eqi_conf.sh` file and are in the proper *virtualenv*.
2. Go into the `~/tutorials/cooling_cup/local_execution`:

```
(easyvvuq-qcgpj)$ cd ~/tutorials/cooling_cup/local_execution
```

3. Execute the workflow:

   (Note that for the local execution we are using a slightly modified version of the core workflow ((not from the `../app` folder)) - since we may test this workflow on a local computer without the queuing system allocation, we define 4 virtual cores to demonstrate how QCG-PilotJob Manager executes tasks in parallel. However, be aware: when Pilot Job Manager is started as an interactive task in the allocation created by Slurm, it will override the settings of virtual resources by the actually allocated real resources. Thus, in order to test parallel execution on a cluster, you need to allocate at least 2 cores for your interactive job. Be aware that the amount of allocated resources should be larger than the requirements of any of the tasks, otherwise the demanding tasks will be blocked in the queue).

```
(easyvvuq-qcgpj)$ python3 test_cooling_pj.py
```

4. When processing completes, check results produced by EasyVVUQ.

5.7.4 Execution using SLURM

This execution is possible only on a cluster with the SLURM queuing system. In this tutorial we assume that EasyVVUQ-QCGPJ has been configured on the Eagle cluster in the way as described in the section Installation of EasyVVUQ-QCGPJ and the tutorial files has been cloned into the `~/tutorial/VECMAtk`.

1. Go into the `~/tutorials/cooling_cup/slurm_execution`

```
$ cd ~/tutorials/cooling_cup/slurm_execution
```

2. Adjust the SLURM job description file: `test_cooling_pj.sh`.
3. Submit the workflow as a SLURM batch job:

```
$ sbatch test_cooling_pj.sh
Submitted batch job 11094963
```

4. You can check the status of your SLURM jobs with:

```
$ squeue -u plguser
             JOBID PARTITION     NAME     USER     ST     TIME  NODES␣
↪NODELIST(REASON)
          11094963 fast          easyvvuq plguser  R      0:02  1     e0022OBID
```

5. Alternatively you can display detailed information for a concrete job:

```
$ sacct -j 11094963
       JobID    JobName  Partition    Account  AllocCPUS      State ExitCode
------------ ---------- ---------- ---------- ---------- ---------- --------
11094963     easyvvuq_+       fast   vecma2019          4  COMPLETED      0:0
11094963.ba+      batch              vecma2019          4  COMPLETED      0:0
11094963.0   .encode_R+             vecma2019          1  COMPLETED      0:0
11094963.1   .encode_R+             vecma2019          1  COMPLETED      0:0
11094963.2   .encode_R+             vecma2019          1  COMPLETED      0:0
11094963.3   .encode_R+             vecma2019          1  COMPLETED      0:0
11094963.4   .execute_+             vecma2019          1  COMPLETED      0:0
11094963.5   .execute_+             vecma2019          1  COMPLETED      0:0
11094963.6   .execute_+             vecma2019          1  COMPLETED      0:0
11094963.7   .encode_R+             vecma2019          1  COMPLETED      0:0
11094963.8   .execute_+             vecma2019          1  COMPLETED      0:0
11094963.9   .execute_+             vecma2019          1  COMPLETED      0:0
11094963.10  .encode_R+             vecma2019          1  COMPLETED      0:0
11094963.11  .encode_R+             vecma2019          1  COMPLETED      0:0
11094963.12  .encode_R+             vecma2019          1  COMPLETED      0:0
11094963.13  .execute_+             vecma2019          1  COMPLETED      0:0
11094963.14  .execute_+             vecma2019          1  COMPLETED      0:0
11094963.15  .execute_+             vecma2019          1  COMPLETED      0:0
11094963.16  .encode_R+             vecma2019          1  COMPLETED      0:0
11094963.17  .execute_+             vecma2019          1  COMPLETED      0:0
```

6. When the job completes, you can check the file `output[jobid].txt`, in which you will find the output produced by EasyVVUQ.

5.7.5 Execution with QCG-Client

This execution can be performed only on a machine with QCG-Client installed and configured to execute jobs on a cluster with SLURM queuing system. In the tutorial we assume the usage of the QCG-Client installed on qcg.man.poznan.pl and the Eagle cluster, which is a part of the PLGrid infrastructure. These two machines share the same $HOME directory where both EasyVVUQ-QCGPJ has been configured in the way described in the section Installation of EasyVVUQ-QCGPJ and the tutorial files has been cloned into the `~/tutorial/VECMAtk`.

1. Login into the machine where qcg-client is installed:

```
$ ssh user@qcg.man.poznan.pl
```

2. Go into the `~/tutorials/cooling_cup/qcg_execution`

```
$ cd ~/tutorials/cooling_cup/qcg_execution
```

3. Adjust QCG job description file: `test_cooling_pj.qcg`.

4. Submit the workflow as a QCG batch job (you may be asked to provide your personal certificate credentials):

```
$ qcg-sub test_cooling_pj.qcg
Enter GRID pass phrase for this identity:
```

(continues on next page)

(continued from previous page)

```
...
test_cooling_pj.qcg {}      jobId = J1559813849509_easyvvuq_pj_qcg_4338
```

5. You can list and check the status of QCG jobs with:

```
$ qcg-list
...
IDENTIFIER       NOTE  SUBMISSION     START   FINISH  STATUS   HOST  FLAGS ␣
↪DESCRIPTION
J1559813849509_easyvv* 06.06.19 11:39                PREPROCESSING
                                                             eagle S UP
```

6. A detailed information about the lastly submitted job can be obtained in the following way:

```
$ qcg-info
...
J1559814286855_easyvvuq_pj_qcg_5894 :
Note:
UserDN: ****
TaskType: SINGLE
SubmissionTime: Thu Jun 06 11:44:47 CEST 2019
FinishTime: Thu Jun 06 11:45:18 CEST 2019
ProxyLifetime: P24DT23H48M33S
Status: FINISHED
StatusDesc:
StartTime: Thu Jun 06 11:44:47 CEST 2019
Purged: true

Allocation:
HostName: eagle
ProcessesCount: 4
ProcessesGroupId: qcg
Status: FINISHED
StatusDescription:
SubmissionTime: Thu Jun 06 11:44:47 CEST 2019
FinishTime: Thu Jun 06 11:45:52 CEST 2019
LocalSubmissionTime: Thu Jun 06 11:44:52 CEST 2019
LocalStartTime: Thu Jun 06 11:45:02 CEST 2019
LocalFinishTime: Thu Jun 06 11:45:18 CEST 2019
Purged: true
WorkingDirectory: gsiftp://eagle.man.poznan.pl//tmp/lustre/plguser/J1559814286855_
↪easyvvuq_pj_qcg_5894_task_1559814287294_978
```

