Embarrassingly Parallel workflows and how to execute them

SURF

Embarrassingly parallel workflows

What?

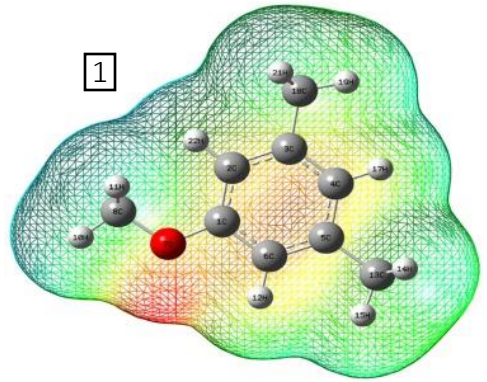
- Executing many independent small jobs simultaneously to use the hardware resource efficiently.

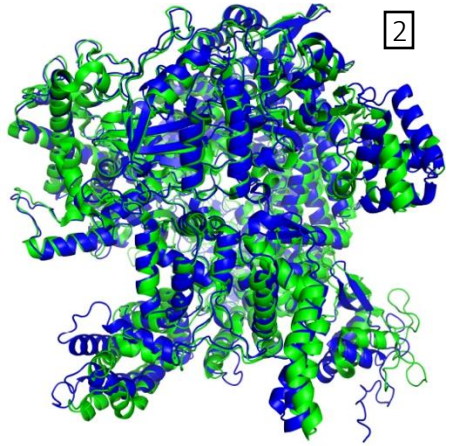
When?

- It is not possible to combine multiple runs/tasks into a single application.
- Large parameter sweep based on the same application or workflow with dependent pipeline steps.
- Workflow involves runs that are depending on several independent software which do not scale beyond a certain number of cores.

Why?

- Nodes are getting denser with respect to resources such as number of cores/threads, GPU cards, Memory.
- Applications do not necessarily scale even within a node: shared memory (OpenMP, MPI SHM, pthreads).


1


2

1. Mathiyalagan, A., Manimaran, K., Muthu, K., Rajakantham, M., Density functional theory study on the electronic structures and spectral properties of 3,5-Dimethylanisole dye sensitizer for solar cell applications, Results in Chemistry, Volume 3, 2021

2. Jumper, J., Evans, R., Pritzel, A. *et al.* Highly accurate protein structure prediction with AlphaFold. *Nature* **596**, 583–589 (2021)

How?

Tools

- Linux background processes (within a node) along with **numactl**.
  - Supported on Snellius and requires prior knowledge of bash scripting and process placement using **numactl**.
  - Provides maximum manual control if one knows what to do.
- SLURM arrays (to perform similar jobs on multiple allocations).
  - Supported on Snellius and requires prior knowledge of SLURM array options.
  - Has some limitations.
- Services for execution of many jobs inside a single allocation, e.g. QCG-PilotJob (https://qcg-pilotjob.readthedocs.io/en/develop/).
  - Supported on Snellius and requires prior knowledge of Python scripting.

SURF

How?

Tools

- Linux background processes (within a node) along with **numactl**.
  - Supported on Snellius and requires prior knowledge of bash scripting and process placement using **numactl**.
  - Provides maximum manual control if one knows what to do.
- SLURM arrays (to perform similar jobs on multiple allocations).
  - Supported on Snellius and requires prior knowledge of SLURM array options.
  - Has some limitations.
- Services for execution of many jobs inside a single allocation, e.g. QCG-PilotJob (https://qcg-pilotjob.readthedocs.io/en/develop/).
  - Supported on Snellius and requires prior knowledge of Python scripting.

SURF

There are many more tools

SLURM arrays

Limitations

- SLURM job arrays offer a mechanism for submitting and managing **collections of similar jobs**.
- All jobs must have the **same initial options** (e.g. size, time limit, etc.).
- The **max array size** depends on the settings defined in the SLURM configuration file. On Snellius it's **30000**.
- The **allocation size** per instance **cannot be below the minimum allocation** within SLURM. For instance, on Snellius it's 16 cores and 28 GiB of memory on the "rome" partition.
- One must **wait for each task** in the array to get allocated and finish.

job.sh

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -n 2
#SBATCH -p rome
#SBATCH -t 1
hostname
```

```
$ sbatch --array=0-10 job.sh
```

SURF

Limitations

- SLURM job arrays offer a mechanism for submitting and managing **collections of similar jobs**.
- All jobs must have the **same initial options** (e.g. size, time limit, etc.).
- The **max array size** depends on the settings defined in the SLURM configuration file. On Snellius it's **30000**.
- The **allocation size** per instance **cannot be below the minimum allocation** within SLURM. For instance, on Snellius it's 16 cores and 28 GiB of memory on the "rome" partition.
- One must **wait for each task** in the array to get allocated and finish.

job.sh

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -n 2
#SBATCH -p rome
#SBATCH -t 1
hostname
```

```
$ sbatch --array=0-10 job.sh
```

Will create **11** tasks!

