# Introduction to High Performance Computing (HPC) – Session 2

## using the **C**omputational **S**hared **F**acility (CSF)

Course materials / slides available from:
https://ri.itservices.manchester.ac.uk/course/rcsf/

Research Infrastructure Team, IT Services

its-ri-team@manchester.ac.uk

https://ri.itservices.manchester.ac.uk/csf3/

Recap slide skipped in training room

https://ri.itservices.manchester.ac.uk/course/rcsf/

# Housekeeping

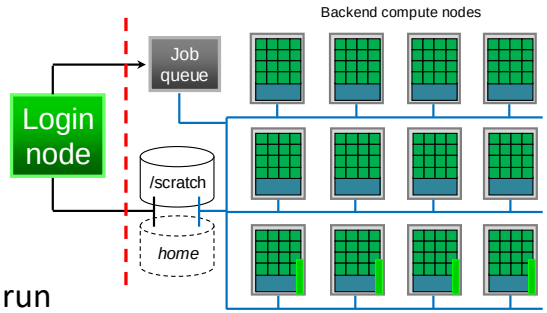
- Please let me know if you're leaving
  - Morning: Session one: 10am – 12:30pm (practicals 1, 2, & 3)
  - Afternoon: Session two: 1:30pm - 4pm (practicals 4 & 5)
- 1-to-1 help is available if needed during exercises. We'll describe how this works before the first one.
- Please give feedback on this course
  - Quick form at https://goo.gl/forms/zfZyTLw4DDaySnCF3 (choose "*Introduction to HPC (Using CSF)*")
  - Feedback is important to help us improve our courses
  - Records your attendance on the course



Recap slide skipped in training room

# Jobs, Jobscripts and the Batch System

- We want to do computational work - "jobs"




- You decide:
  - Which program(s) to run
  - Which directory to run from (within *scratch* :-) )
  - Which resources it needs (#cores, CPU type, memory)
- Write these requirements in a *jobscript*
- Submit your jobscript to the batch system (SGE)
- SGE decides exactly *when* and *where* the job runs



Recap slide skipped in training room

# A simple Jobscript – *Serial* (1 core)

`#!` on first line only (a special line)

First line indicates we use the *bash* script language to write our jobscript.

`myjob.txt`

`#$` indicates a **batch system parameter** to specify our job requirements. We'll use various combinations of these.

```
#!/bin/bash --login

#$ -cwd
#$ -N myjob
#$ -l resource

# Let's do work
date
hostname
sleep 120
date
```

`-cwd` indicates we'll run from our current (working) directory. Input / output files will usually be found here.

`-N` (optional). Set the *jobname*. Otherwise will use name of your jobscript as the name.

`-l` (optional) used to add extra resource requirements e.g. memory, time limits

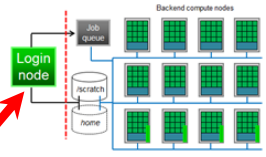
#$ -l course only works on the day of a course.

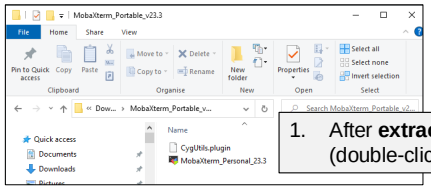
`#` lines are just comments - anything on the line after it will be ignored.

Actual Linux commands we run in our job. They will execute on a compute node.

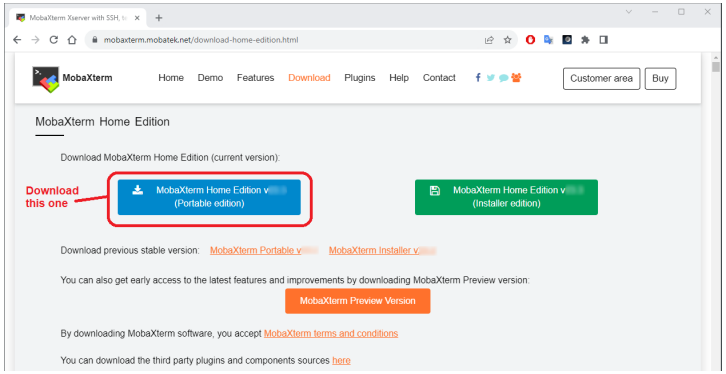
Recap slide skipped in training room

# Connect to CSF from Windows



- Access the CSF from a PC / laptop using an SSH (**S**ecure **Sh**ell) app
  - Sometimes called a "terminal".
  - There's no web-site or other fancy GUI on the CSF – use the "command-line".
- **Windows users** need to install a free *terminal* app called MobaXterm
- https://mobaxterm.mobatek.net/download-home-edition.html
  the Home edition (portable edition) does *not require* Administrator rights - just *extract* the small .zip file in your P-Drive or USB stick for example.




1. Download using the blue box.
2. Once downloaded, *right-click* on the .zip file and select:

   "Extract all …"

   This will *unpack* the .zip file to a folder.

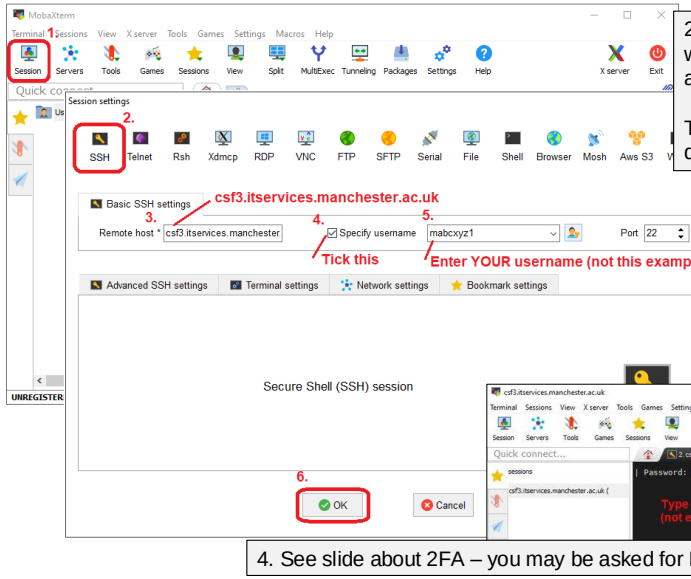


Recap slide skipped in training room

# MobaXterm "Session"

(username saved in the session setup)

1. After **extracting** the .zip file, start MobaXterm_Personal_*xy.z* (double-click on the icon)

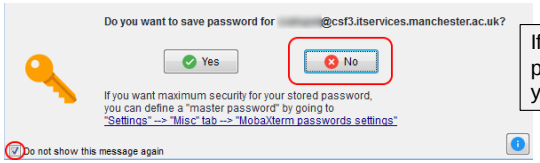



2 (1-6). Create a "Session" which saves the CSF's details along with *your* **username**.

This is needed to make file drag-n-drop work (see later.)

3. This will then start to log you into the CSF – it will ask for your password. Type carefully!

4. See slide about 2FA – you may be asked for DUO after your password




If asked to save your password, we recommend you say "No", for security.

Recap slide skipped in training room

Recap slide skipped in training room

# Connecting from Linux / Mac

- From MacOS using a *Terminal* window (after installing Xquartz)

  **`ssh -Y username@csf3.itservices.manchester.ac.uk`**

  UPPERcase Y

  Central IT Services username.
  Answer 'Yes' to continue *if* asked.
  Enter central IT password when asked (same as for email)

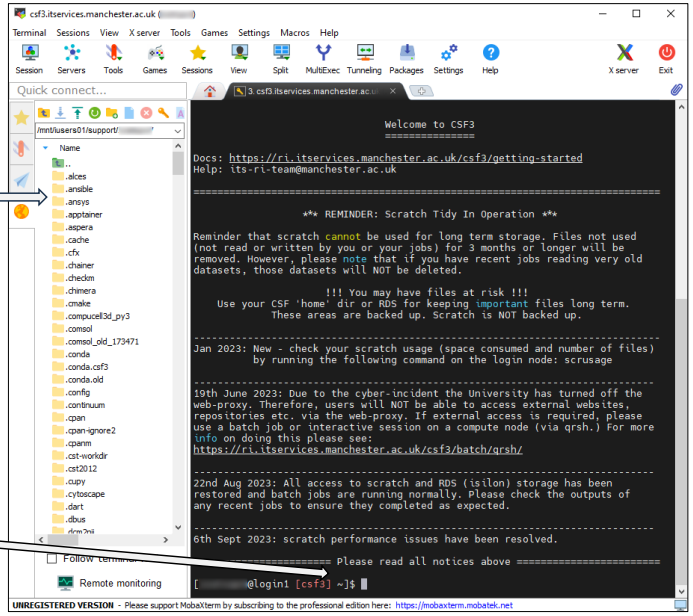
- From Linux using a Terminal window

  **`ssh -X username@csf3.itservices.manchester.ac.uk`**

  UPPERcase X

  Central IT Services username.
  Answer 'Yes' to continue *if* asked.
  Enter central IT password when asked (same as for email)

- Finished using CSF? Log out with: **`logout`** or **`exit`**




Drag-n-drop file browser for upload / download

(new users won't have as many items in the list!)