7. When the job completes, the results are downloaded to `results[JOB_ID]` directory.

5.7.6 Execution with QCG-Now

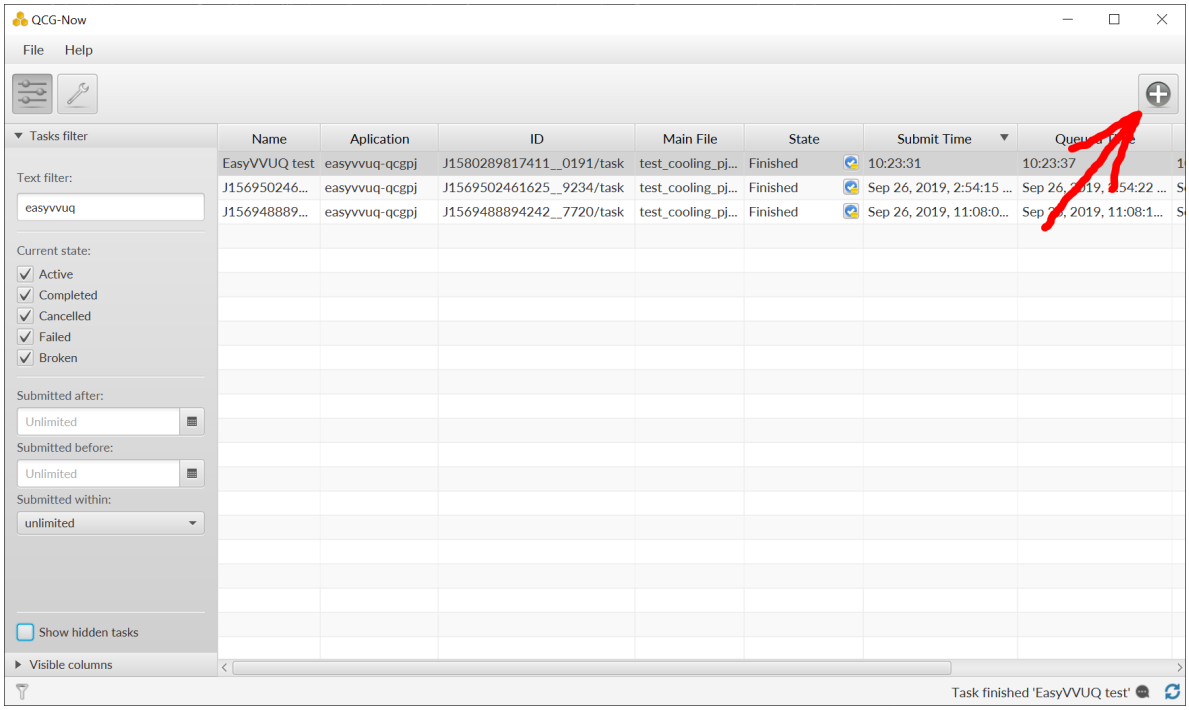
At this moment QCG-Now allows users to submit jobs to PLGrid clusters only, thus in order to use the tool, an account with PLGrid is mandatory. As before, we assume the usage of Eagle.

The installation, configuration and basic usage of QCG-Now is described here:
http://www.qoscosgrid.org/qcg-now/en/instructions/firststeps/elementary

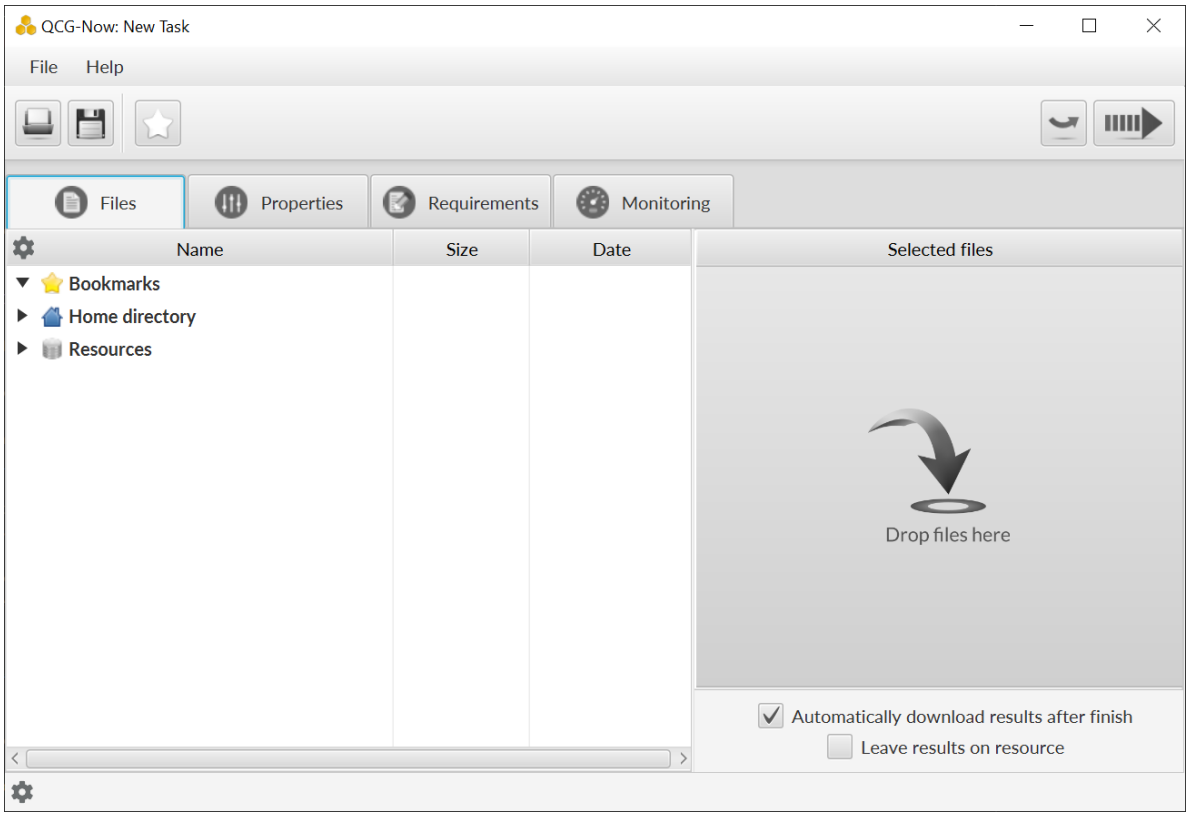
During the configuration you should select **VECMA** as a domain and then whenever QCG-Now asks about user ID/password you should provide your PLGrid credentials.

