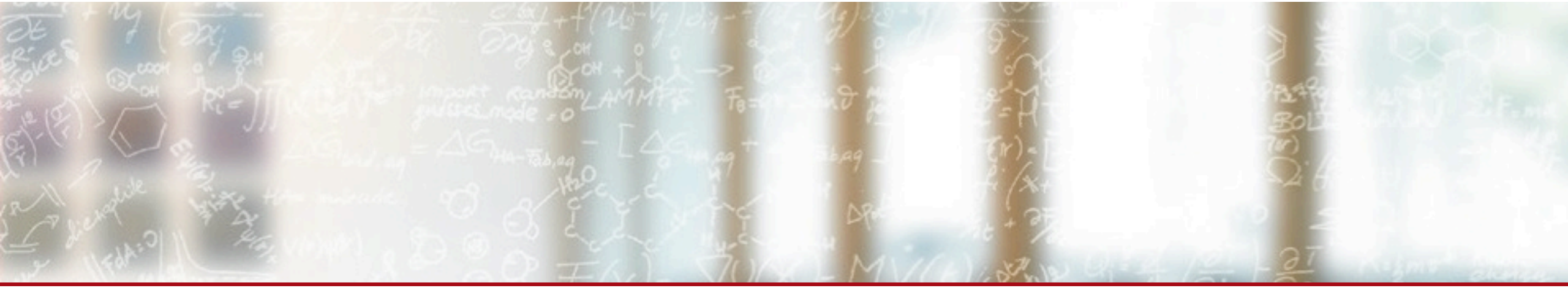
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Workflows and best practices (SLURM, Data transfer, Pre- & Post-processing)

Miguel Gila, Stefano Gorini and Victor Holanda

CSCS User Lab Day – Meet the Swiss National Supercomputing Centre
September 11, 2018

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Who are we?

Ourselves

Stefano Gorini

HPC Operation Group Leader of Data and Network Services

“In my previous life, I have been a Computer Scientist with a strong interest in Storage, Backup and System administration.”

Victor Holanda

Scientific Support and Engagement

“In my current life, I am a PhD in Chemistry with a strong interest in Linux, Parallel Programming and System administration.”

Miguel Gila

HPC Operation

“In my future life, I am a sysadmin for all-things Piz Daint and Slurm.”

SLURM

Slurm

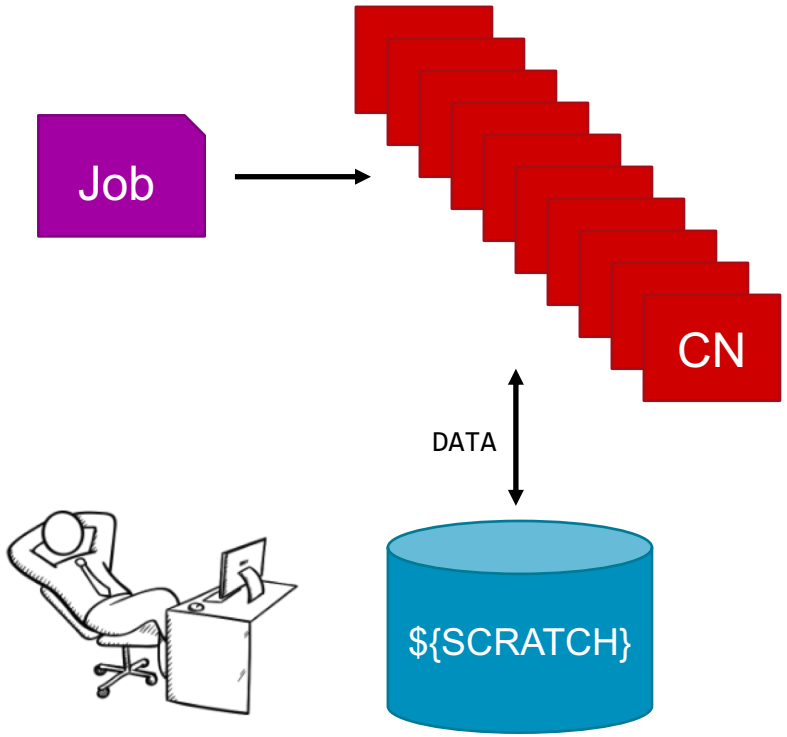
- Slurm is the batch system/scheduler running on Piz Daint
- Permits users to run jobs on multiple nodes on different queues

Name of the queue	Max time	Max nodes	Brief Description
debug	30 min	4	Quick turnaround for test jobs (one per user)
large	12 h	4400	Large scale work, by arrangement only
long	72 h	4	Maximum 5 long jobs in total (one per user)
low	6 h	2400(gpu)/512(mc)	Use only when allocations are exhausted
normal	**24 h**	**2400(gpu)/512(mc)**	**Standard queue for production work**
prepost	30 min	1	High priority pre/post processing
xfer	24h	1	Data transfer queue
total	2 h		CSCS maintenance queue (restricted use)

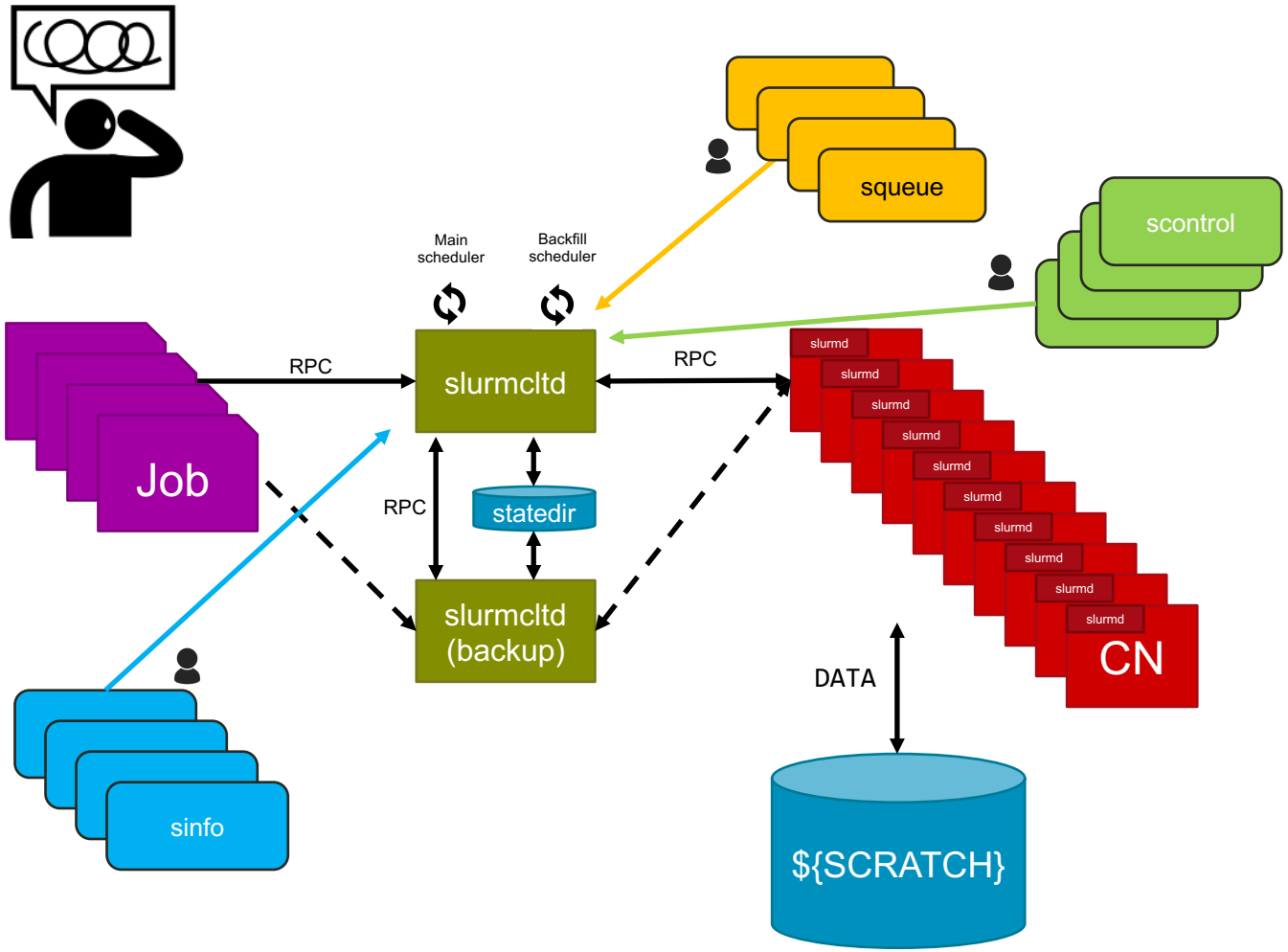
https://user.cscs.ch/access/running/piz_daint/#slurm-batch-queues