We're on (one of) the CSF login nodes. Any commands you use will be typed "at the prompt", which shows your username and current directory (folder.)

https://ri.itservices.manchester.ac.uk/course/rcsf/

https://ri.itservices.manchester.ac.uk/csf3

# ACCESSING APPLICATION S/W

Modules



# Access to Application Software

- Lots of different pieces of software installed
  - Many different applications
  - Different *versions* of an application
  - Need to ensure job knows where an app is installed
    - Try `echo $PATH` to see all directories the CSF will look in
- Use "*modules*" to set up *environment* for software
  - In your jobscript, add some `module` commands
  - Sets up all necessary *environment variables*
  - Apps use these *env vars* to get various settings
  - Can also run `module` commands on the login node (e.g., to check what apps are available)



# Module Commands

- `module avail` – lists all available modules
- `module search` *`keyword`* – lists all modules with *keyword* in their name
- `module list` – lists currently loaded modules
- `module load` *`modulename`* – loads module
- `module unload` *`modulename`* – unloads module
- `module purge` – unload all modules (hopefully)
- `man module` – man pages for the module command
- Examples:

```
module load apps/binapps/matlab/R2024b
module load apps/intel-19.1/amber/20-bf12-at21-bf12
module load apps/gcc/R/4.4.1
module unload apps/binapps/starccm/18.02-double
module help compilers/intel/19.1.2
module load tools/gcc/cmake/3.28.6
```

- See documentation for more info
  https://ri.itservices.manchester.ac.uk/csf3/software/modules/



# Modulefile settings

- What "settings" do modulefiles actually make?
  - Depends on the application (eg the installation instructions)
- Try the following commands on the login node:

```
which matlab
/usr/bin/which: no matlab in(/opt/site/sge.....

module load apps/binapps/matlab/R2024b

which matlab
/opt/apps/apps/binapps/matlab/R2024b/bin/matlab
```

- This shows that the modulefile made the matlab installation available.
- A job can do this to run that version of matlab.
- If interested, to see all of the settings that a modulefile will make:

```
module show apps/binapps/matlab/R2024b
```

But the idea is you don't need to know the settings - modulefiles take care of the details so you can concentrate on what your jobs actually do with the application.

- See documentation for more info
  https://ri.itservices.manchester.ac.uk/csf3/software/modules/

# Loading modulefiles: On login nodes OR in the jobscript

*Inherit* from the login node (**not recommended**)

**Extra flag needed** to *inherit* all settings from login node (settings are copied when job is *submitted*, not when it runs)

myjob.txt

```
#!/bin/bash
#$ -cwd
#$ -l resource
#$ -V   # Inherit login node env
        # (note: UPPERcase V)
        # Settings copied when
        # job is submitted

# Let's do some work
R CMD BATCH myscr.R
```

In the jobscript (**recommended!**)

Extra flag needed to load modulefiles in the jobscript

myjob.txt

```
#!/bin/bash --login
#$ -cwd
#$ -l resource


# Load module inside jobscript
module load apps/R/4.4.1

# Let's do some work
R CMD BATCH myscr.R
```

**On the CSF login node run the following commands**

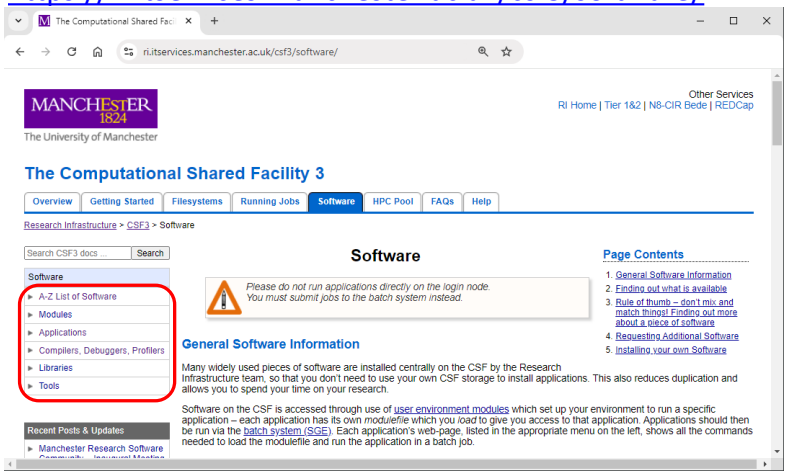
```
module load apps/R/4.4.1
qsub myjob.txt
```

```
qsub myjob.txt
```



# Which Modulefiles to Load

- How do I know which modulefile to load for a particular app?
  - https://ri.itservices.manchester.ac.uk/csf3/software/



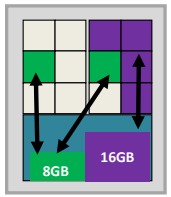



# PARALLEL COMPUTING

Background

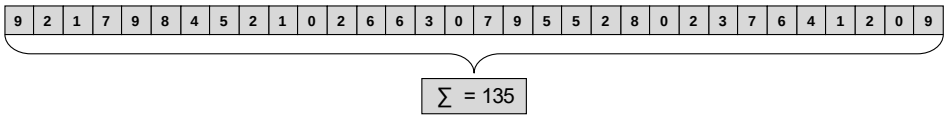
# Motivations for Parallel Computing

- CSF compute nodes have multiple CPU cores (28,32,168)
- Many apps can use multiple cores to speed up the computation
  - Split the "computation" over multiple CPU cores
    - Each core does a small(er) part of the computation, all in parallel
    - "Data parallelism" (same instructions run on each portion of "data")
  - May need to *combine* partial results together at end
  - Should get final result quicker
    - Ideally *N* cores giving results *N* times quicker
- Also provides access to more memory
  - Each core has access to ~4GB RAM (std nodes)
    - Ideally *M* cores for *M* times larger problem
- Both of the above!
- Another "parallel" method: High *Throughput* Computing
  - Multiple instances of an app with different params or data

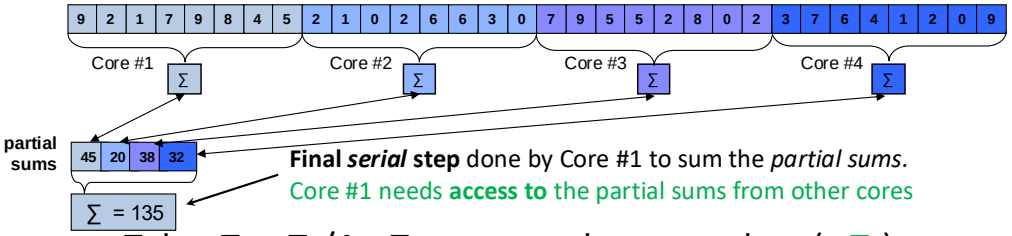


# Simple example: sum a list of numbers

- Could do this example manually with 4 volunteers
- 1-core: sum = sum + number$_i$ (for i = 1 to N)
  - Let's say it takes $T_1$ seconds to complete




