QCG-PilotJob

Jun 10, 2020

Basics

A python service for easy execution of many tasks inside a single allocation.

CHAPTER 1

Overview

The QCG-PilotJob system is designed to schedule and execute many small jobs inside one scheduling system allocation. Direct submission of a large group of jobs to a scheduling system can result in long aggregated time to finish as each single job is scheduled independently and waits in a queue. On the other hand the submission of a group of jobs can be restricted or even forbidden by administrative policies defined on clusters. One can argue that there are available job array mechanisms in many systems, however the traditional job array mechanism allows to run only bunch of jobs having the same resource requirements while jobs being parts of a multiscale simulation by nature vary in requirements and therefore need more flexible solutions.

The core component of QCG-PilotJob system is QCG-PilotJob Manager. From the scheduling system perspective, QCG-PilotJob Manager, is seen as a single job inside a single user allocation. It means that QCG-PilotJob Manager controls an execution of a complex experiment consisting of many jobs on resources reserved for the single job allocation. The manager listens to user's requests and executes commands like submit job, cancel job and report resources usage. In order to manage the resources and jobs the system takes into account both resources availability and mutual dependencies between jobs. Two interfaces are defined to communicate with the system: file-based (batch mode) and API based. The former one is dedicated and more convenient for a static scenarios when a number of jobs is known in advance to the QCG-PilotJob Manager start. The API based interface is more general and flexible as it allows to dynamically send new requests and track execution of previously submitted jobs during the run-time.

To allow user's to test their scenarios, QCG-PilotJob Manager supports *local* execution mode, in which all job's are executed on local machine and doesn't require any scheduling system allocation.

CHAPTER 2

Installation

QCG-PilotJob Manager requires Python version >= 3.6.

Optionally the latest version of *pip* package manager and *virtualenv* can be insalled in user's directory by following commands:

```
$ curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
$ python3 get-pip.py --user
$ pip install --user virtualenv
```

To create private virtual environment for installed packages, type following commands:

```
$ virtualenv venv
$ . venv/bin/activate
```

There are two options for the actual installation of QCG-PilotJob. You can use the PyPi repository or install the package from GitHub.

2.1 PyPi

The installation of QCG-PilotJob from the PyPi repository is as simple as:

```
$ pip install qcg-pilotjob
```

2.2 GitHub

To install QCG-PilotJob directly from github.com type the following command:

```
$ pip install --upgrade git+https://github.com/vecma-project/QCG-PilotJob.git
```

To install specific branch from QCG-PilotJob github.com repository, the following format should be used:

```
$ pip install --upgrade git+https://github.com/vecma-project/QCG-PilotJob.git@branch_
↪name
```

CHAPTER 3

Examples

QCG-PilotJob Manager can be used in two different ways:

- as an service accessible with API
- as a command line utility to execute static, prepared job workflows in a batch mode

The first method allows to dynamically control the jobs execution.

3.1 Example API application

Let's write a simple program that will runs 4 instances of simple bash script.

First, we must create an instance of QCG-PilotJob Manager

```
from qcg.pilotjob.api.manager import LocalManager

manager = LocalManager()
```

This default instance, when launched outside Slurm scheduling system allocation, will use all local available CPU's. To check what resources are available for our future jobs, we call a `resources` method.

```
print('available resources: ', manager.resources())
```

In return we should give something like:

```
available resources: {'total_nodes': 1, 'total_cores': 8, 'used_cores': 0, 'free_cores
↪': 8}
```

where `total_cores` and `free_cores` depends on number of cores on machine where we are running this example. So our programs will have access to all `free_cores`, and QCG-PilotJob manager will make sure that tasks do not interfere with each other, so the maximum number of simultaneously running job's will be exact `free_cores`.

To run jobs, we have to create a list of job descriptions and sent it to the QCG-PilotJob manager.

```
from qcg.pilotjob.api.job import Jobs
jobs = Jobs().add(script='echo "job ${it} executed at `date` @ `hostname`"', stdout=
↪'job.out.${it}', iteration=4)
job_ids = manager.submit(jobs)
print('submited jobs: ', str(job_ids))
```

In this code, we submitted a job with four iterations. The standard output stream should be redirected to file job.out with iteration index as postfix. As a program to execute in job iteration, we passed the simple *bash* command. The above code should print a list with just one element: the submitted job identifier. Because we didn't name our job, the automatically generated name was returned. The job name can passed as keyword argument `name` to `Jobs.add` method.

Now we can check the status of our submitted job:

```
job_status = manager.status(job_ids)
print('job status: ', job_status)
```

The `job_status` should contain dictionary `jobs` with our job status information. Because our job was very short, and should finish immediately, the `state` key of `data` dictionary of our job's status, should contain value `SUCCEED`. For longer jobs, we may want to wait until our submitted jobs finish, to do this we use the `wait4` *Manager* method:

```
manager.wait4(job_ids)
```

Alternatively we can use the `wait4all` method, which will wait until all submitted to the QCG-PilotJob Manager jobs finish:

```
manager.wait4all()
```

If we check current directory, we can see that bunch of `job.out.` files has been created with a proper content. If we want to get detailed information about our job, we can use the `info` method:

```
job_info = manager.info(job_ids)
print('job detailed information: ', job_info)
```

In return we will get information about iterations (how many finished successfully, how many failed) and when our job finished.

It is important to call `finish` method at the end of our program. This method sent a proper command to QCG-PilotJob Manager instance, and terminates the background thread in which the instance has been run.

```
manager.finish()
```

QCG-PilotJob Manager creates a directory *.qcgpjm-service-* where the following files are stored:

- `service.log` - logs of QCG-PilotJob Manager, very useful in case of problems
- `jobs.report` - the file containing information about all finished jobs, by default written in text format, but there is an option for JSON format which will be easier to parse.

See also:

The full documentation of the API methods and it's arguments is available in the *qcg.pilotjob.api package* documentation.

3.2 Example batch usage

The same jobs we can launch using the batch method and prepared input files. In this mode, we have to create JSON file with all requests we want to sent to QCG-PilotJob Manager. For example, the file contains jobs we submitted in previous section will look like this:

```
[
  {
    "request": "submit",
    "jobs": [
      {
        "name": "example",
        "iteration": { "stop": 4 },
        "execution": {
          "script": "echo \"job ${it} executed at `date` @ `hostname`\"",
          "stdout": "job.out.${it}"
        }
      }
    ]
  },
  {
    "request": "control",
    "command": "finishAfterAllTasksDone"
  }
]
```

After placing above content in the JSON file, for example `jobs.json`, we can execute this workflow with:

```
$ python -m qcg.pilotjob.service --file-path jobs.json
```

Alternatively, we can use the `qcg-pm-service` command alias, that is installed with `qcg-pilotjob` Python package.

```
$ qcg-pm-service --file-path jobs.json
```

In the input file, we have placed two requests:

- `submit` - with job description we want to run
- `control` - with `finishAfterAllTasksDone` command, which is required to finish QCG-PilotJob Manager (the service might listen also on other interfaces, like ZMQ network interface, and must explicitly know when no more requests will come and service may be stopped.