RPC: Remote Protocol Call

Slurm, how does it work?



Easy, right?

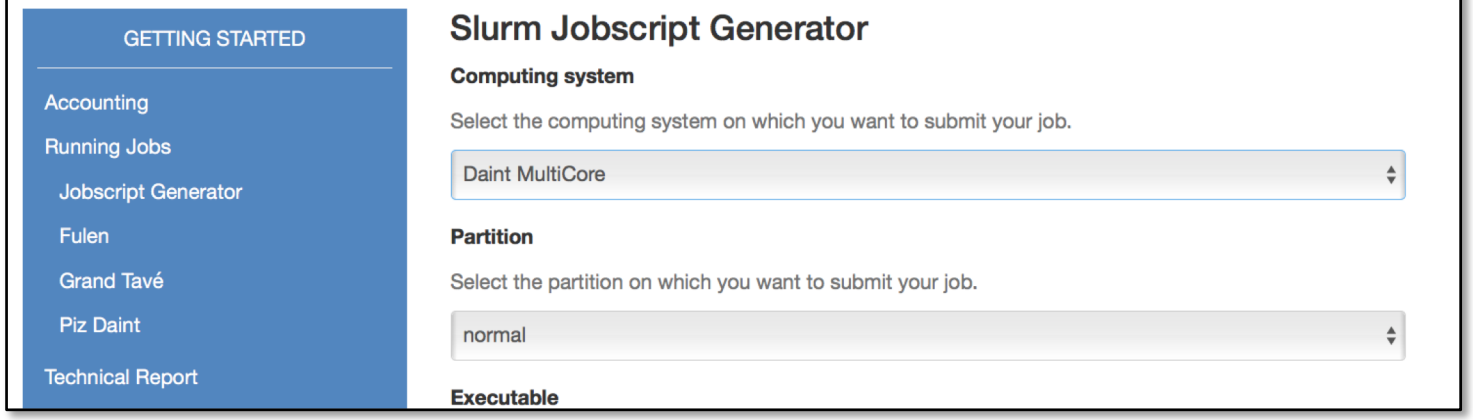


Not really, it's very complex, the end user just doesn't see it!

Good practices when submitting jobs (1)

Use our Slurm Jobscript Generator

https://user.cscs.ch/access/running/jobscript_generator/



Good practices when submitting jobs (2)

Specify accurate wall time

```
#!/bin/bash -l
#SBATCH --nodes=120
#SBATCH --time=0:30:00
#SBATCH --partition=normal
#SBATCH --constraint=gpu
[...]
```

For jobs with *many* tasks, use greasy

```
#!/bin/bash -l
#SBATCH --nodes=120
#SBATCH --time=0:30:00
#SBATCH --partition=normal
#SBATCH --constraint=gpu

module load daint-mc
module load GREASY/2.1-cscs-CrayGNU-17.08
greasy -f greasy_tasks.list
```

Run jobs off ${SCRATCH}

```
#!/bin/bash -l
#SBATCH --nodes=120
#SBATCH --time=0:30:00
#SBATCH --partition=normal
#SBATCH --constraint=gpu

cd ${SCRATCH}
srun ${SCRATCH}/my_binary
```

Make sure your sruns work! (or sleep a little bit between sruns)

```
#!/bin/bash -l
#SBATCH --nodes=120
#SBATCH --time=0:30:00
#SBATCH --partition=normal
#SBATCH --constraint=gpu
function p() {
rt=$?;
if [[ ${rt} -ne 0 ]]; then
  sleep 2
fi
return ${rt}
}
srun mytask ; p
srun mytask2 ; p
```

Good practices when submitting jobs (3)

Use dependencies to link jobs

```
#!/bin/bash -l
#SBATCH --nodes=120
#SBATCH --time=0:30:00
#SBATCH --partition=normal
#SBATCH --constraint=gpu
#SBATCH --dependency=afterok:91999
srun ${SCRATCH}/my_binary
```

Job arrays are awesome!

```
#!/bin/bash -l
#SBATCH --nodes=120
#SBATCH --time=0:30:00
#SBATCH --partition=normal
#SBATCH --constraint=gpu
#SBATCH –array=1-10%2

srun ${SCRATCH}/my_binary $SLURM_ARRAY_TASK_ID
```

Pack nodes by choosing group/cabinet/row

```
#!/bin/bash -l
#SBATCH --nodes=12
#SBATCH --time=0:30:00
#SBATCH --partition=normal
#SBATCH --constraint=“mc&group1”
[...]
```

```
$ scontrol show no |grep 'AvailableFeatures' |sort|uniq
   AvailableFeatures=gpu,row1,c0-1,group5,startx,gpumodedefault,perf
   AvailableFeatures=gpu,row1,c1-1,group5,startx,gpumodedefault,perf
   AvailableFeatures=gpu,row1,c2-1,group6,startx,gpumodedefault,perf
[...]
    AvailableFeatures=mc,row0,c5-0,group2,perf
   AvailableFeatures=mc,row0,c6-0,group3,128gb,perf
[...]
```

Use high memory MC nodes

```
#!/bin/bash -l
#SBATCH --nodes=12
#SBATCH --time=0:30:00
#SBATCH --partition=normal
#SBATCH --constraint=“mc&128g”
[...]
```

