Accessing, Navigating, Running Jobs On Cyclone

Hands On Session

Chris Stylianou

Research Engineer

CaSToRC
eurocc.cyi.ac.cy

Co-funded by the European Union

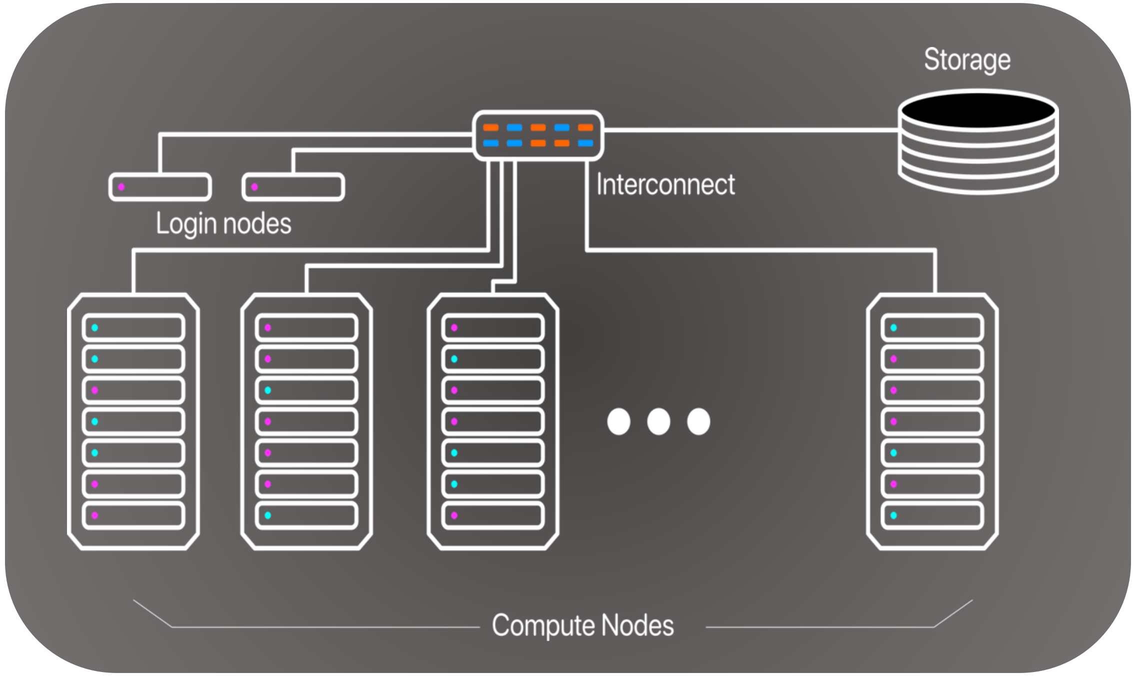
Republic of Cyprus

Outline

- Navigating an HPC Cluster
- Build Environment
- Submitting Jobs

- **Assumptions about this hands-on session**
  - Some familiarity with programming in Python
  - Familiarity with some common command-line tasks, e.g. navigating the file system
  - Can edit text files on a remote server, e.g. text-based (emacs or vim) or [VS Code Remote]

You can find the slides at: [https://github.com/CaSToRC-CyI/EuroCC-HPC-with-Python-2025]



- **Specific configuration of our local system**
- Part of a larger system with ~100 nodes
- 4 nodes reserved for this training
  - Hostnames: cn{04,07-08,15}
  - 2×20-core Intel Xeon
  - 186 GBytes RAM
  - 4×NVIDIA V100 GPUs with 32 GBytes Graphics memory each
- Common storage for our course: /nvme/scratch/edu27/

- Log in to a *login node* or *frontend node* –– login node in our case has hostname `front02`
- To run programs on *compute nodes*, a *job scheduler* is available
- Distinguish between *interactive* and *batch* jobs

SLURM job scheduler

- See currently running and waiting jobs: `squeue`
- Ask for an interactive job: `salloc`
- Submit a batch job: `sbatch`
- Run an executable: `srun`

Accessing the System

- To Log in, open a new terminal and type:

```
[localhost ~]$ ssh <username>@front02.hpcf.cyi.ac.cy
```

⚠ Should have gotten instructions how to get to this point before the training event!

