QMap Documentation

Release 0.4

BBGLab

Jun 21, 2022

Contents:

1 General overview **1**

2 QMap **3**
- 2.1 Tools . . . 3
- 2.2 Jobs map file . . . 5
- 2.3 Installation . . . 5
- 2.4 License . . . 5

3 Concepts **7**
- 3.1 Executor . . . 7
- 3.2 Profile . . . 7
- 3.3 Parameters . . . 8

4 Executors **9**
- 4.1 Local executor . . . 9
- 4.2 SLURM executor . . . 9
- 4.3 SGE executor . . . 10
- 4.4 LSF executor . . . 10

5 run **13**
- 5.1 Usage . . . 13
- 5.2 Examples . . . 13

6 template **15**
- 6.1 Generating a file . . . 15
- 6.2 Using wildcards . . . 15
- 6.3 Usage . . . 16
- 6.4 Examples . . . 16

7 submit **19**
- 7.1 Job parameters . . . 20
- 7.2 Groups . . . 20
- 7.3 How does it work? . . . 21
- 7.4 Usage . . . 22
- 7.5 Examples . . . 22

8 reattach **25**

8.1 Usage 25
8.2 Examples 25

9 info **27**
9.1 Exploring the metadata 27
9.2 Filtering commands 27
9.3 Examples 28

10 Configuration **29**

11 Indices and tables **31**

Index **33**

CHAPTER 1

General overview

CHAPTER 2

QMap

QMap is a tool aimed to run a collection of similar jobs quickly and easily in parallel. It can run standalone or using different HPC schedulers (Slurm, Sun Grid Engine and LFS).

QMap contains 5 different tools:

- *run*: execute commands with extended resources
- *template*: create a jobs map file
- *submit*: submit jobs from a map file
- *reattach*: reattach to a previous QMap execution
- *info*: explore the metadata of your jobs

Warning: **QMap** makes use of the shell using the *subprocess* module of the Python standard library. The shell is invoked making use of `shell=True` which can lead to shell injections vulnerabilities. More details in https://docs.python.org/3/library/subprocess.html#security-considerations

Documentation in: https://qmap.readthedocs.io/en/latest/

2.1 Tools

qmap run Execute a command with more resources maintaining your working environment

```
qmap run -m <memory> -c <cores>  "<command>"
```

qmap template Create a *jobs map file* that works with **qmap submit**.

```
qmap template "<command with wildcards>" -f <jobs map file>
```

The file created uses the current loaded Easy Build modules and the current conda environment as jobs pre-commands[1] if not explicitly provided.

The job commands are all the combinations that result of the expansion of:

{{list,of,items}} comma separated list of items

{{file}} all lines in file

***, ?, [x-y]** wildcards in Python's glob module

Wildcards of the format `{{...}}` are expanded in a first phase and glob wildcards are expanded later on.

As additional feature, any of the above mentioned groups can be named `{{?<name>:...}}` and replaced anywhere using `{{?=<name>}}`.

Note: To name glob wildcards they should be solely in the group. E.g. `{{?myfiles:*}}`

qmap submit Execute all jobs from a *jobs map file*

```
qmap submit -m <memory> -c <cores>  <jobs map file> --logs <logs folder> --max-
↪running <#>
```

`qmap submit` has been implemented to submit a set of jobs to a cluster for execution and control them. It acts as a layer between the workload manager and the user preventing she/he from submitting a huge number of jobs at once (potentially blocking future users). The number of jobs that can be submitted to the workload manager is controlled by the *–max-running* flag.

Warning: If `qmap submit` is closed, jobs that have not been submitted to the workload manager will never be. Thus, it is recommended to run it inside a **screen**.

In addition, in the folder indicated to store the logs with the *–logs* flag the user can find important information about each job execution as well as the logs from STDOUT and STDERR.

Another feature of this tool is the possibility to group your jobs with the *–grouping* option. This option uses the value passed as the number of commands that fit in each job. Thus, several commands can be executed as part of the same job, one after another. This option can be interesting for "small" jobs as they use the same allocation. If any of the commands fail, the associated job will fail.

Finally, any job command can include several values that are substituted before execution. Those values represent the job parameters and additionally, a unique identifier for each job.