What not to do when submitting jobs

Jobs that submit other jobs/tasks in loops

```
#!/bin/bash
#SBATCH ...
while :
do
srun sbatch my_job.sbatch
sleep 1
done
```

Jobs with *bajillions* of tasks

```
# sacct -j XXXXXX |wc -l
25337
```

Jobs that run off ${HOME}

```
#!/bin/bash -l
#SBATCH --nodes=120
#SBATCH --time=0:30:00
#SBATCH --partition=normal
#SBATCH --constraint=gpu

srun ~/my_binary
```

Jobs with hundreds of failing tasks (or many tasks in parallel)

```
#!/bin/bash -l
#SBATCH --nodes=3
#SBATCH --time=0:30:00
#SBATCH --partition=normal
#SBATCH --constraint=gpu

export CRAY_CUDA_MPS=1
cd $SLURM_SUBMIT_DIR

date
srun --nodes=1 --bcast=/tmp/${USER} --ntasks=1 --ntasks-per-node=1 --cpus-
per-task=12 tune_5x16x13_exe0 &
srun --nodes=1 --bcast=/tmp/${USER} --ntasks=1 --ntasks-per-node=1 --cpus-
per-task=12 tune_5x16x13_exe1 &
srun --nodes=1 --bcast=/tmp/${USER} --ntasks=1 --ntasks-per-node=1 --cpus-
per-task=12 tune_5x16x13_exe10 &
[...]
sleep 29m
```

What not to do on login nodes

Some loops are evil!

```
#!/bin/bash
while :
do
clear
squeue | grep JOBID
squeue | grep ${USER}
sleep 1
done
```

Other loops are even more evil!

```
#!/bin/bash
for i in ${var}; do
sbatch my_job.sbatch
done
```

GNU make without number of tasks

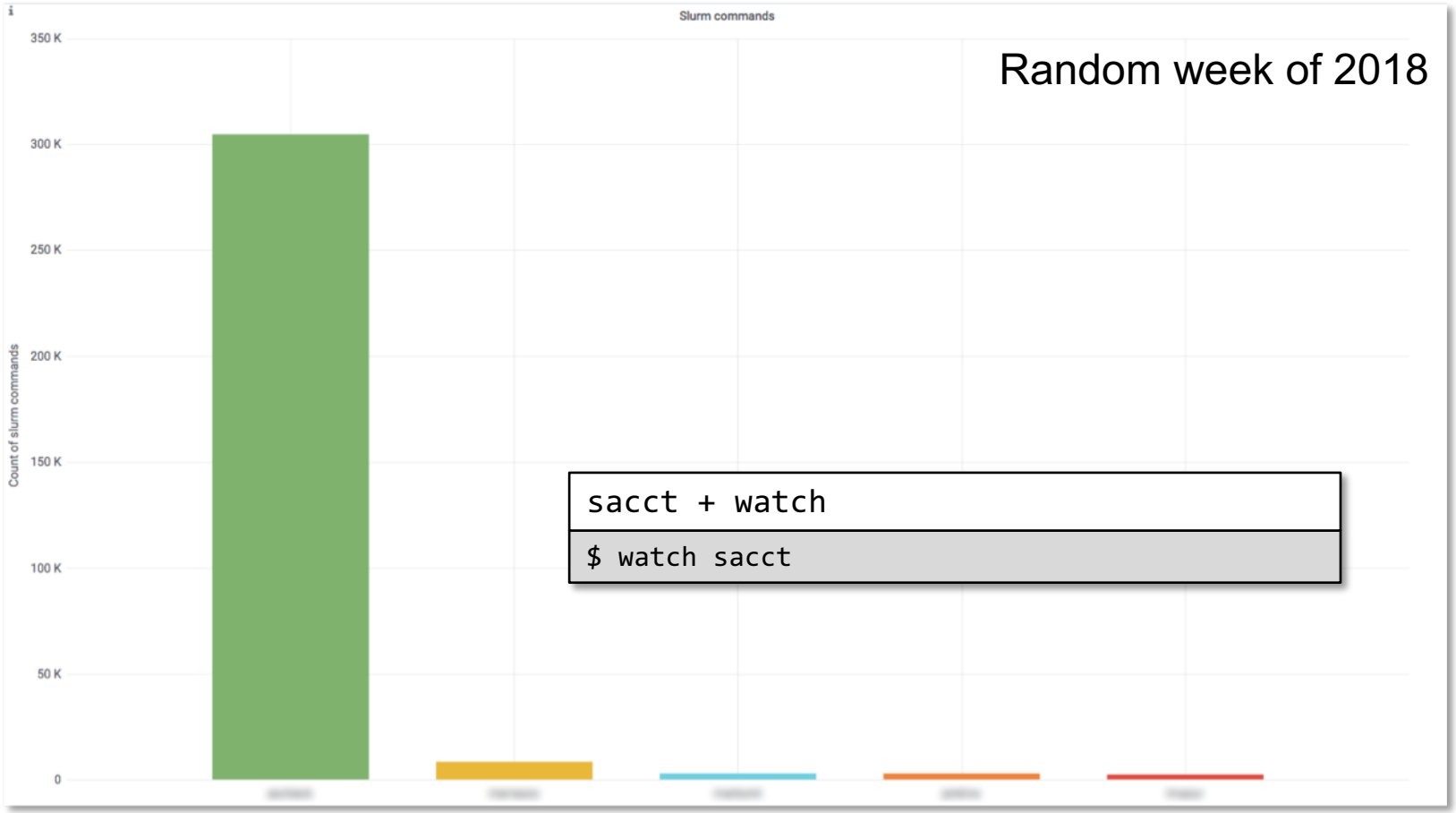
```
$ make -j
```

Watch is not a friend of RPCs

```
$ watch squeue -u ${USER}
```

squeue without filtering (+watch)!

```
$ watch squeue
```



sacct + watch

```
$ watch sacct
```

Job allocations

- CSCS has 3-month allocation periods

- If you want to fully utilize your allocated node hours, it's better to have a constant stream of jobs rather than packing all the jobs at the end of the allocation period

- However, if you've run out of budget, remember that the partition *low* can still be used... If you don't mind waiting ☺

Job Priority

```
$ sprio -w
JOBID PARTITION   PRIORITY       AGE  FAIRSHARE  PARTITION        QOS
Weights                         172800     259200     172800     500000
```