The result of executing QCG-PilotJob Manager with presented example file should be the same as using the API - the bunch of output files should be created, as well as `.qcgpjm-service-` directory with additional files.

CHAPTER 4

Modes of execution

In the previously presented examples we submitted a single CPU applications. However QCG-PilotJob Manager is intended for use in HPC environments, especially with *Slurm* scheduling system. The execution on a cluster is therefore a default mode of execution of QCG-PilotJob. In order to support users in testing their scenarios before the actual execution on a cluster, QCG-PilotJob can be also run in a local environment. Below we present these two modes of execution of QCG-PilotJob.

4.1 Scheduling systems

In case of execution via Slurm we submit a request to scheduling system and when requested resources are available, the allocation is created and our application is run inside it. Of course we might run our job's directly in scheduling system without any pilot job mechanism, but we have to remember about some limitations of scheduling systems such as - maximum number of submitted/executing jobs in the same time, queueing time (significant for large number of jobs), job array mechanism only for same resource requirement jobs. Generally, scheduling systems wasn't designed for handling very large number of small jobs.

To use QCG-PilotJob Manager in HPC environment, we suggest to install QCG-PilotJob Manager via virtual environment in directory shared among all computing nodes (most of home directories are available from computing nodes). On some systems, we need to load a proper Python >= 3.6 module before:

```
$ module load python/3.7.3
```

Next we can create virtual environment with QCG-PilotJob Manager:

```
$ python3 -m virtualenv $HOME/qcgpj-venv
$ source $HOME/qcgpj-venv/bin/activate
$ pip install qcg-pilotjob
```

Now we can use this virtual environment in our jobs. The example job submission script for *Slurm* scheduling system that launched application `myapp.py` that uses QCG-PilotJob Manager API, may look like this:

```
#SBATCH --job-name=qcgpilotjob-ex
#SBATCH --nodes=2
#SBATCH --tasks-per-node=28
#SBATCH --time=60

module load python/3.7.3
source $HOME/qcgpj-venv/bin/activate

python myapp.py
```

Of course, some scheduling system might require some additional parameters like:

- `--account` - name of the account/grant we want to use
- `--partition` - the partition name where our job should be scheduled

To submit a job with QCG-PilotJob Manager in batch mode with JSON jobs description file, we have to change the last line to:

```
python -m qcg.pilotjob.service --file-path jobs.json
```

Note: Once QCG-PilotJob is submitted via Slurm or QCG middleware, it inherits the execution environment set by those systems. Some environment variables, such as the location of a shared directory, may be useful in a user's tasks. In order to get more detailed information on this topic please see *Execution environments*.

4.2 Local execution

QCG-PilotJob Manager supports *local* mode that is suitable for locally testing execution scenarios. In contrast to execution mode, where QCG-PilotJob Manager is executed in scheduling system allocation, all jobs are launched with the usage of scheduling system. In the *local* mode, the user itself can define the size of available resources and execute it's scenario on such defined resources without the having access to scheduling system. It's worth remembering that QCG-PilotJob Manager doesn't verify the physically available resources, also the executed jobs are not launched with any core/processor affinity. Thus the performance of jobs might not be optimal.

The choice between *allocation* (in scheduling system allocation) or *local* mode is made automatically by the QCG PilotJob Manager during the start. If scheduling system environment will be detected, the *allocation* mode will be chosen. In other case, the local mode will be active, and if resources are not defined by the user, the default number of available cores in the system will be taken.

The command line arguments, that also might by passed as argument `server_args` during instantiating the LocalManager , related to the *local* mode are presented below:

- `--nodes NODES` - the available resources definition; the `NODES` parameter should have format:

  ```
  `[NODE_NAME]:CORES[,[NODE_NAME]:CORES]...`
  ```

- `--envschema ENVSCHEMA` - job execution environment; for each job QCG-PilotJob Manager can create environment similar to the Slurm execution environment

Some examples of resources definition:

- `--nodes 4` - single node with 4 available cores
- `--nodes n1:2` - single named node with 2 available cores
- `--nodes 4,2,2` - three unnamed nodes with 8 total cores

- `--nodes n1:4, n2:4, n3:4` - three named nodes with 12 total cores

CHAPTER 5

Parallelism

QCG-PilotJob Manager can handle jobs that require more than a single core. The number of required cores and nodes is specified with `numCores` and `numNodes` parameter of `Jobs.add` method. The number of required resources can be specified either as specific values or as a range of resources (with minimum and maximum values), where QCG-PilotJob Manager will try to assign as much resources from those available in the moment. The environment of parallel job is prepared for *MPI* or *OpenMP* jobs.

5.1 MPI

In case of *MPI* programs only one process is launched by QCG-PilotJob Manager that should call a proper MPI starting program, such as: `mpirun` or `mpiexec`. All the environment for the parallel job, such as hosts file, and environment variables are prepared by QCG-PilotJob Manager. For example to run *Quantum Espresso* application, the example program may look like this:

```
from qcg.pilotjob.api.manager import LocalManager
from qcg.pilotjob.api.job import Jobs

manager = LocalManager()

jobs = Jobs().add(
    name='qe-example',
    exec='mpirun',
    args=['pw.x'],
    stdin='pw.benzene.scf.in',
    stdout='pw.benzene.scf.out',
    modules=['espresso/5.3.0', 'mkl', 'impi', 'mpich'],
    numCores=8)

job_ids = manager.submit(jobs)
manager.wait4(job_ids)

manager.finish()
```

As we can see in the example, we run a single program `mpirun` which is responsible for setup a proper, parallel environment for the destination program and spawn the *Quantum Espresso* executables (`pw.x`).

