COMPAT

Computing Patterns for High Performance Multiscale Computing

D5.3 Report on integration of ComPat services with multiscale coupling libraries and patterns

Due Date	36
Delivery	36
Lead Partner	PSNC
Dissemination Level	Public
Status	Final
Approved	Internal review
Version	1.0

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under grant agreement No 671564.

DOCUMENT INFO

Date and version number	Author	Comments
28.08.2018 v0.1	Bartosz Bosak	Table of contents
29.08.2018 v0.2	Bartosz Bosak	Executive Summary
29.08.2018 v0.3	Piotr Kopta	Pilot jobs description
31.08.2018 v0.4	Tomasz Piontek	Multi-Criteria brokering
01.09.2018 v0.5	Vytautas Jancauskas	Queue Time Prediction Service
03.09.2018 v0.6	Bartosz Bosak	Merge all parts
20.09.2018 v1.0	Tomasz Piontek	Final version after the internal review

CONTRIBUTORS

Contributor	Role
Tomasz Piontek	Technical Leader, WP5 Leader and author of this deliverable
Bartosz Bosak	WP5 contributor
Piotr Kopta	WP5 contributor
Vytautas Jancauskas	WP5 contributor

TABLE OF CONTENTS

LIST OF ABREVIATIONS

EEE	Experimental Execution Environment [Project Testbed]
ES	Extreme Scaling [Pattern]
HMC	Heterogeneous Multiscale Computing [Pattern]
HPMC	High Performance Multiscale Computing
HPMCP	High Performance Multiscale Computing Patterns
RC	Replica Computing [Pattern]
QCG	Quality in Cloud and Grid [middleware]
QTPS	Queue Time Prediction Service

1 Executive summary

This deliverable provides information about the development status of the ComPat middleware services and the progress made to integrate them with the rest of the ComPat software stack (especially coupling libraries and patterns' related components). It concentrates on the new capabilities that were developed in WP5 to improve the trilateral cooperation between the ComPat resource (WP6), middleware (WP5) and pattern layers (WP2) in order to support applications (WP3). Since the document outlines the progress achieved in the Work Package 5 in the last 18 months of the project (M18-M36), it can be seen as a natural extension of the deliverable D5.2 published on M18 [1] and the further implementation of the ComPat's architecture outlined in D5.1 on M9 [2] .

The achievements presented in the document are reflected in the final software release (M36 of the project). This software release is already deployed on the ComPat Experimental Execution Environment and made available to the project's application teams.

During the last 18 months the ComPat middleware layer has been adapted to the new requirements of High Performance Multiscale Computing Patterns (HPMCP). While the support for the previously implemented Extreme Scaling (ES) and Replica Computing (RC) patterns was only slightly adjusted, a set of extensions to the QCG components for the execution of applications based on the concept of the Heterogeneous Multiscale Computing (HMC) pattern is one of the crucial new functionalities incorporated into the middleware stack. Particularly, in order to efficiently support this coupling pattern, QCG has been extended with the brand new mechanism for pilot jobs allowing to easily execute jobs with an unknown number of subtasks. Due to its functionality, the created PilotJob system can be treated, alongside MUSCLE2 and AMUSE, as the other coupling tool/library tightly integrated with QCG and the whole ComPat software stack.

Significant efforts were devoted to advance brokering capabilities of the ComPat system. The multi-criteria brokering plugin recently implemented for QCG-Broker allows selecting resources for applications based on multiple metrics, such as expected time to completion and expected energy usage. Thus a user, depending on its needs, can request the faster completion of a job or the limitation of the energy loss, which is currently innovative and essential for brokering of large computing jobs. In order to precisely estimate the queue time for a job, in cooperation with WP6, a new Queue Time Prediction service (QTPS) was developed and integrated with the QCG-Broker. In short, the service collects in real-time data from local resource managing systems and, based on solid probabilistic foundations, provides an approximation of queuing time usable for QCG-Broker.

In Section 3 the above mentioned results of the activities of WP5 will be presented in details.

2 Status of ComPat middleware services and their integration with other ComPat software components.

The significant role of the ComPat middleware layer in supporting HPMC patterns and efficient execution of multiscale application on HPC resources required a number of extensions and slight improvements to the existing software. However, in order to align with the project roadmap and answer to the new requirements of multiscale applications, it was necessary to develop a few completely new software components and modules. This section tries to describe both aspects in details.

2.1 Extensions to the Extreme Scaling and Replica Computing patterns

The main application developed in ComPat project that fits the extreme scaling computing pattern is the plasma physics simulation created in the Max Planck Institute for Plasma Physics. This application uses the MUSCLE2 framework [3] to couple together single-scale kernels into a multiscale application. The MUSCLE2 framework requires a simulation description file where size of each kernel and its relationships to other kernels are defined. To support the MUSCLE2 description file, the QCG-Broker service has been extended to set the following environment variables, passed transparently and automatically to the launched MUSCLE2 based application:

- `QCG_PLAN_ID` - the identifier of selected scheduling plan
- `QCG_KERNEL_{KERNEL_NAME}` - number of cores allocated for each kernel

By using these variables, a single, universal MUSCLE2 description file can be created and it will be the same across every cluster available in ComPat Project.

2.2 Support for the Heterogeneous Multiscale Computing pattern - Pilot-Job functionality

In order to support scenarios where the number of computing kernels or their resource requirements in multiscale computing applications are not known a priori, a specialised software has to be employed that extends capabilities of typical queuing systems. After the performed analysis and review of existing available options, a decision was made to create from scratch a new module for the QCG middleware stack, called QCG PilotJob Manager tailored to the specific needs of the project and integrated with the rest of the ComPat software stack. This 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 has to wait in a queue for its own slot. On the other hand the submission of a group of jobs can be restricted or even forbidden by administrative policies on the clusters. One can argue that there are job array mechanisms already available on many systems, however, the traditional implementations of job array mechanisms only allow running a bunch of jobs having the same resource requirements while jobs that are parts of a multiscale simulation by nature vary in requirements, might require to be executed in a specific order, and therefore need more flexible solutions.

