QCG-PilotJob

Mar 22, 2023

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 larger workflows 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.

QCG-PilotJob's source code is publicly available at: https://github.com/vecma-project/QCG-PilotJob

1.1 Components

QCG-PilotJob consists of three components:

QCG-PilotJob Core the essential part of the software, provides all basic mechanism needed to use QCG-PilotJob

QCG-PilotJob Command Line Tools a set of command line tools for reporting and analysis of QCG-PilotJob execution

QCG-PilotJob Executor API an alternative, simplified API for QCG-PilotJob

CHAPTER 2

Installation

QCG-PilotJob requires Python version >= 3.6.

All QCG-PilotJob components can be installed by a regular user (without administrative privileges) In the presented instructions we assume such type of installation.

2.1 Preparation of virtualenv (optional step)

In order to make dependency management easier, a good practice is to install QCG-PilotJob into a fresh virtual environment. To do so, we need the latest version of *pip* package manager and *virtualenv*. They can be installed in user's directory by the 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 the following commands:

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

2.2 Installation of QCG-PilotJob packages

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

2.2.1 PyPi

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

```
pip install qcg-pilotjob
```

In a similar way you can install supplementary packages, namely *QCG-PilotJob Command Line Tools* and *QCG-PilotJob Executor API*:

```
pip install qcg-pilotjob-cmds
pip install qcg-pilotjob-executor-api
```

2.2.2 GitHub

To install QCG-PilotJob packages directly from github.com you can use the following commands:

```
pip install --upgrade git+https://github.com/vecma-project/QCG-PilotJob.git
↪#subdirectory=components/core
pip install --upgrade git+https://github.com/vecma-project/QCG-PilotJob.git
↪#subdirectory=components/cmds
pip install --upgrade git+https://github.com/vecma-project/QCG-PilotJob.git
↪#subdirectory=components/executor_api
```

You can also install the packages from a specific branch:

```
pip install --upgrade git+https://github.com/vecma-project/QCG-PilotJob.git@branch_
↪name#subdirectory=components/core
pip install --upgrade git+https://github.com/vecma-project/QCG-PilotJob.git@branch_
↪name#subdirectory=components/cmds
pip install --upgrade git+https://github.com/vecma-project/QCG-PilotJob.git@branch_
↪name#subdirectory=components/executor_api
```

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.

Note: In the example above we presented the simplified API to submit a job. In case of more complicated scenarios we can use the full JSON description to define a submitted job by using `Jobs.add_std` method where all JSON attributes are passed as keyword parameters. The full list of accepted parameters can be found in the `submit` command documentation described in the *File based interface* document.

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: Affinity binding supported by Slurm and used by QCG-PilotJob may not work properly when allocation doesn't contain entire nodes, so we recommended running QCG-PilotJob on allocations with entire nodes reserved.

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 Local-Manager , 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

Running *MPI* programs on HPC systems can be a complex process, as it depends on chosen MPI implementation (*OpenMPI*, *IntelMPI*) and system configuration. Some sites supports launching MPI programs directly with scheduling system client `srun`, but on other ones such applications should be launched with standard `mpirun` command. To get a proper process binding to the specific cores is even harder, especially where programs are launched with `mpirun` command. To support running MPI applications, QCG-PilotJob Manager implements different execution models. The detailed description about those models can be found in *Execution models* section. In following example we are using the default model, where only single process is started by QCG-PilotJob Manager which is typically script that calls `mpirun` or `mpiexec` command. 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:

```python
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'],
```

(continues on next page)

(continued from previous page)

```
    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:

```python
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',
    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:

```json
[
  {
    "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: service.py [-h] [--net] [--net-port NET_PORT]
                                        [--net-pub-port NET_PUB_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]
                                        [--slurm-resources-file SLURM_RESOURCES_FILE]
```

(continues on next page)

(continued from previous page)

```
                                       [--resume RESUME] [--enable-proc-stats] [--
↪enable-rt-stats]
                                       [--wrapper-rt-stats WRAPPER_RT_STATS]
                                       [--nl-init-timeout NL_INIT_TIMEOUT]
                                       [--nl-ready-treshold NL_READY_TRESHOLD] [--
↪disable-pub]
                                       [--nl-start-method NL_START_METHOD]

    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-pub-port NET_PUB_PORT
                                                    port to publish events (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
```

(continues on next page)

(continued from previous page)

```
                                                     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)
      --slurm-resources-file SLURM_RESOURCES_FILE
                                                     path to the file with slurm␣
↪resources description
      --resume RESUME       path to the QCG-PilotJob working directory to resume
      --enable-proc-stats   gather information about launched processes from
                                                     system
      --enable-rt-stats     gather exact start & stop information of launched
                                                     processes
      --wrapper-rt-stats WRAPPER_RT_STATS
                                                     exact start & stop information␣
↪wrapper path
      --nl-init-timeout NL_INIT_TIMEOUT
                                                     node launcher init timeout (s)
      --nl-ready-treshold NL_READY_TRESHOLD
                                                     percent (0.0-1.0) of node␣
↪launchers registered when
                                                     computations should start
      --disable-pub         disable status publisher interface
      --nl-start-method NL_START_METHOD
                                                     method to start node launchers␣
↪(ssh,slurm - default)
```

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

Execution models

The QCG-PJM service manages individual cores, so it assigns specific cores to the tasks. From the performance perspective, binding tasks to the cores is more efficient as it separates tasks from each other.

Note: To support CPU binding, the service must have information about physical available cores in the system. This information is provided by *SLURM* in a created allocation, but it is not available in case of logical resources, i.e. where user defines *virtual* cores and nodes. So currently the binding is supported only when the QCG-PJM service is run inside a *SLURM* allocation.

The binding of single core tasks is achieved with:

- Custom Launching Agent, that uses `taskset` command,
- SLURM built-in mechanism based on `--cpu-bind` option of the `srun` command.

Currently, the parallel tasks that require more than one core are launched only by the `srun` or `mpirun` commands. The *mask_cpu* flag of the `srun`'s `` ` ``--cpu-bind`` parameter will contain the CPU masks for all allocated nodes separated with the comma. When `mpirun` command is used to launch parallel task, either the `--rankfile` parameter is used for OpenMPI model or `I_MPI_PIN_PROCESSOR_LIST` environment variable for Intel MPI model.

Additionally, for all tasks launched by the Slurm with binding supported, the *QCG_PM_CPU_SET* environment variable will be available and set with core identifiers separated with comma.

To support process affinity for different parallel applications, QCG-PJM supports different execution models. Currently the following models are available:

- `default` - in this model only a single process is launched within the allocation, which is prepared based on task's resource requirements, the allocation description can be found in environment variables, such as: - `SLURM_NODELIST` - list of allocated nodes separated by comma character - `SLURM_NTASKS` - total number of cores - `SLURM_TASKS_PER_NODE` - number of allocated cores on following nodes, where each element can be in a form `NODE_NAME` (a single core on a node) or `NODE_NAME(xNUM_OF_CORES)` (many cores on a single node) - `QCG_PM_CPU_SET` - list of core identifiers on following nodes separated by a comma character
- `threads` - is designed for running OpenMP tasks on a single node, the process is started with the `srun` command with the `--cpus-per-task` parameter set according to a number of cores defined in resource requirements

- `openmpi` - the processes are started with the `mpirun` command with rankfile created based on task's resource requirements
- `intelmpi` - the processes are started with the `mpirun` command (from the IntelMPI distribution) with defined multiple components, where each component describing execution node, contains `-host` element and `I_MPI_PIN_PROCESSOR_LIST` arguments set according to the allocated resources
- `srunmpi` - the processes are started with the `srun` command with `--cpu-bind` parameter set according to the allocated resources; this model should be used only on sites that have penMPI/IntelMPI/Slurm environments configured properly.

The `openmpi` and `srunmpi` provide additional configuration options that that can be defined in the element `model_opts`. Currently the following options are supported:

- `mpirun` (str) - path to the *mpirun* command that should be used to launch applications, if not defined the default command (should be in the `PATH` environment variable) is used
- `mpirun_args` (list) - additional arguments that should be passed to the *mpirun* command

We recommend usage of `srunmpi` model for MPI applications on HPC sites wherever srun is properly configured to execute MPI codes.