In the example program we used some additional options of `Jobs.add` method:

- `stdin` - points to the file that content should be sent to job's standard input
- `modules` - environment modules that should be loaded before job start
- `numCores` - how much cores should be allocated for the job

The JSON job description file for the same example is presented below:

```
[
  {
    "request": "submit",
    "jobs": [
      {
        "name": "qe-example",
        "execution": {
          "exec": "mpirun",
          "args": ["pw.x"],
          "stdin": "pw.benzene.scf.in",
          "stdout": "pw.benzene.scf.out",
          "modules": ["espresso/5.3.0", "mkl", "impi", "mpich"]
        },
        "resources": {
          "numCores": { "exact": 8 }
        }
      }
    ]
  },
  {
    "request": "control",
    "command": "finishAfterAllTasksDone"
  }
]
```

5.2 OpenMP

For *OpenMP* programs (shared memory parallel model), where there is one process that spawns many threads on the same node, we need to use special option `model` with `threads` value. To test execution of *OpenMP* program we need to compile a sample application:

```
$ wget https://computing.llnl.gov/tutorials/openMP/samples/C/omp_hello.c
$ gcc -Wall -fopenmp -o omp_hello omp_hello.c
```

Now we can launch this application with QCG-PilotJob Manager:

```
from qcg.pilotjob.api.manager import LocalManager
from qcg.pilotjob.api.job import Jobs

manager = LocalManager()

jobs = Jobs().add(
    name='openmp-example',
    exec='omp_hello',
```

(continues on next page)

(continued from previous page)

```
    stdout='omp.out',
    model='threads',
    numCores=8,
    numNodes=1)

job_ids = manager.submit(jobs)
manager.wait4(job_ids)

manager.finish()
```

The `omp.out` file should contain eight lines with *Hello world from thread* =. It is worth to remember, that OpenMP applications can operate only on single node, so adding `numNodes=1` might be necessary in case where there are more than single node in available resources.

The equivalent JSON job description file for given example is presented below:

```
[
  {
    "request": "submit",
    "jobs": [
      {
        "name": "openmp-example",
        "execution": {
          "exec": "omp_hello",
          "stdout": "omp.ou",
          "model": "threads"
        },
        "resources": {
          "numCores": { "exact": 8 },
          "numNodes": { "exact": 1 }
        }
      }
    ]
  },
  {
    "request": "control",
    "command": "finishAfterAllTasksDone"
  }
]
```

CHAPTER 6

QCG-PilotJob Manager options

The list of all options can be obtained by running either the wrapper command:

```
$ qcg-pm-service --help
```

or directly call the Python module:

```
$ python -m qcg.pilotjob.service –help
```

Those options can be passed to QCG-PilotJob Manager in batch mode as command line arguments, or as an argument `server_args` during instantiating the LocalManager class.

The full list of currently supported options is presented below.

```
$ qcg-pm-service --help
usage: qcg-pm-service [-h] [--net] [--net-port NET_PORT]
                      [--net-port-min NET_PORT_MIN]
                      [--net-port-max NET_PORT_MAX] [--file]
                      [--file-path FILE_PATH] [--wd WD]
                      [--envschema ENVSCHEMA] [--resources RESOURCES]
                      [--report-format REPORT_FORMAT]
                      [--report-file REPORT_FILE] [--nodes NODES]
                      [--log {critical,error,warning,info,debug,notset}]
                      [--system-core] [--disable-nl] [--show-progress]
                      [--governor] [--parent PARENT] [--id ID] [--tags TAGS]
                      [--slurm-partition-nodes SLURM_PARTITION_NODES]
                      [--slurm-limit-nodes-range-begin SLURM_LIMIT_NODES_RANGE_BEGIN]
                      [--slurm-limit-nodes-range-end SLURM_LIMIT_NODES_RANGE_END]

optional arguments:
  -h, --help            show this help message and exit
  --net                 enable network interface
  --net-port NET_PORT   port to listen for network interface (implies --net)
  --net-port-min NET_PORT_MIN
                        minimum port range to listen for network interface if
                        exact port number is not defined (implies --net)
```

(continues on next page)

(continued from previous page)

```
  --net-port-max NET_PORT_MAX
                        maximum port range to listen for network interface if
                        exact port number is not defined (implies --net)
  --file                enable file interface
  --file-path FILE_PATH
                        path to the request file (implies --file)
  --wd WD               working directory for the service
  --envschema ENVSCHEMA
                        job environment schema [auto|slurm]
  --resources RESOURCES
                        source of information about available resources
                        [auto|slurm|local] as well as a method of job
                        execution (through local processes or as a Slurm sub
                        jobs)
  --report-format REPORT_FORMAT
                        format of job report file [text|json]
  --report-file REPORT_FILE
                        name of the job report file
  --nodes NODES         configuration of available resources (implies
                        --resources local)
  --log {critical,error,warning,info,debug,notset}
                        log level
  --system-core         reserve one of the core for the QCG-PJM
  --disable-nl          disable custom launching method
  --show-progress       print information about executing tasks
  --governor            run manager in the governor mode, where jobs will be
                        scheduled to execute to the dependant managers
  --parent PARENT       address of the parent manager, current instance will
                        receive jobs from the parent manaqger
  --id ID               optional manager instance identifier - will be
                        generated automatically when not defined
  --tags TAGS           optional manager instance tags separated by commas
  --slurm-partition-nodes SLURM_PARTITION_NODES
                        split Slurm allocation by given number of nodes, where
                        each group will be controlled by separate manager
                        (implies --governor)
  --slurm-limit-nodes-range-begin SLURM_LIMIT_NODES_RANGE_BEGIN
                        limit Slurm allocation to specified range of nodes
                        (starting node)
  --slurm-limit-nodes-range-end SLURM_LIMIT_NODES_RANGE_END
                        limit Slurm allocation to specified range of nodes
                        (ending node)
```

CHAPTER 7

Key concepts

7.1 Modules

QCG-PilotJob Manager consists of the following internal functional modules:

- *Queue* - the queue containing jobs waiting for resources,
- *Scheduler* algorithm - the algorithm selecting jobs and assigning resources to them.
- *Registry* - the permanent registry containing information about all (current and historical) jobs in the system,
- *Executor* - a module responsible for execution of jobs for which resources were assigned.

7.2 Queue & scheduler

All the jobs submitted to the QCG-PilotJob Manger system are placed in the queue in the order of they arrival. The scheduling algorithm of QCG-PilotJob Manager works on that queue. The goal of the Scheduler is to determine the order of execution and amount of resources assigned to individual jobs to maximise the throughput of the system. The algorithm is based on the following set of rules:

- Jobs being in the queue are processed in the FIFO manner,
- For every feasible (ready for execution) job the maximum (possible) amount of requested resources is determined. If the amount of allocated resources is greater than the minimal requirements requested by the user, the resources are exclusively assigned to the job and the job is removed from the queue to be executed.
- If the minimal resource requirements are greater than total available resources the job is removed from the queue with the `FAILED` status.
- If the amount of resources doesn't allow to start the job, it stays in the queue with the `QUEUED` status to be taken into consideration again in the next scheduling iteration,
- Jobs waiting for successful finish of any other job, are not taken into consideration and stay in the queue with the `QUEUED` state,

- Jobs for which dependency constraints can not be met, due to failure or cancellation of at least one job which they depend on, are marked as `OMITTED` and removed from the queue,
- If the algorithm finishes processing the given job and some resources still remain unassigned the whole procedure is repeated for the next job.