Ways of control

- Jobs which are part of a job array will have the environment variable **SLURM_ARRAY_TASK_ID** set to its array index value.
- A **maximum number of simultaneously running tasks** from the job array may be specified **using a "%" separator**.

```
$ cat job.sh
#!/bin/bash
#SBATCH -N 1
#SBATCH -n 2
#SBATCH -p rome
#SBATCH -t 1
echo "SLURM_ARRAY_TASK_ID: $SLURM_ARRAY_TASK_ID"
hostname

$ sbatch --array=0-10 job.sh
...
$ sbatch --array=0-10%2 job.sh
...
```

SURF

Ways of control

- Jobs which are part of a job array will have the environment variable **SLURM_ARRAY_TASK_ID** set to its array index value
- A **maximum number of simultaneously running tasks** from the job array may be specified **using a "%" separator**.

```
$ cat job.sh
#!/bin/bash
#SBATCH -N 1
#SBATCH -n 2
#SBATCH -p rome
#SBATCH -t 1
echo "SLURM_ARRAY_TASK_ID: $SLURM_ARRAY_TASK_ID"
hostname

$ sbatch --array=0-10 job.sh
...
$ sbatch --array=0-10%2 job.sh
...
```

Returns a task ID

Will tell SLURM to submit at most 2 tasks at the same time

slurm workload manager

SURF

Ways of control

Useful environment variables

- **SLURM_ARRAY_JOB_ID** will be set to the first job ID of the array.
- **SLURM_ARRAY_TASK_ID** will be set to the job array index value.
- **SLURM_ARRAY_TASK_COUNT** will be set to the number of tasks in the job array.
- **SLURM_ARRAY_TASK_MAX** will be set to the highest job array index value.
- **SLURM_ARRAY_TASK_MIN** will be set to the lowest job array index value.

SURF

Ways of control

File names

Two additional options are available to specify a job's **stdin**, **stdout**, and **stderr** file names:

- **%A** will be replaced by the value of **SLURM_ARRAY_JOB_ID**
- **%a** will be replaced by the value of **SLURM_ARRAY_TASK_ID**

```
#!/bin/bash
...
#SBATCH --output slurm-%A_%a.out
...
```

SURF

Hands-on

Hands-on

Clone the repository

- https://github.com/sara-nl/course-module-qcg-pilotjob

```
$ cd ~
$ git clone https://github.com/sara-nl/course-module-qcg-pilotjob
$ cd course-module-qcg-pilotjob
```

Hands-on #1

Task

- Submit a job array and print **hostname, TMPDIR, SLURM_ARRAY_JOB_ID**

Steps

- Go to **hands-on/example_01**
- Take a look at the job script **array_job.sh**
- Submit the SLURM job script, check the SLURM log file

Questions

- How many output files were created?
- Are host names the same?
- What are the names of **TMPDIR** that are reported?

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -p rome
#SBATCH -t 10
#SBATCH --array 0-11%3

echo "Running on "
hostname
echo "TMPDIR $TMPDIR"

echo "SLURM_ARRAY_JOB_ID $SLURM_ARRAY_JOB_ID"
echo "SLURM_ARRAY_TASK_ID $SLURM_ARRAY_TASK_ID"
echo "SLURM_ARRAY_TASK_COUNT $SLURM_ARRAY_TASK_COUNT"
echo "SLURM_ARRAY_TASK_MAX $SLURM_ARRAY_TASK_MAX"
echo "SLURM_ARRAY_TASK_MIN $SLURM_ARRAY_TASK_MIN"
```

SURF

Hands-on #2

Task

- Compute the average of one column in 1000 CSV files
- Aggregate the results in one CSV file.

Steps

- Go to **hands-on/example_02**
- Take a look at the job script **array_job.sh**
- Run **prepare_input.sh**
- Submit the SLURM job script
- Run **aggregate.sh** when all jobs are finished

Questions

- What is the total execution time and SBUs spent on this task?