Note: It is important to define for the `intelmpi`, `srunmpi` and `openmpi` models appropriate IntelMPI/OpenMPI modules in *executable/modules* element of the job description or to load them before staring QCG-PJM.

9.1 Examples

1) To use different MPI implementation applications in a single workflow we can define appropriate options

```
[
  {
    "request": "submit",
    "jobs": [
      {
        "name": "intelmpi_task",
        "execution": {
          "exec": "/home/user/my_intelmpi_application",
          "model": "intelmpi",
          "model_opts": {
              "mpirun": "/opt/exp_soft/local/skylake/intel/compilers_and_libraries_
↪2020.4.304/linux/mpi/intel64/bin/mpirun"
          },
          "modules": [ "impi" ]
        },
        "resources": {
          "numCores": {
            "exact": 8
          }
        }
      },
      {
        "name": "openmpi_task",
        "execution": {
          "exec": "/home/user/my_openmpi_application",
          "model": "openmpi",
```

(continues on next page)

(continued from previous page)

```
          "model_opts": {
              "mpirun": "/opt/exp_soft/local/skylake/openmpi/4.1.0_gcc620/bin/mpirun",
              "mpirun_args": [ "--mca", "rmaps_rank_file_physical", "1" ]
          },
          "modules": [ "openmpi/4.1.0_gcc620" ]
        },
        "resources": {
          "numCores": {
            "exact": 8
          }
        }
      }
    ]
  }
]
```

With this input, QCG-PilotJob service will launch task's *intelmpi_task* application `/home/user/my_intelmpi_application` with the mpirun command path `/opt/exp_soft/local/skylake/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel64/bin/mpirun` and additionally it will load *impi* module. The second task's *openmpi_task* application `/home/user/my_openmpi_application` will be launched with the command `/opt/exp_soft/local/skylake/openmpi/4.1.0_gcc620/bin/mpirun` with additional arguments `--mca rmaps_rank_file_physical 1` and the module `openmpi/4.1.0_gcc620` loaded before the application's start.

The description for the API looks similar:

```
jobs = Jobs()
jobs.add(name = 'intelmpi_task', exec = '/home/user/my_intelmpi_application',␣
↪numCores = { 'exact': 4 }, model = 'intelmpi', model_opts = { 'mpirun': '/opt/exp_
↪soft/local/skylake/intel/compilers_and_libraries_2020.4.304/linux/mpi/intel64/bin/
↪mpirun' }, modules = [ 'impi' ])
jobs.add(name = 'openmpi_task', exec = '/home/user/my_openmpi_application', numCores␣
↪= { 'exact': 4 }, model = 'openmpi', model_opts = { 'mpirun': '/opt/exp_soft/local/
↪skylake/openmpi/4.1.0_gcc620/bin/mpirun', 'mpirun_args': ['--mca', 'rmaps_rank_file_
↪physical', '1']}, modules = [ 'openmpi/4.1.0_gcc620' ])
```

2) Instead of compiled application, it is possible to use Bash script from which the application is called later. It gives us more possibilities to configure the environment for the application. For example using the following input description:

```
[
  {
    "request": "submit",
    "jobs": [
      {
        "name": "openmpi_task",
        "execution": {
          "exec": "bash",
          "args": [ "-l", "./app_script.sh" ],
          "model": "openmpi",
        },
        "resources": {
          "numCores": {
            "exact": 8
          }
        }
```

(continues on next page)

(continued from previous page)

```
            }
        ]
    }
]
```

The script `app_script.sh` could look like the following:

```
#!/bin/bash

module load openmpi/4.1.0_gcc620
/home/user/my_openmpi_application
```

> **Warning:** It is important to remember, that for the parallel task with a model different that default, there will be as many instances created of the script as the required number of cores. Thus the actions that should be executed only once per all application's processes should be enclosed in the following block:

```
if [ "x$OMPI_COMM_WORLD_RANK" == "x0" ] || [ "x$PMI_RANK" == "x0" ]; then
  # actions in this block will be executed only for rank 0 of OpenMPI/IntelMPI␣
↪applications
endif
```

CHAPTER 10

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.

10.1 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.

10.2 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.

10.3 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.

10.3.1 `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 either *start* (*optional*) and *stop* or *values* keys must be defined; the total number of iterations will be *stop* - *start* (the last index of the sub-job will be *stop* - *1*) in case of boundary definition or lenght of *values* array
- `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 following values are supported:
    - `threads` - job's iteration is launched with *srun* command on a single node with as many cpus per task as declared in `resources` element
    - `intelmpi` - job's iteration is launched with *mpirun* command (or command defined in element `model_opts/mpirun`) with the IntelMPI set of arguments, additional arguments for *mpirun* command can be declared in element `model_opts/mpirun_args`
    - `openmpi` - job's iteration is launched with *mpirun* command (or command defined in element `model_opts/mpirun`) with the OpenMPI set of arguments, additional arguments for *mpirun* command can be declared in element `model_opts/mpirun_args`
    - `srunmpi` - job's iteration is launched with *srun* command on as many number of nodes and cores as declared in `resources` element
    - `default` - job's iteration is launched as a single process with environment variable *QCG_PM_CPU_SET* containing allocated cores on a set of declared nodes, the allocated nodes can be obtained from *QCG_PM_NODELIST* environment variables

    - `model_opts` (*optional*) `Dict` - the additional arguments used in some of the models, currently the following keys are supported

        * `mpirun` (*optional*) `String` - the path to the command to be used in `srunmpi` and `openmpi` models

        * `mpirun_args` (*optional*) `Array of String` - the additional arguments that should be passed to the `mpirun` command in `srunmpi` and `openmpi` models

- `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",
          "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"
    ]
  }
}
```

10.3.2 `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
      }
    }
  }
}
```

10.3.3 `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).

10.3.4 `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": {
        "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"
        }
      }
    }
  }
}
```

10.3.5 `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"
}
```

10.3.6 cancelJob

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

10.3.7 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
}
```

10.3.8 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
}
```

10.3.9 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 11

Executor API

Beta

Executor API is an alternative, simplified programming interface for QCG-PilotJob. In some aspects it mimics an interface of `concurrent.futures.Executor` and may therefore be appealing to many Python programmers. However, since this interface is still under development, it is dedicated mostly for less-demanding use-cases.

Executor API is based on the basic API of QCG-PilotJob and therefore it inherits core elements from that API. On the other hand, in order to support definition of the common execution scenarios, many elements of basic API have been hidden behind simplified interface.

11.1 Installation

Executor API can be installed from PyPi, with the following command:

```
$ pip install qcg-pilotjob-executor-api
```

11.2 Usage

Before we present more details about the usage of Executor API, let's outline a minimal working example:

```python
from qcg.pilotjob.executor_api.qcgpj_executor import QCGPJExecutor
from qcg.pilotjob.executor_api.templates.basic_template import BasicTemplate

with QCGPJExecutor() as e:
    f = e.submit(BasicTemplate.template, name='tj', exec='date')
    f.result()
```

This example shows how *Executor API* can be used to run specific command, here `date`, within a QCG-PilotJob task.

The interesting part starts on the 4th line. Here we create `QCGPJExecutor`, which is an entry point to QCG-PilotJob. Actually, behind the scenes `QCGPJExecutor` initialises the QCG-PilotJob manager service and it plays a role of a proxy to its methods.

Once created, `QCGPJExecutor` allows us to submit tasks for the execution within QCG-PilotJob. An example invocation of the `submit` method is shown on the 5th line. The first and the most interesting argument to this method is template. The template is actually a *Callable* that returns a tuple consisting of string and dictionary. The string need to be a QCG-PilotJob submit request description written in a JSON format with optional placeholders for substitution of specific parameters, while the dictionary may be used to set default values for placeholders. The next parameters of the method are optional and dependent on the selected template - their role is to provide values for the actual substitution of placeholders.

In the example above we use a predefined template called `BasicTemplate.template`, which requires only two parameters to be provided, namely `name` and `exec`.