⚠

- Type hostname. This tells you the name of the node you are currently logged into:

```
[cstyl@front02 ~]$ hostname
front02
```

this is the login node!

- Ask for one node (`-N 1`):

```
[cstyl@front02 ~]$ salloc -N 1 -p cpu --reservation=edu27 -A edu27
salloc: Granted job allocation 397828
salloc: Waiting for resource configuration
salloc: Nodes cn04 are ready for job
[cstyl@cn04 ~]$
```

- `-A edu27`: charge project with name `edu27`
- `--reservation=edu27`: use reservation `edu27` (reservation and project names need not be the same)
- `-p=cpu`: requests a node from the `cpu` partition –– the partition that includes all `cn[04,07-08,15]` nodes

- Type `hostname` again:

```
[cstyl@cn04 ~]$ hostname
cn04
```

`hostname` is the hostname of a compute node.

- Release the node (type `exit` or hit `ctrl-d`):

```
[cstyl@cn04 ~]$ exit
salloc: Relinquishing job allocation 397828
[cstyl@front02 ~]$
```

we're back on `front02`

⚠ Please **do not** hold nodes unnecessarily; when you have nodes `salloc`ed you may be blocking other users from using those nodes. ⚠

Submitting Jobs via SLURM

- Use `srun` instead of `salloc`:

```
[cstyl@front02 ~]$ srun -N 1 -p cpu -A edu27 --reservation=edu27 hostname
cn07
```

- Allocates a node, runs the specified command (in this case `hostname`), and then exits the node, releasing the allocation
- The output, `cn07`, reveals that we were allocated node `cn07` for this specific –– very short –– job!

Submitting Jobs via SLURM (2)

- Run multiple instances of `hostname` in parallel:

```
[cstyl@front02 ~]$ srun -N 1 -n 2 -p cpu -A edu27 --reservation=edu27 hostname
cn04
cn04
```

`-N 1`: use one node
`-n 2`: use two processes

- Run on **more than one node**:

```
[cstyl@front02 ~]$ srun -N 2 -n 2 -p cpu -A edu27 --reservation=edu27 hostname
cn04
cn07
```

runs one instance of `hostname` on each node.

- Try:

```
[cstyl@front02 ~]$ srun -N 2 -n 1 -p cpu -A edu27 --reservation=edu27 hostname
```

```
srun: Warning: can't run 1 processes on 2 nodes, setting nnodes to 1
cn04
```

- Make a directory. **List** it, see that it is there:

```
[cstyl@front02 ~]$ mkdir eurocc-training
[cstyl@front02 ~]$ ls
eurocc-training
```

- **Change** into it:

```
[cstyl@front02 ~]$ cd eurocc-training
[cstyl@front02 eurocc-training]$
```

- `pwd` will tell you where you are in the file system:

```
[cstyl@front02 eurocc-training]$ pwd
/nvme/h/cstyl/eurocc-training
```

- `/` is referred to as the *root* directory
- `.` is an alias for the *current* directory
- `..` is an alias for the directory one level above
- `~` is an alias for your home directory

E.g.:

```
[cstyl@front02 ~]$ cd ../../
[cstyl@front02 eurocc-training]$ pwd
/nvme/h
```

Navigating Directories (2)

- `cd` without any additional arguments takes you home (equivalent to `cd ~`)

```
[cstyl@front02 ~]$ cd
[cstyl@front02 ~]$ pwd
/nvme/h/cstyl
```

- Make a subdirectory under `eurocc-training` for our first Python program:

```
[cstyl@login02 ~]$ cd eurocc-training
[cstyl@login02 eurocc-training]$ mkdir 01
[cstyl@login02 eurocc-training]$ cd 01
```