```
#!/bin/bash
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -p rome
#SBATCH -t 10
#SBATCH --array 0-999%10

module load 2023
module load Python/3.11.3-GCCcore-12.3.0

# copy input data and scripts to working directory
cp -r $SLURM_SUBMIT_DIR/input_mod $TMPDIR/
cp -r $SLURM_SUBMIT_DIR/average.py $TMPDIR/
cp -r $SLURM_SUBMIT_DIR/average.py $TMPDIR/

cd $TMPDIR

echo "TMPDIR = $TMPDIR”

# average data
input_file_path=input_mod/${SLURM_ARRAY_TASK_ID}_*
echo "Analyzing file: $input_file_path"
python3 average.py $input_file_path
```

QCG-PilotJob

SURF

QCGPilot-Job

Job manager inside SLURM

- Submits jobs within a single large allocation across multiple nodes.
- Does not have to wait for small individual allocations like in SLURM arrays.
- Tries to optimize resources for each job based on requirements such that the queue is always full.
- Possibility of restarting failed/timed out jobs.
- One can also specify complex result-based job dependencies.

Some caveats and drawbacks

- One needs to estimate, the total time of all the jobs combined.
- Job/system state files get produced in the submission directory.
- Might bring some complexity into execution of a Singularity containers with embedded MPI-based codes.
- In the case of problems, the log files may not be very expressive.

SURF

Workflow

Describes resources, loads the QCG-PilotJob module and runs the QCG-PilotJob python script

Invokes the QCG-PilotJob Manager that aggregates all your jobs along with resources and conditions based on which jobs need to be invoked

This is the main application that one wants to execute

Job output with QCG-PilotJob metrics

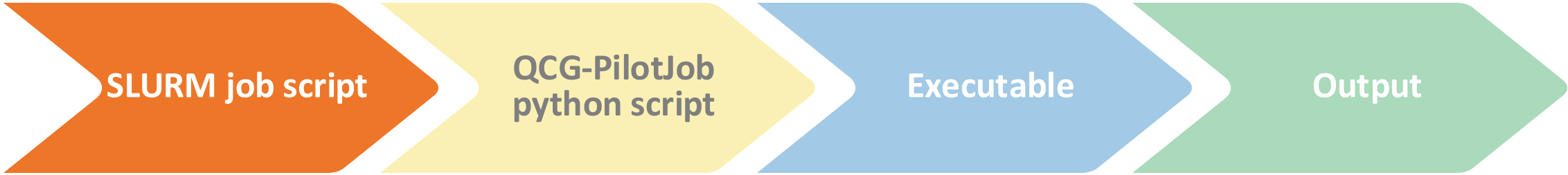
SURF

QCGPilot-Job: Set up the environment

- The **QCG-PilotJob** package is available as a module in the **2023** environment.
- Note that the environment from the login node is ***not transferred*** to the compute nodes, unless defined in, e.g., the **.bashrc** file. Thus, you have to re-load the QCG-PilotJob package in your job script.

```
$ module purge
$ module load 2023
$ module load QCG-PilotJob/0.14.1-foss-2023a
```

Workflow



Describes resources, loads the QCG-PilotJob module and runs the QCG-PilotJob python script

Invokes the QCG-PilotJob Manager that aggregates all your jobs along with resources and conditions based on which jobs need to be invoked

This is the main application that one wants to execute

Job output with QCG-PilotJob metrics

SURF

QCGPilot-Job: SLURM job script

run.sh

```
#!/bin/bash

#SBATCH -N 1
#SBATCH -n 16
#SBATCH -p rome
#SBATCH -t 10

module load 2023
module load QCG-PilotJob/0.14.1-foss-2023a

python test.py
```

Workflow

Describes resources, loads the QCG-PilotJob module and runs the QCG-PilotJob python script

Invokes the QCG-PilotJob Manager that aggregates all your jobs along with resources and conditions based on which jobs need to be invoked

This is the main application that one wants to execute

Job output with QCG-PilotJob metrics

SURF

QCGPilot-Job: Python script

- Work with the **QCG-PilotJob** manager consists of a few mandatory steps:
  - Creating an instance of the **QCG-PilotJob Manager**
    - creation of a class object
  - Waiting for all submitted jobs to finish
    - will wait until some or all submitted to the **QCG-PilotJob Manager** jobs finish
  - Calling for the **finish()** method at the end of the Python script
    - this method terminates the background thread in which the instance of a **QCG-PilotJob Manager** has been run

SURF

QCGPilot-Job: Python script

test.py

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

manager = LocalManager()
print('available resources: ', manager.resources())
manager.wait4all()
manager.finish()
```

Example output

```
available resources:  {'total_nodes': 1, 'total_cores': 96, 'used_cores': 0,
'free_cores': 96}
```

QCGPilot-Job: Python script

Create an object

Wait for all jobs to finish

Finalize the execution

test.py

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

manager = LocalManager()
print('available resources: ', manager.resources())
manager.wait4all()
manager.finish()
```

Example output