7.3 Executors

QCG-PilotJob Manager module named Executor is responsible for execution and control of jobs by interacting with the cluster resource management system. The current implementation contains three different methods of executing jobs:

- as a local process - this method is used when QCG-PilotJob Manager either has been run outside a Slurm allocation or when parameter `--resources local` has been defined,
- through internal distributed launcher service - currently used only in Slurm allocation for single core jobs,
- as a Slurm sub job - the job is submitted to the Slurm to be run in current allocation on scheduled resources.

The modular approach allows for relatively easy integration also with other queuing systems. The QCG-PilotJob Manager and all jobs controlled by it are executed in a single allocation. To hide this fact from the individual job and to give it an impression that it is executed directly by the queuing system QCG-PilotJob overrides some of the environment settings. More on this topic is available in *Execution environments*

CHAPTER 8

Execution environments

In order to give an impression that an individual QCG-PilotJob task is executed directly by the queuing system a set of environment variables, typically set by the queuing system, is overwritten and passed to the job. These variables give the application all typical information about a job it can be interested in, e.g. the amount of assigned resources. In case of parallel application an appropriate machine file is created with a list of resources for each task. Additionally to unify the execution regardless of the queuing system a set of variables independent from a queuing system is defined and passed to tasks.

8.1 Slurm execution environment

For the SLURM scheduling system, an execution environment for a single job contains the following set of variables:

- `SLURM_NNODES` - a number of nodes
- `SLURM_NODELIST` - a list of nodes separated by the comma
- `SLURM_NPROCS` - a number of cores
- `SLURM_NTASKS` - see `SLURM_NPROCS`
- `SLURM_JOB_NODELIST` - see `SLURM_NODELIST`
- `SLURM_JOB_NUM_NODES` - see `SLURM_NNODES`
- `SLURM_STEP_NODELIST` - see `SLURM_NODELIST`
- `SLURM_STEP_NUM_NODES` - see `SLURM_NNODES`
- `SLURM_STEP_NUM_TASKS` - see `SLURM_NPROCS`
- `SLURM_NTASKS_PER_NODE` - a number of cores on every node listed in `SLURM_NODELIST` separated by the comma,
- `SLURM_STEP_TASKS_PER_NODE` - see `SLURM_NTASKS_PER_NODE`
- `SLURM_TASKS_PER_NODE` - see `SLURM_NTASKS_PER_NODE`

8.2 QCG Execution environment

To unify the execution environment regardless of the queuing system the following variables are set:

- `QCG_PM_NNODES` - a number of nodes
- `QCG_PM_NODELIST`- a list of nodes separated by the comma
- `QCG_PM_NPROCS` - a number of cores
- `QCG_PM_NTASKS` - see `QCG_PM_NPROCS`
- `QCG_PM_STEP_ID` - a unique identifier of a job (generated by QCG-PilotJob Manager)
- `QCG_PM_TASKS_PER_NODE` - a number of cores on every node listed in `QCG_PM_NODELIST` separated by the comma
- `QCG_PM_ZMQ_ADDRESS` - an address of the network interface of QCG-PilotJob Manager (if enabled)

CHAPTER 9

File based interface

The *File* interface allows a static sequence of commands (called requests) to be read from a file a nd performed by the system.

9.1 File interface usage

To use QCG-PilotJob Manager with the *File* interface we should call either the wrapper command:

```
$ qcg-pm-service
```

or directly call the Python module:

```
$ python -m qcg.pilotjob.service
```

with the `--file-path FILE_PATH` parameter, where `FILE_PATH` is a path to the requests file. For example, the command:

```
$ qcg-pm-service --file-path reqs.json
```

will run QCG-PilotJob Manager on requests written in `reqs.json` file.

9.2 Requests file

The requests file is a JSON format file containing a sequence of commands (requests). The file must be staged into the working directory of the QCG-PilotJob Manager job and passed as an argument of this job invocation. The requests are read in an order they are placed in the file. In the file mode, QCG-PilotJob Manager outputs all responses to the log file.

9.3 Commands

The request is a JSON dictionary with the `request` key containing a request command. The additional data format depends on a specific request command. The following commands are currently supported.

9.3.1 `submit`

Submit a list of jobs to be processed by the system. The `jobs` key must contain a list of formalised descriptions of jobs.

The Job description is a dictionary with the following keys:

- `name` (*required*) `String` - job name, must be unique among all other submitted jobs
- `iteration` (*optional*) `Dict` - defines a loop for iterative jobs, the *start* (*optional*) and *stop* keys must be defined; the total number of iterations will be *stop* - *start* (the last index of the sub-job will be *stop* - *1*)
- `execution` (*required*) `Dict` - execution description with the following keys:
  - `exec` (*optional*) `String` - executable name (if available in *$PATH*) or absolute path to the executable,
  - `args` (*optional*) `Array of String` - list of arguments that will be passed to the executable,
  - `script` (*optional*) `String` - commands for *bash* environment, mutually exclusive with `exec` and `args`
  - `env` (*optional*) `Dict (String: String)` - environment variables that will be appended to the execution environment,
  - `wd` (*optional*) `String` - a working directory, if not defined the working directory (current directory) of QCG-PilotJob Manager will be used. If the path is not absolute it is relative to the QCG-PilotJob Manager working directory. If the directory pointed by the path does not exist, it is created before the job starts.
  - `stdin`, `stdout`, `stderr` (*optional*) `String` - path to the standard input , standard output and standard error files respectively.
  - `modules` (*optional*) `Array of String` - the list of environment modules that should be loaded before start of the job
  - `venv` (*optional*) `String` - the path to the virtual environment inside in job should be started
  - `model` (*optional*) `String` - the model of execution, currently only *threads* explicit model is supported which should be used for OpenMP jobs; if not defined - the default, dedicated to the multi processes execution model is used
- `resources` (*optional*) `Dict` - resource requirements, a dictionary with the following keys:
  - `numCores` (*optional*) `Dict` - number of cores,
  - `numNodes` (*optional*) `Dict`- number of nodes,

    The specification of `numCores`/`numNodes` elements may contain the following keys:

    * `exact` (*optional*) `Number` - the exact number of cores,
    * `min` (*optional*) `Number` - minimal number of cores,
    * `max` (*optional*) `Number` - maximal number of cores,
    * `scheduler` (*optional*) `Dict` - the type of resource iteration scheduler, the key *name* specify type of scheduler and currently the *maximum-iters* and *split-into* names are supported, the optional *params* dictionary specifies the scheduler parameters (the `exact` and `min` / `max` are mutually exclusive).