${QMAP_LINE} identifier of the line the job command has in the input file

${QMAP_CORES} cores for the execution

qmap reattach Once a `qmap submit` execution is closed, you can reconnect from its logs directory

```
qmap reattach --logs <logs folder>
```

Note: If in the previous execution there were jobs that have not been submitted to the workload manager `qmap reattach` can submit them, but the execution is halted except for the `no-console` interface.

[1] Commands executed before any actual job

qmap info `qmap submit` generates a file for each job with metadata information. `qmap info` is designed to explore them and retrieve the requested data. Information is stored in json format and the user can request any fields:

```
qmap info --logs <logs folder> <field 1> <field 2>.<subfield 1> ...
```

In addition, the *–status* option can be used to filter the jobs by their status (completed|failed|other|pending|running|unsubmitted|all).

If you do not pass any field, then the return value is the input commands of the jobs. This feature can be used to generate a new jobs file a subset of the original one.

2.2 Jobs map file

This file contains a list of the commands to be executed as well as commands to be executed before and after each job (e.g. loading Easy Build modules or conda environments). The format of the file is:

```
[pre]
# command to be executed before any job

[post]
# command to be executed after any job

[params]
# parameters for all the jobs
cores = 7
memory = 16G

[jobs]
job command
job command
```

2.3 Installation

QMap depends on Python >3.5 and some external libraries.

You can install it directly from our github repository:

```
pip install git+https://github.com/bbglab/qmap.git
```

2.4 License

Apache Software License 2.0.

CHAPTER 3

Concepts

There are a few concepts we will like to explain so that it is easier to follow the documentation.

3.1 Executor

The executors are gluing pieces between **QMap** and the associated workload manager (if any). For example the *slurm* executor is in charge of interacting with a SLURM workload manager, while the *sge* executor does the same thing with a SGE (Sun Grid Engine) cluster.

Currently, only 4 executors are implemented:

- **dummy**: executor that does nothing. It is used for testing.
- **local**: executor that uses `bash` on Linux systems.
- **slurm**: executor for SLURM.
- **sge**: executor for SGE.
- **lsf**: executor for Platform LSF.

To select the executor you want to use read about the *profile*.

Find information about all executors in the *executors section*.

3.2 Profile

When executing a job or a set of jobs, **QMap** requires to know to which executor you are looking to. That is done using the **profile**. Essentially, the *profile* is a configuration file that only need to indicate the executor. E.g.:

```
executor = slurm
```

However, the profile can also be used to indicate the default parameters for the jobs using that profile:

```
[params]
cores = 7
```

In addition, you can also indicate which parameters can be changed during execution (this change will have effect on unsubmitted jobs):

```
[editable_params]
cores = Cores
```

QMap has built-in profiles for all the executors it supports. Those are empty, but you can edit them on `~/.config/qmap/`. Moreover, you can add any new profile in that folder and then pass them to **QMap** by just providing the name of the file (e.g. *slurm*).

Other variables that can be set are:

- show_usage = True/False. Indicate whether to call the get_usage function of the executor or not.
- max_ungrouped = integer. Maximum number of jobs allowed without grouping.

The profile that is passed to **QMap** is a combination of the profile indicated and the *default* profile (also available in `~/.config/qmap/`). The latter has lower priority. This feature can be useful for sysadmins to configure some defaults.

Finally, it is important to mention that the profile parameters (in the commands that use it) is not required. By default it is read from the `QMAP_PROFILE` environment variable (which can contain the name of a profile in `~/.config/qmap/` or a path to another file). If that variable is not set, and no profile is passed, the local executor will be used (without reading its profile).

3.3 Parameters

Parameters (also referred as *params*) indicate how a job is submitted to the corresponding workload manager. Thus, each executor might be able to receive a different set of parameters. The parameters can be passed using the *jobs file* or the *profile*.

Some parameters can be passed through the command line interface, thus they are equal for all executors. Those parameters are:

- cores: integer representing the number of cores to request
- memory: requested memory (units are T|G|M|K). Units are optional.
- time: wall time for the job (units are d|h|m|s). Units are optional.