The `submit` method returns a `QCGPJFuture` object, which provides methods associated with the execution of submission. For instance, the invocation `f.result()` in the example above, blocks processing until the task is not completed and then returns the status of its execution.

11.2.1 QCGPJExecutor

`QCGPJExecutor` is an approximate implementation of the `concurrent.futures.Executor` interface, but instead of execution of functions using threads or multiprocessing module like it takes place in case of python build-in executors, here we execute QCG-PilotJob's tasks.

Technically, `QCGPJExecutor` is a kind of proxy over the QCG-PilotJob manager and at the expense of some flexibility of the covered service, it provides simpler interface. `QCGPJExecutor`'s constructor can be invoked without any parameters and then it is started with default settings.

However, in order to enable easy configuration of the commonly changed settings, several optional parameters are provided. One of such parameters is `resources` which may be useful for testing QCG-PilotJob on a local laptop.

`QCGPJExecutor` implements *ContextManager*'s methods that allow for its easy usage with the `with` statements. When the `with` statement is used, python will automatically take care of releasing `QCGPJExecutor`'s resources.

When the `QCGPJExecutor` is constructed outside the `with` statement, it needs to be released manually, using the `shutdown` method.

For the full reference of the `QCGPJExecutor` module see *`qcg.pilotjob.executor_api.qcgpj_executor`*.

11.2.2 Submission of tasks

The key method offered by QCGPJExecutor is `submit`. The call of this method adds a new task (or tasks, depending on the usage scenario) to the QCG-PilotJob's queue to be executed once resources are available and dependencies satisfied. The method takes the following arguments:

1. **`fn:`** a callable that returns a tuple representing a template. The first element of the tuple should be a string containing a QCG-PilotJob submit request expressed in a JSON format compatible with the *QCG-PilotJob's interface*. The string can include placeholders (identifiers preceded by $ symbol) that are the target for substitution. The second element of a tuple is dictionary which may be used to assign default values for substitution of selected placeholders.
2. **`*args:`** a set of dicts which contain parameters that will be used to substitute placeholders defined in the template.
3. **`**kwargs:`** a set of keyword arguments that will be used to substitute placeholders defined in the template.

Note: In the process of substitution `**kwargs` overwrite `*args` and `*args` overwrite defaults

11.2.3 Example template

In order to understand how to use or create templates, possibly the best option is to look at the example. `BasicTemplate` class, which is delivered with the QCG-PilotJob Executor API, provides a predefined template method that was already used in the example above. It is a simple example, but can give a good overview.

```
class BasicTemplate(QCGPJTemplate):
    @staticmethod
    def template() -> Tuple[str, Dict[str, Any]]:
        template = """
        {
            'name': '${name}',
            'execution': {
                'exec': '${exec}',
                'args': ${args},
                'stdout': '${stdout}',
                'stderr': '${stderr}'
            }
        }
         """

        defaults = {
            'args': [],
            'stdout': 'stdout',
            'stderr': 'stderr'
        }

        return template, defaults
```

Here, accordingly with the expectations, the function returns `template` and `defaults`. The `template` is a JSON dictionary representing a QCG-PilotJob *submit request*. What is important, it includes a set of `${}` placeholders. These placeholders may be substituted by the parameters provided to the `submit` method. For some of the placeholders, default values are already predefined in a `defaults` dictionary, and these parameters don't need to be substituted if there is no concrete reason for this. The rest of placeholders, namely `{name}` and `{exec}`, don't have default values and therefore they need to be substituted by parameters provided to the `submit`.

Let's see how example invocations of the `submit` method for this template can look like:

```
e.submit(BasicTemplate.template, name='tj', exec='date')
e.submit(BasicTemplate.template, name='tj', exec='sleep', args=['10'])
```

11.2.4 QCGPJFuture

The `submit` method returns `QCGPJFuture` object, which plays a role of a handler for the submission. Thus, using the returned `QCGPJFuture` object it is possible to make queries to check if the submitted task has been finished, with the `done` method, or request the cancellation of an execution with the `cancel` method. As it was presented in the attached example, it is also possible to invoke blocking wait until the task is finished with the `result` method. For the full reference of methods provided by `QCGPJFuture` see *`qcg.pilotjob.executor_api.qcgpj_future`*.

CHAPTER 12

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`

12.1 `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.

12.2 `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 13

Resuming prematurely interrupted computations

13.1 General

The QCG-PilotJob Manager service implements mechanism for resuming prematurely interrupted computations. All incoming job submission requests, as well as the finished job iterations are recorded to allow resuming job execution. The current state is placed in files `track.*` in auxiliary directory (the `.qcgpjm-service-*` in the working directory). It is worth to mention that started, but not finished job iterations will be started again, so if they don't implement automatic computation checkpointing, they will re-start from begin.

13.2 Invocation

To resume the QCG-PilotJob Manager with previous jobs, the `resume` command line option must be used with path either direct to the auxiliary QCG-PilotJob Manager directory or to the working directory where auxiliary directory is placed (in case where there are many auxiliary directories in the working directory, the last modified one will be automatically selected).

Note: Currently during the resume operation, non of previously used command line option will be re-used. So if for example the working directory has been specified in original QCG PilotJob Manager start, the same working directory should be used during resuming.

Example invocation:

```
qcg-pm-service --wd prev_work_dir --resume prev_work_dir/.qcgpjm-service-LAPTOP-
↪CNT0BD0F.5091
```

13.3 Operation

After resume, the QCG-PilotJob Manager will re-use the pointed auxiliary directory, so all log files, current tracking status and job reports will be appended to the previous files. Thus there is no problem to resume, already resumed computations.

13.4 Issues

Currently the resume mechanism is not supported in resource partitioning mode.

CHAPTER 14

Performance statistics

The QCG-PilotJob service provides tools which, based on service logs, allow you to analyse the efficiency of resource use. Please note that the current implementation is the first version of these tools and may not be free from errors.

14.1 Performance measurements and data collection

Due to the potential load on the QCG-PilotJob service and its asynchronous nature, in order to accurately measure its performance it is necessary to use external metrics to determine when a particular job started and ended. In order to generate such metrics, a wrapper program was created in the C language. The role of this program is to register the moment when the application indicated by the arguments started, and to register the moment when it ended. The collected data is sent through a named pipe (data sent through such a communication channel is not stored in the file system, so the file system performance does not affect the efficiency of communication) to the QCG-PilotJob service agent running on the local compute node. When all calculations are completed, the collected data are written to the QCG-PilotJob service logs.

Note that currently all parallel tasks (requiring more than one core) are started using the "srun" command (provided by the Slurm queue system). Therefore, the wrapper used with the "srun" command will register the moment the "srun" command itself is run, and not the target processes of the parallel application. Therefore, please note that the metrics and analysis presented here do not take into account the latency of the queueing system itself

14.1.1 Wrapper installation

As the wrapper itself is written in C, it requires compilation on the target computing cluster.

Source code (`qcg_pj_launch_wrapper.c`) can be downloaded from the **develop** branch of QCG-PilotJob project repository on GitHub, e.g:

```
$ wget https://raw.githubusercontent.com/vecma-project/QCG-PilotJob/develop/utils/qcg_
↪pj_launch_wrapper.c
```

and then compiled using the compiler of your choice:

```
$ gcc -Wall -o qcg_pj_launch_wrapper qcg_pj_launch_wrapper
```

or

```
$ icc -Wall -o qcg_pj_launch_wrapper qcg_pj_launch_wrapper
```

The compiled program should be placed in a path accessible from all compute nodes.

14.1.2 Launch of QCG-PilotJob service with collection of external metrics

In order to collect external metrics related to application execution time, two arguments must be passed when starting the QCG-PilotJob service:

- `--enable-rt-stats` - enabling the collection of statistics
- `--wrapper-rt-stats` - indication of wrapper location (the path to the installed wrapper program)

An example call to the QCG-PilotJob service with the job description placed in a JSON file and the collection of external metrics looks like this:

```
$ qcg-pm-service --wd out-dir --file-path mpi_iter_mixed.json --enable-rt-stats --
↪wrapper-rt-stats /home/piotrk/runtime_wrapper/qcg_pj_launch_wrapper
```

In this case, all logs and metrics will be in the 'out-dir' subdirectory when the calculation is complete.