- Copy the program from our shared space

```
[cstyl@login02 01]$ cp /nvme/scratch/edu27/Intro/01/prog-01.py .
```

- Use `emacs` or `vim` (or any other text editor you ‘re comfortable with) to inspect the program
- **E.g.:**

```
[cstyl@login02 01]$ vim prog-01.py
```

Running your First Program (1)

```python
import os # For getpid()
import socket # For gethostname()

def main():
    # Get the system's hostname (equivalent to gethostname() in C)
    hostname = socket.gethostname()

    # Get the current process ID (equivalent to getpid() in C)
    pid = os.getpid()

    # Print the hostname and process ID
    print(f"Hostname: {hostname}, pid: {pid}")

if __name__ == "__main__":
    main()
```

Running your First Program (2)

- Time to *execute* the program.
- It is common on HPC systems, as is the case for this system, to have multiple versions and releases of software installed
- This is also the case for ***build environments***, i.e. various versions of interpreters, compilers, linkers, debuggers, and libraries
- The user can select the version that is available to them in each session via a so-called ***modules*** *system*
- List all available modules (long listing truncated in this slide)

```
[cstyl@front02 01]$ module avail
------------------------------------------------------------------ /eb/modules/all ------------------------------------------------
ABAQUS/2024                                          attr/2.5.1-GCCcore-11.3.0
ANTLR/2.7.7-GCCcore-8.3.0-Java-11                    binutils/2.32-GCCcore-8.3.0
...
Use "module spider" to find all possible modules and extensions.
Use "module keyword key1 key2 ..." to search for all possible modules matching any of the "keys".
```

- `module list` should show the currently loaded modules. In a fresh environment with no modules loaded, you should see:

```
[cstyl@front02 01]$ module list
No modules loaded
```

- `module purge` will unload all modules!

- We will be using the `Python` module to load the Python Interpreter. See info:

```
[cstyl@front02 01]$ module show Python
…
```

- Now let's load the module before running the program:

```
[cstyl@front02 01]$ module load Python
[cstyl@front02 01]$ module list
Currently Loaded Modules:
  1) GCCcore/13.3.0                 4) bzip2/1.0.8-GCCcore-13.3.0       7) Tcl/8.6.14-GCCcore-13.3.0        10) libffi/3.4.5-GCCcore-13.3.0
  2) zlib/1.3.1-GCCcore-13.3.0      5) ncurses/6.5-GCCcore-13.3.0       8) SQLite/3.45.3-GCCcore-13.3.0     11) OpenSSL/3
  3) binutils/2.42-GCCcore-13.3.0   6) libreadline/8.2-GCCcore-13.3.0   9) XZ/5.4.5-GCCcore-13.3.0          12) Python/3.12.3-GCCcore-13.3.0
```

- Type `ls` to make sure the program is in the correct directory. Then run it on the frontend node:

```
[cstyl@front02 01]$ ls
prog-01.py
[cstyl @front02 01]$ python prog-01.py
Hostname: front02, pid: 14848
```

Running your First Program (5)

- Run the program using `srun` on two nodes with two processes each:

```
[cstyl@front02 01]$ srun -N 2 -n 4 -p cpu --reservation=edu27 -A
edu27 python prog-01.py
Hostname: cn04, pid: 129145
Hostname: cn07, pid: 241544
Hostname: cn07, pid: 241543
Hostname: cn07, pid: 241545
[cstyl@front02 01]$
```

1 process ran on `cn04` and 3 processes ran **in parallel** on `cn07`

The Fletcher 32 Checksum – Setting Up

- Time for step 2 –– a simple program to compute the Fletcher 32 checksum of several files
- First, make a new directory under `~/eurocc-training/.`

```
[cstyl@front02 01]$ cd ../
[cstyl @front02 eurocc-training]$ mkdir 02
[cstyl @front02 eurocc-training]$ cd 02/
[cstyl @front02 02]$
```