In addition, each executor can receive an additional set of parameters. Check the *executors section* to find more details. Those parameters are only allowed in the profile.

More over, as some parameters are not supported out of the box by **QMap**, the `extra` parameters is also allowed. It is a string where you can set any of the native supported options.

CHAPTER 4

Executors

4.1 Local executor

The local executor is a simple executor that runs commands on a linux system using **bash**.

Important: If you close **QMap** all running jobs will be killed.

4.1.1 Params

Regarding the *parameters* this executors only cares about the *working directory* and the *extra parameter*.

4.2 SLURM executor

The SLURM executor is used to run jobs on a cluster using the SLURM workload manager.

When a jobs file is submitted, the `sbatch` command is used for each job. The `--no-requeue` option is always used.

When the `run` command is invoked, this executor makes use of `salloc` and `srun`.

4.2.1 Parameters

Accepted parameters for a job and its conversion into SLURM parameters:

- cores: -c
- memory : –mem
- time: -t

- nodes: -N
- tasks: -n
- working_directory: -D
- name: -J

In addition, the *extra parameter* is also available. In the time parameter, you can either use the **QMap** *format* or any of the SLURM accepted formats.

4.3 SGE executor

The SGE executor is used to run jobs on a cluster using the Sun Grid Engine workload manager.

When a jobs file is submitted, the `qsub` command is used for each job.

When the `run` command is invoked, this executor makes use of `qrsh`.

4.3.1 Parameters

Accepted parameters for a job and its conversion into SGE parameters:

- cores: `-pe` if penv is passed, otherwise `-l slots=`
- penv: option for *-pe‘*
- memory : `-l h_vmem=`
- queue: `-q`
- time: `-l h_rt=`
- tasks: `-n`
- working_directory: `-wd`
- name: `-N`

Important: configure the appropiate `penv` option in your *profile file*.

In addition, the *extra parameter* is also available. In the time parameter, you can either use the **QMap** *format* or any of the SGE accepted formats.

4.4 LSF executor

The LSF executor is used to run jobs on a cluster using Platform LSF.

When a jobs file is submitted, the `bsub` command is used for each job.

When the `run` command is invoked, this executor makes use of `bsub -I`.

4.4.1 Parameters

Accepted parameters for a job and its conversion into SGE parameters:

- cores: `-n` and adding `-R span[hosts=1]` to limit the job to a single node
- memory : `-M` and adding `-R select[mem>={}] rusage[mem={}]` (replacing `{}` with the memory value
- queue: `-q`
- time: `-W`
- working_directory: `-cwd`
- name: `-J`

In addition, the *extra parameter* is also available. In the time parameter, you can either use the **QMap** *format* or any of the LSF accepted formats.

CHAPTER 5

run

The `qmap run` command is aimed to be use to execute a single command in a cluster with extended resources.

In certain cluster managers, you can ask for job resources to have a interactive console running on a worker node. Typically, the resources of such a job are quite limited, so few resources are taken even if people leave that console open.

`qmap run` allows users to run one specific command as another job and then return so that resources are optimized and only taken for the time that the job requires them.

Note: `qmap run` keeps your working directory and environment variables for the execution.

Once the job finishes, `qmap run` will try to provide the user with some job statistics (if available) like the memory consumed or the elapsed time.

5.1 Usage

Basic usage:

```
qmap run -m <memory> -c <cores> "<command>"
```

Check all options using **`qmap run --help`**.

5.2 Examples

Usage example:

```
$ qmap run -c 6 -m 12G "sleep 5 && echo 'hello world'"
Executing sleep 5 && echo 'hello world'
salloc: Granted job allocation 31707
```

(continues on next page)

(continued from previous page)

```
hello world
salloc: Relinquishing job allocation 31707
Elapsed time:  00:00:05
Memory  0G
```

Jobs that require more resources can be easily re-run:

```
$ python test/python_scripts/memory.py 10
1 Gb
2 Gb
...
8 Gb
Killed

$ qmap run -m 12 "python test/python_scripts/memory.py 10"
Executing python test/python_scripts/memory.py 10
salloc: Granted job allocation 36015
1 Gb
...
10 Gb
salloc: Relinquishing job allocation 36015
Elapsed time:  00:00:36
Memory  10G
```

CHAPTER 6

template

`qmap template` is a tool aimed to ease the creation of a *jobs file* to be used with **qmap submit**.