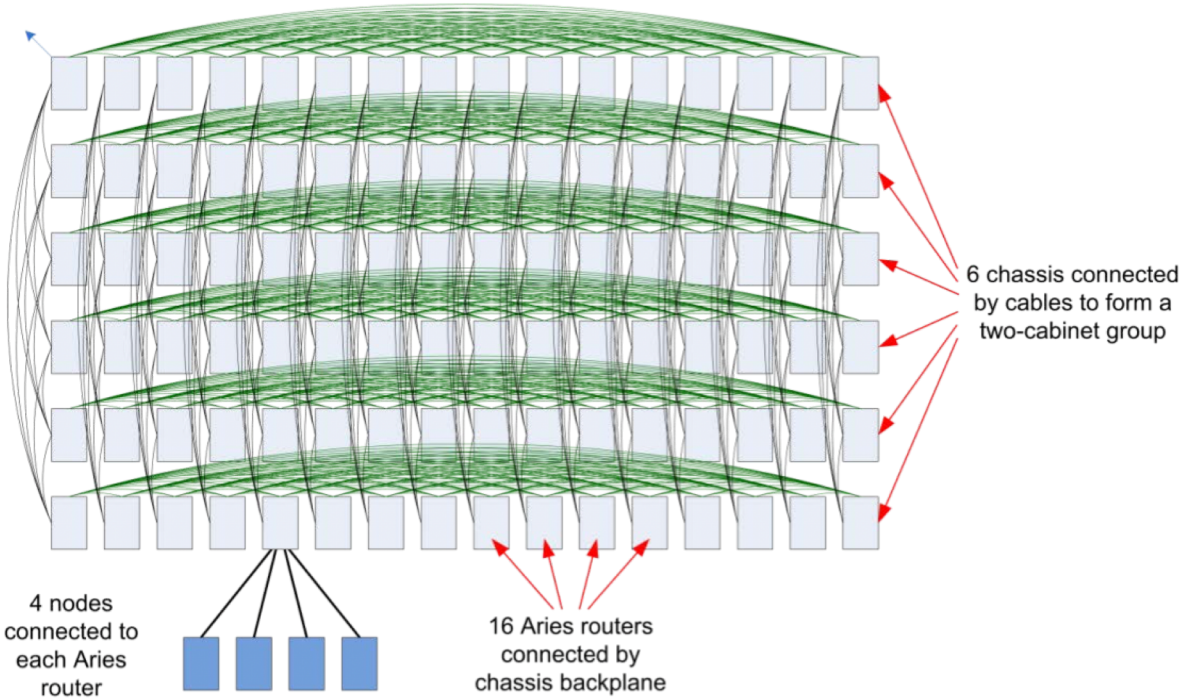
```
$ sbucheck

Per-project usage at CSCS since Sun 1 July, 12am CEST
Allocation period ending Sun 30 September, 11pm CEST

* csstaff:   Authorized Daint constraints: gpu, mc
           DAINT Usage:      ------ NODE HOURS (NH)   Quota:    ∞ NH     0.0%
             DOM Usage:           0 NODE HOURS (NH)   Quota:    ∞ NH     0.0%
           LEONE Usage:           0 NODE HOURS (NH)   Quota:    ∞ NH     0.0%
            TAVE Usage:       ----- NODE HOURS (NH)   Quota:    ∞ NH     0.0%
```

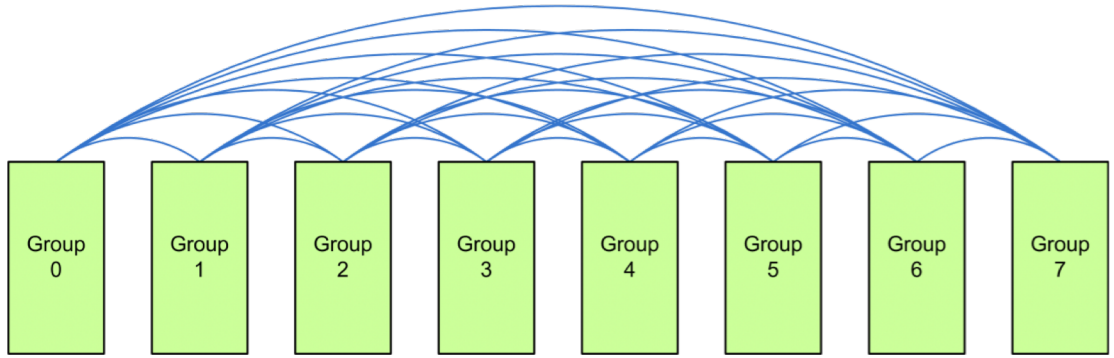
- Job priority is based on partition,
- If your job has been in the queue
- Even if you still have lots of hours left, there may be other users/accounts with less usage and/or more hours allocated
- So if your jobs are waiting *for too long*, just wait more! ☺
- Also, make sure there are no reservations in the system (maintenances, large runs, etc.)

```
$ scontrol show reservations
```

Considerations about the network

- Dragonfly network topology
- A group consists of 2 cabinets, 384 nodes
- In a group, all2all links
- A job requiring more nodes needs to go thru optical uplinks
- Performance when crossing groups might be reduced if links are heavily used (heavy I/O, heavy usage of MPI, etc.)
- Remember: this is a **shared** machine, so network performance varies
- Use constraints to request specific parts of the system if your workload requires it!





Data transfer

Why a Data Transfer Service?

The need to move in/out

Big Datasets (TB)

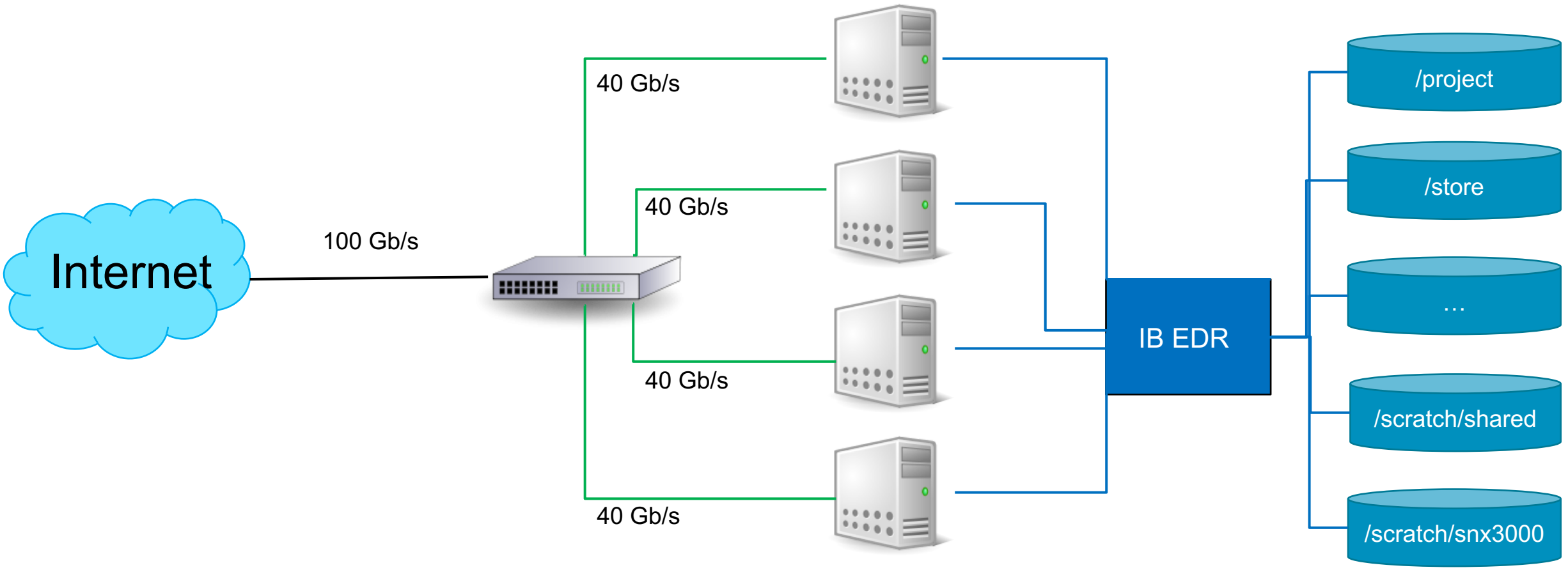
to/from CSCS with

high throughput (>1 GB/s)