From the scheduling system perspective, QCG PilotJob Manager is seen on a cluster as a single job inside a single user allocation. In other words, the manager controls an execution of a complex experiment consisting of many computing jobs on resources reserved for a single job allocation. The manager listens to user's requests and executes commands like submit job, cancel job and report resource use. In order to manage resources and jobs the system takes into account both resource availability and mutual dependencies between jobs. Two interfaces are defined to communicate with the system: file-based and network-based. The former one is dedicated to and more convenient for static scenarios when a number of jobs is known in advance. The network 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. Noticeably, the QCG PilotJob Manager's interfaces are independent from the underlying resource management systems and provide an abstraction layer over them, in turn, providing the users with a single, unified interface to resources.

Due to its functionality and place in the ComPat architecture, the PilotJob, in spite of its generality, belongs to the specific ComPat Pattern Services and Tools supporting the concept of the High Performance Multiscale Patterns. The PilotJob was designed and implemented to support specific requirements of the Heterogeneous Multiscale Computing Pattern, but can be also used for Replica Pattern. The PilotJob Manager allows one in an efficient and easy way to build a single multiscale application from many potentially dynamic tasks, which number can be unknown in advance. All the functional and technical details concerning the QCG PilotJob system can be found in the technical documentation of the system provided to the end-users and attached to this deliverable as Appendix 1 - QCG PilotJobs.

2.3 Multi-criteria scheduling

Modern computational problems, including multiscale applications, belonging to the class of grand challenge applications, require significant computational power to achieve accurate results in acceptable time. With increasing demands for computing power from users, the operational costs of infrastructure maintenance grows on the side of resource providers. In recent years, the cost of electricity, needed to both power the computers and to cool them, in order to provide optimal operating conditions for the hardware, has become an important factor. In many cases, this cost determines the possibility for further development of the infrastructure or may even lead to a reduction in its use. Therefore, many HPC resource providers consider a departure from the typical user billing model for the core-hours used and the transition to a new model in which a user is assigned an energy budget. Such energy-oriented approach makes the user more interested in optimal utilisation of the infrastructure. For these reasons, it is necessary to provide novel software solutions in order to reconcile the conflicting requirements of users expecting their tasks to be completed as soon as possible with those of Computation Centres trying to minimise their expenses. The optimization of energy consumption can be achieved in many ways and on different levels of the software and hardware layers.

2.3.1 QCG-Broker scheduling plugin for time and energy criteria

As part of the ComPat project, a multi-criteria approach for assigning in an optimal way of an entire multiscale application or its components to the heterogeneous resources offered by the project's test and production infrastructure, has been designed and developed as a new module to the QCG-Broker service. Different performance and energy consumption characteristics collected from resources allow savings to be made while maintaining the quality of the service required by the user. The proposed approach has evolved over the course of the project. In the first prototype, only a single criterion on the time to completion of calculations was taken into account with a user-defined limit for the use of electricity. In the next step, the implementation of a complementary scenario was planned where energy would be optimized while maintaining the requirements for timely delivery of results. Ultimately, the decision was made to implement the most general solution. Instead of taking individual criteria into consideration, the system performs multi-criteria optimization with regard to limits on given criteria. This approach guarantees full flexibility in the definition of criteria and limits by the user when submitting tasks. The user can choose any single criterion or a group of many criteria. The same applies to limits giving full control on the quality of the service from the user's point of view.

Through the arrangements with users, it was decided to include in the ComPat system the three criteria, covering to the best of our knowledge the majority of real and potential scenarios:

- electricity needed to carry out calculations. (This criterion currently does not take into account other infrastructure-related costs such as cooling, for example)
- time to completion - defined as the sum of the time of waiting for the resources and the time of calculation,
- use of infrastructure in time - defined as the product of the number of cores and the hours of their use. This criterion has been added to support the most popular method of accounting for used core-hours.

The proposed approach is easily expandable with further criteria. Each of the criteria is optional and can be defined with an optional limitation. Solutions that violate even one constraint (limit) are rejected. Since the simultaneous optimization of many objectives for non-trivial cases is usually impossible, it

was decided to aggregate them through a single objective function, which is minimized. This function is determined as a weighted sum of many objective functions related to individual criteria, where the weights of individual criteria can be understood as a conversion of a given criterion into a common unit, e.g. monetary.
The system allows for an optimal selection of resources in a scenario in which the total cost of commercial or scientific calculations depends on both the energy consumed and the use of resources, and at the same time penalty fees are charged for delays in exceeding the set date for the delivery of results. In addition, hard limits, which cannot be exceeded, can be set on all three criteria. The approach described above has been implemented as a new QCG-Broker resource allocation module and made available to the users as part of the project.

The rest of this section is devoted to the implementation details of the solution.
Along with the implementation of the above multi-criteria approach in the QCG-Broker service, it was necessary to modify the QCG-Computing services providing information about the current state of infrastructure, including the availability of nodes of particular types. It was also needed to design and implement a service the role of which is to gather performance and energy-consumption results to determine the performance and energy profiles of the applications. The service is described in details in the deliverables D4.3 [4] and D6.3[5]. Additionally, it was necessary to modify the task description used by QCG-Broker and adapt it to the specific requirements of the project.