Features:

- find your current loaded *EasyBuild modules* and adds them to the output as pre-command
- find your current *conda environment* and add them to the output as pre-command
- the *job parameters* passed through command line are added to the generated jobs_file

6.1 Generating a file

By default, the output is printed to the standard output. You can provide a file with the `-o` flag or redirect the output to a file (`> file.txt`).

6.2 Using wildcards

`qmap template` accepts two types of wildcards:

- **user wildcards**: indicated with `{{...}}` They can contain:
  - list of `,` separated items: the wildcard is replaced by each item
  - file name: he wildcard is replaced by each line in the file
- **glob wildcards**: if any of `*` and `**` is found, it is assumed to be a *glob* wildcard and therefore expanded using the python glob module.

6.2.1 How it works

The expansion of wildcards is a two step process. First *user wildcards* are expanded and *glob wildcards* are expanded in a second phase. For the latter, any set of characters surrounded by blanks is analysed. If it contains one or more of the mentioned wildcards, a glob search is performed.

Note: Use \ before glob wildcards to avoid their expansion

6.2.2 Named groups

`qmap template` contains a special feature that allows the user to replace the values of **any of the wildcards** in different parts of the command.

To use this feature, the wildcard needs to be named using `{{?<name>:<value>}}` and it can be replaced anywhere using `{{?=<name>}}`.

The *name* can be anything, but a *glob wildcard* character. It cannot start with a number.

Tip: We recommend to limit the names to characters in a-z, A-Z and 0-9.

The *value* can be **anything** in a *user wildcard* or a *glob wildcard*.

Note: Even if it is possible to use a glob wildcard in a user wildcard (e.g. {{a,*.txt}}) we do not recommend this use for named groups as the result might differ from the expected.

Warning: As mentioned, in a user group you can place anything that is is a user or glob wildcard. Thus, `{{?group:*}}.txt` recognize the glob wildcard and and will do a glob search for all `.txt` files. On the other hand, `{{?group:*.txt}}` assumes it is a user wildcard (as it is not only a glob wildcard) and will try to open a file named `*.txt` which most likely will not exits and will fail.

6.3 Usage

Basic usage:

```
qmap template "<command>" -m <memory> -c <cores> -o <output file>
```

Check all options using **`qmap template --help`**.

6.4 Examples

Easybuild modules and **conda environments** are recognized:

```
$ qmap template "sleep 5"
[pre]
module load anaconda3/4.4.0
```

(continues on next page)

(continued from previous page)

```
source activate test_qmap

[jobs]
sleep 5
```

Job parameters can also be added:

```
$ qmap template "sleep 5" -c 1 -m 1G
[params]
memory=1G
cores=1

[jobs]
sleep 5
```

Using **user wildcards** with lists:

```
$ qmap template "sleep {{5,10}}"
[jobs]
sleep 5
sleep 10
```

Using **user wildcards** with files:

```
$ qmap template "sleep {{sleep_times.txt}}"
[jobs]
sleep 5
sleep 10
```

Using **glob wildcards**:

```
$ qmap template "mypgrog --input *.txt"
[jobs]
mypgrog --input file1.txt
mypgrog --input file2.txt
```

Using **named wildcards**:

```
$ qmap template "myprog --input {{?f_name:*}}.txt --variable {{?v_name:a,b}} --output
↪{{?=f_name}}_{{?=v_name}}"
[jobs]
myprog --input file1.txt --variable a --output file1_a
myprog --input file2.txt --variable a --output file2_a
myprog --input file1.txt --variable b --output file1_b
myprog --input file2.txt --variable b --output file2_b
```

CHAPTER 7

submit

`qmap submit` launches a bunch of commands to the workload manager for each execution. The commands to be executed come from a file with the following format:

```
[pre]
pre-command 1
pre-command 2
...
pre-command l

[post]
post-command 1
post-command 2
...
post-command n

[params]
job parameters

[jobs]
job 1
job 2  ## job specific parameters
...
job m
```