14.2 Analysis tool - qcg-pm-report

The QCG-PilotJob package provides a qcg-pm-report program to enable performing a number of analyses based on the data collected by the wrapper and available in a working directory after the QCG-PilotJob's run. It allows to assess the performance of the entire workflow, elementary tasks as well as to get information about resources utilisation.

A description of the available commands of qcg-pm-report is given below.

14.2.1 stats

The stats command displays basic metrics about the executed workflow as a whole, such as:

- `total jobs`, `failed jobs` - number of running jobs and number of failed jobs
- `total cores`, `total nodes` - number of available nodes and total number of available cores on all nodes
- `service started`, `service finished`, `service runtime` - the start and end time of the QCG-PilotJob service and the difference between these dates; note that the start time of the QCG-PilotJob service may be slightly delayed with respect to the start time of the job in the Slurm service, as the time reported by the queue system does not take into account the time associated with loading the Python environment and the QCG-PilotJob service modules; in practice, however, this difference should not be greater than 1-2 seconds.
- `init overhead`, `finish overhead`, `total overhead` - these are respectively the time markups for the initiation of the QCG-PilotJob service (the time from the start of the service to the start of the first job within it), the finish time of the QCG-PilotJob service (the time from the end of the last job within it to the end of the QCG-PilotJob service) and the sum of the two previously mentioned
- `overhead ratio` - the ratio of `total overhead` to `service runtime`
- `overhead core-hours` - it is a product of available cores and `total overhead` value (expressed in hours)

An example of a generated report:

```
$ qcg-pm-report stats out/intelmpi-mpi-iter-mixed-large-10128902/
                                  total jobs: 2000
                                 failed jobs: 0
                                 total cores: 960
                                 total nodes: 40
                          service started: 2021-04-14 14:20:08.223523
                         service finished: 2021-04-14 14:23:21.028799
                          service runtime: 192.81 secs
                                init overhead: 4.08 secs
                          finish overhead: 1.57 secs
                           total overhead: 5.65 secs
                           overhead ratio: 2.9
                  overhead core-hours: 1.51
```

14.2.2 launch-stats

The launch-stats command is used to generate a report showing delays in launching and recording the completion of jobs by the QCG-PilotJob service. The following metrics are generated:

- `total start overhead` - total (for all jobs) time difference between starting the job by the QCG-PilotJob service and the actual start of the job (information registered by the running wrapper), expressed in seconds
- `total finish overhead` - total (for all jobs) difference in time between the actual finish of the job (information registered by the running wrapper) and the moment when it was registered by the QCG-PilotJob service, expressed in seconds
- `total start and finish overhead` - sum of the two previous metrics
- `average job start overhead` - average delay in starting a single job
- `average job finish overhead` - average delay in handling the completion of a single job
- `average job total overhead` - average total delay in starting and handling the completion of a single job
- `average real job run time` - average real run time of a single job (determined by metrics sent by wrapper)
- `average qcg job run time` - average duration of a single job (determined from the times recorded by the QCG-PilotJob service)
- `average job overhead per runtime` - percentage ratio of `average job total overhead` to the `average real job run time`
- `generated for total jobs` - the number of jobs for which a report was generated, i.e. the number of all jobs for which metrics were recorded (these were provided by the wrapper

An example of a generated report:

```
$ qcg-pm-report launch-stats out/intelmpi-mpi-iter-mixed-large-10128902/
                                                       total start overhead: 29.2834
                                                      total finish overhead: 62.1390
                                 total start and finish overhead: 91.4224
                                          average job start overhead: 0.0146
                                         average job finish overhead: 0.0311
                                          average job total overhead: 0.0457
                                               average real job run time: 16.5664
```

(continues on next page)

(continued from previous page)

```
                                  average qcg job run time: 16.6121
                      average job overhead per runtime (%): 0.29
                                  generated for total jobs: 2000
```

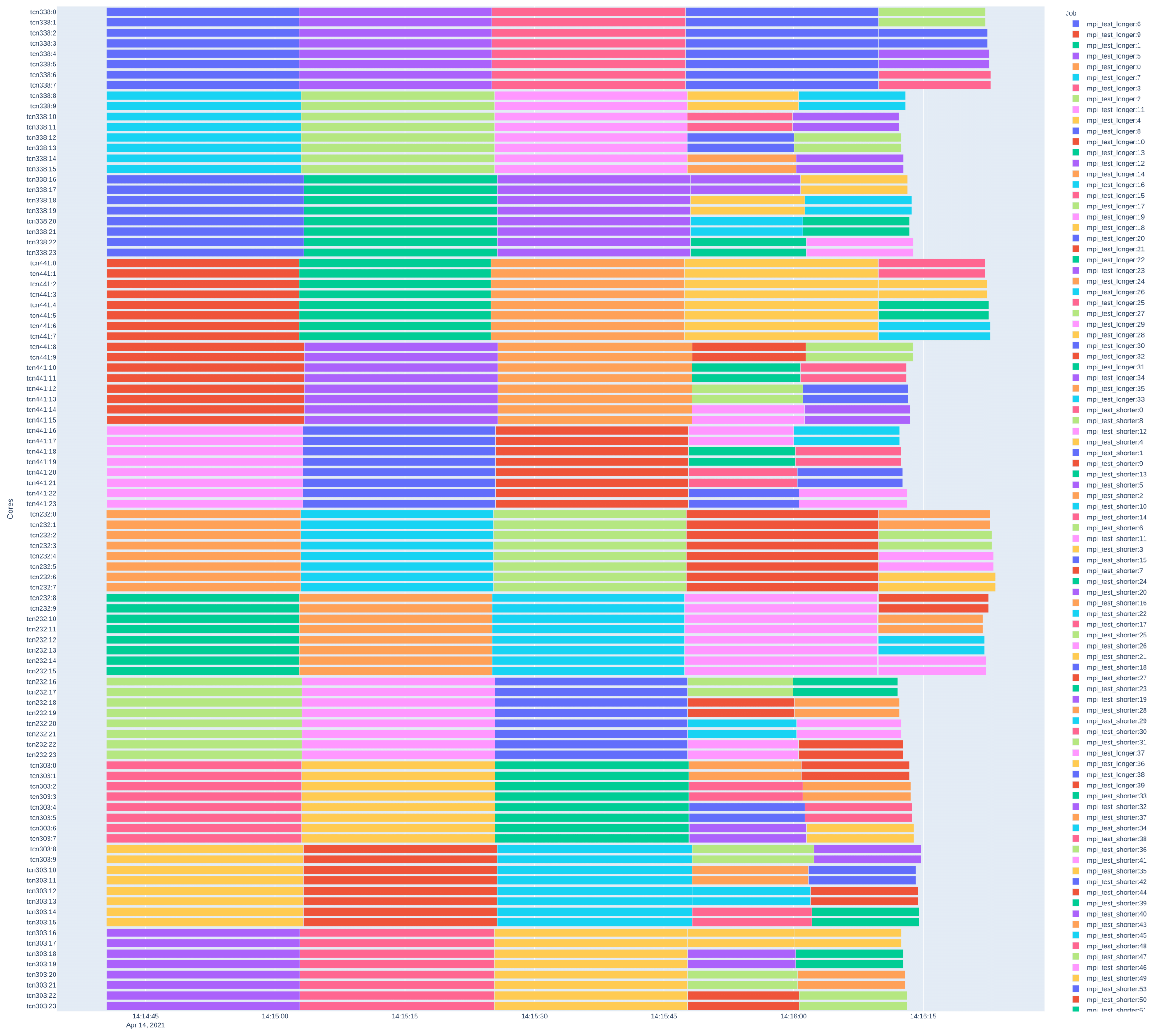
14.2.3 gantt

The `gantt` command is used to generate a timeline plot with the start and end of each task running on the allocated resources marked. This plot shows an overall view of the scheduling plan. In order to generate the plot, as an additional argument, in addition to the path to the working directory of the task, the name of the target file where the plot should be saved should be given. Supported files include: pdf, png, jpeg. Note - the time to generate the graph depends on the number of resources, the number of tasks and the duration of the entire workflow, and for larger scenarios can be a time-consuming operation. In case of a scenario running 2000 tasks on 960 cores and 40 nodes, the graph generation may take up to 3 minutes.