Thus, in a typical scenario addressed in the project, the QCG-Broker service receives, as an element of the task description, a list of resource allocation plans with static cost values on defined criteria. These costs are determined for each application module on the basis of the aforementioned benchmarks. The final selection of the plan to be executed is based on full set of criteria and on the basis of dynamic knowledge about the current state of infrastructure unknown a priori for the Pattern Optimization Service generating the plans.

Figure 1 shows the final structure of the task description element developed to satisfy the needs of the ComPat project (multiCriteriaPatternTopology type), taking into account the plans and their costs as well as defining the criteria and limits taken into account in the optimization of resource allocation.

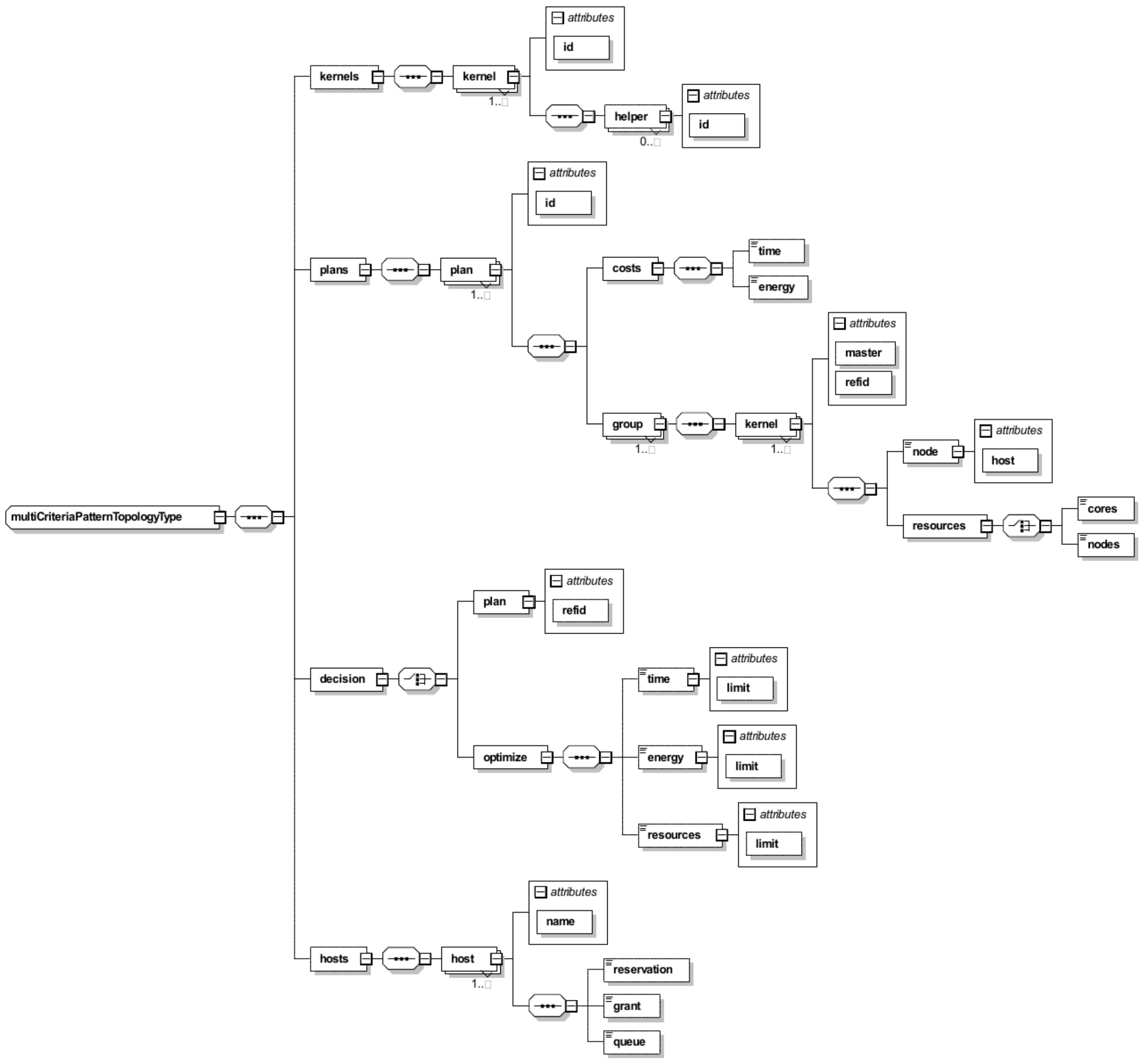

Figure 1 The new QCG task description scheme for multi-criteria brokering

The meaning of individual elements is as follows:

- <kernels> - a list of appropriate computational kernels (single scale models) along with auxiliary modules that make up a multiscale application,
- <plans> - a list of alternative execution plans generated by Pattern Optimization Service. The QCG-Broker service selects the best plan according to the definition of optimization parameters. Each plan has its own unique identifier and the <costs> element defining the known from benchmarks costs of executing a given plan on the resources assigned to it. The <group> element logically groups kernels, which for some reason (most often resulting from communication topology) must be performed on one machine. For each kernel, resource requirements defining the type of resource (host and node type) and the number of resources required in cores or nodes are defined.

- <decision> - the control element of the optimization parameters. It allows the user to either select a specific plan (useful for testing purposes) or define criteria and limits for the process of automatic selection of hardware configuration performed by the system (primary scenario). The individual criteria and their limits are optional. With each criterion, the weight of this criterion in the global objective function is provided.
- <hosts> - this element allows to pass to the QCG-Broker service additional execution parameters specific for a given host, such as reservation ID, grant or queue/partition of the queueing system.

An example of a task description for the Fusion application is presented at the end of the document in Appendix 2 - Fusion QCG Job Description.

Based on the description of the task and the current state of the infrastructure, the QCG-Broker service selects the optimal assignment of tasks to the resources taking into account the given criteria.