When installed and configured, the steps to submit an EasyVVUQ / QCG-PilotJob task from QCG-Now are as follows:

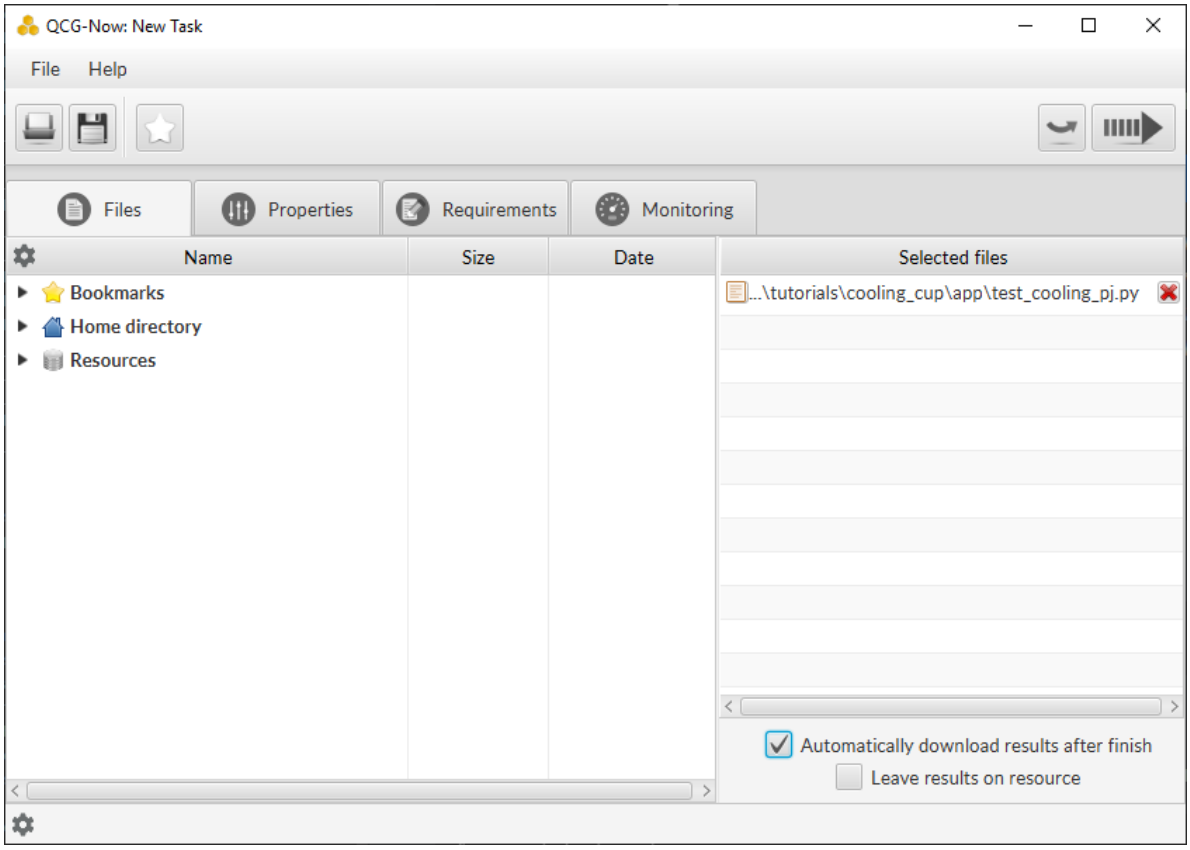
1. Get the tutorial files using GIT or download them zipped from https://github.com/vecma-project/EasyVVUQ-QCGPJ/archive/master.zip - then extract the files.
2. In the main window of QCG-Now click "+"



3. The New Task definition window should open. When you select the Files tab it should look as follows:

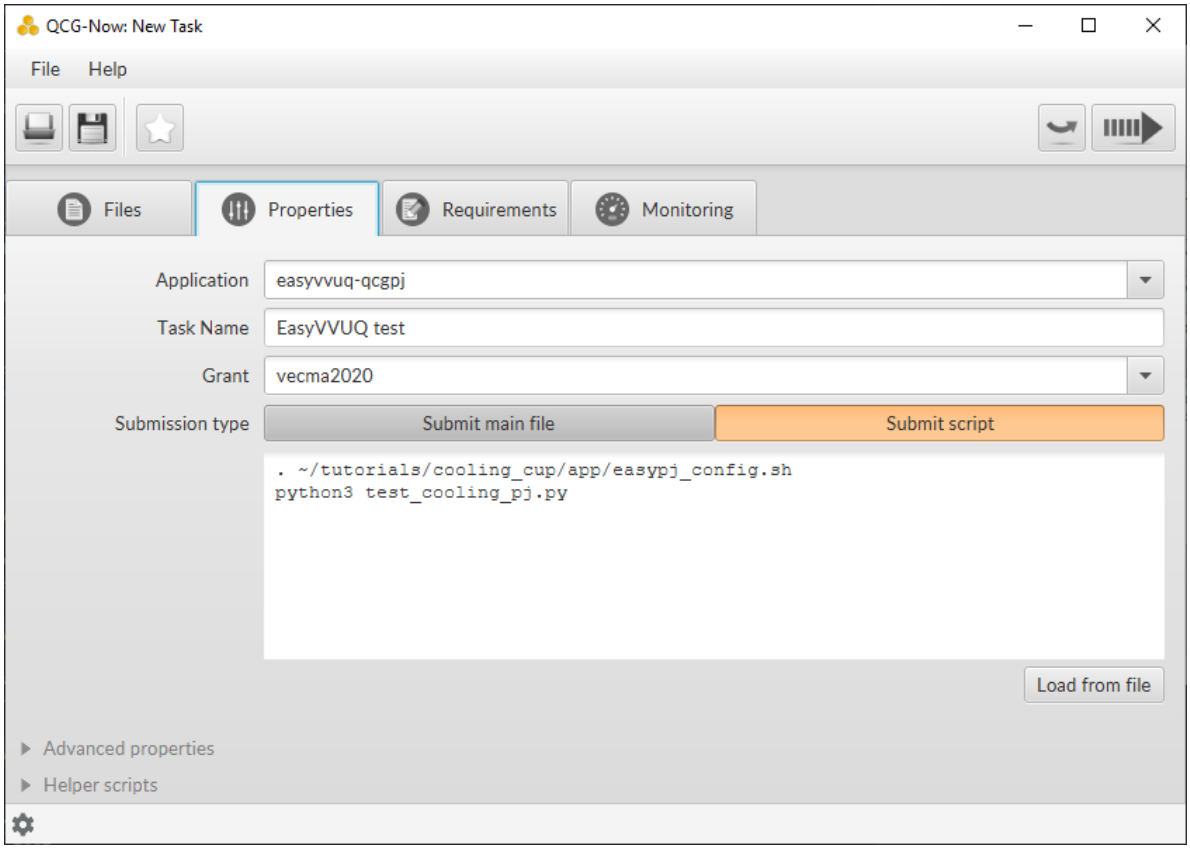


4. Drag&drop the /tutorials/cooling_cup/app/test_cooling_pj.py file from the extracted zip file into “DROP FILES HERE” space:

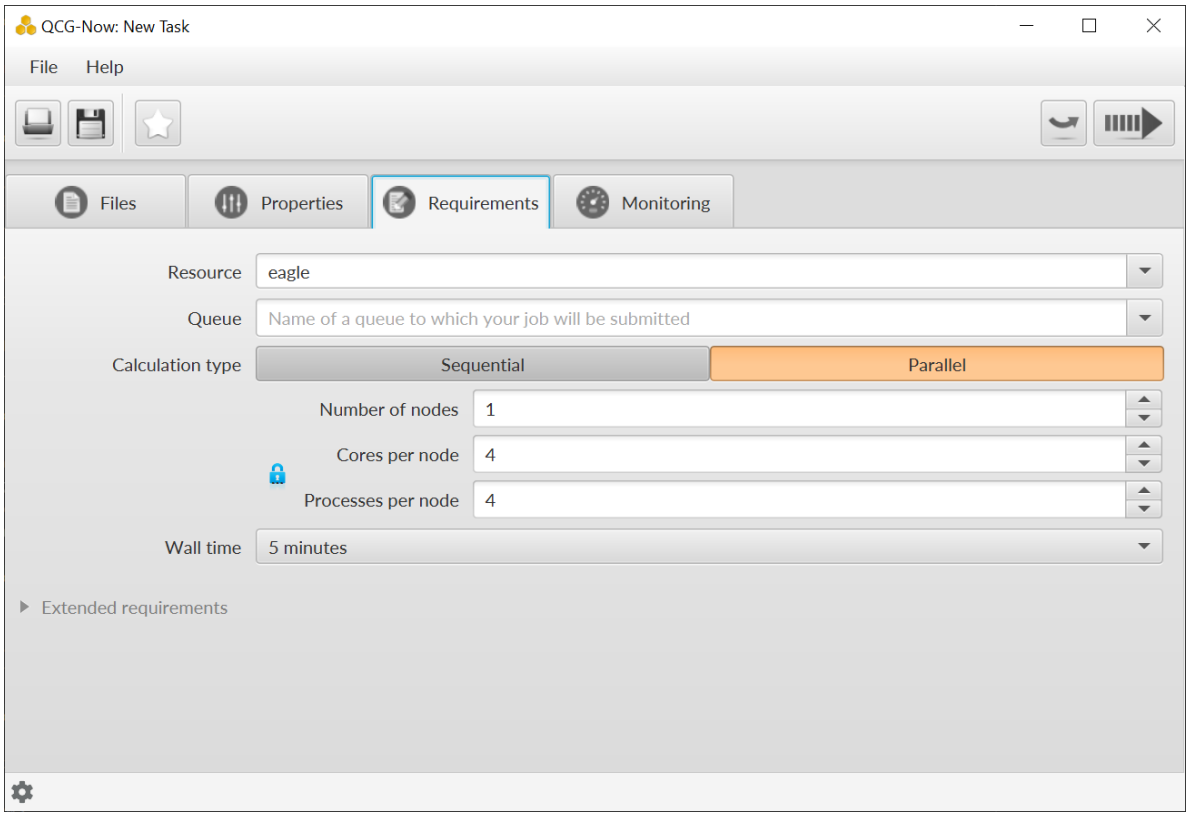


5. In the Properties tab select:
   - Application: **easyvvuq-qcgpj**
   - Task Name: EasyVVUQ test
   - Grant: leave blank to use a default one or select another
   - Submission type: **Submit script**
   - In the opened textarea write:

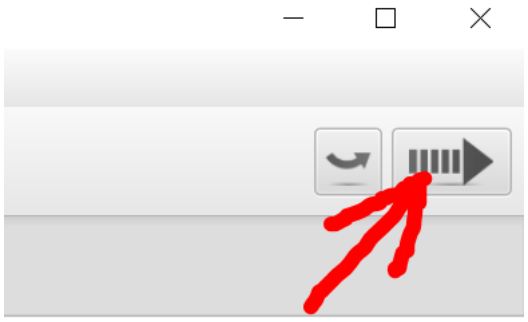
```
. ~/tutorials/cooling_cup/app/eqi_config.sh
python3 test_cooling_pj.py
```



6. In the Requirements tab select:
   - Resource: **eagle**
   - Calculation type: **Parallel** (Number of nodes: **1**, Cores per node: **4**, Processes per node **4**)
   - Walltime: **5 minutes**

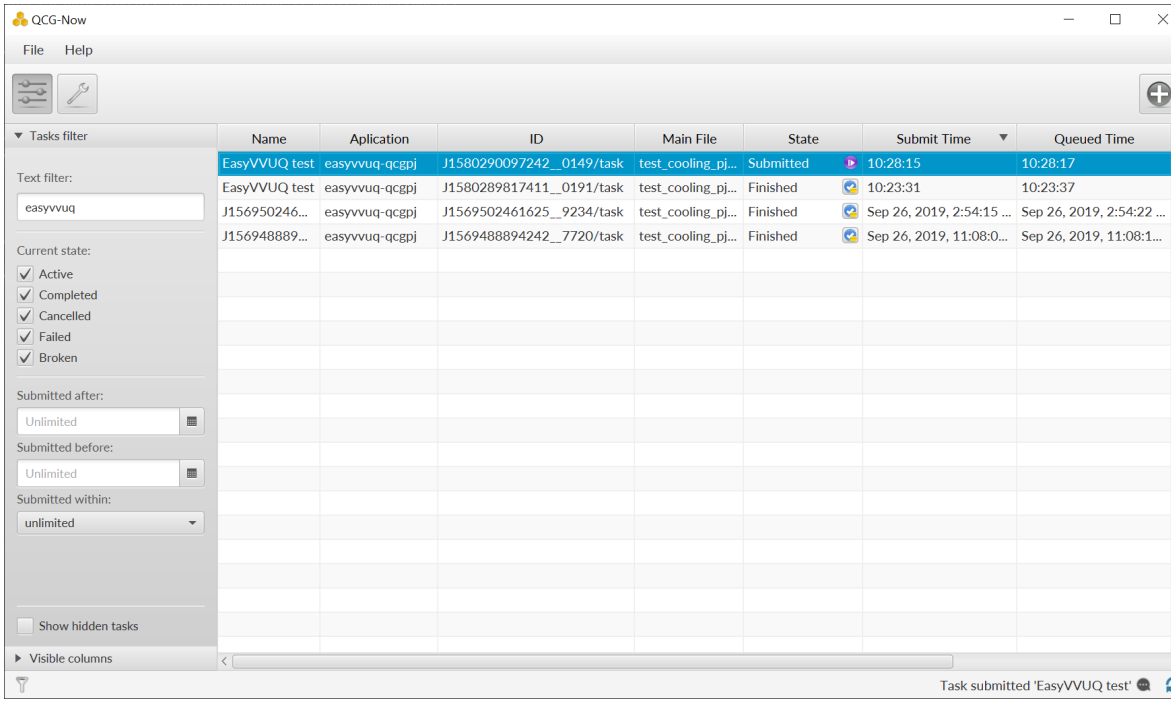


7. Click the submit button (the arrow in the top-right corner). At this moment QCG-Now initiates a data transfer to the computing resources and requests the QCG middleware for the task execution.