Example of chart generation:

```
$ qcg-pm-report gantt out/intelmpi-mpi-iter-mixed-10128873/ gantt.pdf
```

A sample chart generated:



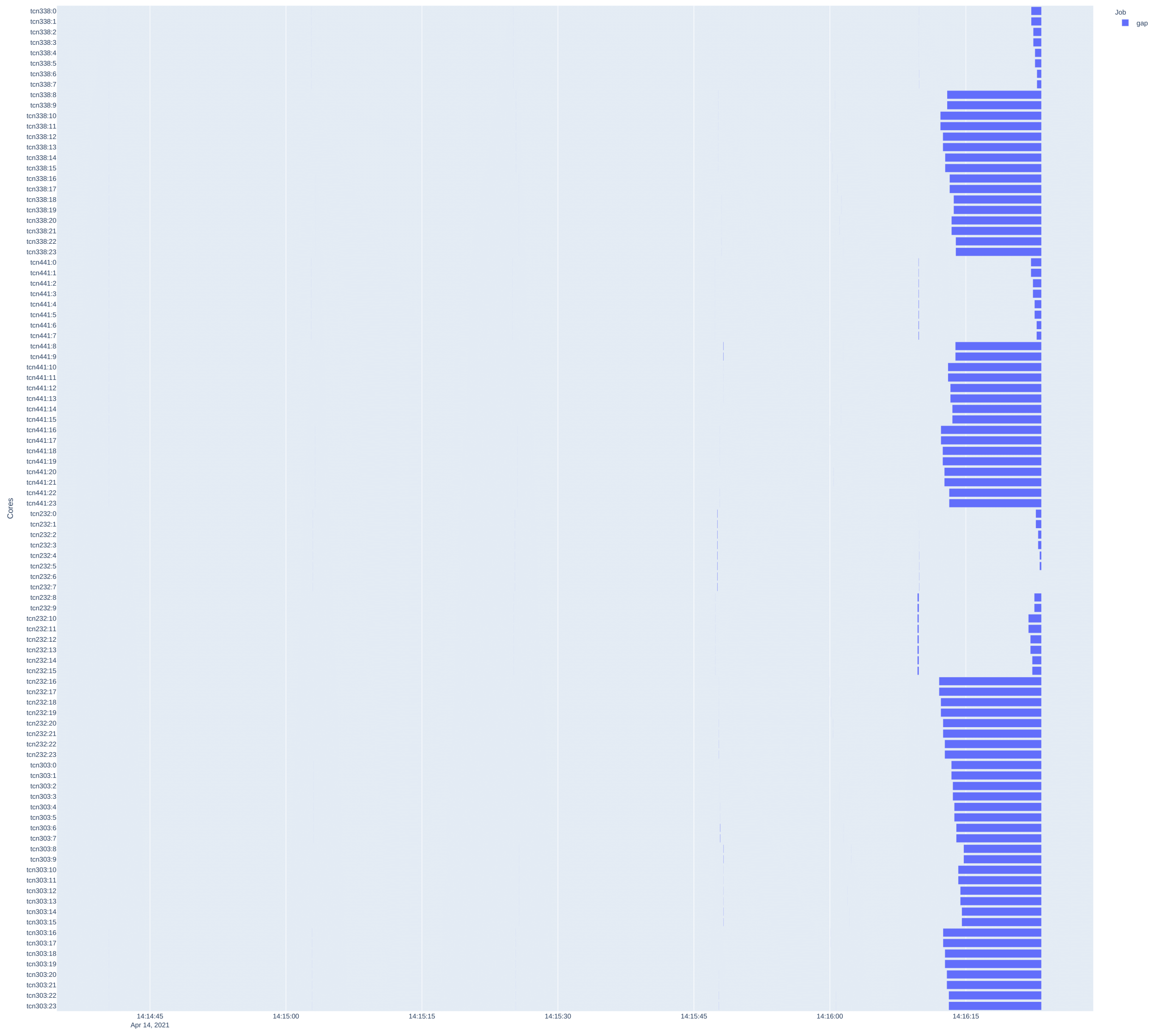
14.2.4 gantt-gaps

The `gantt-gaps` command is used to generate a timeline plot with marked moments when resources were not used. This plot is in a way the negative of the plot generated by the "gantt" command. As an additional argument, in addition to the path to the working directory of the task, you should give the name of the target file where the plot should be saved. The following files are supported: pdf, png, jpeg. Note - the time to generate the graph depends on the number of resources, the number of tasks and the duration of the entire workflow, and for larger scenarios can be a time-consuming operation. In the case of a scenario running 2000 tasks on 960 cores and 40 nodes, graph generation can take up to 3 minutes.

Example of chart generation:

```
$ qcg-pm-report gantt-gaps out/intelmpi-mpi-iter-mixed-10128873/ gantt-gaps.pdf
```

A sample chart generated:



14.2.5 rusage

The `rusage` command is used to generate a report showing the usage of available resources. In the basic version, it displays two metrics:

- `used cores` - number of cores on which tasks were running
- `average core utilization (%)` - percentage core utilization, it is calculated as the average value of the percentage utilization of a single core (on which at least one task was running)

The single core utilisation percentage is calculated as the ratio of the time during which a job was actually running on a given node (based on metrics sent by the wrapper) to the total time of running the QCG-PilotJob service (see `service runtime` in the `stats` command).

An example of a generated report:

```
$ qcg-pm-report rusage out/intelmpi-mpi-iter-mixed-large-10128902/
                              used cores: 960
             average core utilization (%): 94.2%
```

Running the `rusage` command with the `--details` parameter will list the usage percentages for each core.

For example, a generated report containing details:

```
$ qcg-pm-report rusage --details out/intelmpi-mpi-iter-mixed-large-10128902/
                              used cores: 960
             average core utilization (%): 94.2%
 tcn1261
         0    : 95.9%, unused 7.8734 s
         1    : 95.9%, unused 7.8734 s
         2    : 96.1%, unused 7.4926 s
         3    : 96.1%, unused 7.4926 s
         4    : 96.2%, unused 7.3733 s
         5    : 96.2%, unused 7.3733 s
         6    : 90.0%, unused 19.3723 s
         7    : 90.0%, unused 19.3723 s
         8    : 90.4%, unused 18.5259 s
         9    : 90.4%, unused 18.5259 s
         10 : 90.1%, unused 19.0094 s
         11 : 90.1%, unused 19.0094 s
         12 : 90.3%, unused 18.7818 s
         13 : 90.3%, unused 18.7818 s
         14 : 90.3%, unused 18.7562 s
         15 : 90.3%, unused 18.7562 s
         16 : 95.4%, unused 8.8197 s
         17 : 95.4%, unused 8.8197 s
         18 : 95.5%, unused 8.6833 s
         19 : 95.5%, unused 8.6833 s
         20 : 95.6%, unused 8.5297 s
         21 : 95.6%, unused 8.5297 s
         22 : 95.8%, unused 8.0063 s
         23 : 95.8%, unused 8.0063 s
```

(due to the length of the report, data for one computational node only are included).

14.2.6 efficiency

Command `efficiency` is used to show the percentage of resource usage, excluding the time when the resource was inactive due to a scheduling plan. Resource usage time is counted as time when any task was running or when another task was waiting for another resource to free up. The efficiency metric only takes into account delays due to QCG-PilotJob's job launching and termination handling.