- Copy the program `fletcher32.py` from `/nvme/scratch/edu27/Intro/02/fletcher32.py`

```
[cstyl @front02 02]$ cp /nvme/scratch/edu27/Intro/02/fletcher32.py .
```

- Inspect `fletcher32.py`, e.g.:

```
[cstyl @front02 02]$ vim fletcher32.py .
```

The Fletcher 32 Checksum – Frontend Run

- To run on the frontend:

```
[cstyl@front02 02]$ python fletcher32.py
Usage: python fletcher32.py FNAME
```

- Requires as argument the filename to compute the checksum over. There are 10 files you can use under `/nvme/scratch/edu27/Intro/data/`

```
[cstyl@front02 02]$ ls -1 /nvme/scratch/edu27/Intro/data/
00.txt
01.txt
02.txt
03.txt
04.txt
05.txt
06.txt
07.txt
[cstyl@front02 02]$
```

- For example:

```
[cstyl@front02 02]$ python fletcher32.py /nvme/scratch/edu27/Intro/data/03.txt
/nvme/scratch/edu27/Intro/data/03.txt: 0c40e2d2
```

Takes ~15-20s per file!

The Fletcher 32 Checksum - Objective

- Our objective is to compute the checksums of all 8 files in parallel
- We will use the compute nodes available to us and the 40 available cores per node
- We will start by using a ***Slurm batch script*** as opposed to running ***interactively*** which we have been doing so far

The Fletcher 32 Checksum – Creating the Batch Script

- Copy from `/nvme/scratch/edu27/Intro/02/fletcher.sh`

```
#!/bin/bash
#SBATCH -J fletcher32
#SBATCH -o f32.txt
#SBATCH -e f32.err
#SBATCH -p cpu
#SBATCH -A edu19
#SBATCH --reservation=edu27
#SBATCH -t 00:02:00
#SBATCH -n 1
#SBATCH -N 1
#SBATCH --ntasks-per-node=1

module load Python

date +"%T.%6N"
srun python fletcher32.py
/nvme/scratch/edu27/Intro/data/03.txt
date +"%T.%6N"
```

- `date +"%T.%6N"` prints the current time, including milliseconds (you can try this on the command line interactively)
- The lines starting with `#SBATCH` are parsed by Slurm, even though they are ignored as comments by bash.
- The options are the same as you would pass to `srun`. Notice that `srun` now takes no options

The Fletcher 32 Checksum – Running on the Compute Node

- Submit the script via `sbatch`:

```
[cstyl@front02 02]$ sbatch fletcher.sh
Submitted batch job 397847
```

- Check the status via `squeue`:

```
[cstyl@front02 02]$ squeue -u $USER
JOBID     PARTITION NAME     USER  ST TIME NODES NODELIST(REASON)
397850    cpu       fletcher cstyl R  0:01 1     cn04
```

  `R` means "Running". You may see `PD` for "Pending" (not yet running) or `CG` for "Completing"
- When the job has finished, `squeue -u $USER` should not show any jobs
- The output is now in `f32.txt`. Any errors will be in `f32.err`
- This is the output that would be shown on the terminal had you run the program interactively. Slurm redirects the output and error to those two files
- Check the Output:

```
[cstyl@front02 02]$ cat f32.txt
22:19:41.134499
/nvme/scratch/edu27/data/session1/03.txt: 0c40e2d2
22:19:55.669549
```

Now let's loop over the 8 files

- Could just copy-paste the line to have 8 `srun` lines
- But here we will go with something a little more elegant:

```
date +"%T.%6N"
for((i=0; i<8; i++)); do
    filename=$(printf "%02.0f.txt" $i)
    srun python fletcher32.py /nvme/scratch/edu27/Intro/data/$filename
done
date +"%T.%6N"
```

- The `#SBATCH` lines not shown are unchanged compared to before.