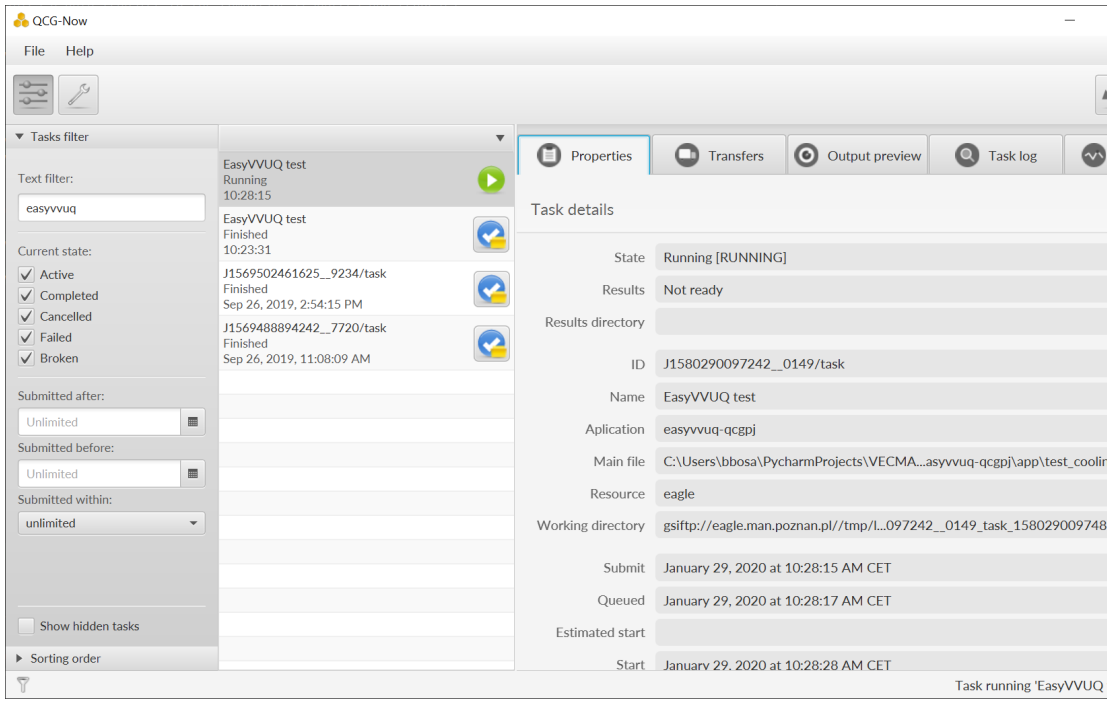


8. When submitted, the task is added to the list of tasks in the main window, where it is possible to track the state and progress of its execution in two complementary views:

The **Tabular** view:

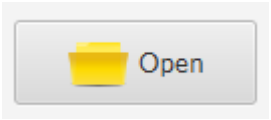


The **Task's Details** view:



The views can be switched by double-clicking on a task.

9. When the task completes successfully, the output data is transferred back to a user's computer and user can open a directory with results using one of dedicated buttons from the main window.