Job pre-commands Command to be executed before any job

Job parameters Resources asked to the workload manager (e.g. memory or cores)

Job command Bash command to be executed. One command corresponds to one job unless *groups* are made

Job post-commands Commands to be executed before any job

An example of such file:

```
# module load anaconda3
# source activate oncodrivefml
```

(continues on next page)

(continued from previous page)

```
## cores=6, memory=25G

oncodrivefml -i acc.txt -e cds.txt -o acc.out
oncodrivefml -i blca.txt -e cds.txt -o blca.out
```

`qmap submit` is a tool that is not only intended to easy the job submission, but also tries to limit the jobs that one user submits at once, preventing that he/she takes the whole cluster.

7.1 Job parameters

Using the command line interface, your profile or the jobs file, you can set the *parameters for you jobs*.

The **command line parameters** override *general job parameters* (the ones coming from the combination of the ones in your jobs file and the ones in your profile) but not *specific job parameters* (*see below*).

There are few *"global" parameters* that you can with any executor. Amongst these, there are two of them that are special:

- `working_directory`: is always set by the command line, and thus, setting it in your profile is useless. The default is the directory where you make the submission.
- `prefix`: is used to create a job name as <prefix>.<job ID> (or in you are inside a *screen* it uses the name of such *screen*)

One of the features of **QMap** is that you can access the parameters of the job from your job command. E.g. Using **`qmap submit --cores 5 jobs.map``** you can have a `jobs.map` file with commands like:

```
python parallel1.py --cores ${QMAP_CORES}
python parallel2.py --cores ${QMAP_CORES}
python parallel3.py --cores ${QMAP_CORES}
```

And the variable `QMAP_CORES` will be exported.

Note: It is possible to use `${QMAP_LINE}` in your command. It will provide a unique identifier for each job command. It will be replaced (before the job command is created).

7.2 Groups

Optionally, there can be a limit to commands per submission without **grouping**. Grouping involves that a set of `x` commands is executed one after the other as part of the same job. If one fails, the job is terminated.

Warning: *Job specific parameters* are ignored in grouped submissions (if the group is bigger than 1).

7.3 How does it work?

7.3.1 Reading the jobs file

Lines starting with `#` are assumed to be comments. Then, commands and parameters are read from their sections as explained in the *above*.

If no sections (using `[section]`) are present, all non-blank lines are assumed to be job commands.

If the job command contains `##` anything from there is interpreted as *specific job parameters*.

7.3.2 Generating the jobs

Once the *jobs file* is parsed, the jobs are created. This process involves:

- creating an **output directory** and copying the *jobs file*

  Note: If the output directory is not empty, `qmap` will fail

- each job receives and **id** that correspond to its line in the submit folder
- for each job command one file with the job **metadata** is created (named as *<job id>.info*)

  Warning: To prevent lots of writing to the `.info` file, `qmap` only writes to disk on special cases, when explicitly asked or before exiting.

- for each job a **script file** with the commands to be executed is created. The file is typically named as *<job id>.sh* and consists on:
  - all the *pre-commands*
  - all the *commands* in the group or a single command if not groups are made
  - all the *post-commands*

  Note: The job commands can contain some environment variables:
  - ${QMAP_LINE}: identifier of the line of the job (unique for each command). This wildcard is expanded before job submission.
  - ${QMAP_<PARAM>}: all job parameters that are explicitly passed are available to the commands, and they are exported as environment variables.

7.3.3 Running the jobs

- the jobs start to be submitted to the executor. Only certain amount of jobs are submitted according to the `--max-running` parameter. This parameter accounts for running and pending jobs.
- each job requests certain **resources** to the executor. The order of priority is: *command line parameters*, *general job parameters* from the *jobs file* and *default parameters*.

Note: If no grouping is perform (or group are of size 1) and the job contains **specific job parameters** those have the highest priority.

- the job output to the standard error is logged in a file named as *<job id>.out* and the job output to the standard error is logged in *<job id>.err*.

7.4 Usage

Basic usage:

```
qmap submit -m <memory> -c <cores> <jobs file>
```

Check all options using **`qmap submit --help`**.