```
available resources:  {'total_nodes': 1, 'total_cores': 96, 'used_cores': 0,
'free_cores': 96}
```

SURF

QCGPilot-Job: Python script

test.py

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

manager = LocalManager()
print('available resources: ', manager.resources())
manager.wait4all()
manager.finish()
```

Create an object

Wait for all jobs to finish

Finalize the execution

Example output

```
available resources:  {'total_nodes': 1, 'total_cores': 96, 'used_cores': 0,
'free_cores': 96}
```

Accessible cores

Total number of nodes and cores within an allocation

SURF

QCGPilot-Job: Create jobs

- To execute multiple jobs within a single SLURM allocation we need to:
  - Create an object of the **Jobs** class
  - Populate it with some parameters (describe jobs)
  - Submit the created object to the **QCG-PilotJob Manager**

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

jobs = Jobs()
jobs.add(...)
jobs_ids = manager.submit(jobs)
print('submited jobs: ', str(job_ids))
...
```

SURF

QCGPilot-Job: Job description (main parameters)

- **name** - the job name
- **exec** - path to the executable program
- **args** - executable program arguments
- **script** - bash script content
- **stdin** - path to file which content should be passed to the standard input stream
- **stdout** - path to the file where standard output stream should be saved
- **stderr** - path to the file where standard error stream should be saved
- **modules** - list of modules that should be loaded before job start
- **numCores** - number of required cores specification
- **numNodes** - number of required nodes specification
- **iteration** - iterations definition
- **model** - model of execution

SURF

QCGPilot-Job: Job description (main parameters)

- **name** - the job name
- **exec** - path to the executable program
- **args** - executable program arguments
- **script** - bash script content
- **stdin** - path to file which content should be passed to the standard input stream
- **stdout** - path to the file where standard output stream should be saved
- **stderr** - path to the file where standard error stream should be saved
- **modules** - list of modules that should be loaded before job start
- **numCores** - number of required cores specification
- **numNodes** - number of required nodes specification
- **iteration** - iterations definition
- **model** - model of execution

QCGPilot-Job: execution models

- **default** - only a single process is launched within the allocation
- **threads** - is designed for running OpenMP tasks on a single node
- **openmpi** - the processes are started with the mpirun command
- **intelmpi** - same but for Intel MPI
- **srunmpi** - the processes are started with the srun command

Read more about execution models in the official docs:
https://qcg-pilotjob.readthedocs.io/en/develop/execution_models.html

SURF

QCGPilot-Job: Calling for an executable

test.py

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

manager = LocalManager()
jobs = Jobs()

jobs.add(script='echo "job ${it} executed at `date` @ `hostname`"',
        name='test_job',
        stdout='job.out.${it}',
        iteration=4)

job_names = manager.submit(jobs)
print('submited job names: ', str(job_names))
job_status = manager.status(job_names)
print('job status: ', job_status)

manager.wait4all()
manager.finish()
```

Hands-on

Hands-on #3

Task

- Execute **512 independent short jobs** on two nodes on the “rome” partition. Note that one node on the “rome” partition has 128 cores.

Steps

- Go to **hands-on/example_03**
- Take a look at **qcg_job.py** and **qcg_job.sh**
- Submit the SLURM job script, check the results

Questions

- What host names are reported?
- What additional files and directories are created?

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

# create the QCG manager and the Jobs object
manager = LocalManager()
jobs = Jobs()

# create 512 independent jobs
for job in range(512):
    # define a job
    jobs.add(name="job_{}".format(job),
             exec='hostname',
             stdout='output/job.{}.out'.format(job),
             stderr='output/job.{}.err'.format(job),
             model='default',
             numCores={ "exact" : 1},
             iteration=1)