The process of selecting the optimal plan consists of the following steps:

1. Elimination of resources - at this stage the resources that will not be able to be used in the task handling are rejected (removed from the list of potential resources). For example, they may be resources that are unavailable to the given user, do not offer the required application or are deliberately rejected by the user. The list of conditions taken into account in the resource selection process is configurable and easily expandable via a system of plug-ins with a defined interface.
2. Elimination of plans - at this stage, plans that are not possible to be implemented are eliminated from the list of potential plans. The reason for the rejection of the plan may be, among others:
   - Undefined or unavailable host or node type,
   - Insufficient amount of resources of a specific type,
   - Violation of one of the optimization limits,
   - Missing costs in the plan for any criterion or limitation,
   - Manual selection of another plan by the user,
   - Violation of the requirements for the execution of a group of kernels on a single host.

   The list of conditions taken into account in the plan selection process is configurable and easily expandable through a plug-in system with a defined interface.
3. Evaluation of plans - determination of the global value of the objective function for the remaining plans. This stage may require referring to external services such as Queue Time Prediction Service (QTPS). QTPS is a service that enables the prediction of queued waiting time of a task with specific resource requirements based on historical data. The QTPS service is one of the services designed, implemented and deployed as part of the ComPat project. The general

information about QTPS is provided in the chapter 2.3.2 of this deliverable, but the technical details of the service are described in Deliverable D6.3 [5] and in the publication [6].

4. Sorting plans by the global value of the objective function.
5. Creation (in a formalized format) of an internal QCG execution assignment plan for a given selected plan. The assignment plan contains information about the allocation of kernels to resources and is passed to the QCG-Broker Job Launcher module.

When implementing the task allocation mechanism in the QCG-Broker service, it was assumed that the user can manually modify the file generated by the Pattern Optimization Service. This results in the need to validate the correctness of the generated plans not only in terms of availability and status of resources, but also the compliance of static costs with the accepted limits.

The described mechanism of multi-criteria allocation of resources for tasks has been designed, implemented and deployed as part of the ComPat project work as well as made available as a default in the final distribution of the services created within the project.

2.3.2 Queue Time Prediction Service

Queue wait time (the time a job will spend in the scheduler queue) is an important consideration in the kind of multi-criteria brokering we want to do. To this end we have worked with WP6 to develop a queue wait time prediction service. The service, upon receiving description of required resources, will calculate probabilities that those resources will be available within some specific time. For example, if we request queue wait times for 8 nodes and 128 cores, the service will find all the queues that can provide that amount of resources and, for each, will return a list of queue wait times and probabilities. Each queue wait time has an associated probability that signifies the chances that the job will be started before that time. This is achieved on the queue wait time prediction service's end by maintaining a database of all jobs submitted to each of the systems. Around 40 attributes describing each job are stored. From these, statistical techniques are used to determine the probabilities. This information can then be used in the decision making process. In particular it is used in the multi-criteria optimization process when deciding where to submit a particular part of the simulation. Full description of the queue wait time prediction service, including the API, can be found in deliverable D6.3 [5].

3 Conclusions

The main goal of WP5 in ComPat was to provide an efficient and easy to use environment for development, testing and execution of pattern-based multiscale applications on existing and future HPC resources. By creating a number of novel tools and services, as well as the work spent on the integration of middleware layer with other ComPat components, the WP5 has made a significant advancements in all those areas. This document concentrates on a progress made in WP5 over the last 18 months of the project. The major outcomes from this period include development of pilot job mechanism for QCG middleware, implementation of the pattern-aware multi-criteria brokering plugin for QCG-Broker as well as joint with WP6 development of Queue Time Prediction service and its integration with QCG-Broker.

The confirmation of the joint achievements of work packages WP2, WP4, WP5 and WP6 at the end of the project is the successful, common final-release of all key tools and services on month 36 (milestone M20). The released packages are available from the project's GitHub repository: https://github.com/compat-project. They were also deployed on EEE and effectively used by a set of HPMC applications developed by ComPat. See the Deliverable D3.3 document [8] for more information on the status reached by each of the ComPat applications.

4 Appendixes

Appendix 1 - QCG PilotJobs

1 The QCG Pilot Manager v 0.3

Author: Piotr Kopta *pkopta@man.poznan.pl*, Tomasz Piontek *piontek@man.poznan.pl*, Bartosz Bosak *bbosak@man.poznan.pl*

1.1 Overview

The QCG PilotJob Manager 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 job array mechanisms are available in many systems, however the traditional job array mechanism only allows to run a bunch of jobs that have the same resource requirements while jobs being parts of a multiscale simulation by nature vary in requirements and therefore need more flexible solutions.

From the scheduling system perspective, the QCG PilotJob Manager is seen as a single job inside a single user allocation. In other words, the manager controls the execution of a complex experiment consisting of many jobs on resources reserved for one single job allocation. The manager reads user's requests and executes commands like submit job, cancel job and report resource use. In order to manage the resources and jobs the system takes into account both resource availability and mutual dependencies between jobs. Two interfaces are defined to communicate with the system: file-based and network-based. The former 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 network 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.

1.2 Modules

The QCG Pilot Job 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.

1.3 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.

1.4 Executor

The 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 is integrated with the SLURM system, but the modular approach allows for relatively easy integration also with other queuing systems. The PilotJob Manager and all the 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 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 job.

1.4.1 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 a 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 a comma,
- *SLURM_STEP_TASKS_PER_NODE* - see *SLURM_NTASKS_PER_NODE*
- *SLURM_TASKS_PER_NODE* - see *SLURM_NTASKS_PER_NODE*

1.4.2 QCG Execution Environment

To unify the execution environment regardless of the queuing system a set of variables independent from a queuing system is passed to the individual job:

- *QCG_PM_NNODES* - a number of nodes
- *QCG_PM_NODELIST*- a list of nodes separated by a 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 a comma
- *QCG_PM_ZMQ_ADDRESS* - an address of the network interface of QCG PilotJob Manager (if enabled)

1.5 File Based Interface

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

1.5.1 Request File

The request 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.