- 4-cores: Each core sums a smaller list of numbers

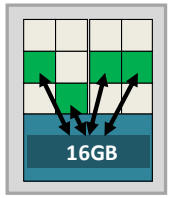



  - Takes $T_4 \approx T_1/4 + T_{serial}$ seconds to complete ($< T_1$)



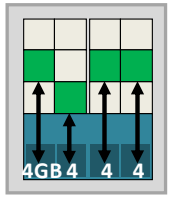
# Parallel Job Type #1 - single node

- A program runs on *multiple CPU cores* of **one** compute node
- Two common techniques used by apps:
  - Typically, one copy of the program runs
    - *"Shared memory"* (all cores see same memory)
    - Cores synchronize access to shared memory (data)
    - Look for "OpenMP" / "multi-threaded" / "Java threads" … in an application's docs



Shared Memory

  - Or coordinated copies of the program run, each communicating with each other
    - *"Distributed memory"* (each core has its own mem)
    - They communicate to share data, results
    - Look for "MPI" or "message passing" in the application's docs



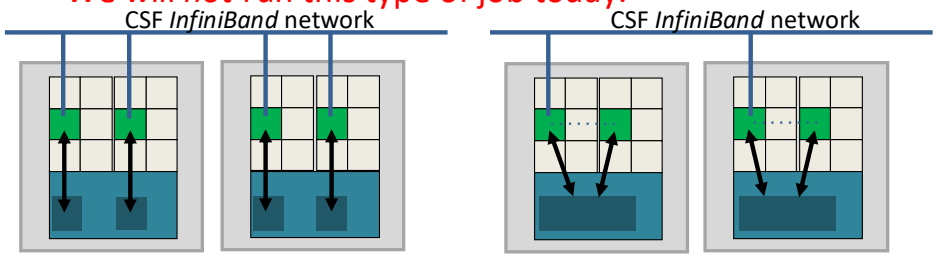
Distributed Memory

- **Your app must have been written to use one (or both) of the above parallel techniques!**
- We'll run this "single compute-node" type of job today



# Parallel Job Type #2 - multi-node

- Running a program over *several* compute nodes (and the many cores on those nodes)
  - Must be the "MPI" / "message passing" style of app (as before)
  - Uses *more* cores than in a single compute node
    - On CSF we require you to use *all* of the cores in *each* compute node!
  - They communicate to share data, results etc (as before)
    - Over the fast internal *InfiniBand* network
    - Possibly via shared memory as before, if on same compute node
- **Your app must have been written to support this!**
- We will *not* run this type of job today.




**Note: the diagrams only show a few cores in use for simplicity. On the CSF you must use *all* cores in *each* node.**

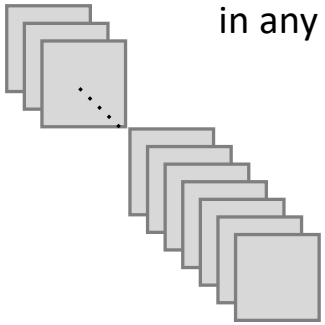


# Parallel Job Type #3 - High Throughput Computing (HTC)

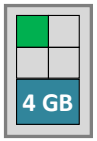
- Lots of *independent* computations. EG:
  - Processing lots of data files (e.g., image files)
  - Running the same simulation many times over with different parameters ("parameter sweeps")
- Run many copies of your program
  - Programs may be serial (single core) but running lots of them at once. They don't communicate.
- Easy to do on CSF. See also the UoM Condor Service (formerly the EPS Condor Pool)
  - Free resource, uses UoM idle desktops over night

# Example: Image Analysis

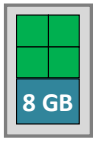
- High Throughput Computing
  - Not all s/w is "HPC" / parallel
  - But you might have *lots* of data
  - Each image takes 1hr to process (and are *independent* - process in any order)



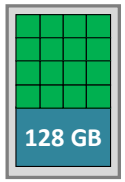
Example: 10,000 image scans to be analysed by an image processing application. Each image takes *1 hour* to process.



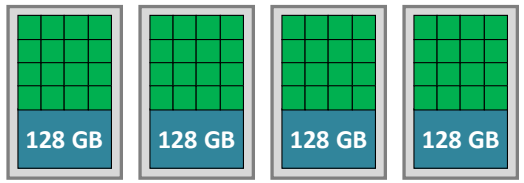

Laptop: 1 copy of software running.
**Over 1 year to complete!!**




Desktop: 4 cores, 4 copies of software running.
**~100 days to complete!**




Single HPC compute node: 16 copies of software running.
**~26 days to complete**




Multiple HPC compute nodes: 64 copies of software running.
**~6 days to complete**



# Which style of parallel job to use

- Mostly determined by the capability of your app
  - Is it serial (1-core) only? Is it multi-core (single-node) only? Is it multi-node capable?
- A serial app will only ever use 1 core
  - But run as an HTC job, you can still process lots of data in parallel
    - Use many cores, running multiple *independent* jobs (see later)
- Parallel app using only *shared memory*
  - "OpenMP", "multithreaded", "Java threads", "shared memory"
  - Can only use 1 compute node (2--32 Intel or 2--168 AMD cores)
- Parallel app using *distributed memory*
  - "MPI" (message passing interface), "distributed memory"
  - Can use many cores across multiple compute nodes
  - But consider: the network
    - Communication faster *within* same compute node
    - Communication slower on network *between* nodes
    - Apps may not speed up, the more cores (and nodes) you use (see later)



# Parallel Jobscript on CSF

- Use a jobscript to ask the batch system to find *N* free cores
  - While matching other requirements (memory, architecture, fast networking, GPU etc).

1. Add one extra line in jobscript to request:
   - *parallel environment* (multi-core or multi-node)
   - *and number of cores* to reserve
2. Inform your app how many cores to use
   - Remember, the jobscript says how many cores your job requires (the batch system will allocate those cores to your job.)
   - **But** *you* must still ensure your app uses no more!!
     - This is not automatic and how you do it varies from app to app



# Parallel Jobscript – Multi-core (*single-node*)

myparajob.txt

```
#!/bin/bash --login
#$ -cwd
#$ -pe smp.pe 4

# Set up to use a chemistry app
module load apps/intel-17.0/gromacs/2018.4/double

# Inform app how many cores to use
export OMP_NUM_THREADS=4

# This job runs "gromacs"
mdrun_d
```

#! and #$ see serial jobscript earlier.

**smp.pe** is the **p**arallel **e**nvironment name. This one means: app will use a single compute node (2 to 32 Intel cores.)

**-pe** indicates we'll run a parallel job in a particular **p**arallel **e**nvironment.

**4** is the number of **cores** we want to *reserve* in the system. Each **PE** has a maximum allowed.

**#** indicates line is a comment, so does nothing.

Any commands we run in our job. They will execute on a compute node that has required number of cores free. `mdrun_d` is Gromacs.

**Key concept!**

Must *somehow* inform the app how many cores we *reserved*. Must use the number **(4)** given on the **-pe** line. **Our app** wants `OMP_NUM_THREADS` environment variable setting.
**Your app** might use a different method!

# Avoid a common mistake

- Can use **$NSLOTS** for correct number of cores

  (Check: *your* app might not use `OMP_NUM_THREADS`)

```
#!/bin/bash --login
#$ -cwd
#$ -pe smp.pe 4       # Can be 2 to 32

# Set up to use "gromacs"
module load apps/intel-17.0/gromacs/2018.4/double

# Inform app how many cores to use
export OMP_NUM_THREADS=$NSLOTS

# This job runs "gromacs"
mdrun_d
```

**Our app** wants `OMP_NUM_THREADS` environment variable setting.
**Your app** might use a different method!

**$NSLOTS** is automatically set to the number, **4** in this case, given on **-pe** line. Will be **1** in a serial job (no **-pe** line).