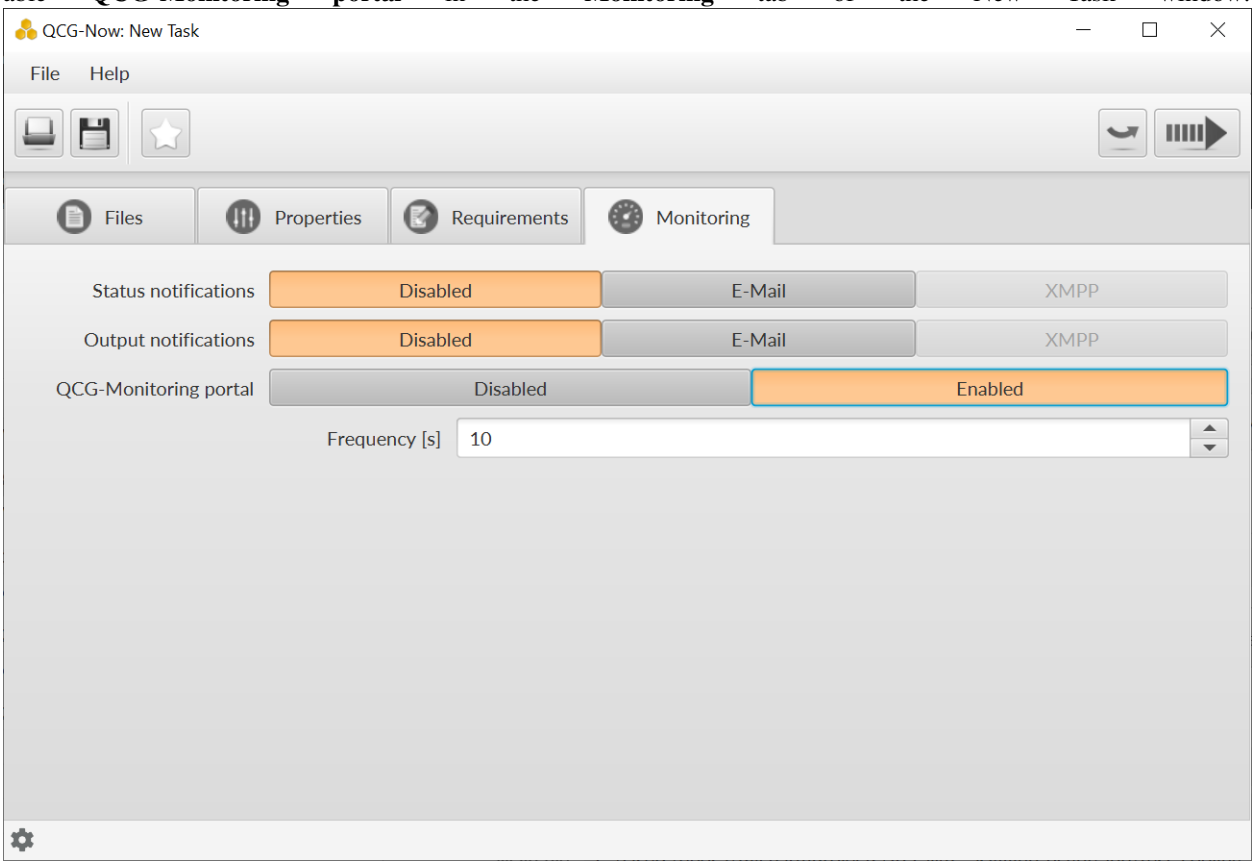


or

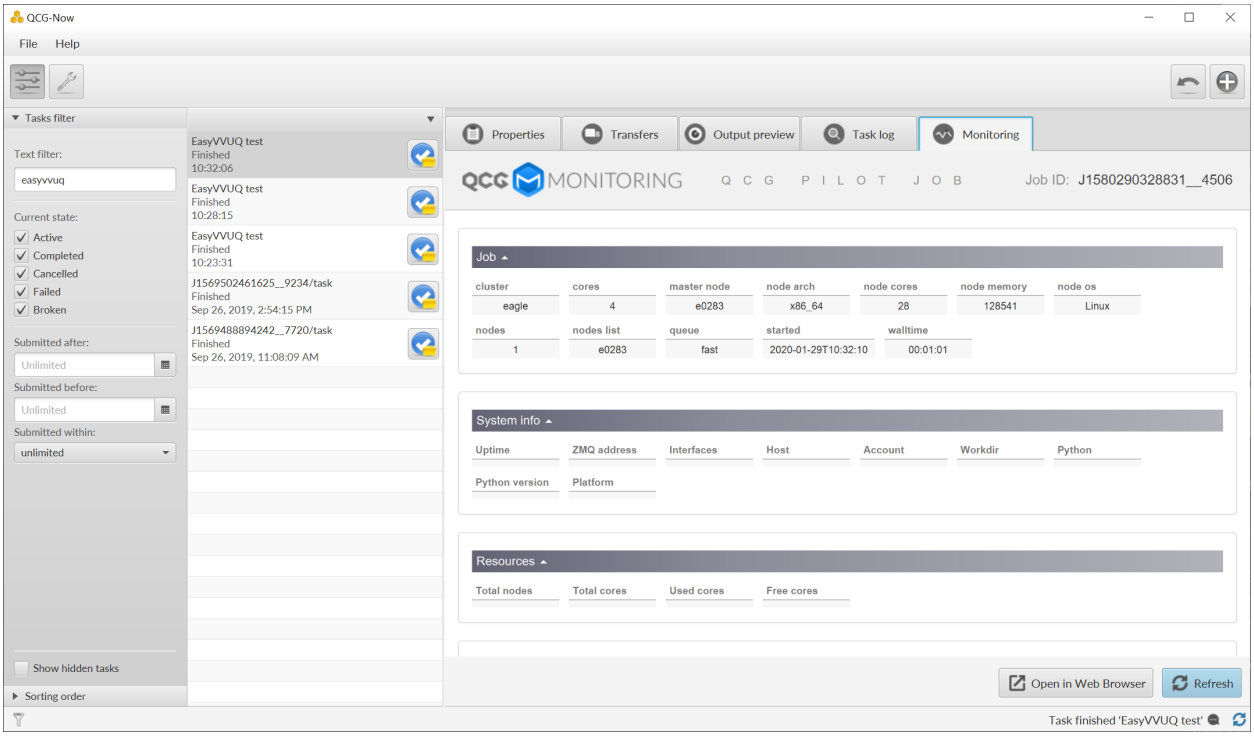
QCG-Monitoring (Experimental)

In order to provide users with the functionality of live monitoring of their tasks, QCG-Now has been experimentally integrated with the QCG-Monitoring solution. This integration allows to display basic data about the users tasks directly in QCG-Now. Currently, for the easyvvuq-qcgpj application the monitoring provides generic information about Pilot Job execution, but it will be tuned for specifics of EasyVVUQ in a near future.

In order to switch on the monitoring for a task, a user needs to enable **QCG-Monitoring portal** in the **Monitoring** tab of the New Task window:



Once the easyvvuq-qcgpj application starts it is possible to use a dedicated **Monitoring** tab of the *Task's Details* view to display monitored information:



5.8 References

CHAPTER 6

Interactive tutorial

It is possible to play with EasyVVUQ-QCGPJ using the PSNC's Jupyter Notebook platform. In order to get free access to this platform please write an e-mail to bbosak_at_man.poznan.pl.

CHAPTER 7

eqi package

class eqi.**Executor**(*campaign*, *config_file=None*, *resume=True*, *log_level='info'*)

Bases: object

Integrates EasyVVUQ and QCG-PilotJob Manager

Executor allows to process the most demanding operations of EasyVVUQ in parallel using QCG-PilotJob.

create_manager(*resources=None*, *reserve_core=False*, *enable_rt_stats=False*, *wrapper_rt_stats=None*, *log_level='info'*)

Creates new QCG-PilotJob Manager and sets is as the Executor's engine.

Parameters

- **resources** (*str, optional*) – The resources to use. If specified forces usage of Local mode of QCG-PilotJob Manager. The format is compliant with the NODES format of QCG-PilotJob, i.e.: [node_name:]cores_on_node[,node_name2:cores_on_node][,...]. Eg. to run on 4 cores regardless the node use *resources="4"* to run on 2 cores of node_1 and on 3 cores of node_2 use *resources="node_1:2,node_2:3"*
- **reserve_core** (*bool, optional*) – If True reserves a core for QCG-PilotJob Manager instance, by default QCG-PilotJob Manager shares a core with computing tasks Parameters.
- **enable_rt_stats** (*bool, optional*) – If True, QCG-PilotJob Manager will collect its runtime statistics
- **wrapper_rt_stats** (*str, optional*) – The path to the QCG-PilotJob Manager tasks wrapper program used for collection of statistics
- **log_level** (*str, optional*) – Logging level for QCG-PilotJob Manager (for both service and client part).

Returns

Return type None

set_manager(*qcgpjm*)

Sets existing QCG-PilotJob Manager as the Executor's engine

Parameters **qcgpjm** (*qcg.pilotjob.api.manager.Manager*) – Existing instance of a QCG-PilotJob Manager

Returns

Return type None

add_task(*task*)
Add a task to execute with QCG PJ

Parameters **task** (Task) – The task that will be added to the execution workflow

Returns

Return type None

run(*processing_scheme=<ProcessingScheme.SAMPLE_ORIENTED: 'Submits a workflow of EasyVVUQ operations as separate QCG PJ tasks for a sample (e.g. encoding -> execution) and then goes to the next sample'>*)
Executes demanding parts of EasyVVUQ campaign with QCG-PilotJob

A user may choose the preferred execution scheme for the given scenario.

Parameters **processing_scheme** (ProcessingScheme) – Tasks processing scheme

Returns

Return type None

print_resources_info()
Displays resources assigned to QCG-PilotJob Manager

terminate_manager()
Terminates QCG-PilotJob Manager

class eqi.**Task**(*type*, *requirements=None*, *name=None*, *model='default'*, *resume_level=<ResumeLevel.BASIC: 'For the tasks creating run directories, the resume checks if an unfinished task created such directory. If such directory is available, this is automatically removed before the start of the resumed task'>*, ***params*)
Bases: object

Represents a piece of work to be executed by QCG-PilotJob Manager

Parameters

- **type** (TaskType) –

  The type of the task. Allowed tasks are: ENCODING, EXECUTION, ENCODING_AND_EXECUTION, and OTHER (currently not supported)
- **requirements** (TaskRequirements, *optional*) – The requirements for the Task
- **name** (*str, optional*) – name of the Task, if not provided the name will take a value of type
- **model** (*str, optional*) – Allows to set the flavour of execution of task adjusted to a given resource. At the moment of writing a user can select from the following models: *threads, intelmpi, openmpi, srunmpi, default*
- **resume_level** (ResumeLevel, *optional*) – The resume level applied for a task.
- **params** (*kwargs*) – additional parameters that may be used by specific Task types

get_type()

get_requirements()

get_model()

get_resume_level()

get_params()

get_name()

class eqi.**TaskType**
Bases: enum.Enum

An enumeration.