1.5.2 Requests (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:

1.5.2.1 Submit Command

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

Job description format

The Job description is a dictionary with the following keys:

- `name` (required) `String` - job name, must be unique
- `iterate` (optional) `Array of Number` - defines a loop for iterative jobs, the two element array with the start and stop index; the total number of iterations will be *stop_index* - *start_index* (the last index of the sub-job will be *stop_index - 1*)
- `execution` (required) `Dict` - execution description with the following keys:
- `exec` (required) `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,
- `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.
- `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:
    - o *exact* (optional) *Number* - the exact number of cores,
    - o *min* (optional) *Number* - minimal number of cores,
    - o *max* (optional) *Number* - maximal number of cores,

    (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. Both elements, `numCores` and `numNodes`, may contain special, optional key `split-into` `Number` along with `min` key. The purpose of this element is to split total number of nodes/cores into smaller chunks for a single job. The quotient of a total number of available cores/nodes by the `split-into` value means number of requested cores/nodes per single chunk. The `min` keyword restricts number of requested resources to the range: <min, Total_Num / split-into\>
- *Dependencies* (optional) *Dict* - a dictionary with the following items:
    - o *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) the current job is taken into consideration by the scheduler and can be executed.

Job description variables

The job description may contain variables 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)
- *its* - a number of iterations (only for iterative jobs)

- `it_start` - a start index of iterations (only for iterative jobs)
- `it_stop` - a stop index of iterations (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": "2 jobs submitted",
  "data": {
    "jobs": [
      "echo",
      "msleep2"
    ],
    "submitted": 2
  }
}
```

1.5.2.2 listJobs Command

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:

```
{
  "data": {
    "jobs": {
      "msleep2": {
        "status": "EXECUTING"
      },
      "echo": {
        "status": "EXECUTING"
      },
      "env_1": {
        "status": "EXECUTING"
      }
    },
    "length": 3
  },
  "code": 0
}
```

1.5.2.3 jobStatus Command

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", "echo" ]
  }
```

The example response is presented below:

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

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).

1.5.2.4 jobInfo Command

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:

```
{
  "data": {
    "jobs": {
      "msleep2": {
        "status": 0,
        "data": {
          "history": "\n2018-03-14 14:51:05.115960: QUEUED\n2018-03-14 14:5
1:05.116719: EXECUTING",
          "jobName": "msleep2",
          "runtime": {
            "allocation": "e0128:1",
            "wd": "/home/plgrid/plgkopta/qcg-pilotmanager/qcg-pilotmanager"
          },
          "status": "EXECUTING"
        }
      },
      "echo": {
        "status": 0,
        "data": {
          "history": "\n2018-03-14 14:51:05.116166: QUEUED\n2018-03-14 14:5
1:05.117643: EXECUTING",
          "jobName": "echo",
          "runtime": {
            "allocation": "e0128:1",
            "wd": "/home/plgrid/plgkopta/qcg-pilotmanager/qcg-pilotmanager"
          },
          "status": "EXECUTING"
        }
      }
    }
  },
  "code": 0
}
```

1.5.2.5 control Command

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"
  }
```

1.5.2.6 cancelJob Command

Cancel a job with a name specified in the *jobName* key. Currently not supported. The example cancel job request is presented below:

```
{
    "request": "cancelJob",
    "jobName": "msleep2"
}
```

1.5.2.7 removeJob Command

Remove a job from the registry. The name of a job to be removed must be placed in the `jobName` 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", "echo" ]
}
```

The example response is presented below:

```
{
  "data": {
    "removed": 2
  },
  "code": 0
}
```

1.5.2.8 resourcesInfo Command

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": {
    "totalCores": 8,
    "totalNodes": 1,
    "usedCores": 2,
    "freeCores": 6
  },
  "code": 0
}
```

1.5.2.9 finish Command

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"
}
```

1.6 Additional Output Files

QCG PilotJob Manager creates the following files in a working directory:

- `service.log` - containing service logs, very useful to debug service and jobs execution
- `jobs.report` - holding information about all finished (with success or failure) jobs along with the details on status, scheduled nodes/cores, working directory and wall time; the sample content is presented below:

```
msleep2 (SUCCEED)
    2018-01-11 16:02:27.740711: QUEUED
    2018-01-11 16:02:27.741276: EXECUTING
    2018-01-11 16:02:33.832568: SUCCEED
    allocation: e0003:2
    wd: /home/plgrid/plgkopta/qcg-pilotmanager-tests/sandbox/sleep2.s
andbox
    rtime: 0:00:06.082136
```

1.7 Python Version Issues

The current implementation of the QCG PilotJob Manager service is based on Python 3. Due to the fact that all jobs controlled by the Manger inherit an environment, the default version of Python for all jobs is also v3. In situation where different version of Python is required by a job, the Python module should be changed in a job's script.