7.5 Examples

Using this jobs file:

```
sleep 5 && echo 'hello world after 5'
sleep 10 && echo 'hello world after 10'
```

Basic example:

```
$ qmap submit -m 1 -c 1 hello.jobs --no-console
Finished vs. total: [0/2]
Job 0 done. [1/2]
Job 1 done. [2/2]
Manager finished
```

In the **output directory** of qmap, you can find a copy of the input file (as `qmap_input`) and for each job 4 up to for different files as explained above:

```
$ ls hello_20180905
0.err   0.info   0.out   0.sh   1.err   1.info   1.out   1.sh   qmap_input default.env
```

The **output directory** must not exist before the submission:

```
$ qmap submit hello.jobs --no-console
QMapError: Output folder [hello_20870905] is not empty. Please give a different␣
↪folder to write the output files.
```

Grouping reduces the number of jobs, but specific job execution parameters are ignored:

```
$ qmap submit hello.jobs -g 2 --no-console
Specific job execution parameters ignored
Finished vs. total: [0/1]
Job 0 done. [1/1]
Manager finished
```

The following examples make use of this other *jobs file*:

```
[pre]
module load anaconda3/4.4.0

[params]
memory=8G

[jobs]
python memory.py 8
python memory.py 10 ## memory=10G
```

The **working directory** is helpful when your jobs file does not contain the full path to your script

```
$ qmap submit memory.jobs --no-console
Finished vs. total: [0/2]
Job 2 failed. [1/2]
Job 3 failed. [2/2]
Manager finished

$ qmap submit memory.jobs -w test/python_scripts/ --no-console
Finished vs. total: [0/2]
Job 2 done. [1/2]
Job 3 done. [2/2]
Manager finished
```

CHAPTER 8

reattach

In order to re-open a previous **qmap** submission, `qmap reattach` does so using the data provided by the output (logs files) of that **qmap submit** execution.

Important: When using the GUI interface, the execution will be halted on reattachment.

8.1 Usage

Basic usage:

```
qmap reattach [--logs <folder>]
```

Check all options using **`qmap reattach --help`**.

8.2 Examples

```
$ qmap reattach --no-console
Finished vs. total: [2/2]
Manager finished
```

CHAPTER 9

info

`qmap info` is a tool aimed to explore the metadata of your jobs or to get the commands of certain jobs.

9.1 Exploring the metadata

`qmap info` can explore the metadata of your jobs and retrieve the fields of interest.

The first column corresponds to the **job id** and each of the other columns correspond to the requested fields. *Missing fields* return and empty string.

To check what fields you can return, you can simply take a look at one of the `.info` files. To access nested elements use `.` to divide the levels (e.g. `usage.time.elapsed`).

The return data is *tab separated* or `|` *separated* if the *collapse flag* is provided.

9.1.1 Usage

```
qmap info -s <status> -l <qmap logs folder> <field 1> <field 2> ... <field n>
```

9.2 Filtering commands

`qmap info` can also return a subset of your commands file that corresponds to the ones whose job have a certain status.

This option is enabled when no fields are passed in the command line. Moreover, in this case, the *collapse flag* removes empty lines from the output.

9.2.1 Usage

Basic usage:

```
qmap info -s <status> -l <qmap logs folder>
```

Check all options using **`qmap info --help`**.

9.3 Examples

Get the fields of interest from your jobs:

```
$ qmap info -s completed usage.time.elapsed retries
id      usage.time.elapsed      retries
1       00:00:12        0
0       00:00:07        0
```

CHAPTER 10

Configuration

Profiles are useful for configuring your jobs execution defaults. However, **QMap** as also some extra layer of configuration possible.

The default profile for you executions can be set using the environment variable `QMAP_PROFILE`. This can be either the *name* of the profile in the corresponding configuration folder (`~/.config/qmap/`) or a *path* a profile file.

The default folder for your configuration files can also be set-up using the `QMAP_HOME`. This feature can be useful for sysadmins that provide one installation of **QMap** for all the users.

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Index

E

environment variable
    QMAP_CORES, 20
    QMAP_HOME, 29
    QMAP_PROFILE, 29

Q

QMAP_CORES, 20
QMAP_HOME, 29
QMAP_PROFILE, 29