If `resources` is not defined, the `numCores` with `exact` set to 1 is taken as the default value.

The `numCores` element without `numNodes` specifies requested number of cores on any number of nodes. The same element used along with the `numNodes` determines the number of cores on each requested node.

The `scheduler` optional key defines the iteration resources scheduler. It is futher described in section *Iteration resources schedulers*.

- `dependencies` (*optional*) `Dict` - a dictionary with the following items:
  - `after` (*required*) `Array of String` - list of names of jobs that must finish before the job can be executed. Only when all listed jobs finish (with `SUCCESS` status) the current job is taken into consideration by the scheduler and can be executed.

The job description may contain variables (except the job name, which cannot contain any variable or special character) in the format:

```
${ variable-name }
```

which are replaced with appropriate values by QCG-PilotJob Manager.

The following set of variables is supported during a request validation:

- `rcnt` - a request counter that is incremented with every request (for iterative sub-jobs the value of this variable is the same)
- `uniq` - a unique identifier of each request (each iterative sub-job has its own unique identifier)
- `sname` - a local cluster name
- `date` - a date when the request was received
- `time` - a time when the request was received
- `dateTime` - date and time when the request was received
- `it` - an index of a current sub-job (only for iterative jobs)
- `jname` - a final job name after substitution of all other used variables to their values

The following variables are handled when resources has been already allocated and before the start of job execution:

- `root_wd` - a working directory of QCG-PilotJob Manager, the parent directory for all relative job's working directories
- `ncores` - a number of allocated cores for the job
- `nnodes` - a number of allocated nodes for the job
- `nlist` - a list of nodes allocated for the job separated by the comma

The sample submit job request is presented below:

```
{
    "request": "submit",
    "jobs": [
    {
        "name": "msleep2",
        "execution": {
          "exec": "/bin/sleep",
          "args": [
            "5s"
          ],
          "env": {},
          "wd": "sleep.sandbox",
```

(continues on next page)

(continued from previous page)

```
          "stdout": "sleep2.${ncores}.${nnodes}.stdout",
          "stderr": "sleep2.${ncores}.${nnodes}.stderr"
        },
        "resources": {
          "numCores": {
            "exact": 2
          }
        }
    }
    ]
}
```

The example response is presented below:

```
{
  "code": 0,
  "message": "1 jobs submitted",
  "data": {
    "submitted": 1,
    "jobs": [
      "msleep2"
    ]
  }
}
```

9.3.2 `listJobs`

Return a list of registered jobs. No additional arguments are needed. The example list jobs request is presented below:

```
{
    "request": "listJobs"
}
```

The example response is presented below:

```
{
  "code": 0,
  "data": {
    "length": 1,
    "jobs": {
      "msleep2": {
        "status": "QUEUED",
        "inQueue": 0
      }
    }
  }
}
```

9.3.3 `jobStatus`

Report current status of a given jobs. The `jobNames` key must contain a list of job names for which status should be reported. A single job may be in one of the following states:

- `QUEUED` - a job was submitted but there are no enough available resources

- `EXECUTING` - a job is currently executed
- `SUCCEED` - a finished with 0 exit code
- `FAILED` - a job could not be started (for example there is no executable) or a job finished with non-zero exit code or a requested amount of resources exceeds a total amount of resources,
- `CANCELED` - a job has been cancelled either by a user or by a system
- `OMITTED` - a job will never be executed due to the dependencies (a job which this job depends on failed or was cancelled).

The example job status request is presented below:

```
{
  "request": "jobStatus",
  "jobNames": [ "msleep2" ]
}
```

The example response is presented below:

```
{
   "code": 0,
   "data": {
     "jobs": {
       "msleep2": {
         "status": 0,
         "data": {
           "jobName": "msleep2",
           "status": "SUCCEED"
         }
       }
     }
       }
}
```

The `status` key at the top, job's level contains numeric code that represents the operation return code - 0 means success, where other values means problem with obtaining job's status (e.g. due to the missing job name).

9.3.4 `jobInfo`

Report detailed information about jobs. The `jobNames` key must contain a list of job names for which information should be reported.

The example job status request is presented below:

```
{
  "request": "jobInfo",
  "jobNames": [ "msleep2", "echo" ]
}
```

The example response is presented below:

```
 {
  "code": 0,
  "data": {
    "jobs": {
      "msleep2": {
```

(continues on next page)

(continued from previous page)

```
        "status": 0,
        "data": {
          "jobName": "msleep2",
          "status": "SUCCEED",
          "runtime": {
            "allocation": "LAPTOP-CNT0BD0F[0:1]",
            "wd": "/sleep.sandbox",
            "rtime": "0:00:02.027212",
            "exit_code": "0"
          },
          "history": "\n2020-06-08 12:56:06.789757: QUEUED\n2020-06-08 12:56:06.
↪789937: SCHEDULED\n2020-06-08 12:56:06.791251: EXECUTING\n2020-06-08 12:56:08.
↪826721: SUCCEED"
        }
      }
    }
  }
}
```

9.3.5 control

Controls behaviour of QCG-PilotJob Manager. The specific command must be placed in the‘‘command‘‘ key. Currently the following commands are supported: - `finishAfterAllTasksDone` This command tells QCG-PilotJob Manager to wait until all submitted jobs finish.

> By default, in the file mode, the QCG-PilotJob Manager application finishes as soon as all requests are read from the request file.

The sample control command request is presented below:

```
{
  "request": "control",
  "command": "finishAfterAllTasksDone"
}
```

9.3.6 cancelJob

Cancel a jobs with a list of their names specified in the `jobNames` key. Currently this operation is not supported.

9.3.7 removeJob

Remove a jobs from the registry. The list of names of a jobs to be removed must be placed in the `jobNames` key. This request can be used in case when there is a need to submit another job with the same name - because all the job names must be unique a new job cannot be submitted with the same name unless the previous one is removed from the registry. The example remove job request is presented below:

```
{
  "request": "removeJob",
  "jobNames": [ "msleep2" ]
}
```

The example response is presented below:

```
{
  "data": {
    "removed": 1
  },
  "code": 0
}
```

9.3.8 resourcesInfo

Return current usage of resources. The information about a number of available and used nodes/cores is reported. No additional arguments are needed. The example resources info request is presented below:

```
{
  "request": "resourcesInfo"
}
```

The example response is presented below:

```
{
  "data": {
    "total_cores": 8,
    "total_nodes": 1,
    "used_cores": 2,
    "free_cores": 6
  },
  "code": 0
}
```

9.3.9 finish

Finish the QCG-PilotJob Manager application immediately. The jobs being currently executed are killed. No additional arguments are needed.