# Parallel jobscript - Multi-core (cont…)

- That was a multicore (*single* compute node) example
- Using an app named Gromacs as an example

  https://ri.itservices.manchester.ac.uk/csf3/software/applications/gromacs/
- Requested a parallel environment (-pe) & 4 cores

  **$# -pe smp.pe 4**

  Will run the app on a single node (Intel CPUs), allocating multiple cores
  - smp.pe=symmetric multi-processor parallel environment
- Then informed the app to use 4 cores via **OMP_NUM_THREADS** environment variable (very common).
  - Special **$NSLOTS** variable always set to number of cores on PE line



# Parallel jobscript - Multi-core (cont…)

- As with the serial job submit it to the system with `qsub` and monitor with `qstat`
- It may take longer for *more* cores to become free in the system
- You'll get the usual output files
  - *jobname*`.o`*JobID* and *jobname*`.e`*JobID*



# New AMD nodes – October 2024

- New AMD EPYC "Genoa" nodes added Oct 2024. Up to 168 cores on a single node!

Only thing you need to change is the PE name: **amd.pe** and can increase number of cores used.

```
#!/bin/bash --login
#$ -cwd
#$ -pe amd.pe 4       # Can be 2 to 168

# Set up to use "gromacs"
module load apps/intel-17.0/gromacs/2018.4/double

# Inform app how many cores to use
export OMP_NUM_THREADS=$NSLOTS

# This job runs "gromacs"
mdrun_d
```

# Parallel jobscript - Multi-core (cont…)

- That was a multicore (*single* compute node) example
- Using an app named Gromacs as an example
  https://ri.itservices.manchester.ac.uk/csf3/software/applications/gromacs/
- Requested a parallel environment (-pe) & 4 cores

```
$# -pe amd.pe 4
```

  Will run the app on a single node (AMD CPUs), allocating multiple cores
  - amd.pe name is easy to remember!
- Then informed the app to use 4 cores via `OMP_NUM_THREADS` environment variable (very common).
  - Special `$NSLOTS` variable always set to number of cores on PE line



# Parallel Jobscript – multi-*node*

#! and #$ lines from serial jobscript earlier.

**hpc.pe** is the **p**arallel **e**nvironment name. This one means: app will use multiple compute nodes (all 32 cores **must** be used on each) and has fast InfiniBand networking between the nodes.

**128** is the total number of **cores** we want to *reserve* in the system. In this PE, 128 cores = 4 x 32-core nodes.

**-pe** indicates we'll run a parallel job in a particular **p**arallel **e**nvironment.

**-P** gives a *project code*, needed for this restricted PE.

```
#!/bin/bash --login
#$ -cwd
#$ -pe hpc.pe 128
#$ -P hpc-projectcode

# Set up to use MrBayes
module load apps/gcc/mrbayes/3.2.6

# App uses MPI to run across nodes
mpirun -n $NSLOTS pmb myinput.nex
```

**#** indicates line is a comment, so does nothing.

The commands we run in our job. They will execute on a compute node that has required number of cores free. `pmb` is the app name.

Must somehow inform the app how many cores we *reserved*. `$NSLOTS` is automatically set to number (**128**) given on **-pe** line. Our app is started via `mpirun` which has a `-n numcores` flag

# Parallel jobscript - Multi-node (cont…)

- A multi-node (but also multi-core) example
- Using an app named gulp as an example
  https://ri.itservices.manchester.ac.uk/csf3/software/applications/mrbayes/
- Requested a parallel environment (pe) & 128 cores

```
$# -pe hpc.pe 128
$# -P hpc-projectcode
```

  - Informed the app to use 128 cores via `mpirun -n $NSLOTS` (very common – lots of apps use this method.)
  - **`mpirun`** starts multiple copies of an MPI app on allocated nodes
  - Special `$NSLOTS` variable always set to number of cores on PE line
- Access to the "HPC Pool" requires an application form, completed by PI/Supervisors on a per-project basis
  - https://ri.itservices.manchester.ac.uk/csf3/hpc-pool/application-questions/



# Parallel Environments (PE)

https://ri.itservices.manchester.ac.uk/csf3/batch/parallel-jobs/

| PE Name | Description |
|---|---|
| **smp.pe** *N* | 2-32 cores, single compute node. ~4-5GB per core. Jobs will be placed on Intel "broadwell" (max 24 cores/job) or Skylake (max 32 cores/job) |
| `-l` *architecture* | **Ignore!** (`broadwell` or `skylake` or `cascadelake` or `icelake`) |
| `-l short` | 4GB/core "haswell" (**1 hour runtime limit**). For dev/test work. Max job size of 12 cores. |
| `-l mem512` | **32GB/core** Intel "haswell". Max job size of 16 cores. |
| `-l mem1500` | **46GB/core** Intel "skylake" or "cascadelake". Max 32 cores. |
| `-l mem2000` | **62GB/core** Intel "icelake". Max 32 cores. |
| `-l mem4000` | **125GB/core** Intel "icelake". Max 32 cores. **RESTRICTED ACCESS**. |

| PE Name | Description |
|---|---|
| **amd.pe** *N* | 2-168 cores, single compute node. 8GB per core. Jobs will be placed on AMD EPYC "Genoa" (max 168 cores/job) |
| `-l short` | **1 hour runtime limit**. For dev/test work. Max 28 cores. |

- **7-day runtime limit** on jobs unless otherwise indicated in table.
- Our simple jobscript did *not* use any of the above. Not needed in most cases.
- If you limit a job by *architecture* or memory it may wait longer in the queue.

# Choosing your Parallel Environment (PE)

- Choosing the PE is fairly simple, but:
  - Check the app's webpage for advice and examples
    https://ri.itservices.manchester.ac.uk/csf3/software
  - Check the PE page for limits on number of cores
    https://ri.itservices.manchester.ac.uk/csf3/batch/parallel-jobs
  - Only use `#$ -l resource` if necessary
- Use Intel (smp.pe) or AMD (amd.pe) nodes?
  - Most (all) apps will run on both, but AMD nodes are newer
  - The high memory nodes are all Intel CPUs (e.g., `-l mem2000`)
  - There are now *a lot* more AMD CPUs available than Intel CPUs
    - Submitting to **amd.pe** may result in shorter wait times
    - **amd.pe** nodes have 8GB/core (**smp.pe** std Intel have ~4-5GB/core)



# Parallel Software Performance

- You'll probably be running an app many times
- Worth small investigation to find optimal performance parameters (#cores & #nodes)
  - How many cores should I use?
- Do a few runs, vary the number of cores
  - Plot time versus num cores
  - Easy to do on CSF: remove PE setting from jobscript (and -N *name* if used), add PE to qsub command instead:

```
qsub -pe smp.pe 2 myjobscript.txt
qsub -pe smp.pe 4 myjobscript.txt
qsub -pe smp.pe 8 myjobscript.txt
```



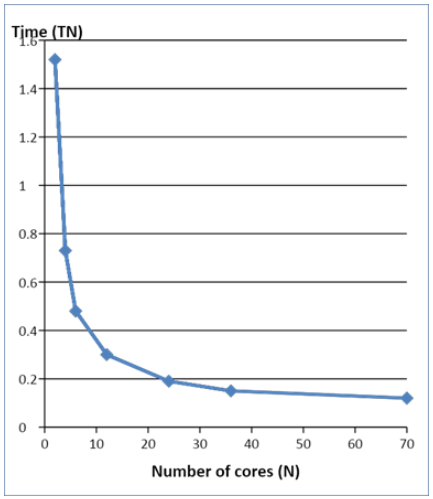
# To Assess Parallelism

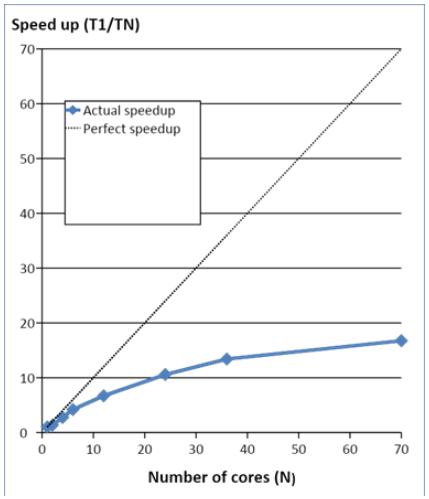
- Plot the following against "Number of Cores":
  - "Speed-up" or "Parallel Efficiency"
  - Total memory usage?
- Look for the sweet-spot
- Calculate: Speed-up = $T_1 / T_N$
  - Compare results against "ideal" scaling (where *N*-cores makes it go *N*-times faster)
- Calculate: Parallel Efficiency = $T_1 / (N \times T_N)$
  - N = number of cores, $T_N$ = time take on N cores
- Pick a typical problem size for your work



# Examples of Speed-up

- Data for popular Finite Element app on CSF
  - The 'Time' graphs shows it getting faster. But…



# Examples of Speed-up & Efficiency

- Example showing Speed-up and Efficiency values
  - App multiplies two square matrices
    - Measured a single multiplication of two 2000x2000 matrices

| No. cores | Time (Seconds) | Speed-up | Efficiency |
|---|---|---|---|
| 1 | 45.0 | 1x | 1.00 |
| 2 | 22.8 | 1.97x | 0.99 |
| 4 | 11.7 | 3.84x | 0.96 |
| 8 | 7.1 | 6.33x | 0.80 |

- The speed-up is reasonably close to "perfect" & efficiency is reasonably close to 100% but…
  - How will this scale as we go multi-node?
  - How will this scale as the problem size increases?
  - How will this scale on other hardware?