An example of a generated report:

```
$ qcg-pm-report efficiency out/intelmpi-mpi-iter-mixed-large-10128902/
                       used cores: 960
     average core utilization (%): 99.6%
```

CHAPTER 15

Performance tuning

15.1 Node launcher agents

To launch user jobs in Slurm allocation, the QCG PilotJob service is using its own services that are started on each of the allocation's node. This sub-service is called node launcher agent. When used on big allocations, that contains hundred of nodes, the process of starting node launcher agents can therefore take longer. Also there is a chance that due to some circumstances (software or hadrware), the process of node launching agent fail. To deal with such cases, there are command line options to control the process of starting node launcher agents:

- `--nl-init-timeout NL_INIT_TIMEOUT` - the `NL_INIT_TIMEOUT` specify number of seconds the service should wait for all node launcher agents start (`600` by default),
- `--nl-ready-treshold NL_READY_TRESHOLD`- the `NL_READY_TRESHOLD` value (from range 0.0 - 1.0) control the ration of ready node launcher agents when process of executing workflow can be started (`1.0` by default).

After starting of all node launcher agents, the QCG PilotJob service waits up to `NL_INIT_TIMEOUT` until `NL_READY_TRESHOLD` * *total number of agents* report it's successfull start. When it happen, the execution of the workflow begins, and jobs are submitted only to those nodes where launcher agents successfully started. All other agents may register after this time enabling their nodes for exection. When, from some reason the required number of agents did not register in given interval, the QCG PilotJob service should report the error and exit without starting workflow execution.

15.2 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 16

Processes statistics

QCG-PilotJob Manager support gathering metrics about launched processes. This feature enables analysis of application behavior such as:

- process tree inspection, from the lauching by QCG-PilotJob to the final application process (with all intermediate steps, such as `srun`, `mpirun`, `orted` etc. with time line (delays between following processes in tree)
- process localization, such as: node name and cpu affinity
- coarse process metrics: cpu and memory utilization

Note: Please note that this is initial version of gathering process metrics, and due to the implementation obtained data might be precise.

16.1 How it works

When `--enable-proc-stats` has been used with QCG-PilotJob (either as a command line argument or as argument to the `server_args` parameter of `LocalManager` class), the launcher agent started on each of the Slurm allocation's node stars thread that periodically query about processes started in the local system. Because collecting statistics about all processes in the system would take too much time, and thus reduce the frequency of queries, launcher agent only checks the descendants of the `slurmstepd` process. This process is responsible for starting every user process in Slurm, including launcher agent. Therefore we register all process started by launcher agent, and also processes started by MPI that is configured with Slurm (in such situation, Slurm asks `slurmstepd` daemon to launch instancesof MPI application). Every descendant process of the `slurmstepd` is registered with it's identifier (`pid`) and basic statistics, such as:

- `pid` - process identifier
- process name (in most cases name of the executable file)
- command line arguments
- parent process name

- parent process identifier
- cpu affinity - list of available cores
- accumulated process times in seconds (the detailed description of the format is available at https://psutil.readthedocs.io/en/latest/index.html?highlight=cpu_times#psutil.cpu_times)
- memory information (the detailed description of the format is available at https://psutil.readthedocs.io/en/latest/index.html?highlight=cpu_times#psutil.Process.memory_info)

Currently the data from each query is stored in in-memory database and saved to the file at launcher agent finish. The destination file, created in QCG-PilotJob working directory, will have name of following pattern: `ptrace_{node_name}_{current_date}_{random_number}.log`. In the future releases we are planing to send those information to external service that will allow to run-time monitoring of gathered statistics.

It is worth to mention about some shortcoming of such approach:

- because the processes are queried with some frequency (currently every 1 second), there is a chance that very short living process will not be registered,
- there is a possibility that after finish of some process, another one with the same identifier will be created later

16.2 How to use

First of all the `--enable-proc-stats` arument of the QCG-PilotJob service must be used either as command line argument, or as one of the `server_args` element in `LocalManager` class. When all QCG-PilotJob workflow finish, the working directory should contain one or many (for each of allocation nodes there should be instance of this file) files named `ptrace_{node_name}_{current_date}_{random_number}.log`. Those files contains information about processes statistics in JSON format. To analyze this data, QCG-PilotJob provides `qcg-pm-processes` command line tool. Documentation about this tool is available with:

In this example we submitted 6 instances of `mpi_iter` job, where each instance is an MPI application started on 8 cores.

To get process tree of the first instance of this job:

```
$ qcg-pm-processes tree out-api-mpi-iter mpi_iter:0
job mpi_iter:0, job process id 28521, application name openmpi_3.1_gcc_6.2_app
  --28521:bash (bash -c source /etc/profile; module purge; module load openmpi/3.1.4_
↪gcc620; exe) node(e0025) created 2021-03-25 17:18:34.350000
        --29537:openmpi_3.1_gcc_6.2_app (/tmp/lustre_shared/plgkopta/qcgpjm-altair/
↪examples/openmpi_3.1_gcc_6.2_app) node(e0025) after 3.83 secs
        --29542:openmpi_3.1_gcc_6.2_app (/tmp/lustre_shared/plgkopta/qcgpjm-altair/
↪examples/openmpi_3.1_gcc_6.2_app) node(e0025) after 3.86 secs
        --29638:openmpi_3.1_gcc_6.2_app (/tmp/lustre_shared/plgkopta/qcgpjm-altair/
↪examples/openmpi_3.1_gcc_6.2_app) node(e0025) after 4.01 secs
        --29608:openmpi_3.1_gcc_6.2_app (/tmp/lustre_shared/plgkopta/qcgpjm-altair/
↪examples/openmpi_3.1_gcc_6.2_app) node(e0025) after 3.98 secs
        --29547:openmpi_3.1_gcc_6.2_app (/tmp/lustre_shared/plgkopta/qcgpjm-altair/
↪examples/openmpi_3.1_gcc_6.2_app) node(e0025) after 3.88 secs
        --29579:openmpi_3.1_gcc_6.2_app (/tmp/lustre_shared/plgkopta/qcgpjm-altair/
↪examples/openmpi_3.1_gcc_6.2_app) node(e0025) after 3.95 secs
        --29554:openmpi_3.1_gcc_6.2_app (/tmp/lustre_shared/plgkopta/qcgpjm-altair/
↪examples/openmpi_3.1_gcc_6.2_app) node(e0025) after 3.91 secs
        --29567:openmpi_3.1_gcc_6.2_app (/tmp/lustre_shared/plgkopta/qcgpjm-altair/
↪examples/openmpi_3.1_gcc_6.2_app) node(e0025) after 3.94 secs
```

To get detail process info:

```
$ qcg-pm-processes apps out-api-mpi-iter mpi_iter:0
found 8 target processes
29537:openmpi_3.1_gcc_6.2_app
                created: 2021-03-25 17:18:38.180000
                cmdline: /tmp/lustre_shared/plgkopta/qcgpjm-altair/examples/openmpi_3.
↪1_gcc_6.2_app
                parent: 28521:mpirun
                cpu affinity: [0]
                cpu times: [0.04, 0.03, 0.0, 0.0, 0.0]
                cpu memory info: [25219072, 525488128, 12701696, 8192, 0, 153374720,␣
↪0]
                cpu memory percent: 0.018710530527870046
29542:openmpi_3.1_gcc_6.2_app
                created: 2021-03-25 17:18:38.210000
                cmdline: /tmp/lustre_shared/plgkopta/qcgpjm-altair/examples/openmpi_3.
↪1_gcc_6.2_app
                parent: 28521:mpirun
                cpu affinity: [1]
                cpu times: [0.06, 0.03, 0.0, 0.0, 0.0]
                cpu memory info: [25206784, 391258112, 12693504, 8192, 0, 153370624,␣
↪0]
                cpu memory percent: 0.01870141381655226
29638:openmpi_3.1_gcc_6.2_app
                created: 2021-03-25 17:18:38.360000
                cmdline: /tmp/lustre_shared/plgkopta/qcgpjm-altair/examples/openmpi_3.
↪1_gcc_6.2_app
                parent: 28521:mpirun
                cpu affinity: [7]
                cpu times: [0.05, 0.03, 0.0, 0.0, 0.0]
                cpu memory info: [25202688, 391258112, 12689408, 8192, 0, 153370624,␣
↪0]
                cpu memory percent: 0.018698374912779966
29608:openmpi_3.1_gcc_6.2_app
                created: 2021-03-25 17:18:38.330000
                cmdline: /tmp/lustre_shared/plgkopta/qcgpjm-altair/examples/openmpi_3.
↪1_gcc_6.2_app
                parent: 28521:mpirun
                cpu affinity: [6]
                cpu times: [0.04, 0.04, 0.0, 0.0, 0.0]
                cpu memory info: [25206784, 391258112, 12693504, 8192, 0, 153370624,␣
↪0]
                cpu memory percent: 0.01870141381655226
29547:openmpi_3.1_gcc_6.2_app
                created: 2021-03-25 17:18:38.230000
                cmdline: /tmp/lustre_shared/plgkopta/qcgpjm-altair/examples/openmpi_3.
↪1_gcc_6.2_app
                parent: 28521:mpirun
                cpu affinity: [2]
                cpu times: [0.06, 0.03, 0.0, 0.0, 0.0]
                cpu memory info: [25206784, 391258112, 12693504, 8192, 0, 153370624,␣
↪0]
                cpu memory percent: 0.01870141381655226
29579:openmpi_3.1_gcc_6.2_app
                created: 2021-03-25 17:18:38.300000
                cmdline: /tmp/lustre_shared/plgkopta/qcgpjm-altair/examples/openmpi_3.
↪1_gcc_6.2_app
                parent: 28521:mpirun
```