The example finish command request is presented below:

```
{
  "request": "finish"
}
```

CHAPTER 10

Iteration resources schedulers

The aim of iteration resources schedulers is to optimise resources usage for iterative tasks. To this end, the schedulers assign an exact number of resources based on single iteration resource requirements described as minimum number of resources and number of available resources in allocation. What is important, the job's resource requirements for iterative tasks do not have to be changed for different allocations. The resource requirements can apply to both: number of cores and number of nodes specifications.

Currently, two schedulers are implemented:

- `maximum-iters`
- `split-into`

10.1 `maximum-iters`

The iteration resource scheduler for maximizing resource usage. The `maximum-iters` iteration resource scheduler is trying to launch as many iterations in the same time on all available resources. In case where number of iterations exceeds the number of available resources, the `maximum-iters` schedulers splits iterations into *steps* minimizing this number, and allocates as many resources as possible for each iteration inside *step*. The `max` attribute of resource specification is not allowed when `maximum-iters` scheduler is used.

10.2 `split-into`

The iteration resource scheduler for partitioning available resources. This simple iteration resource scheduler splits all available resources into given partitions, and each iteration will be executed inside whole single partition.

CHAPTER 11

Performance tuning

11.1 Reserving a core for QCG PJM

We recommend to use `--system-core` parameter for workflows that contains many small jobs (HTC) or bigger allocations (>256 cores). This will reserve a single core in allocation (on the first node of the allocation) for QCG PilogJob Manager service.

CHAPTER 12

Log files

QCG-PilotJob Manager creates a sub directory *.qcgpjm-service-* in working directory where the following files are stored:

- `service.log` - logs of QCG-PilotJob Manager, very useful in case of problems
- `jobs.report` - the file containing information about all finished jobs, by default written in text format, but there is an option for JSON format which will be easier to parse
- `final_status` - created at the finish of QCG-PilotJob Manager with general statistics about platform, available resources and jobs in registry (not removed) that finished, failed etc.

The verbosity of log file can be controlled by the `--log` parameter where `debug` value is the most verbose mode, and `critical` the most silent mode. We recommend to not set the `debug` for large HTC workflows, as it additionally loads the file system.

CHAPTER 13

Dictionary

Scheduling system A service that controls and schedules access to the fixed set of computational resources (aka. queuing system, workload manager, resource management system). The current implementation of QCG-PilotJob supports SLURM cluster management and job scheduling system.

Job A sequential or parallel program with defined resource requirements

Job array A mechanism that allows to submit a set of jobs with the same resource requirements to the scheduling system at once; commonly used in parameter sweep scenarios

Allocation A set of resources allocated by the scheduling system for a specific time period; resources assigned to an allocation are static and do not change in time

QCG-PilotJob Manager A service started inside a scheduling system allocation that schedules and controls execution of jobs on the same allocation

QCG-PilotJob Manager API An interface in the form of Python module that provides communication with QCG-PilotJob Manager

Application Controller A user's program run as one of jobs inside QCG-PilotJob Manager that, using the QCG-PilotJob Manager API, dynamically submits and synchronizes new jobs

CHAPTER 14

qcg.pilotjob.api package

14.1 Submodules

14.1.1 qcg.pilotjob.api.errors module

exception qcg.pilotjob.api.errors.**QCGPJMAError**
Bases: Exception

exception qcg.pilotjob.api.errors.**InternalError**
Bases: *qcg.pilotjob.api.errors.QCGPJMAError*

exception qcg.pilotjob.api.errors.**InvalidJobDescriptionError**
Bases: *qcg.pilotjob.api.errors.QCGPJMAError*

exception qcg.pilotjob.api.errors.**JobNotDefinedError**
Bases: *qcg.pilotjob.api.errors.QCGPJMAError*

exception qcg.pilotjob.api.errors.**ConnectionError**
Bases: *qcg.pilotjob.api.errors.QCGPJMAError*

exception qcg.pilotjob.api.errors.**WrongArgumentsError**
Bases: *qcg.pilotjob.api.errors.QCGPJMAError*

exception qcg.pilotjob.api.errors.**FileError**
Bases: *qcg.pilotjob.api.errors.QCGPJMAError*

exception qcg.pilotjob.api.errors.**ServiceError**
Bases: *qcg.pilotjob.api.errors.QCGPJMAError*

14.1.2 qcg.pilotjob.api.job module

class qcg.pilotjob.api.job.**Jobs**
Bases: object

Group of job descriptions to submit

_list
map with added job descriptions

Type dict(str,dict)

_job_idx
counter which is used to return ordered lists

Type int

Initialize instance.

add(*job_attrs=None*, ***kw_attrs*)
Add a new, simple job description to the group.

If both arguments are present, they are merged and processed as a single dictionary. The following job attributes are currenlty supported:

- `name` (str, optional): the job name
- `exec` (str, optional): path to the executable program
- `script` (str, optional): bash script content
- `args` (str or list(str), optional): executable program arguments
- `stdin` (str, optional): path to file which content should be passed to the standard input stream
- `stdout` (str, optional): path to the file where standard output stream should be saved
- `stderr` (str, optional): path to the file where standard error stream should be saved
- `wd` (str, optional): path to the working directory where job should be started
- `modules` (str or list(str), optional): list of modules that should be loaded before job start
- `venv` (str, optional): path to the virtual environment that should be initialized before job start
- `model` (str, optional): model of execution
- `numCores` (int or dict, optional): number of required cores specification
- `numNodes` (int or dict, optional): number of required nodes specification
- `wt` (str, optional): job's maximum wall time
- `iteration` (int or dict, optional): number of job's iterations
- `after` (str or list(str), optional): name of the job's that must finish successfully before current one start

The attributes `exec` (with optional `args`) are mutually exclusive with `script`.

The `numCores` and `numNodes` atrributes may contain dictionary with following keys:

- `min` (int, optional): minimum number of resources
- `max` (int, optional): maximum number of resources
- `exact` (int, optional): exact number of resources
- `scheduler` (str, optional): name of iteration resource scheduler

The `min`, `max` attributes are mutually exclusive with `exact`. The description of iteration resource schedulers can be found in documentation.

The `iteration` argument may contain dictionary with following keys:

- `start` (int, optional): iterations start index

- `stop` (int, optional): iterations stop index

The total number of iterations will be `stop - start` (the last iteration index will be `stop - 1`).