ENCODING = 'ENCODING'

EXECUTION = 'EXECUTION'

ENCODING_AND_EXECUTION = 'ENCODING_AND_EXECUTION'

OTHER = 'OTHER'

class eqi.**ProcessingScheme**(*description*, *iterative=False*)
Bases: enum.Enum

Specifies scheme of processing of tasks with QCG-PJ

Parameters

- **description** (*str*) – Description of the ProcessingScheme
- **iterative** (*bool*) – Defines if the ProcessingScheme uses iterative tasks of QCG-PJ

STEP_ORIENTED = 'Submits specific EasyVVUQ operation (e.g. encoding) for all samples a

STEP_ORIENTED_ITERATIVE = ('Submits an iterative task for execution of specific EasyVV

SAMPLE_ORIENTED = 'Submits a workflow of EasyVVUQ operations as separate QCG PJ tasks

SAMPLE_ORIENTED_CONDENSED = 'Submits all EasyVVUQ operations for a sample as a single

SAMPLE_ORIENTED_CONDENSED_ITERATIVE = ('Submits an iterative QCG PJ task for all sampl

EXEC_ONLY = 'Submits a workflow of EasyVVUQ operations as separate QCG PJ tasks for ex

EXEC_ONLY_ITERATIVE = ('Submits an iterative QCG PJ task for all samples, where a sing

is_iterative()
Checks if ProcessingScheme makes use of iterative QCG-PJ tasks

Returns bool

Return type True if ProcessingScheme uses iterative QCG-PJ tasks

class eqi.**TaskRequirements**(*cores: Union[int, eqi.core.task_requirements.Resources, None] = None*, *nodes: Union[int, eqi.core.task_requirements.Resources, None] = None*)
Bases: object

Requirements for a task executed within QCG Pilot Job

Parameters

- **cores** (*int or eqi.Resources*) – the resource requirements for cores
- **nodes** (*int or eqi.Resources*) – the resource requirements for nodes

get_resources()
Allows to get resource requirements in a form of dictionary understandable by QCG Pilot Job Manager

Returns dict

Return type dictionary with the resource requirements specification

class eqi.**Resources**(*exact=None*, *min=None*, *max=None*, *split_into=None*)
Bases: object

Stores typical for QCG Pilot Job resource requirements

Parameters

- **exact** (*Number*) – The exact number of resources
- **min** (*Number*) – The minimal acceptable number of resources
- **max** (*Number*) – The maximal acceptable number of resources
- **split_into** (*Number*) – The anticipated number of chunks to which the total resources should be split. The minimal number of resources in a chunk will be restricted by the value of 'min'.

get_dict()

Returns Dictionary of resource requirements.

Return type Dict

class eqi.**ResumeLevel**
Bases: enum.Enum

Typically the resumed task will start in a working directory of the previous, not-completed run. Sometimes the partially generated output or intermediate files could be a problem. Therefore EQI tries to help in this matter by providing dedicated mechanisms for automatic recovery. However, this depends on the use case, how much the automatism can interfere with the resume logic. Therefore there are a few levels of automatic recovery available.

DISABLED = 'Automatic resume is fully disabled'

BASIC = 'For the tasks creating run directories, the resume checks if an unfinished ta

MODERATE = "This level processes all operations offered by BASIC level, and adds the f

class eqi.**StateKeeper**(*directory*)
Bases: object

EQI_CAMPAIGN_STATE_FILE_NAME = '.eqi_campaign_state.json'
Stores information about EQI execution

Parameters

- **directory** (*string*) – the root directory where the information will be stored
- **campaign** (*Campaign*) – the EasyVVUQ Campaign object from which the StateKeeper will be inited

setup(*campaign*)
Initialises StateKeeper with campaign

Parameters **campaign** (*Campaign*) – the EasyVVUQ Campaign object from which the StateKeeper will be inited

write_to_state_file(*data*)

get_from_state_file()

7.1 Subpackages

7.1.1 eqi.core package

Submodules

eqi.core.executor module

class eqi.core.executor.**Executor**(*campaign*, *config_file=None*, *resume=True*, *log_level='info'*)

Bases: object

Integrates EasyVVUQ and QCG-PilotJob Manager

Executor allows to process the most demanding operations of EasyVVUQ in parallel using QCG-PilotJob.

create_manager(*resources=None*, *reserve_core=False*, *enable_rt_stats=False*, *wrapper_rt_stats=None*, *log_level='info'*)

Creates new QCG-PilotJob Manager and sets is as the Executor's engine.

Parameters

- **resources** (*str, optional*) – The resources to use. If specified forces usage of Local mode of QCG-PilotJob Manager. The format is compliant with the NODES format of QCG-PilotJob, i.e.: [node_name:]cores_on_node[,node_name2:cores_on_node][,...]. Eg. to run on 4 cores regardless the node use *resources="4"* to run on 2 cores of node_1 and on 3 cores of node_2 use *resources="node_1:2,node_2:3"*
- **reserve_core** (*bool, optional*) – If True reserves a core for QCG-PilotJob Manager instance, by default QCG-PilotJob Manager shares a core with computing tasks Parameters.
- **enable_rt_stats** (*bool, optional*) – If True, QCG-PilotJob Manager will collect its runtime statistics
- **wrapper_rt_stats** (*str, optional*) – The path to the QCG-PilotJob Manager tasks wrapper program used for collection of statistics
- **log_level** (*str, optional*) – Logging level for QCG-PilotJob Manager (for both service and client part).

Returns

Return type None

set_manager(*qcgpjm*)

Sets existing QCG-PilotJob Manager as the Executor's engine

Parameters **qcgpjm** (*qcg.pilotjob.api.manager.Manager*) – Existing instance of a QCG-PilotJob Manager

Returns

Return type None

add_task(*task*)

Add a task to execute with QCG PJ

Parameters **task** (Task) – The task that will be added to the execution workflow

Returns

Return type None

run(*processing_scheme=<ProcessingScheme.SAMPLE_ORIENTED: 'Submits a workflow of EasyVVUQ operations as separate QCG PJ tasks for a sample (e.g. encoding -> execution) and then goes to the next sample'>*)
Executes demanding parts of EasyVVUQ campaign with QCG-PilotJob

A user may choose the preferred execution scheme for the given scenario.