Data Transfer Service

- 4 nodes:
  - Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz Processor (Sandy Bridge)
  - 128GB Memory
  - 1 Port EDR Infiniband
  - 1 Port 40 Gb/s Ethernet
- **GridFTP** for transfer in/out CSCS
- **Slurm queue (xfer)** for internal transfer
- Bandwidth:
  - 4x40 Gbit/s (internal transfer),
  - 100 Gbit/s (external transfer)
- File systems: /store, /project, /scratch/snx3000, …

Data Transfer Infrastructure



Outside CSCS transfer

GridFTP:

- Easy setup
- Very good performance
- Large base install
- Command line interface
- Compatible with Globus Online

What is/why GridFTP

- FTP version enhanced by the Globus project
- Secure: control channel encrypted with SSH keys
- High performance: multi-stream transfers, TCP buffers, block size …
- Nice features like continuation of partial transfers, 3rd party control
- Wide install base
- Easy setup

SLURM queue for Data transfer (xfer)

- Limits and Quotas on /scratch:
  - /scratch is not a FS where to store long term files
  - /scratch has no quota but if you have more than 100K files you cannot submit jobs on Piz Daint
- The ideal workflow on a chain of Slurm jobs:
  1. Slurm data transfer job to move data into /scratch 
  2. Slurm computing job on Piz Daint 
  3. Slurm data transfer job to move data out of /scratch 
- Why we do not charge and have a different queue for data transfer jobs?
  - ✓ We want to help our users to get the best of CSCS performance and optimization reducing human action when possible.

Last remarks about Data Mover Service

Don’t move trillions of tiny little files

A good practice is to use libraries such as HDF5 or similar to create large files in order to optimize data transfers

Visit our User portal for more details about CSCS services:

https://user.cscs.ch/storage/transfer/

Pre- & Post-processing

Pre- & Post-processing – Ideas and usage

- Historically created to serve some specific disciplines
  - On preparing input files (e.g. fast system minimization in molecular dynamics)
  - Post processing results (e.g. large trajectory to process)
- Resources are charged in the same way as in the normal partition
- Majority of the cases, we see users doing something similar to
  - `srun python script.py`
  - Where `script.py` is a single core python application
- SLURM supports python submission scripts
  - The prepost queue is a good place to use this type of submission scripts
  - The concept can be applied in any queue
  - Examples shown here are for Python > 3.5
    - Make sure to `module load cray-python/3.x.x.x.x` before submitting the jobs

Pre- & Post-processing – Ideas and usage

- Historically created to serve sc
  - On preparing input files (e.g. fast
  - Post processing results (e.g. larg
- Resources are charged in the
- Majority of the cases, we see u
  - `srun python script.py`
  - Where `script.py` is a single c
- SLURM supports python submission scripts
  - The prepost queue is a good place to use this type of submission scripts
  - The concept can be applied in any queue
  - Examples shown here are for Python > 3.5
    - Make sure to `module load cray-python/3.x.x.x.x` before submitting the jobs

BASH jobscript for python

```
#!/bin/bash –l
#SBATCH --job-name=python_test
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-core=1
#SBATCH --ntasks-per-node=12
#SBATCH --cpus-per-task=3
#SBATCH --partition=prepost
#SBATCH --constraint=mc

module load cray-python/3.6.1.1
python script.py
```

Pre- & Post-processing – Ideas and usage

- Historically created to serve sc
  - On preparing input files (e.g. fast
  - Post processing results (e.g. larg
- Resources are charged in the
- Majority of the cases, we see u
  - `srun python script.py`
  - Where `script.py` is a single c
- SLURM supports python submission scripts
  - The prepost queue is a good place to use this type of submission scripts
  - The concept can be applied in any queue
  - Examples shown here are for Python > 3.5
    - Make sure to `module load cray-python/3.x.x.x.x` before submitting the jobs

Python jobscript

```
#!/usr/bin/env python3
#SBATCH --job-name=python_test
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-core=1
#SBATCH --ntasks-per-node=12
#SBATCH --cpus-per-task=3
#SBATCH --partition=prepost
#SBATCH --constraint=mc

import ...
# write your python code here!
print("Hello, world!")
```

Pre- & Post-processing – Python jobscripts

Python jobscript

```
#!/usr/bin/env python3
#SBATCH --job-name=python_test
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-core=1
#SBATCH --ntasks-per-node=12
#SBATCH --cpus-per-task=3
#SBATCH --partition=prepost
#SBATCH --constraint=mc

import os
# write your python code here!
print("Hello, world!")
```

Instead of `#!/bin/bash -l`

`SBATCH` options are the same

Write directly the python code

- We encourage Python 3, version greater than 3.5
- The output should be the same for `srun python script.py`, if `script.py` is NOT an MPI4Py application
- There is no additional shell language to distract (focus just on Python)

Pre- & Post-processing – Python jobscripts

Python jobscript

```
#!/usr/bin/env python3
#SBATCH --job-name=python_test
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-core=1
#SBATCH --ntasks-per-node=12
#SBATCH --cpus-per-task=3
#SBATCH --partition=prepost
#SBATCH --constraint=mc

import os
# write your python code here!
print("Hello, world!")
```

Job output

```
Hello, world!
```

Pre- & Post-processing – Python jobscripts - highlights

BASH jobscript for python

```
#!/bin/bash –l
#SBATCH --job-name=python_test
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-core=1
#SBATCH --ntasks-per-node=12
#SBATCH --cpus-per-task=3
#SBATCH --partition=prepost
#SBATCH --constraint=mc

module load cray-python/3.6.1.1
python script.py
```

Instead of `#!/usr/bin/env python3`

python environment CAN be loaded inside jobscript

python environment MUST be loaded outside jobscript

- If you do not change PrgEnvs frequently, you could consider loading the environment thru `$HOME/.modulerc`

Python jobscript

```
#!/usr/bin/env python3
#SBATCH --job-name=python_test
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-core=1
#SBATCH --ntasks-per-node=12
#SBATCH --cpus-per-task=3
#SBATCH --partition=prepost
#SBATCH --constraint=mc

import os
# write your python code here!
print("Hello, world!")
```

Pre- & Post-processing – Python jobscripts - highlights

BASH jobscript for python

```
#!/bin/bash –l
#SBATCH --job-name=python_test
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-core=1
#SBATCH --ntasks-per-node=12
#SBATCH --cpus-per-task=3
#SBATCH --partition=prepost
#SBATCH --constraint=mc

module load cray-python/3.6.1.1
python script.py
```