print("-- submit jobs --")
manager.submit(jobs)
print("-- wait for all jobs --")
manager.wait4all()
manager.finish()
print("-- finished")
```

Hands-on #4

Task

- Compute the average of one column in 1000 CSV files.
- Aggregate the results in one CSV file.

Steps

- Go to **hands-on/example_04**
- Take a look at **README.md**, **qcg_job_v0.py** and **qcg_job.sh**
- Extract the input data (**input.tar.gz**)
- Submit the SLURM job script, check the results

Steps

- The ***v0** script has a problem. Can you spot it?
- Execute **qcg_job_v1.py**, compare results with the results from **qcg_job_v0.py**

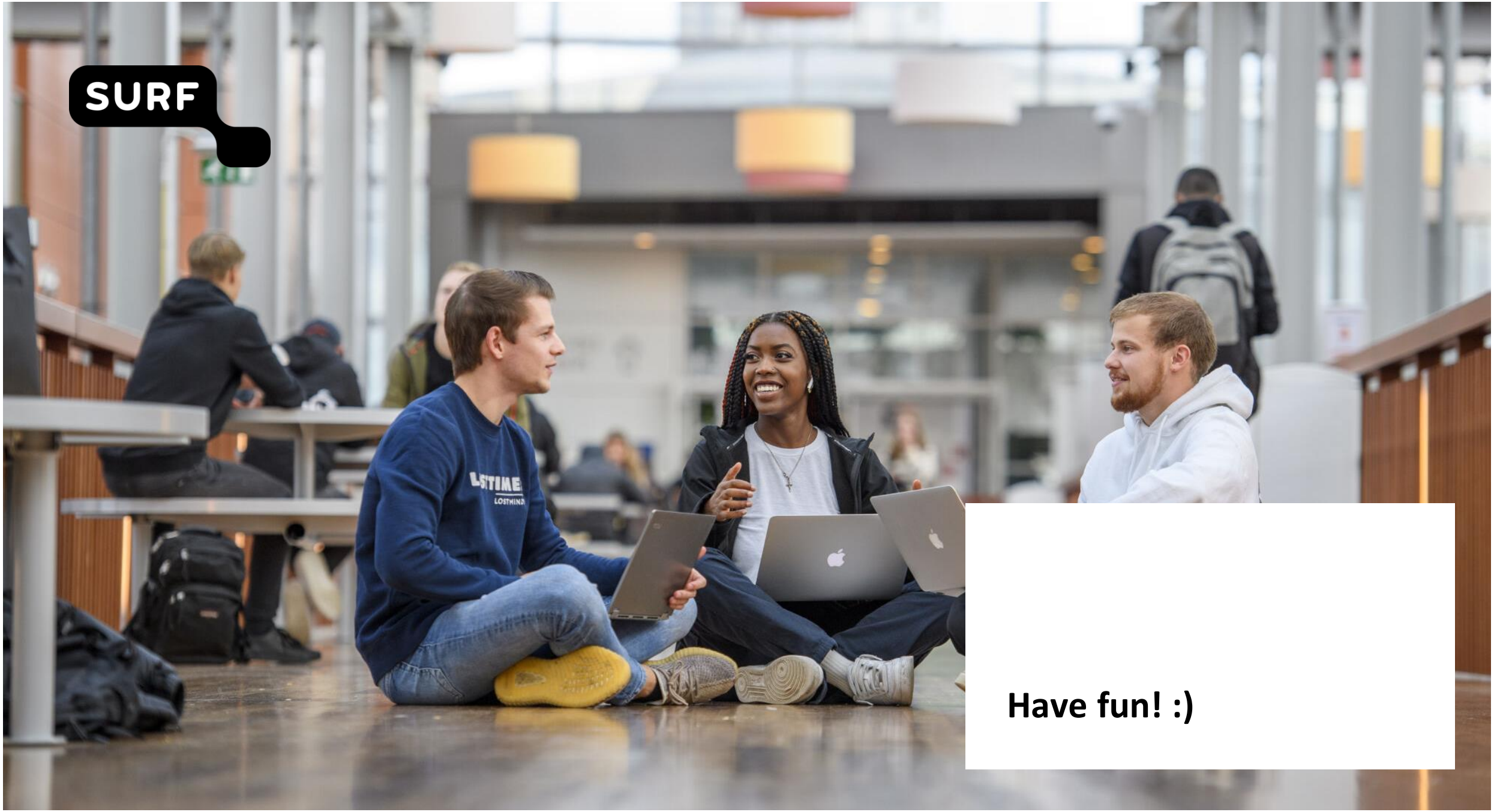
```
job_names = []

# scan input directory for CVS files
for filename in glob(os.path.join("input/*.csv"), recursive = False):
    # determine the job name and append it to the list
    job_names.append(os.path.basename(filename))

# create the QCG manager and the Jobs object
manager = LocalManager()
jobs = Jobs()

# loop over all files and populate the list of jobs
for jname in job_names:
    print("submit {}".format(jname))
    jobs.add(name=jname,
             exec='python3',
             args=["average.py",
                   os.path.join("input", jname)],
             stdout='average_{}'.format(jname),
             stderr='job.{}.err'.format(jname),
             modules=[
            "2023", "Python/3.11.3-GCCcore-12.3.0"],
             iteration=1
             )

#  add a job for the aggregation of the results
jobs.add(name="aggregate",
         script='cat average_*.csv | sort',
         stdout='result.csv',
         stderr='aggregate.err')
```



Have fun! :)