1.8 Usage

The QCG PilotJob Manager service is executed as a regular job in a queuing system. It exploits a single system allocation for itself and all jobs it controls. For the user convenience, the service has been registered as QCG application (currently only on cluster Eagle @ PSNC) under the name `qcg-pm`. The only and required argument for this application is the name (path) of the requests file. The example script (in QCG Simple description format) is presented below:

```
#QCG host=eagle
#QCG note=QCG PM bac16

#QCG output=${JOB_ID}.output
#QCG error=${JOB_ID}.error

#QCG stage-in-file=bac16.json
#QCG stage-in-file=bac-namd.sh
#QCG stage-in-file=bac-amber.sh
#QCG stage-in-file=bac-16-input.tgz
```

```
#QCG stage-out-dir=.->out/eagle-pm-bac16-${JOB_ID}

#QCG preprocess=tar --strip-components=1 -x -z -f bac-16-input.tgz

#QCG nodes=4:28
#QCG queue=fast
#QCG walltime=PT60M
#QCG grant=compatpsnc2

#QCG application=qcg-pm
#QCG argument=bac16.json
```

In the presented example the job requires 4 nodes (28 cores each), a set of input files is staged into the working directory (the `bac16.json` file is the request file containing jobs descriptions). The name of this file is the only argument for the QCG PilotJob Manager application.

In the `bac16.json` we define the following requests (commands):

```
[
{
    "request": "submit",
    "jobs": [
    {
        "name": "namd_bac16_${it}",
        "iterate": [ 1, 17 ],
        "execution": {
          "exec": "bash",
          "args": [ "${root_wd}/bac-namd.sh", "${it}" ],
          "stdout": "logs/${jname}.stdout",
          "stderr": "logs/${jname}.stderr"
        },
        "resources": {
          "numNodes": {
                "min": 1,
                "max": 2
          }
        }
    },
    {
        "name": "amber_bac16_${it}",
        "iterate": [ 1, 17 ],
        "execution": {
          "exec": "bash",
          "args": [ "${root_wd}/bac-amber.sh", "${it}" ],
          "stdout": "logs/${jname}.stdout",
          "stderr": "logs/${jname}.stderr"
        },
        "resources": {
          "numCores": {
                "exact": 4
          }
        },
        "dependencies": {
          "after": [ "namd_bac16_${it}" ]
        }
    }
    ]
},
{
    "request": "control",
    "command": "finishAfterAllTasksDone"
}
]
```

The file contains two request: submit jobs and control ones. The submit request defines two iterative jobs. Each iteration of the second job (amber) depends on the corresponding iteration of the first sub-job (namd). The control request is used to tell the Manager to finish after all jobs will be handled and finished (with any result).

1.9 API

QCG PilotJob Manager provides a client side Python API which allows users to dynamically control the instance of the Pilot Job and all its jobs. To avoid communication issues, we assume that the user's program (Application Controller), that refers via API to the QCG PilotJob Manager, is run inside an allocation as one of the jobs. The communication is done through the network interface provided by QCG PilotJob Manager. The Python API is provided in the *qcg.appscheduler.api* package.

1.9.1 Manager

The *qcg.appscheduler.api.Manager* class is the main actor of the API. This class is responsible for the initialisation of the client and handles communication with a single instance of QCG PilotJob Manager. The class provides interface for:

- determining the resources status (the *resources* method)
- job submission (the *submit* method),
- listing all jobs (the *list* method),
- obtaining the current status of jobs (the *status* method),
- obtaining the detailed information about jobs (the *info* method),
- removing jobs (the *remove* method),
- synchronisation of the job execution (the *wait4* method).

1.9.1.1 Initialization

```
def __init__(address = None, cfg = { })
```

To properly work, the *Manager* class needs to know the address of the QCG PilotJob Manager instance. In case where the client application is run as one of jobs in QCG PilotJob Manager this address is stored in the *QCG_PM_ZMQ_ADDRESS* environment variable, and is automatically used by the API when it was not provided explicitly.

During the initialisation of the *Manager* class the user can specify some variables that configure behaviour of the class. Currently the following configuration keys are supported:

- *poll_delay* - a delay in seconds between successive updates of the job status in the *wait* method,
- *log_file* - a location of the log file
- *log_level* - a logging level (e.g. *DEBUG*); by default the logging level is set to INFO. Levels allowed and supported so far are: *DEBUG* and *INFO*.

1.9.1.2 *resources*

```
"""
Raises:
        InternalError - in case of request processing internal error
        ConnectionError - in case of lack of connection to the service or n
onzero response code.
"""
def resources()
```

This method returns information about the current status of resources available (allocated) in QCG PilotJob Manager. The output format is described in *this section*

1.9.1.3 *submit*

```
    """
    Submit jobs.


    Args:
        jobs (qcg.appscheduler.api.Jobs) - a list of job descriptions


    Returns:
        list - a list of names (identifiers) of submitted jobs


    Raises:
        InternalError - in case of request processing internal error
        ConnectionError - in case of lack of connection to the service or n
onzero response code.
    """
```

```
def submit(jobs)
```

Submit a list of jobs. The jobs to be submitted are stored in a `Jobs` object. In case of success the list of names (identifiers) of submitted jobs is returned. The returned list can be used in other methods for example to get actual statuses of jobs or to synchronise their execution.

1.9.1.4 *qcg.appscheduler.api.Jobs*

```
    """
    Validates and adds a new job description in a simple format to the grou
p of jobs.
    If both arguments are present, they are merged and processed as a singl
e dictionary.


    Args:
        dAttrs (dict) - simple format attributes as a dictionary
        attrs (dict) - simple format attributes as a named arguments


    Raises:
        InvalidJobDescriptionError - in case of non-unique job name or inva
lid job description
    """
    def add(self, dAttrs = None, **attrs)

    """
    Validates and adds a new job description in a standard format (acceptab
le to the QCG PJM) to the group.
    If both arguments are present, they are merged and processed as a singl
e dictionary.


    Args:
        dAttrs (dict) - standard format attributes as a dictionary
        stdAttrs (dict) - standard format attributes as a named arguments


    Raises:
        InvalidJobDescriptionError - in case of non-unique job name or inva
lid job description

    """
    def addStd(self, dAttrs = None, **stdAttrs)

    """
    Remove a job from the group.


    Args:
        name (str) - name of the job to be removed


    Raises:
        JobNotDefinedError - in case of lack in a group of a job with the g
iven name
    """
    def remove(self, name)
    """
    Read job descriptions (in a standard format acceptable to the QCG-PJM)
from a JSON file


    Args:
        filePath (str) - path to the file containing job descriptions in a
standard format


    Raises:
        InvalidJobDescriptionError - in case of invalid job description
    """
    def loadFromFile(self, filePath)


    """
    Save job list to a file in a JSON format.


    Args:
        filePath (str) - path to the destination file


    Raises:
        FileError - in case of problems with opening / writing output file.
    """
    def saveToFile(self, filePath)
```