(continues on next page)

(continued from previous page)

```
                cpu affinity: [5]
                cpu times: [0.05, 0.03, 0.0, 0.0, 0.0]
                cpu memory info: [25206784, 391258112, 12693504, 8192, 0, 153370624,␣
↪0]
                cpu memory percent: 0.01870141381655226
29554:openmpi_3.1_gcc_6.2_app
                created: 2021-03-25 17:18:38.260000
                cmdline: /tmp/lustre_shared/plgkopta/qcgpjm-altair/examples/openmpi_3.
↪1_gcc_6.2_app
                parent: 28521:mpirun
                cpu affinity: [3]
                cpu times: [0.05, 0.04, 0.0, 0.0, 0.0]
                cpu memory info: [25202688, 391258112, 12689408, 8192, 0, 153370624,␣
↪0]
                cpu memory percent: 0.01869837491277966
29567:openmpi_3.1_gcc_6.2_app
                created: 2021-03-25 17:18:38.290000
                cmdline: /tmp/lustre_shared/plgkopta/qcgpjm-altair/examples/openmpi_3.
↪1_gcc_6.2_app
                parent: 28521:mpirun
                cpu affinity: [4]
                cpu times: [0.06, 0.03, 0.0, 0.0, 0.0]
                cpu memory info: [25206784, 391258112, 12693504, 8192, 0, 153370624,␣
↪0]
                cpu memory percent: 0.01870141381655226
```

It is worth to mention, that analysis with the `qcg-pm-processes` tool can be done at any time outside the Slurm allocation. The only input data is the working directory.

CHAPTER 17

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 18

Slurm performance

18.1 srun command

QCG-Pilot job uses the Slurm's `srun` client to run applications within a created allocation. Thanks to the tight integration with the queueing system, `srun` is able to properly run an application on the specified node of our allocation using, for example, cpu binding mechanisms. The usage of *srun* seems to be particularly convenient for running parallel applications using the MPI library. It provides a unified way of running such applications, regardless of the vendor and version of MPI library (note that the commands used to run MPI-based applications provided by different MPI libraries such as *OpenMPI/IntelMPI/MPICH* have a different name, syntax and way of running the target application). Unfortunately, when starting an application, the `srun` client communicates with the queueing system controller and creates a step for each running application. It turns out that with too frequent use of this client, the queue system controller struggles with quite a heavy load which affects the performance of the whole queueing system.

The QCG-PilotJob service uses the `srun` client by default in two cases:

- during service initialization to launch agents* running on each allocation node
- in the `srunmpi` model when launching user applications with the `srunmpi` model.

It is possible to replace `srun` with alternatives for both these cases as presented below.

18.2 Recommendations

18.2.1 Agents

When running QCG-PilotJob on large allocations (containing more than a few dozen nodes) it is recommended to use the `--nl-start-method` call parameter with the value `ssh` which will cause the QCG-PilotJob service agents to be started on the allocation nodes using the *ssh* protocol.

Note: Ensure that logging in using the *ssh* protocol is done using a public key without requiring a password. Information on how to configure the ssh service in this way should be available in the documentation of the computing system, and usually boils down to generating an ssh key and adding its public signature to the ~/.ssh/authorized_keys file.

18.2.2 User parallel applications

For scenarios containing a significant number of parallel user jobs, we recommend that you resign from the srunmpi tasks startup model and use one of the following:

- intelmpi
- openmpi

These are models that use native *IntelMPI* and *OpenMPI* library commands to run parallel applications. Additionally, they allow to configure call parameters using model_opts/mpirun and model_opts/mpirun_args elements. An example syntax of such commands is as follows:

1) example of running a LAMMPS application compiled with the *IntelMPI* library using the *ssh* protocol on a *SupermucNG* system

```
....
"name": "lammps-bench",
"execution": {
        "exec": "lmp",
        "args": ["-log", "none", "-i", "in.lammps"],
        "stderr": "stderr",
        "stdout": "stdout",
        "model": "intelmpi",
        "model_opts": { "mpirun_args": [ "-launcher", "ssh" ] }
},
"resources": {
        "numCores": {
                        "exact": 24
        }
}
....
```

2) example of running an application compiled with the *OpenMPI* library

```
        ....
        "name": "mpi-app",
"execution": {
    "exec": "mpiapp",
    "stderr": "stderr",
    "stdout": "stdout",
    "model": "openmpi",
    "model_opts": {
                        "mpirun": "/opt/exp_soft/local/skylake/openmpi/4.1.0_gcc620/
↪bin/mpirun",
                        "mpirun_args": ["--mca", "rmaps_rank_file_physical", "1"]
                },
                modules = [ "openmpi/4.1.0_gcc620" ],
},
"resources": {
    "numCores": {
        "exact": 24
    }
```

(continues on next page)

(continued from previous page)

```
}
             ....
```

CHAPTER 19

FAQ

19.1 How is QCG-PilotJob better than a BASH script?

QCG-PilotJob has been designed to simplify definition of common scenarios of execution of large number of tasks on computing resources. Typically these scenarios were done by application developers in a custom and often far from an optimal way. With QCG-PilotJob users are offered with ready to use efficient mechanisms as well as nice API that can be recognized as much more natural solution than sophisticated *BASH* scripts, for both direct human use and integration with other software components.

The particular advantage of QCG-PilotJob is visible in case of dynamic scenarios with dynamic number of jobs, dynamic requirements of these jobs and a need to start / cancel these jobs depending on the intermediary results of calculations. For these scenarios the core capabilities of QCG-PilotJob and easy to use constructs offered by QCG-PilotJob API seem to be exceptionally sound.

For all kinds of scenarios, also for the static use-cases (where we know in advance a number of tasks, their requirements, and we have a static allocation) QCG-PilotJob provides a few advantages, like built-in mechanism to resume prematurely stopped workflow, tools for collecting timings from the execution and generation of the reports for the analysis (e.g. Gantt chart), or a custom launcher for single-core tasks, which is more efficient (at least on some resources) than the *srun* command run from *BASH*. QCG-PilotJob delivers also different predefined models of running tasks with *srun*, *intelmpi*, *openmpi* as well as a with *openmp*, which simplify execution of MPI and OpenMP based applications across different computing resources.

However, the target powerfulness of the QCG-PilotJob should be achieved when we release the common queue service that will provide the possibility to combine resources from many allocations into one QCG-PilotJob. Then it will be easy to extend the resources depending on the dynamic needs of the scenario, taking them even from many HPC facilities.

19.2 How is QCG-PilotJob better than existing Workflow / Pilot Job implementations?

The strategic decision for the development of QCG-PilotJob was to ensure simplicity of the entire process related to the tool's use: from its installation, through defining workflows, to the actual execution of tasks. Thus, in contrast to many existing products, QCG-PilotJob not only simplifies definition of execution scenarios, but also comes very easy to install and can be run without problems across different environments, even conservative and variously restricted ones.