# PRACTICAL SESSION 4

Parallel job and scaling (no handout)



# Practical Session 4 (Intro)

- We will measure parallel efficiency for a similar matrix multiplication program
- But this time
  - Same problem size: 2000 x 2000 matrices
  - Repeats 5 times with additional maths ops on elements
  - (*sort of simulates an app solving equations*)
- Hardware reserved today
  - Intel 32-core compute nodes
  - We'll run multi-core (single node) jobs.



# Practical Session 4 (Intro)

- This is a distributed memory MPI program written in C
  - Already compiled: executable named `pmm.exe`
  - The `pmm_jobscript` can be edited as needed
- The jobscript for a parallel job must specify:
  - Parallel environment (where job runs on CSF)
  - Number of cores (2 or more)
- For today, use an Intel compute node (2-32 cores):
  - Shared memory parallel env: **`smp.pe`**

# Practical Session 4

- Inspect the jobscript
  - `cat pmm_jobscript`
  - Notice: initially it will use 2 cores (`-pe smp.pe 2`) and the job name, and hence output filenames, is "myjob" (`-N myjob`).
- Edit the jobscript (`gedit`) and change the job name (the `-N` line) to be "`pmm_2cores`"
- Submit the job to the batch system
  - `qsub pmm_jobscript`
- Immediately edit `pmm_jobscript` to change number of cores then resubmit (you *don't* need to wait for the previous job to run/finish)
  - Use 1, (2), 4, 8, 16, 32 cores.
  - Change the job name (EG: "`-N pmm_4cores`") to make `.o` and `.e` output filenames different (change the number of cores in the name - can't use `$NSLOTS` here sadly).
- The `pmm.exe` app times itself and reports how long it took to run, in its output:
  - Look in the `pmm_1,2,4,8,16,32cores.o`*`JobID`* files (use `cat`, `less`, or `gedit`)
  - Or, can always check the `ru_wallclock` (seconds) using `qacct -j` *`JobID`*
- Calculate the speed-up (or efficiency) for your runs – see slide 35 for the formulae.



# MULTIPLE SIMILAR JOBS

High Throughput Computing and "Job arrays"



# Multiple Runs of Same App

- We want to make many runs of an application to process *many different* input files
  - For example, on a desktop PC you might run

```
myapp.exe -in mydata.1.tif -out myresult.1.tif
(wait for it to finish)
myapp.exe -in mydata.2.tif -out myresult.2.tif
(wait for it to finish)
myapp.exe -in mydata.3.tif -out myresult.3.tif
…
myapp.exe -in mydata.1000.tif -out myresult.1000.tif
```

  - If it takes 5 minutes to process one file, it will take 1000 x 5 minutes to process them all (~3.5 days)



# How **Not** To Do It on the CSF (1)

- Inefficient method 1: one after another in *one* job? `qsub jobscript-all.txt`

jobscript-all.txt

```
#!/bin/bash --login
#$ -cwd

myapp.exe -in mydata.1.tif -out myresult.1.tif
(will wait for it to finish)
myapp.exe -in mydata.2.tif -out myresult.2.tif
(will wait for it to finish)
myapp.exe -in mydata.3.tif -out myresult.3.tif
…
myapp.exe -in mydata.1000.tif -out myresult.1000.tif
```

- This is *no better* than the desktop PC method

# How **Not** To Do It on the CSF (2)

- Inefficient method 2: lots of individual jobscripts?

jobscript1.txt

```
#!/bin/bash --login
#$ -cwd
myapp.exe -in mydata.1.tif -out myresult.1.tif
```

```
qsub jobscript1.txt
qsub jobscript2.txt
qsub jobscript3.txt
...
qsub jobscript1000.txt
```

**Make 1000 copies** of this jobscript, edit each one to process a different file (mydata.2.tif, ...)

Then **submit each job**

- Strains the batch system queue manager
- But, you will get many jobs running in parallel
  - EG: approx 100-200 jobs running at same time



# How To Do It - "Job Array" Jobscript

**-t** makes the jobscript automatically run a specified number of times. These are called **tasks**. Each is numbered uniquely 1,2,3....1000.

**1-1000** (start-end) says how many tasks to run and how they should be numbered. Note: **Cannot** start at **0**. Can use start-end:increment to increase the ID by more than 1.

arrayjob.txt

```
#!/bin/bash --login
#$ -cwd
#$ -t 1-1000

echo "I am task ${SGE_TASK_ID}"
myapp.exe -in mydata.${SGE_TASK_ID}.tif \
          -out myresult.${SGE_TASK_ID}.tif
```

The commands we run in our job. They will execute on backend nodes (different cores and nodes for different tasks).

`${SGE_TASK_ID}` is automatically set by the batch system and tells us which task we are (1,2,...). We can use this to do something different for each task.



# "Job Array" Jobscript

- Our app is serial (1-core) so no **`#$ -pe`** line
  - But you *could* add one if your app is multi-core
- The total number of tasks can be 100s, 1,000s, 10,000s (seen over 50,000 on CSF)
- The system will run *many* of the tasks in parallel
  - Usually 100s - "High-throughput Computing"
  - You get lots of work done sooner
  - It will eventually churn through all of them
  - They are started in numerical order but no guarantee they'll finish in that order!
- The extra jobscript **`#$ -t`** line is easy. Using the *task id* number *creatively* is the key to job arrays.



# The `$SGE_TASK_ID` variable (1)

- Want to do something different in each task. EG:
  - Read a different data file to process
  - Pass a different parameter to an application
- You can get this different "thing" in many ways:
  - EG: Use the $SGE_TASK_ID in filenames:

```
#$ -t 1-1000
imgapp -i image_${SGE_TASK_ID}.dat \
       -o image_${SGE_TASK_ID}.png
```


Task 1 reads `image_1.dat` writes `image_1.png`
Task 2 reads `image_2.dat` writes `image_2.png`
...
Task 1000 reads `image_1000.dat` writes `image_1000.png`

# The $SGE_TASK_ID variable (2)

- Or have a "master" list (a text file) of names etc
- The N[th] task reads the N[th] line from that text file:

```
#$ -t 1-4000
# Read the Nth line of filenamelist.txt and save in variable MYFILE
MYFILE=$(awk "NR==${SGE_TASK_ID} {print}" filenamelist.txt)

# Now use whatever the value of variable is in the next command
myapp.exe -input ${MYFILE} -output ${MYFILE}.out
```

**filenamelist.txt**

```
ptn1511.dat
ptn7235.dat
ptn7AFF.dat
ptn6E14.dat
ptn330D.dat
…
```

Task 1 reads `ptn1511.dat` writes `ptn1511.dat.out`
Task 2 reads `ptn7235.dat` writes `ptn7235.dat.out`
…

- Number of lines in file **must** match number of tasks
- To get number of lines in master file use:
  **`wc -l filenamelist.txt`**
- NB: VAR=$(*command arg1 arg2…*) captures output from *command* and assigned to variable VAR