The `Jobs` object holds a list of job descriptions. The names of jobs must be unique. The jobs can be described in a format presented in *this section*, as well as in a simplified one. The main difference between formats is that the simplified one has a more flat structure, where `execution` and `resources` elements have been moved to the top level. The next difference is that the `dependencies` section, has been simplified and replaced with the `after` element, that contains a list of ancestor jobs.

The simplified job description format contains the following keys:

- `name` (required - `True`, allowed types - `str`) - a job's name,
- `exec` (required - `True`, allowed types - `str`) - a path to the executable,
- `args` (required - `False`, allowed types - `list`, `str`) - an argument or a list of arguments,
- `stdin` (required - `False`, allowed types - `str`) - a path to the standard input file,
- `stdout` (required - `False`, allowed types - `str`) - a path to the standard output file,
- `stderr` (required - `False`, allowed types - `str`) - a path to the standard error file,
- `wd` (required - `False`, allowed types - `str`) - a path to the working directory,
- `numNodes` (required - `False`, allowed types - `dict`) - number of nodes requirements (described in *section*),
- `numCores` (required - `False`, allowed types - `dict`) - number of cores requirements (described in *section*),
- `wt` (required - `False`, allowed types - `str`) - a wall-time specification,
- `iterate` (required - `False`, allowed types - `list`) - a list describing iterations, it should contain two or three elements: `start iteration`, `end iteration` and optionally, `the step iteration`,
- `after` (required - `False`, allowed types - `list`, `str`) - a list of (or single entry) of tasks that should finish before current one starts.

1.9.1.5 `list`

```
    """
    List all the jobs.
    Return a list of all job names along with their statuses.


    Returns:
        list - list of jobs with additional data in format described in 'li
stJobs' method in QCG PJM.


    Raises:
        InternalError - in case of request processing internal error
        ConnectionError - in case of lack of connection to the service or n
onzero response code.
    """
    def list(self)
```

This method returns a list of all submitted and registered (not yet removed) jobs from the QCG PilotJob Manager instance along with their statuses. The output format is described *in*.

1.9.1.6 *status*

```
    """
    Return current statuses of the given jobs.


    Args:
        names (list, str) - list of job names


    Returns:
        list - a list of job statuses in the format described in 'jobStatus
' method of QCG PJM.


    Raises:
        InternalError - in case of request processing internal error
        ConnectionError - in case of lack of connection to the service or n
onzero response code.
    """
    def status(self, names)
```

This method returns a statuses of the given jobs. The output format is described *in*.

1.9.1.7 *info*

```
    """
    Return detailed information about the given jobs.


    Args:
        names (list, str) - a list of job names


    Returns:
        list - a list of job's detailed information in the format described
in 'jobStatus' method of
          QCG PJM.


    Raises:
        InternalError - in case of request processing internal error
        ConnectionError - in case of lack of connection to the service or n
onzero response code.
    """
    def info(self, names)
```

The *info* method returns detailed information about specified jobs. The output format is described *in*.

1.9.1.8 *remove*

```
    """
    Remove jobs from registry.


    Args:
        names (list, str) - a list of job names


    Raises:
        InternalError - in case of request processing internal error
        ConnectionError - in case of lack of connection to the service or n
onzero response code.
    """
    def remove(self, names)
```

The *remove* method removes specified jobs from the registry of QCG PilotJob Manager. The output format is described [in][#removejob-command).

1.9.1.9 *wait4*

```
    """
    Wait until the given jobs finish.
    This method waits until all specified jobs finish its execution (succes
sfully or not).
    The QCG PJM is periodically polled about statuses 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 constructor.


    Args:
        names (list, str) - a list of job names


    Returns:
        dict - a map with job names and their terminal statuses


    Raises:
        InternalError - in case of request processing internal error
        ConnectionError - in case of lack of connection to the service or n
onzero response code.
    """
    def wait4(self, names)
```

The *wait4* method synchronises the execution of jobs - it waits (blocks) until all specified jobs are completed. The example output returned by the method is presented below:

```
{
   "msleep2": "SUCCEED",
   "echo": "SUCCEED",
}
```

1.9.2 Examples

An example user program utilising the QCG PilotJob Manager API is presented below:

```
import zmq


from qcg.appscheduler.api.manager import Manager
from qcg.appscheduler.api.job import Jobs


# switch on debugging (by default in api.log file)
m = Manager( cfg = { 'log_level': 'DEBUG' } )


# get available resources
print("available resources:\n%s\n" % str(m.resources()))


# submit jobs and save their names in 'ids' list
ids = m.submit(Jobs().
        add( name= 'host', exec = '/bin/hostname', args = [ '--fqdn' ], std
out = 'host.stdout' ).
        add( name = 'env', exec = '/bin/env', stdout = 'env.stdout', numCor
es = { 'exact': 2 } )
        )


# list submited jobs
print("submited jobs:\n%s\n" % str(m.list()))


# wait until submited jobs finish
m.wait4(ids)


# get detailed information about submited and finished jobs
print("jobs details:\n%s\n" % str(m.info(ids)))
```

1.9.3 Running with QCG

The QCG instance available at the Eagle cluster contains registered application *qcg-pm-client*. As a single argument, the application requires a user's Python application. During the execution of user's application, the environment is setup with the required Python compiler (version 3.5) as well as all needed modules (*qcg.appscheduler.api*). An example QCG job description is presented below:

```
#QCG note=pjm-client


#QCG host=eagle
#QCG walltime=PT1H
#QCG nodes=1:28


#QCG stage-in-file=api_ex.py
#QCG stage-out-dir=.->eagle.wd.${JOB_ID}


#QCG application=qcg-pm-client
#QCG argument=api_ex.py
```

In this case, the program from the file `api_ex.py` will be executed in a QCG PilotJob Manager environment.