SBATCH options are the same

Python jobscript

```
#!/usr/bin/env python3
#SBATCH --job-name=python_test
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-core=1
#SBATCH --ntasks-per-node=12
#SBATCH --cpus-per-task=3
#SBATCH --partition=prepost
#SBATCH --constraint=mc

import os
# write your python code here!
print("Hello, world!")
```

Pre- & Post-processing – Python jobscripts - highlights

BASH jobscript for python

```
#!/bin/bash –l
#SBATCH --job-name=python_test
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-core=1
#SBATCH --ntasks-per-node=12
#SBATCH --cpus-per-task=3
#SBATCH --partition=prepost
#SBATCH --constraint=mc

module load cray-python/3.6.1.1
python script.py
```

Contains the actual python code

Python jobscript

```
#!/usr/bin/env python3
#SBATCH --job-name=python_test
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-core=1
#SBATCH --ntasks-per-node=12
#SBATCH --cpus-per-task=3
#SBATCH --partition=prepost
#SBATCH --constraint=mc
```

```
import os
# write your python code here!
print("Hello, world!")
```

Contents of `script.py`

Pre- & Post-processing – Python jobscripts

Read SLURM variables

```
#!/usr/bin/env python3
#SBATCH --job-name=python_test
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-core=1
#SBATCH --ntasks-per-node=12
#SBATCH --cpus-per-task=3
#SBATCH --partition=prepost
#SBATCH --constraint=mc

import os
# you can capture any os variables
print(os.environ.get('HOME', None))
# even SLURM ones
print(os.environ.get('SLURM_JOB_CPUS_PER_NODE', None))
print(os.environ.get('SLURM_SUBMIT_HOST', None))
print(os.environ.get('SLURM_SUBMIT_DIR', None))

# avoid capturing SLURM_STEP_* variables
# they are only valid inside srun
#print(os.environ.get('SLURM_STEP_NUM_NODES', None))
```

- Check "**OUTPUT ENVIRONMENT VARIABLES**" from **man sbatch** for a complete list of variables

Pre- & Post-processing – Python jobscripts

Run applications with srun using full path

```
#!/usr/bin/env python3
#SBATCH ...

import os
import subprocess

srun_cmd = ['srun', '/scratch/snx3000/username/my_time_consuming_app.x']
proc = subprocess.Popen(args=srun_cmd, # your srun command here
                        stdout=subprocess.PIPE,
                        stderr=subprocess.PIPE,
                        universal_newlines=True)

# you can overlap other python computations here
# do other stuff

# wait until srun finishes
proc.wait()

if proc.returncode != 0:
    print('proc failed')
else:
    print('proc has passed')
```

- You can programmatically control asynchronous execution of parallel applications
- Full application path is a disadvantage of using a system call inside Python
- Can still use `srun` for running scientific applications or heavy loaded processes

Pre- & Post-processing – Python jobscripts

Use **start_new_session=True** for tasks that spawns processes

```
#!/usr/bin/env python3
#SBATCH ...

import os
import subprocess

srun_cmd = ['srun', '/scratch/snx3000/username/my_time_consuming_app.sh']
proc = subprocess.Popen(args=srun_cmd, # your srun command here
                        stdout=subprocess.PIPE,
                        stderr=subprocess.PIPE,
                        universal_newlines=True,
                        start_new_session=True)
# you can overlap other python computations here
# wait until srun finishes
proc.wait()

if proc.returncode != 0:
    print('proc failed')
else:
    print('proc has passed')
```

- You can programmatically control asynchronous execution of shell scripts
- If the shell script spawns child process, use the `start_new_session` argument option
- Good practice is to always use this option

Pre- & Post-processing – Python jobscripts

Run module commands

```
#!/usr/bin/env python3
#SBATCH --job-name=python_test
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-core=1
#SBATCH --ntasks-per-node=12
#SBATCH --cpus-per-task=3
#SBATCH --partition=prepost
#SBATCH --constraint=mc

import os
import subprocess

try:
  module_cmd = ['modulecmd', 'python', 'load', 'daint-gpu']
  proc = subprocess.run(args=module_cmd, # your module command here
                        stdout=subprocess.PIPE, # use PIPE for Python3
                        stderr=subprocess.PIPE, # use PIPE for Python3
                        universal_newlines=True) # Open stds as text
except OSError as e:
  print('module error:', e)
```

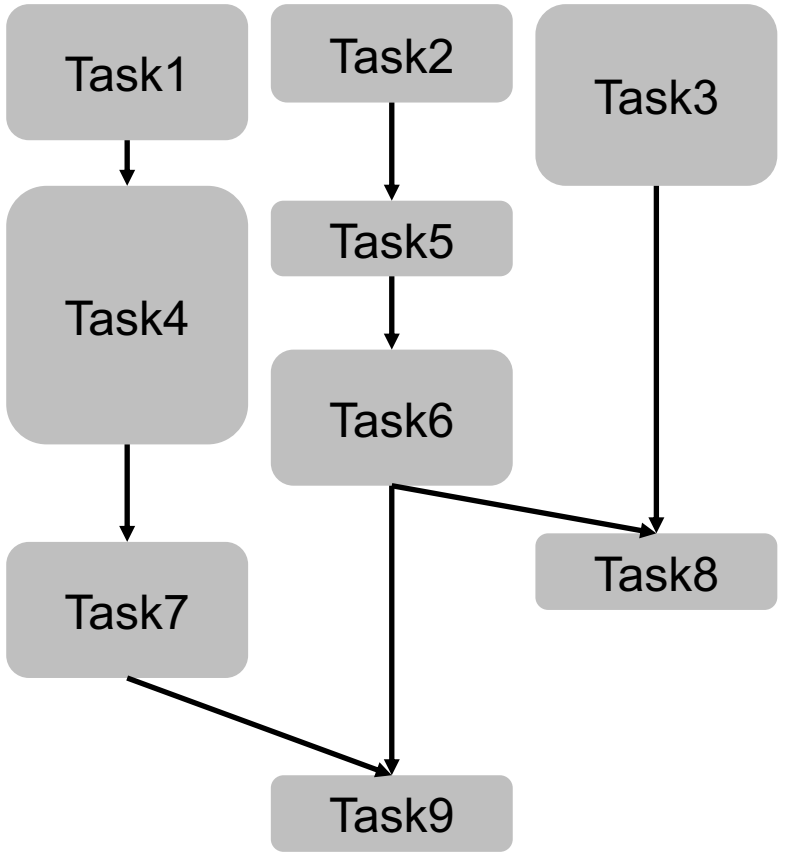
- Make use of environment modules
- Use synchronous executions
- `subprocess.run` is valid only for Python > 3.5

Do not forget to load cray-python with version > 3.5

```
Traceback (most recent call last):
  File "/var/spool/slurmd/job9499780/slurm_script", line 16, in <module>
    proc = subprocess.run(args=module_cmd, # your module command here
AttributeError: 'module' object has no attribute 'run'
```