Parameters

- **job_attrs** (*dict*) – job description attributes in a simple format
- **kw_attrs** (*dict*) – job description attributes as a named arguments in a simple format

Raises `InvalidJobDescriptionError` – in case of non-unique job name or invalid job description

add_std(*job_attrs=None*, ***kw_attrs*)

Add a new, standard job description (acceptable by the QCG PJM) to the group.

If both arguments are present, they are merged and processed as a single dictionary.

Parameters

- **job_attrs** (*dict*) – job description attributes in a standard format
- **kw_attrs** (*dict*) – job description attributes as a named arguments in a standard format

Raises `InvalidJobDescriptionError` – in case of non-unique job name or invalid job description

remove(*name*)

Remote a job from the group.

Parameters **name** (*str*) – name of the job to remove

Raises `JobNotDefinedError` – in case of missing job in a group with given name

clear()

Remove all jobs from the group.

Returns number of removed elements

Return type int

job_names()

Return a list with job names in group.

Returns job names in group

Return type list(str)

ordered_job_names()

Return a list with job names in group in order they were appended.

Returns ordered job names

Return type list(str)

jobs()

Return job descriptions in format acceptable by the QCG-PJM

Returns a list of jobs in the format acceptable by the QCG PJM (standard format)

Return type list(dict)

ordered_jobs()

Return job descriptions in format acceptable by the QCG-PJM in order they were appended.

Returns a list of jobs in the format acceptable by the QCG PJM (standard format)

Return type list(dict)

load_from_file(*file_path*)
Read job's descriptions from JSON file in format acceptable (StdJob) by the QCG-PJM

Parameters **file_path** (*str*) – path to the file with jobs descriptions in a standard format

Raises InvalidJobDescriptionError – in case of invalid job description

save_to_file(*file_path*)
Save job list to JSON file in a standard format.

Parameters **file_path** (*str*) – path to the destination file

Raises FileError – in case of problems with opening / writing output file.

14.1.3 qcg.pilotjob.api.jobinfo module

class qcg.pilotjob.api.jobinfo.**JobInfo**
Bases: object

Object to store parsed job informations.

name
job name

Type str

status
job status

Type str

nodes
dictionary with node names and list of allocated cores

Type dict(str, int[]), optional

total_cores
number of total allocated cores

Type int

wdir
working directory path

Type str

time
job run time

Type timedelta, optional

iteration
iteration index

Type int, optional

iterations
info about iterations

Type dict, optional

childs
a list of child jobs

Type *JobInfo*[], optional

history
list of job status change moments

Type str[], optional

messages

Type str

static from_child(*job_name*, *child_data*)
Parse information about a sub job.

Parameters

- **job_name** (*str*) – job name
- **child_data** (*dict*) – element of 'childs' from job info response

Returns instance of job info

Return type *JobInfo*

static from_job(*job_data*)
Parse job info response.

Parameters **job_data** (*dict*) – job information obtained with jobInfo request

Returns parsed information

Return type *JobInfo*

14.1.4 qcg.pilotjob.api.manager module

class qcg.pilotjob.api.manager.**Manager**(*address=None*, *cfg=None*)
Bases: object

The Manager class is used to communicate with single QCG-PilotJob manager instance.

We assume that QCG-PilotJob manager instance is already running with ZMQ interface. The communication with QCG-PilotJob is fully synchronous.

Initialize instance.

Parameters

- **address** (*str*) – [proto://]host[:port] the default values for 'proto' and 'port' are respectively - 'tcp' and '5555'; if 'address' is not defined the following procedure will be performed:
  a) if the environment contains QCG_PM_ZMQ_ADDRESS - the value of this var will be used,

  else

  b) the tcp://127.0.0.1:5555 default address will be used
- **cfg** (*dict*) – 'poll_delay' - the delay between following status polls in wait methods 'log_file' - the location of the log file 'log_level' - the log level ('DEBUG'); by default the log level is set to INFO

DEFAULT_ADDRESS_ENV = 'QCG_PM_ZMQ_ADDRESS'

DEFAULT_ADDRESS = 'tcp://127.0.0.1:5555'

DEFAULT_PROTO = 'tcp'

DEFAULT_PORT = '5555'

DEFAULT_POLL_DELAY = 2

send_request(*request*)
Method for testing purposes - allows to send any request to the QCG PJM. The received response is validated for correct format.

Parameters **request** (*dict*) – the request data to send

Returns validated response

Return type dict

resources()
Return available resources.

Return information about current resource status of QCG PJM.

Returns data in format described in 'resourceInfo' method of QCG PJM.

Return type dict

Raises see _send_and_validate_result

submit(*jobs*)
Submit jobs.

Parameters **jobs** (Jobs) – the job descriptions to submit

Returns list of submitted job names

Return type list(str)

Raises

- InternalError - in case of unexpected result format
- see _send_and_validate_result

list()
List all jobs.

Return a list of all job names registered in the QCG PJM. Beside the name, each job will contain additional data, like:

status (str) - current job status messages (str, optional) - error message generated during job processing inQueue (int, optional) - current job position in scheduling queue

Returns dictionary with job names and attributes

Return type dict

Raises

- InternalError - in case of unexpected result format
- see _send_and_validate_result

status(*names*)
Return current status of jobs.

Parameters **names** (*str|list(str)*) – list of job names to get status for

Returns

dictionary with job names and status data in format of dictionary with following keys:
status (int): 0 - job found, other value - job not found message (str): an error description data (dict):

jobName: job name status: current job status

Return type dict

Raises see _send_and_validate_result

info(*names*, ***kwargs*)
Return detailed information about jobs.

Parameters

- **names** (*str|list(str)*) – list of job names to get detailed information about
- **kwargs** (**dict) – additional keyword arguments to the info method, currently following attributes are supported:

withChilds (bool): if True the detailed information about all job's iterations will be returned

Returns

dictionary with job names and detailed information in format of dictionary with following keys:
status (int): 0 - job found, other value - job not found message (str): an error description data (dict):

jobName (str): job name status (str): current job status iterations (dict, optional): the information about iteration job

start: start index of iterations stop: stop index of iterations total: total number of iterations finished: already finished number of iterations failed: already failed number of iterations

childs (list(dict), optional): only when 'withChilds' option has been used, each entry contains:
iteration (int): the iteration index state (str): current state of iteration runtime (dict): runtime information

messages (str, optional): error description runtime (dict, optional): runtime information, see below history (str): history of status changes, see below

The runtime information can contains following keys:

allocation (str): information about allocated resources in form:

NODE_NAME0[CORE_ID0[:CORE_ID1+]][,NODE_NAME1[CORE_ID0[:CORE_ID1+]]....]

the nodes are separated by the comma, and each node contain CPU's identifiers separated by colon : enclosed in square brackets

wd (str): path to the working directory rtime (str): the running time (set at the job's or job's iteration finish) exit_code (int): the exit code (set at the job's or job's iteration finish)

The history information contains multiple lines, where each line has format: YEAR-MONTH-DAY HOUR:MINUTE:SECOND.MILLIS: STATE

The first part is a job's or job's iteration status change timestamp, and second is the new state.