1.9.4 Glossary

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 Pilot Job 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 the 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

Appendix 2 - Fusion QCG Job Description

```
<qcgJob appId="compat-FUSION-test" project="compat"
xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <task persistent="true" taskId="ES-task-2">
    <requirements>
      <multiCriteriaPatternTopology>
        <kernels>
          <kernel id="turb">
            <helper id="init"/>
            <helper id="transp"/>
            <helper id="equil"/>
            <helper id="f2dv"/>
            <helper id="dupEquil"/>
            <helper id="dupCorep"/>
          </kernel>
        </kernels>
        <plans>
          <plan id="plan1">
            <costs>
              <time>PT14H00M</time>
              <energy>100</energy>
            </costs>
            <group>
              <kernel refid="turb">
                <node host="supermuc"> thin </node>
                <resources>
                  <cores>1024</cores>
                </resources>
              </kernel>
            </group>
          </plan>
          <plan id="plan2">
            <costs>
              <time>PT28H00M</time>
              <energy>200</energy>
            </costs>
            <group>
              <kernel refid="turb">
                <node host="eagle"> haswell_128 </node>
                <resources>
                  <cores>512</cores>
                </resources>
              </kernel>
            </group>
          </plan>
        </plans>
        <decision>
          <optimize>
            <time limit="PT100H">1</time>
            <energy limit="10000">2</energy>
            <resources limit="200">2</resources>
          </optimize>
        </decision>
        <hosts>
          <host name="eagle">
            <reservation>eagle-res-1</reservation>
          </host>
          <host name="supermuc">
            <reservation>sm-res-1</reservation>
          </host>
        </hosts>
      </multiCriteriaPatternTopology>
    </requirements>
    <execution type="compat">
      <executable>
        <application name="muscle2" version="compat-1.2"/>
      </executable>
      <arguments>
        <value>test.cxa.rb</value>
      </arguments>
      <stdout>
        <directory>
          <location
type="URL">gsiftp://qcg.man.poznan.pl//home/plgrid-
groups/plggcompat/Fusion/FastTrack/qcg-test/results</location>
        </directory>
      </stdout>
      <stderr>
        <directory>
          <location
type="URL">gsiftp://qcg.man.poznan.pl//home/plgrid-
groups/plggcompat/Fusion/FastTrack/qcg-test/results</location>
        </directory>
      </stderr>
      <stageInOut>
        <file name="test.cxa.rb" type="in">
          <location
type="URL">gsiftp://qcg.man.poznan.pl//home/plgrid-
groups/plggcompat/Fusion/FastTrack/qcg-test/test.cxa.rb</location>
        </file>
        <file name="inputs.tgz" type="in">
          <location
type="URL">gsiftp://qcg.man.poznan.pl//home/plgrid-
groups/plggcompat/Fusion/FastTrack/qcg-test/inputs.tgz</location>
        </file>
        <file name="extract_inputs.sh" type="in">
          <location
type="URL">gsiftp://qcg.man.poznan.pl//home/plgrid-
groups/plggcompat/Fusion/FastTrack/qcg-
test/extract_inputs.sh</location>
        </file>
        <directory name="outputs" type="out">
          <location
type="URL">gsiftp://qcg.man.poznan.pl//home/plgrid-
groups/plggcompat/Fusion/FastTrack/qcg-test/results</location>
        </directory>
      </stageInOut>
      <environment>
        <variable
name="QCG_MODULES_LIST">compat/apps/fusion</variable>
        <variable name="QCG_PREPROCESS">extract_inputs.sh</variable>
      </environment>
    </execution>
  </task>
</qcgJob>
```

5 References

[1] Deliverable *D5.2 – ComPat middleware services*: http://www.compat-project.eu/wp-content/uploads/2018/01/d5.2-firstreportoncompatmiddlewareservices.pdf

[2] Deliverable *D5.1 – Architecture of the ComPat system*: http://www.compat-project.eu/wp-content/uploads/2016/07/ComPat_D5.1.pdf

[3] J. Borgdorff, M. Mamonski, B. Bosak, K. Kurowski, M. Ben Belgacem, B. Chopard, D. Groen, P. V. Coveney, and A. G. Hoekstra, "Distributed Multiscale Computing with MUSCLE 2, the Multiscale Coupling Library and Environment," Journal of Computational Science. 5 (2014) 719–731. doi:10.1016/j.jocs.2014.04.004

[4] Deliverable D4.3 – *Final report on high level tools, including performance profiling and modelling*

[5] Deliverable D6.3 – *Final report on the Experimental Execution Environment*

[6] Vytautas Jancauskas, Tomasz Piontek, Piotr Kopta, Bartosz Bosak, "Predicting Queue Wait Time Probabilities for Multi-Scale Computing", submitted to Philosophical Transactions journal.

[7] Alowayyed, Saad, et al. "Multiscale Computing in the Exascale Era." *arXiv preprint arXiv:1612.02467,* 2016.

[8] Deliverable D3.3 – *Report on instantiating computing patterns and performance measurements and prediction of HPMC application*