# The $SGE_TASK_ID variable (3)

- Or have a "master" list (a text file) of names etc
- The N[th] task reads the N[th] line from that text file:

```
#$ -t 1-50
# Read the Nth line of dirnamelist.txt and save in variable SUBDIR
FOLDER=$(awk "NR==${SGE_TASK_ID} {print}" dirnamelist.txt)

# Now use whatever the value of variable is in the next command
cd ~/scratch/experiments/${FOLDER}
mdrun_d
```

**dirnamelist.txt**

```
znc24/100p/a1
znc24/200p/b2
ag80/100p/b1
ag81/100q/c1
ptn2/50a/a1
ptn3/50b/c1
…
```

Task 1 reads `znc24/100p/a1` as folder name
Task 2 reads `znc24/200p/b2` as folder name
…

- Number of lines in file **must** match number of tasks
- To get number of lines in master file use:
  **`wc -l dirnamelist.txt`**
- NB: VAR=$(*command arg1 arg2…*) captures output from *command* and assigned to variable VAR



# Jobarrays and qstat, qdel

- qstat shows running tasks and tasks still waiting

```
[mxyzabc1@login1 ~]$ qstat
job-ID  prior   name       user         state submit/start at     queue                          slots ja-task-ID
-----------------------------------------------------------------------------------------------------------------
 675199 0.35028 exjobarr.q mxyzabc1     r     02/09/2015 18:24:31 C6100-STD-serial.q@node395.dan     1 1
 675199 0.35028 exjobarr.q              r     02/09/2015 18:24:31 C6100-STD-serial.q@node370.dan     1 2
 675199 0.35028 exjobarr.q              r     02/09/2015 18:24:31 C6100-STD-serial.q@node357.dan     1 3
 675199 0.35028 exjobarr.q              r     02/09/2015 18:24:31 C6100-STD-serial.q@node342.dan     1 4
 675199 0.35028 exjobarr.q              r     02/09/2015 18:24:31 C6100-STD-serial.q@node358.dan     1 5
 675199 0.35028 exjobarr.q              r     02/09/2015 18:24:31 C6100-STD-serial.q@node402.dan     1 6
 675199 0.35028 exjobarr.q              r     02/09/2015 18:24:31 C6100-STD-serial.q@node402.dan     1 7
 675199 0.35028 exjobarr.q              r     02/09/2015 18:24:31 C6100-STD-serial.q@node402.dan     1 8
 675199 0.35028 exjobarr.q              r     02/09/2015 18:24:31 C6100-STD-serial.q@node402.dan     1 9
 675199 0.35028 exjobarr.q              r     02/09/2015 18:24:31 C6100-STD-serial.q@node401.dan     1 10
 675199 0.35028 exjobarr.q              r     02/09/2015 18:24:31 C6100-STD-serial.q@node401.dan     1 11

 675199 0.35028 exjobarr.q              r     02/09/2015 18:24:33 C6100-STD-serial.q@node395.dan     1 239
 675199 0.35028 exjobarr.q              r     02/09/2015 18:24:33 C6100-STD-serial.q@node395.dan     1 240
 675199 0.35000 exjobarr.q              qw    02/09/2015 18:24:23                                    1 241-5000:1
[mxyzabc1@login1 ~]$
```

- qdel can remove all tasks or just some

`qdel 675199` Remove all running and waiting
`qdel 675199 -t 300` Remove task 300 (a bit strange)
`qdel 657199 -t 4000-5000` Remove last 1000 tasks



# Jobarray Output Files

- You'll get the usual output .o file and error .e file (hopefully empty) but
  - One per task
  - Potentially a lot of files!
- Look for
  *jobname*.o*JobID*.*TaskID* and
  *jobname*.e*JobID*.*TaskID*
- You should delete empty / unwanted files soon and often

# PRACTICAL SESSION 5

Job array examples



## Practical Session 5 (job array)

- Write a small job array to process some images
- Go to `~/training/RCSF/examples/hudf_images/`
  - Has some images from Hubble Ultra Deep Field
    https://esahubble.org/images/heic0611b/
    **Credit:** NASA, ESA, and S. Beckwith (STScI) and the HUDF Team
  - To list them: `ls -l` To view one: `eog hudf_1.png`
  - Write a serial jobscript to process an image using:
    `module load apps/binapps/anaconda3/2021.11`
    `python process.py filename.png`
  - Add the jobarray `#$ -t` line to it (with *start* and *end*) and use `$SGE_TASK_ID` in the image filename
- Check the results in `xxxxx.o`*`JobID`*`.`*`TaskID`*
- **Q**: Which image has most objects detected?
- On login node run: `eog` *`filename`*`.png` to see images
- No exercise sheet again ;-)



## Practical Session 5 (advanced job array)

- Write a small job array to run an app with different input parameters (taken from a list of input params).
- Go to `~/training/RCSF/examples/`
- You should now be able to
  - Get number-of-lines in the `numberlist.txt` file (the list of inputs)
  - Begin writing a serial jobscript
  - Add the jobarray `#$ -t` line to it (with *start* and *end*)
  - Optional: Use CSF3 website to find the `#$` flag to "join" .o and .e outputs into only the .o file (for each task) to reduce number of files.
- Each task should read a line from `numberlist.txt` (each line in the file contains an integer)
  - Use that integer as a command-line param to a prime-factor program:
    `./prime_factor.exe`
- Check the results in `xxxxx.o`*`JobID`*`.`*`TaskID`*
- No exercise sheet again ;-)



# JOB PIPELINES

Ordering jobs

# A Job Pipeline (aka workflow)

- Suppose you have several jobs that:
  - Need to run in a specific order - a job "pipeline"
    - There is a *dependency* between jobs
  - Each might have different CPU-core or memory requirements
  - Each might take different amounts of time to run

1. Pre-processing job:
**raw.dat** to **clean.dat**
- Serial (1-core)
- Low memory
- Runs for several hours

→

2. Main processing job:
Analyse **clean.dat** to **result.dat**
- Parallel (multi-core)
- High memory
- Runs for many days

→

3. Post-processing job:
Graphs from **result.dat** to **graphs.png**
- Serial (1-core)
- Low memory
- Runs for under one hour



# How **not** to do it on the CSF (1)

- Put all steps in one job?
  - Wastes resources (some cores and mem)
  - May go over 7-day runtime limit

Only one command uses all of the cores

**mypipeline_bad.txt**

```
#!/bin/bash --login
#$ -cwd
#$ -l mem2000           # Uses a high-memory node and
#$ -pe smp.pe 16        # … reserves 16 cores
                        # … for duration of job
module load apps/……

# First 'job' (serial)
preproc -in raw.dat -out clean.dat
# Second 'job' (parallel, needs lots of memory)
mapper -p $NSLOTS -in clean.dat -out result.dat
# Third 'job' (serial)
drawGraphs -in result.dat -out graphs.png
```



# Better but still not perfect

- Split into multiple jobs, notice when jobs finish, submit next…?
  - Log in to CSF, check if previous job has finished…. wastes time!

A serial job (no wasted cores)

**firstjob.txt**

```
#!/bin/bash --login
#$ -cwd
module load apps/……
# First 'job' (serial)
preproc -i raw.dat -o clean.dat
```

A parallel, high-mem job

**secondjob.txt**

```
#!/bin/bash --login
#$ -cwd
#$ -l mem2000           # Uses a high-memory node and
#$ -pe smp.pe 16        # … reserves 16 cores
module load apps/……
# Second 'job' (parallel)
mapper -p $NSLOTS -i clean.dat -o result.dat
```

A serial job (no wasted cores)

**thirdjob.txt**

```
#!/bin/bash --login
#$ -cwd
module load apps/……
# Third 'job' (serial)
drawGraphs -i result.dat -o graphs.png
```

**qsub firstjob.txt**
(now wait until this job has finished before submitting the next one!)
**qsub secondjob.txt**
(now wait until this job has finished before submitting the next one!)
**qsub thirdjob.txt**
(now wait until this job has finished before submitting the next one!)



# How **to** do it - Job Dependencies

- Split in to multiple jobs, submit all jobs, let SGE manage it!