It should be stressed that QCG-PilotJob is a fully user-space solution, and as such, can be installed by an ordinary user, in its home directory (e.g. in a virtual environment). At any step there is no need to bother administrators: to install something or to open some ports.

CHAPTER 20

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 21

License

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

(continues on next page)

(continued from previous page)

CHAPTER 22

qcg.pilotjob.api package

22.1 Submodules

22.1.1 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`*

exception `qcg.pilotjob.api.errors.`**`TimeoutElapsed`**
    Bases: `Exception`

22.1.2 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
- `model_opts` (dict, optional): model options
- `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, dict or list, optional): iterations definition
- `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 either:

- dictionary with following keys:
  - `start` (int, optional): iterations start index
  - `stop` (int, optional): iterations stop index
- `values` list with following iteration names

The total number of iterations will be:

- `stop - start` (the last iteration index will be `stop - 1`) for boundary definition
- length of `values` list

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.

22.1.3 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*

22.1.4 qcg.pilotjob.api.manager module

`class` `qcg.pilotjob.api.manager.`**`TimeStamp`**(*manager*, *timeout_secs=None*)
Bases: `object`

Timestamp utility to trace timeouts and compute the poll times that do not exceed defined timeouts.

Create timestamp. During initialization the timestamp start moment is set to current time.

Parameters

- **`(Manager)`** (*`manager`*) – the manager instance with defined default poll and publisher timeout setting
- **`(int|float)`** (*`timeout`*) – the timeout in seconds for operation related with this timestamp, the default value *None* means the timeout is not defined (infinity)

secs_from_start
Return number of seconds elapsed since start

check_timeout()
Check if timeout has been reached. If *timeout_secs* has been defined check if *timeout_secs* have elapsed from *started* datetime. If *timeout_secs* is not defined always return False

Returns True if timeout reached, False otherwise

get_poll_time()
Return the poll time that do not exceed timeout. If timeout already reached, the 0 will be returned.

Returns poll time in seconds

get_events_timeout()
Return the subscribe timeout that do not exceed total operation timeout. If timeout already reached, the 0 will be returned.

Returns the timeout time in seconds

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*) – 'default_poll_delay' - the default delay between following status polls in wait methods 'default_pub_timeout' - the default timeout for waiting on published events '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 = 5

DEFAULT_PUB_TIMEOUT = 300

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 top logger.

`system_status`()

wait4(*names*, *timeout=None*)

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
- **timeout** (*int|float*) – maximum number of seconds to wait

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

Raises

- `TimeoutElapsed` – in case of timeout elapsed
- `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(*timeout_secs=None*)

Wait for finish of all submitted jobs.

Parameters **(int|float)** (*timeout_secs*) – optional timeout setting in seconds

:raise TimeoutElapsed when timeout elapsed (if defined as argument)

This method waits until all jobs submitted to service finish its execution (successfully or not).

wait4_any_job_finish(*timeout_secs=None*)

Wait for finish one of any submitted job.

This method waits until one of any jobs submitted to service finish its execution (successfully or not).

:arg timeout_secs (float|int) - timeout in milliseconds, endlessly (None) by default

:raise TimeoutElapsed when timeout elapsed (if defined as argument)

:return (str, str) identifier of finished job and it's status or None, None if timeout has been reached.

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`) –

  'init_timeout' - the timeout (in seconds) client should wait for QCG-PilotJob manager start until it raise
  error, 300 by default

  '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.

`static is_notebook``()`

CHAPTER 23

qcg.pilotjob.executor_api package

23.1 Subpackages

23.1.1 qcg.pilotjob.executor_api.templates package

Submodules

qcg.pilotjob.executor_api.templates.basic_template module

class `qcg.pilotjob.executor_api.templates.basic_template.`**`BasicTemplate`**

Bases: *`qcg.pilotjob.executor_api.templates.qcgpj_template.QCGPJTemplate`*

static template() → Tuple[str, Dict[str, Any]]

qcg.pilotjob.executor_api.templates.qcgpj_template module

class `qcg.pilotjob.executor_api.templates.qcgpj_template.`**`QCGPJTemplate`**

Bases: `object`

static template() → Tuple[str, Dict[str, Any]]

23.2 Submodules

23.2.1 qcg.pilotjob.executor_api.qcgpj_executor module

class qcg.pilotjob.executor_api.qcgpj_executor.**QCGPJExecutor**(**other_args, wd='.', resources=None, reserve_core=False, enable_rt_stats=False, wrapper_rt_stats=None, log_level='info'*)

Bases: `concurrent.futures._base.Executor`

QCG-PilotJob Executor. It provides simplified interface for common uses of QCG-PilotJob

Parameters

- **wd** (*str, optional*) – Working directory where QCG-PilotJob manager should be started, by default it is a current directory
- **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 define 4 cores on an unnamed node use *resources="4"*, to define 2 nodes: node_1 with 2 cores and node_2 with 3 cores, 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).
- **other_args** (*optional*) – Optional list of additional arguments for initialisation of QCG-PilotJob Manager

Returns

Return type None

shutdown(*wait=True*)

Shutdowns the QCG-PJ manager service. If it is already closed, the method has no effect.

submit(*fn: Callable[[...], Union[str, Tuple[str, Dict[str, Any]]]], *args, **kwargs*)

Submits a specific task to the QCG-PJ manager using template-based, executor-like interface.

Parameters

- **fn** (*Callable*) – A callable that returns a tuple representing a task's template. The first element of the tuple should be a string containing a QCG-PilotJob task's description with placeholders (identifiers preceded by $ symbol) and the second a dictionary that assigns default values for selected placeholders.

- ***args** (*variable length list with dicts, optional*) – A set of dicts which contain parameters that will be used to substitute placeholders defined in the template. Note: *args overwrite defaults, but they are overwritten by **kwargs
- ****kwargs** (*arbitrary keyword arguments*) – A set of keyword arguments that will be used to substitute placeholders defined in the template. Note: **kwargs overwrite *args and defaults.

Returns The QCGPJFuture object assigned with the submitted task

Return type *QCGPJFuture*

qcgpj_manager
Returns current QCG-PilotJob manager instance

class qcg.pilotjob.executor_api.qcgpj_executor.**ServiceLogLevel**
Bases: enum.Enum

An enumeration.

CRITICAL = 'critical'

ERROR = 'error'

WARNING = 'warning'

INFO = 'info'

DEBUG = 'debug'

class qcg.pilotjob.executor_api.qcgpj_executor.**ClientLogLevel**
Bases: enum.Enum

An enumeration.

INFO = 'info'

DEBUG = 'debug'

23.2.2 qcg.pilotjob.executor_api.qcgpj_future module

class qcg.pilotjob.executor_api.qcgpj_future.**QCGPJFuture**(*ids*, *qcgpjm*)
Bases: object

QCG-PilotJob Future tracks execution of tasks submitted to QCG-PilotJob via QCGPJExecutor.

Parameters

- **ids** (*list (str)*) – list of identifiers of tasks submitted to a QCG-PilotJob manager
- **qcgpjm** (LocalManager) – QCG-PilotJob manager instance, to which tasks have been submitted

Returns

Return type None

result(*timeout=None*)
Waits for finish of tasks assigned to this future and once finished results their statuses.

This method waits until all tasks assigned to the future are executed (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 **`timeout`** (*`int`*) – currently not used

Returns

Return type dict - a map with tasks names and their terminal status

`done``()`

Checks if the future has been finished

Checks if all tasks assigned to the future are already finished.

Returns

Return type True if all tasks are finished, False otherwise

`running``()`

Checks if the future is still running

Checks if any of tasks assigned to the future are still running.

Returns

Return type True if any of tasks is still running, False otherwise

`cancel``()`

Cancels the future

Cancels all tasks assigned to the future.

Returns

Return type True if the operation succeeded.

`cancelled``()`

Checks if the future has been already cancelled

Checks if the future, and by consequence all the tasks assigned to this future, have been cancelled.

Returns

Return type True if the future has been cancelled, False otherwise.

CHAPTER 24

Indices and tables

- genindex
- modindex
- search

CHAPTER 25

Authors

Bartosz Bosak, Piotr Kopta, Tomasz Piontek (PSNC)

Python Module Index

Index