Parameters **processing_scheme** (ProcessingScheme) – Tasks processing scheme

Returns

Return type None

print_resources_info()
Displays resources assigned to QCG-PilotJob Manager

terminate_manager()
Terminates QCG-PilotJob Manager

class eqi.core.executor.**ServiceLogLevel**
Bases: enum.Enum

An enumeration.

CRITICAL = 'critical'

ERROR = 'error'

WARNING = 'warning'

INFO = 'info'

DEBUG = 'debug'

class eqi.core.executor.**ClientLogLevel**
Bases: enum.Enum

An enumeration.

INFO = 'info'

DEBUG = 'debug'

eqi.core.processing_scheme module

class eqi.core.processing_scheme.**ProcessingScheme**(*description*, *iterative=False*)
Bases: enum.Enum

Specifies scheme of processing of tasks with QCG-PJ

Parameters

- **description** (*str*) – Description of the ProcessingScheme
- **iterative** (*bool*) – Defines if the ProcessingScheme uses iterative tasks of QCG-PJ

STEP_ORIENTED = 'Submits specific EasyVVUQ operation (e.g. encoding) for all samples a

STEP_ORIENTED_ITERATIVE = ('Submits an iterative task for execution of specific EasyVV

SAMPLE_ORIENTED = 'Submits a workflow of EasyVVUQ operations as separate QCG PJ tasks

SAMPLE_ORIENTED_CONDENSED = 'Submits all EasyVVUQ operations for a sample as a single

SAMPLE_ORIENTED_CONDENSED_ITERATIVE = ('Submits an iterative QCG PJ task for all sampl

EXEC_ONLY = 'Submits a workflow of EasyVVUQ operations as separate QCG PJ tasks for ex

EXEC_ONLY_ITERATIVE = ('Submits an iterative QCG PJ task for all samples, where a sing

is_iterative()
Checks if ProcessingScheme makes use of iterative QCG-PJ tasks

Returns **bool**

Return type True if ProcessingScheme uses iterative QCG-PJ tasks

eqi.core.resume module

class eqi.core.resume.**ResumeLevel**
Bases: enum.Enum

Typically the resumed task will start in a working directory of the previous, not-completed run. Sometimes the partially generated output or intermediate files could be a problem. Therefore EQI tries to help in this matter by providing dedicated mechanisms for automatic recovery. However, this depends on the use case, how much the automatism can interfere with the resume logic. Therefore there are a few levels of automatic recovery available.

DISABLED = 'Automatic resume is fully disabled'

BASIC = 'For the tasks creating run directories, the resume checks if an unfinished ta

MODERATE = "This level processes all operations offered by BASIC level, and adds the f

eqi.core.task module

class eqi.core.task.**TaskType**
Bases: enum.Enum

An enumeration.

ENCODING = 'ENCODING'

EXECUTION = 'EXECUTION'

ENCODING_AND_EXECUTION = 'ENCODING_AND_EXECUTION'

OTHER = 'OTHER'

class eqi.core.task.**Task**(*type*, *requirements=None*, *name=None*, *model='default'*, *resume_level=<ResumeLevel.BASIC: 'For the tasks creating run directories, the resume checks if an unfinished task created such directory. If such directory is available, this is automatically removed before the start of the resumed task'>*, ***params*)
Bases: object

Represents a piece of work to be executed by QCG-PilotJob Manager

Parameters

- **type** (TaskType) –

  The type of the task. Allowed tasks are: ENCODING, EXECUTION, ENCODING_AND_EXECUTION, and OTHER (currently not supported)

- **requirements** (TaskRequirements, *optional*) – The requirements for the Task

- **name** (*str, optional*) – name of the Task, if not provided the name will take a value of type
- **model** (*str, optional*) – Allows to set the flavour of execution of task adjusted to a given resource. At the moment of writing a user can select from the following models: *threads, intelmpi, openmpi, srunmpi, default*
- **resume_level** (ResumeLevel, *optional*) – The resume level applied for a task.
- **params** (*kwargs*) – additional parameters that may be used by specific Task types

get_type()

get_requirements()

get_model()

get_resume_level()

get_params()

get_name()

eqi.core.task_requirements module

class eqi.core.task_requirements.**Resources**(*exact=None*, *min=None*, *max=None*, *split_into=None*)

Bases: object

Stores typical for QCG Pilot Job resource requirements

Parameters

- **exact** (*Number*) – The exact number of resources
- **min** (*Number*) – The minimal acceptable number of resources
- **max** (*Number*) – The maximal acceptable number of resources
- **split_into** (*Number*) – The anticipated number of chunks to which the total resources should be split. The minimal number of resources in a chunk will be restricted by the value of 'min'.

get_dict()

Returns Dictionary of resource requirements.

Return type Dict

class eqi.core.task_requirements.**TaskRequirements**(*cores: Union[int, eqi.core.task_requirements.Resources, None] = None*, *nodes: Union[int, eqi.core.task_requirements.Resources, None] = None*)

Bases: object

Requirements for a task executed within QCG Pilot Job

Parameters

- **cores** (*int or* eqi.Resources) – the resource requirements for cores
- **nodes** (*int or* eqi.Resources) – the resource requirements for nodes

get_resources()
Allows to get resource requirements in a form of dictionary understandable by QCG Pilot Job Manager

Returns **dict**

Return type dictionary with the resource requirements specification

eqi.core.tasks_manager module

class eqi.core.tasks_manager.**TasksManager**(*campaign*, *eqi_dir*, *config_file=None*)
Bases: object

Manages tasks for execution with QCG-PJ

This is the main class for definition and management of tasks for execution by QCG-PilotJob. The tasks usually maps to certain steps of EasyVVUQ

add_task(*task*)

get_task(*name*, *key=None*, *key_min=None*, *key_max=None*, *after=None*)

7.1.2 eqi.utils package

Submodules

eqi.utils.state_keeper module

class eqi.utils.state_keeper.**StateKeeper**(*directory*)
Bases: object

EQI_CAMPAIGN_STATE_FILE_NAME = '.eqi_campaign_state.json'
Stores information about EQI execution

Parameters

- **directory** (*string*) – the root directory where the information will be stored
- **campaign** (*Campaign*) – the EasyVVUQ Campaign object from which the StateKeeper will be inited

setup(*campaign*)
Initialises StateKeeper with campaign

Parameters **campaign** (*Campaign*) – the EasyVVUQ Campaign object from which the StateKeeper will be inited

write_to_state_file(*data*)

get_from_state_file()

7.2 Submodules

7.2.1 eqi.external_encoder module

eqi.external_encoder.**encode**(*params*)

CHAPTER 8

Indices and tables

- genindex
- modindex

8.1 Authors

Bartosz Bosak <bbosak@man.poznan.pl> (PSNC)
Piotr Kopta <pkopta@man.poznan.pl> (PSNC)
Tomasz Piontek <pkopta@man.poznan.pl> (PSNC)
Jalal Lakhlili <jalal.lakhlili@ipp.mpg.de> (IPP)

Python Module Index

e

Index

I

M

O

P

R

S

T

W