The jobscripts are as before, but …

**firstjob.txt**

```
#!/bin/bash --login
#$ -cwd
module load apps/……
# First 'job' (serial)
preproc -i raw.dat -o clean.dat
```

The jobscript filename is used for the name of the job (if no **#$ -N *name*** flag supplied.

… added a job dependency

**secondjob.txt**

```
#!/bin/bash --login
#$ -cwd
#$ -l mem2000           # Uses a high-memory node and
#$ -pe smp.pe 16        # … reserves 16 cores
#$ -hold_jid firstjob.txt
module load apps/……
# Second 'job' (parallel)
mapper -p $NSLOTS -i clean.dat -o result.dat
```

… added a job dependency

**thirdjob.txt**

```
#!/bin/bash --login
#$ -cwd
#$ -hold_jid secondjob.txt
module load apps/……
# Third 'job' (serial)
drawGraphs -i result.dat -o graphs.png
```

**-hold_jid *name* (or *jobid*)** makes the job automatically wait for the named (earlier) job **to finish**. The name can be a **job name** or a **job ID** number.

- Submit all of your jobs in one go

```
qsub firstjob.txt
qsub secondjob.txt
qsub thirdjob.txt
```

# Job Dependencies

- You *must* submit the jobs in the correct order
  - EG: If `secondjob.txt` is submitted first, it runs immediately (no dependency job exists to wait for)
- `qstat` shows `hqw` for jobs on hold

```
job-ID  prior   name       user         state submit/start at     queue                          slots ja-task-ID
-----------------------------------------------------------------------------------------------------------------
 857177 0.35002 firstjob.t             r     11/12/2019 17:46:16 short-interactive.q@node406.pr     1
 857178 0.00000 secondjob.             hqw   11/12/2019 17:46:12                                    1
 857180 0.00000 thirdjob.t             hqw   11/12/2019 17:46:13                                    1
```

- Later jobs may still *wait* to be scheduled
  - They don't always run *immediately* after earlier jobs finish



# Job Dependencies

- Using job names can become messy
  - Generalise using the job ID and `qsub` command-line
  - Firstly, remove all **`#$ -hold_jid name`** lines from the jobscripts!
  - Then add `-hold_jid name` to `qsub` command-line
  - Use `-terse` flag to get *just* the job ID of the submitted job (instead of 'long' message):
    - **`qsub myjobscript`**
      `Your job 19886 ("myjobscript") has been submitted`
    - **`qsub -terse myjobscript`**
      `19886`
  - Capture output of command into shell variable

```
JOBID=$(qsub -terse firstjob.txt)
JOBID=$(qsub -terse -hold_jid $JOBID secondjob.txt)
JOBID=$(qsub -terse -hold_jid $JOBID thirdjob.txt)
```



# Job-Array Dependencies (1)

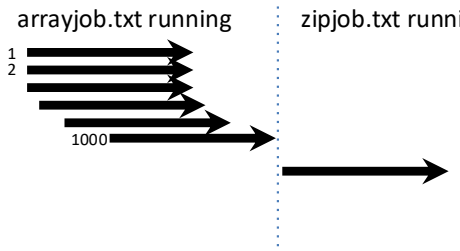
- An ordinary job can wait for a job array to finish
  - All tasks in the job array must have finished

arrayjob.txt
```
#!/bin/bash --login
#$ -cwd
#$ -t 1-1000            # Job array with 1000 tasks
convert img.${SGE_TASK_ID}.tif img.${SGE_TASK_ID}.pdf
```

zipjob.txt
```
#!/bin/bash --login
#$ -cwd
#$ -hold_jid arrayjob.txt
zip conference.zip img.*.pdf
```

Add a job dependency

```
qsub arrayjob.txt
qsub zipjob.txt
```






# Job-Array Dependencies (2)

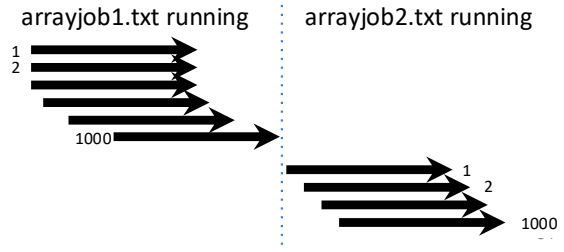
- A *job array* can wait for a job array to finish
  - *All* tasks in the *first* job array must have finished

arrayjob1.txt
```
#!/bin/bash --login
#$ -cwd
#$ -t 1-1000            # Job array with 1000 tasks
someapp data.${SGE_TASK_ID}.xyz data.${SGE_TASK_ID}.dat
```

arrayjob2.txt
```
#!/bin/bash --login
#$ -cwd
#$ -t 1-1000            # Job array with 1000 tasks
#$ -hold_jid arrayjob1.txt
someotherapp data.${SGE_TASK_ID}.dat res.${SGE_TASK_ID}.dat
```

Add a job dependency

```
qsub arrayjob1.txt
qsub arrayjob2.txt
```



# Job-Array Dependencies (3)

- Job array *tasks* can wait for other *tasks* to finish
  - A task in *second* job array waits for same task in first

arrayjob1.txt

```
#!/bin/bash --login
#$ -cwd
#$ -t 1-1000            # Job array with 1000 tasks
someapp data.${SGE_TASK_ID}.xyz data.${SGE_TASK_ID}.dat
```

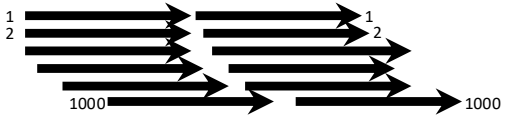
arrayjob2.txt

```
#!/bin/bash --login
#$ -cwd
#$ -t 1-1000            # Job array with 1000 tasks
#$ -hold_jid_ad arrayjob1.txt
someotherapp data.${SGE_TASK_ID}.dat res.${SGE_TASK_ID}.dat
```

Add a job *array* (_ad) dependency

```
qsub arrayjob1.txt
qsub arrayjob2.txt
```