Return type dict

Raises

- InternalError – in case the response format is invalid

- `ConnectionError` – in case of non zero exit code, or if connection has not been established yet

info_parsed(*names*, ***kwargs*)

Return detailed and parsed information about jobs.

The request sent to the QCG-PilotJob manager instance is the same as in `info`, but the result information is parsed into more simpler to use `JobInfo` object.

Parameters

- **names** (*str|list(str)*) – list of job names to get detailed information about
- **kwargs** (**dict) – additional keyword arguments to the info method, currently following attributes are supported:

  withChilds (bool): if True the detailed information about all job's iterations will be returned

Returns a dictionary with job names and information parsed into JobInfo object

Return type dict(str, *JobInfo*)

Raises

- `InternalError` – in case the response format is invalid
- `ConnectionError` – in case of non zero exit code, or if connection has not been established yet

remove(*names*)

Remove jobs from QCG-PilotJob manager instance.

This function might be useful if we want to submit jobs with the same names as previously used, or to release memory allocated for storing information about already finished jobs. After removing, there will be not possible to get any information about removed jobs.

Parameters **names** (*str|list(str)*) – list of job names to remove from QCG-PilotJob manager

Raises

- `InternalError` – in case the response format is invalid
- `ConnectionError` – in case of non zero exit code, or if connection has not been established yet

cancel(*names*)

Cancel jobs execution.

This method is currently not supported.

Parameters **names** (*str|list(str)*) – list of job names to cancel

Raises `InternalError` – always

finish()

Send finish request to the QCG-PilotJob manager, close connection.

Sending finish request to the QCG-PilotJob manager result in closing instance of QCG-PilotJob manager (with some delay). There will be not possible to send any new requests to this instance of QCG-PilotJob manager.

Raises

- `InternalError` – in case the response format is invalid

- `ConnectionError` – in case of non zero exit code, or if connection has not been established yet

cleanup()
Clean up resources.

The custom logging handlers are removed from root logger.

wait4(*names*)
Wait for finish of specific jobs.

This method waits until all specified jobs finish its execution (successfully or not). The QCG-PilotJob manager is periodically polled about status of not finished jobs. The poll interval (2 sec by default) can be changed by defining a 'poll_delay' key with appropriate value (in seconds) in configuration of instance.

Parameters **names** (*str|list(str)*) – list of job names to get detailed information about

Returns dict - a map with job names and their terminal status

Raises

- `InternalError` – in case the response format is invalid
- `ConnectionError` – in case of non zero exit code, or if connection has not been established yet

wait4all()
Wait for finish of all submitted jobs.

This method waits until all specified jobs finish its execution (successfully or not). See 'wait4'.

static is_status_finished(*status*)
Check if status of a job is a terminal status.

Parameters **status** (*str*) – a job status

Returns true if a given status is a terminal status

Return type bool

class qcg.pilotjob.api.manager.**LocalManager**(*server_args=None*, *cfg=None*)
Bases: *qcg.pilotjob.api.manager.Manager*

The Manager class which launches locally (in separate thread) instance of QCG-PilotJob manager

The communication model as all functionality is the same as in `Manager` class.

Initialize instance.

Launch QCG-PilotJob manager instance in background thread and connect to it. The port number for ZMQ interface of QCG-PilotJob manager instance is randomly selected.

Parameters

- **server_args** (*list(str)*) – the command line arguments for QCG-PilotJob manager instance

  --net enable network interface

  --net-port NET_PORT port to listen for network interface (implies –net)

  --net-port-min NET_PORT_MIN minimum port range to listen for network interface if exact port number is not defined (implies –net)

--net-port-max NET_PORT_MAX maximum port range to listen for network interface if exact port number is not defined (implies –net)

--file enable file interface

--file-path FILE_PATH path to the request file (implies –file)

--wd WD working directory for the service

--envschema ENVSCHEMA job environment schema [auto|slurm]

--resources RESOURCES source of information about available resources [auto|slurm|local] as well as a method of job execution (through local processes or as a Slurm sub jobs)

--report-format REPORT_FORMAT format of job report file [text|json]

--report-file REPORT_FILE name of the job report file

--nodes NODES configuration of available resources (implies –resources local)

–log {critical,error,warning,info,debug,notset} log level

--system-core reserve one of the core for the QCG-PJM

--disable-nl disable custom launching method

--show-progress print information about executing tasks

--governor run manager in the governor mode, where jobs will be scheduled to execute to the dependant managers

--parent PARENT address of the parent manager, current instance will receive jobs from the parent manaqger

--id ID optional manager instance identifier - will be generated automatically when not defined

--tags TAGS optional manager instance tags separated by commas

--slurm-partition-nodes SLURM_PARTITION_NODES split Slurm allocation by given number of nodes, where each group will be controlled by separate manager (implies –governor)

--slurm-limit-nodes-range-begin SLURM_LIMIT_NODES_RANGE_BEGIN limit Slurm allocation to specified range of nodes (starting node)

--slurm-limit-nodes-range-end SLURM_LIMIT_NODES_RANGE_END limit Slurm allocation to specified range of nodes (ending node)

each command line argument and (optionaly) it's value should be passed as separate entry in the list

- **cfg** (*dict*) – 'poll_delay' - the delay between following status polls in wait methods 'log_file' - the location of the log file 'log_level' - the log level ('DEBUG'); by default the log level is set to INFO

finish()

Send a finish control message to the manager and stop the manager's process.

Sending finish request to the QCG-PilotJob manager result in closing instance of QCG-PilotJob manager (with some delay). There will be not possible to send any new requests to this instance of QCG-PilotJob manager.

If the manager process won't stop in 10 seconds it will be terminated. We also call the 'cleanup' method.

Raises

- `InternalError` – in case the response format is invalid
- `ConnectionError` – in case of non zero exit code, or if connection has not been established yet

kill_manager_process()

Terminate the manager's process with the SIGTERM signal.

In normal conditions the `finish` method should be called.

CHAPTER 15

Indices and tables

- genindex
- modindex
- search

Python Module Index

q

Index

Symbols

A

C

D

F

H

I

J

K

L

M