Pre- & Post-processing – Complex Workflows

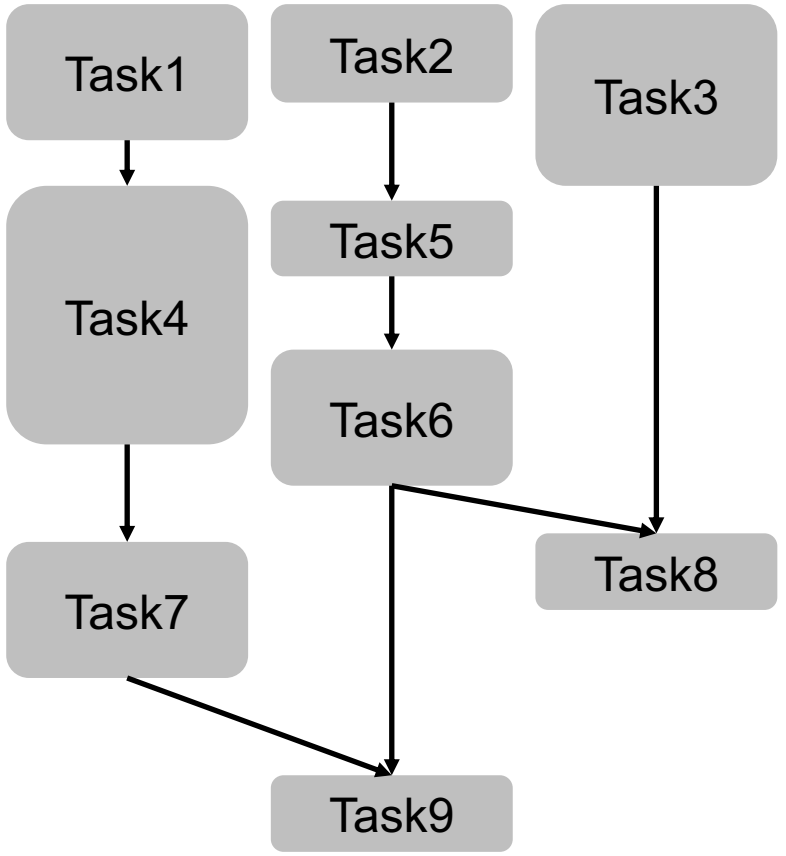
- Embarrassingly parallel problem
  - Controlling the execution of each individual process is a tedious task and it has already been solved several times
- Your workflow is a complex dependency graph
  - Efficient implementation may take a while in Python
- Efficient consumption of resources is a requirement
  - No core should be idle
  - Few jobs submitted to the queue
- You may not know or may not like Python
- You may simply do not want to implement anything



Welcome to meta schedulers

Pre- & Post-processing – GREASY

- Meta schedulers are schedulers that typically run inside other schedulers (in our case SLURM)
- There are several implementations of meta schedulers. We support GREASY on Piz Daint
- GREASY build to manage high throughput simulations.
- Aims to simplify the execution of embarrassingly parallel simulations in any environment.
- Developed by the Barcelona SuperComputing Center

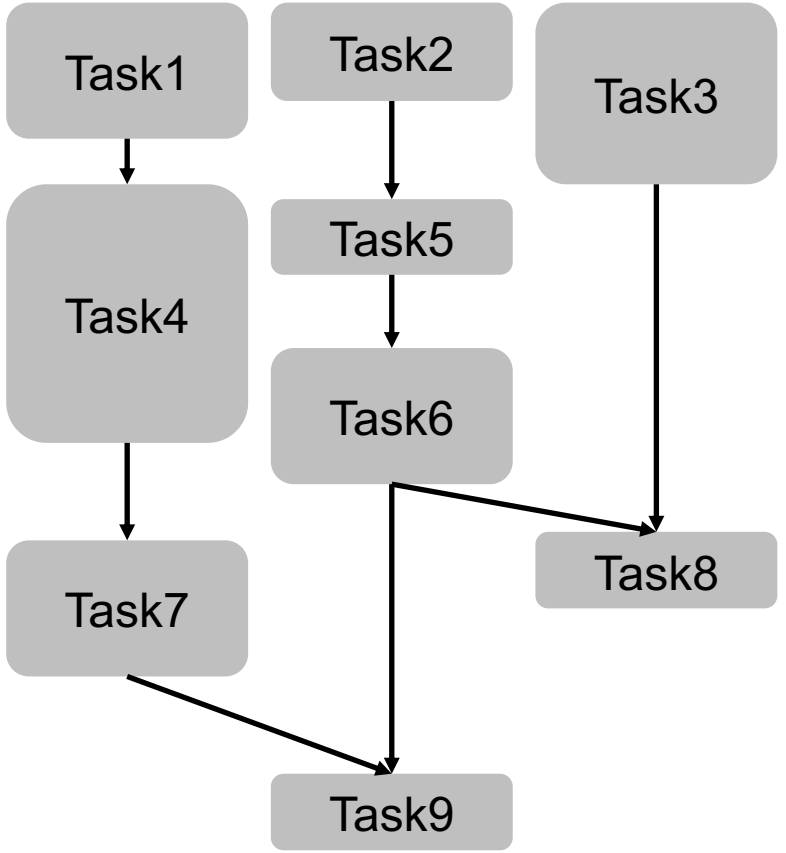


https://user.cscs.ch/tools/high_throughput/

Pre- & Post-processing – GREASY

Run GREASY

```
#!/bin/bash -l
#SBATCH --job-name=greasy_test
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-core=1
#SBATCH --ntasks-per-node=12
#SBATCH --cpus-per-task=3
#SBATCH --partition=prepost
#SBATCH --constraint=mc

module load daint-mc
module load GREASY/2.1-cscs-CrayGNU-17.08
greasy -f tasks.txt
```



- Line numbers represent the task ids, and those are used for later reference
- Use the `-f` flag when using CSCS version

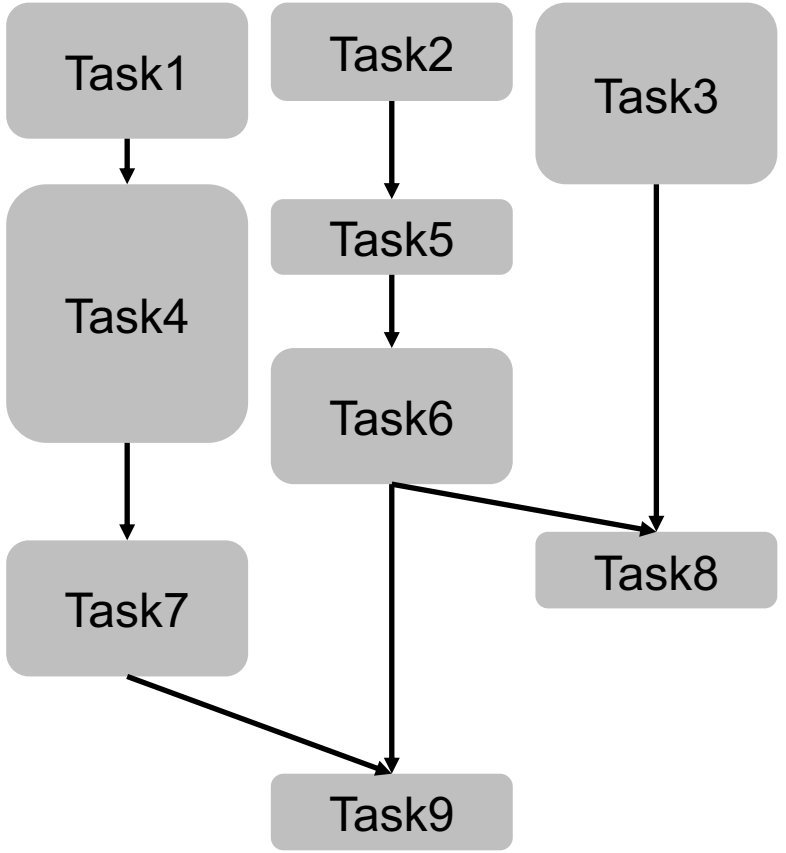
Tasks file

```
task1
task2
task3
[# 1 #] task4
[# 2 #] task5
[# 5 #] task6
[# 4 #] task7
[# 3, 6 #] task8
[# 6, 7 #] task9
```