arrayjob1.txt tasks running then arrayjob2.txt tasks






# INTERACTIVE AND GPU COMPUTING

Compute apps with GUIs



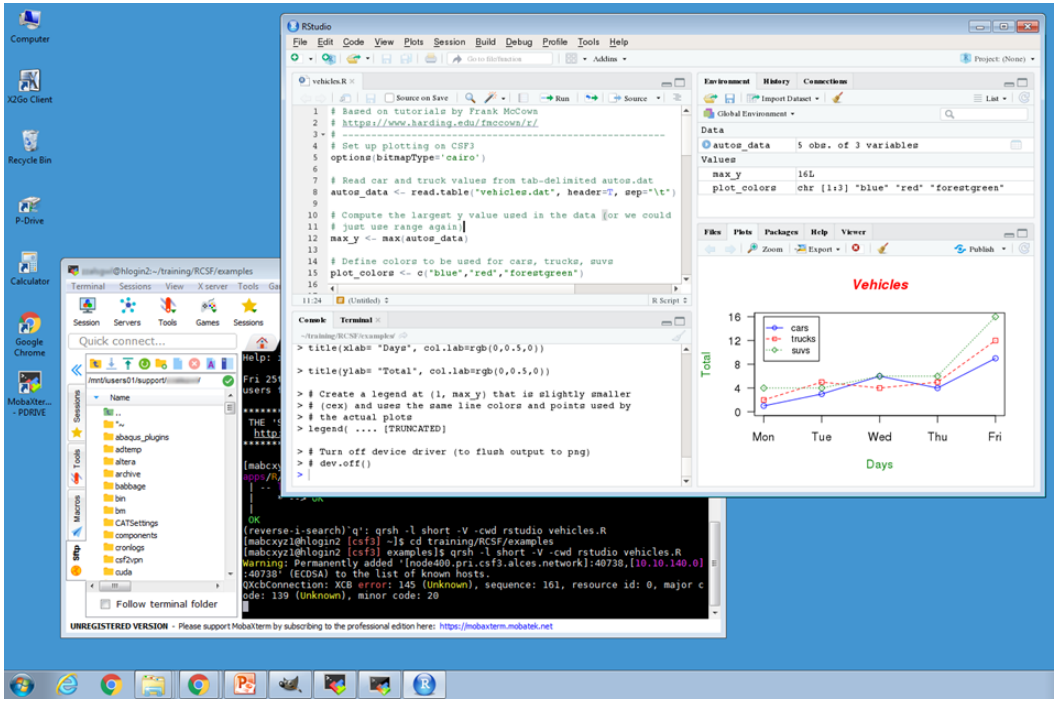
# Interactive work

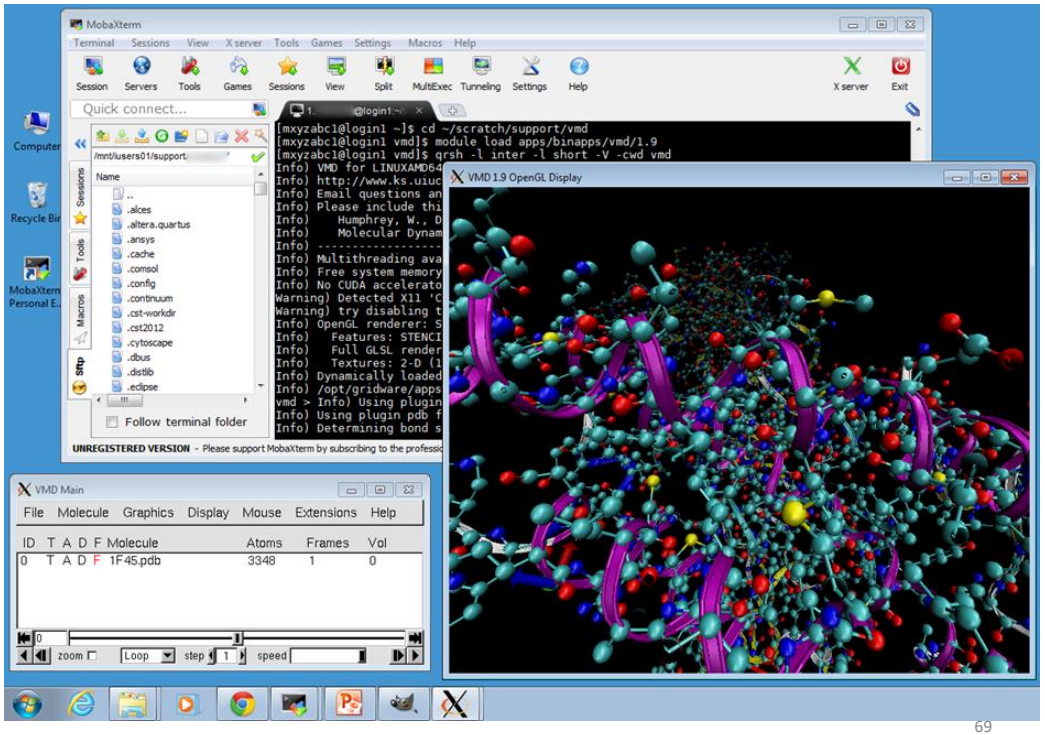
- Some apps (eg Rstudio, VMD, molden, paraview) may have a GUI but should not be run on the login node!!
- Use the `qrsh` command to get an *interactive session* on a compute node

```
module load apps/binapps/rstudio/1.1.463
qrsh -l short -V -cwd rstudio vehicles.R
```

- No dedicated resource, priority to batch jobs
- Only 4GB per core (contact its-ri-team@manchester.ac.uk if you need more)
- Remember - it is a GUI app, as with `gedit` you need Xwindows running on your PC (MobaXTerm, X-Quartz, Linux)
- Remember to exit your GUI app when you have finished so the resource is made available for others
- Better options: Virtual Desktop Service and InCLine (Interactive Computational LInux Environment) also known as iCSF.










# Nvidia GPUs

- CSF3 has 152 x Nvidia GPUs

**68 x Volta v100 GPUs** in total – 4 GPUs/node
16GB GPU memory, Mem bandwidth 900GB/s
5120 CUDA cores (80 Multiprocessors, 64 cores/MP)
640 Tensor cores
Peak FP64 7.5 TFLOPS
32-core Intel "Skylake"
192GB RAM host node + InfiniBand

**72 x Ampere A100 GPUs** in total – 4 GPUs/node
80GB or 40GB GPU memory, Mem bandwidth 2TB/s
6912 CUDA cores (108 Multiprocessors, 64 cores/MP)
432 Tensor cores
Peak FP64 9.7 TFLOPS
48-core AMD Epyc "Milan"
512GB RAM host node + InfiniBand

- Also some L40s GPUs (for a specific research group)
- Faster for certain tasks
  - All cores perform same instruction
  - Operating on different items of data
- Code can be difficult to write (CUDA, OpenCL)
- Several CSF apps already support GPUs



# OTHER PARALLEL HARDWARE

What else is available?



# HPC Pool

- Dedicated pool for “true” HPC jobs
  - 4096 cores of Infiniband connected Skylake
  - Minimum 128-core job size, maximum 1024
  - Frontend shared with CSF3
    - You just submit HPC jobs like any other CSF job (with a different "PE" name and an account code.)
  - Lightweight application process – must be made by PI
  - Currently free

https://ri.itservices.manchester.ac.uk/csf3/hpc-pool

# ITS Condor Service

- Formerly EPS Condor Pool
  - Condor manager HTC workflow
  - Condor pool is a group of cores available for use
  - Condor sends out jobs to the pool (similar to SGE)
  - Often cores become available when PCs are idle
    - UoM public clusters over night
    - Dedicated pool always available
- Condor pool available to all researchers for free
  - More than 2000 cores (if all configured PCs available)
  - Suitable for short lightweight computations
  - Can now burst to the cloud (AWS)!!!
  - See https://ri.itservices.manchester.ac.uk/htccondor/



# ARCHER2

- National supercomputer funded by UK Research Councils
  - Archer2 has replaced Archer which was 118,080 cores
  - Now 5,848 compute nodes, each with dual AMD EPYC Zen2 (Rome) 64 core CPUs at 2.2GHz, giving 748,544 cores in total.
  - Estimated peak performance of 28 PFLOP/s
- Mostly open source / free HPC software
- See https://www.archer2.ac.uk/
  - Info for how to apply for access
    - Applications assessed for suitability
- IT Services can help you apply for access



# Scafell Pike

- Hartree Centre
  - 25,728 Intel Skylake + ~55,680 Xeon Phi cores
- Common open source HPC software installed
- Focus on industry / academia collab. projects
- Contact Research IT for advice



# N8 Bede (NICE)

- 32 IBM Power 9 dual-CPU nodes
  - Each node comprises 4 NVIDIA V100 GPUs and high performance interconnect.
- 5 Nvidia GH200 Grace Hopper nodes
  - Each node comprises 1x NVIDIA H100 96GB with 900 GB/s NVLink-C2C and 1x NVIDIA Grace aarch64 CPU @ 3.483 GHz (72 Arm Neoverse V2 cores)
- Same architecture as the US government's SUMMIT and SIERRA supercomputers which occupied the top two places in a recently published list of the world's fastest supercomputers.
- Contact Research IT for advice
- https://n8cir.org.uk/supporting-research/facilities/bede/docs/

# FINAL POINTS

Further info



# News

- MOTD when you log into the CSF - please read it
- Problems e.g. system down, can't log in, minor changes to the service (and other services - e.g storage):
  https://ri.itservices.manchester.ac.uk/services-news/
- Prolonged problems or major changes emailed to all users



## its-ri-team@manchester.ac.uk

- More SGE options/parameters
  https://ri.itservices.manchester.ac.uk/csf3/batch/qsub-options/
- Job Arrays - multiple similar jobs from a single submission script
  https://ri.itservices.manchester.ac.uk/csf3/batch/job-arrays/
- SSHFS - another means of file transfer
  https://ri.itservices.manchester.ac.uk/userdocs/file-transfer/
  Virtual Desktop Service – another means of connecting and running GUIs and logging in from off campus
  https://ri.itservices.manchester.ac.uk/virtual-desktop-service/
- Please give feedback: Quick form at https://goo.gl/forms/zfZyTLw4DDaySnCF3 (choose "*Introduction to HPC (Using CSF)*")

## Thank you!