Pre- & Post-processing – GREASY

Run serial version of GREASY

```
#!/bin/bash -l
#SBATCH --job-name=greasy_test
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-core=1
#SBATCH --ntasks-per-node=12
#SBATCH --cpus-per-task=3
#SBATCH --partition=prepost
#SBATCH --constraint=mc

module load daint-mc
module load GREASY/2.1-cscs-CrayGNU-17.08
greasy -f tasks.txt
```



- Line numbers represent the task ids, and those are used for later reference
- CSCS version has extended GREASY syntax (`[@ folder @]`). This allows the execution folder to be changed

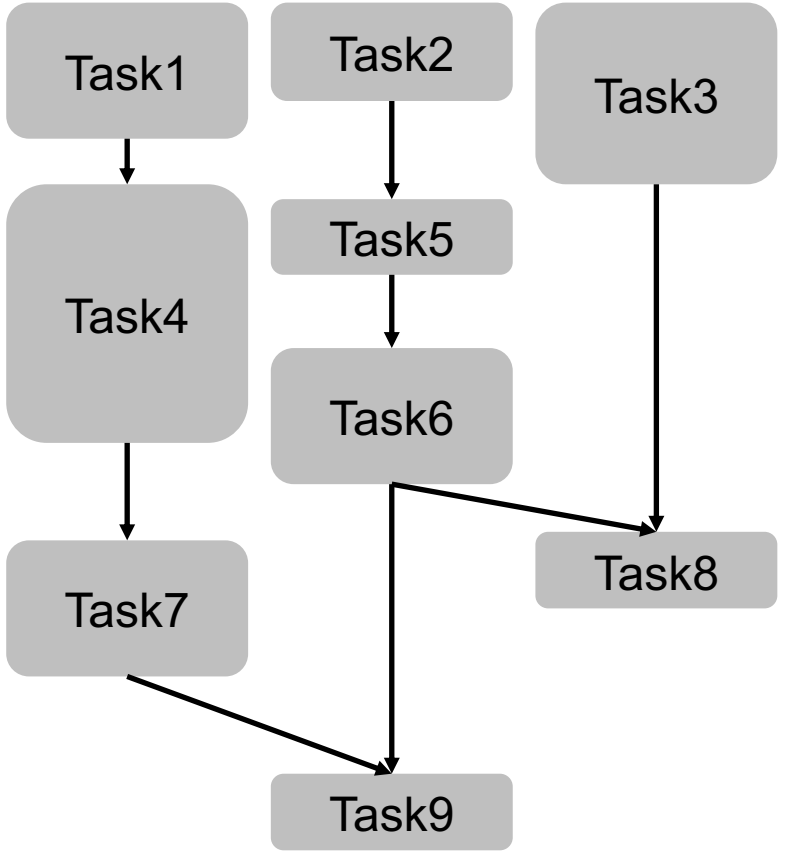
Tasks file

```
[@ /path/to/folder/to/execute/task1 @] task1
task2
[@ /path/to/folder/to/execute/task3 @] task3
[@ /path/to/folder/to/execute/task4 @] [# 1 #] task4
[# 2 #] [@ /path/to/folder/to/execute/task5 @] task5
[@ /path/to/folder/to/execute/task6 @][# 5 #] task6
[# 4 #] task7
[# 3, 6 #] task8
[# 6, 7 #] task9
```

Pre- & Post-processing – GREASY

Run MPI+OpenMP version of GREASY

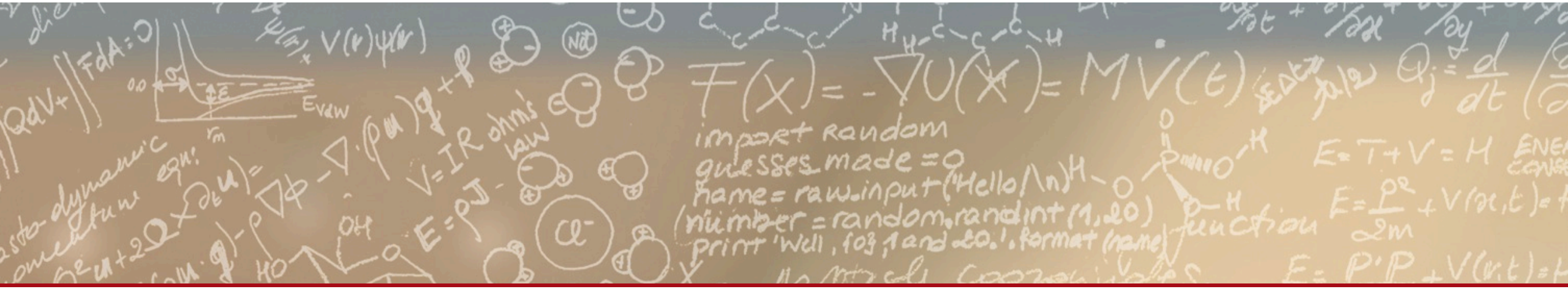
```
#!/bin/bash -l
#SBATCH --job-name=greasy_test
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-core=1
#SBATCH --ntasks-per-node=2
#SBATCH --cpus-per-task=6
#SBATCH --partition=prepost
#SBATCH --constraint=mc

module load daint-mc
module load GREASY/2.1-cscs-CrayGNU-17.08

export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
export CRAY_CUDA_MPS=1

greasy -f tasks.txt –n 2
```



- It is should run a single MPI task with 2 two MPI ranks per node

Tasks file

```
[@ /path/to/folder/to/execute/task1 @] task1
task2
[@ /path/to/folder/to/execute/task3 @] task3
[@ /path/to/folder/to/execute/task4 @] [# 1 #] task4
[# 2 #] [@ /path/to/folder/to/execute/task5 @] task5
[@ /path/to/folder/to/execute/task6 @][# 5 #] task6
[# 4 #] task7
[# 3, 6 #] task8
[# 6, 7 #] task9
```



Thank you for your attention.

Please remember, we are here to help!

help@cscs.ch