- Submit and check output once completed (will overwrite previous output). Note that the 8 iterations need about 2 minutes:

```
[cstyl@front02 02]$ cat f32.txt
22:36:50.344613
/nvme/scratch/edu27/Intro/data/00.txt: 04d70552
/nvme/scratch/edu27/Intro/data/01.txt: 19708cd4
/nvme/scratch/edu27/Intro/data/02.txt: ed737a1c
/nvme/scratch/edu27/Intro/data/03.txt: 0c40e2d2
/nvme/scratch/edu27/Intro/data/04.txt: f7bde74d
/nvme/scratch/edu27/Intro/data/05.txt: 562ddd6c
/nvme/scratch/edu27/Intro/data/06.txt: 6f2cd2f1
/nvme/scratch/edu27/Intro/data/07.txt: 016db6c6
22:38:46.696467
```

The Fletcher 32 Checksum – Batch Script for Running in Parallel (1)

- Now we will modify the script to run the 8 iterations ***in parallel***

```
#!/bin/bash
#SBATCH -J fletcher32
#SBATCH -o f32.txt
#SBATCH -e f32.err
#SBATCH -p cpu
#SBATCH -A edu27
#SBATCH --reservation=edu27
#SBATCH -t 00:02:00
#SBATCH -n 8
#SBATCH -N 2
#SBATCH --ntasks-per-node=4
module load Python
date +"%T.%6N"
for((i=0; i<8; i++)); do
    filename=$(printf "%02.0f.txt" $i)
    srun --exclusive -n 1 -N 1 python fletcher32.py /nvme/scratch/edu27/Intro/data/$filename &
done
wait
date +"%T.%6N"
```

The Fletcher 32 Checksum – Batch Script for Running in Parallel (2)

- 8 processes in total, 2 nodes, 4 processes per node
  - `-n 8`
  - `-N 2`
  - `--ntasks-per-node=4`
- Each instance of `srun` will run on one node and a single process
  - `srun -n 1 -N 1`)
- `--exclusive` means that each process will be dedicated a processor
  - i.e., a "CPU core"
- The ampersand "&" at the end of the `srun` line indicates that the command in this line should be ***sent to the background***
  - i.e. the loop will go to the next iteration without waiting for the current iteration to complete
- `wait` means wait for all background processes to finish before moving to the next line

The Fletcher 32 Checksum –Running in Parallel

- Submit and check the output once completed!

```
[cstyl@front02 02]$ cat f32.txt
22:31:47.057209
/nvme/scratch/edu27/Intro/data/05.txt: 562ddd6c
/nvme/scratch/edu27/Intro/data/03.txt: 0c40e2d2
/nvme/scratch/edu27/Intro/data/07.txt: 016db6c6
/nvme/scratch/edu27/Intro/data/00.txt: 04d70552
/nvme/scratch/edu27/Intro/data/04.txt: f7bde74d
/nvme/scratch/edu27/Intro/data/01.txt: 19708cd4
/nvme/scratch/edu27/Intro/data/06.txt: 6f2cd2f1
/nvme/scratch/edu27/Intro/data/02.txt: ed737a1c
22:32:03.237708
```

- The 8 checksums should be identical to before
- The order is random. Which checksum completes first is undetermined. The only guarantee is that the last line in the script (the second `date +"%T.%6N`) is run after all 8 are complete (because of the preceding wait)
- All 8 checksums complete within ~15 seconds

Outline of major points covered

- Job scheduling system for reserving and running on compute nodes
- Interactive versus batch jobs
- Modules system for selecting build environment
- Running programs concurrently on multiple cores and multiple nodes

Important points *not* covered

- Here we parallelized the execution of otherwise ***scalar*** programs
- We have not covered implementing parallelism and concurrency ***within*** the programs, which is the standard approach in parallel computing
- Examples:
  - Multi-threading, using a shared memory paradigm. For example using OpenMP
  - Distributed memory parallelization, e.g. using MPI

Up next: Distributed Memory Parallelisation in Python

Thank you for your attention!

More information:

https://eurocc.cyi.ac.cy/
https://www.linkedin.com/company/eurocc2

Contact us at:
eurocc-contact@cyi.ac.cy

Funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) and Germany, Bulgaria, Austria, Croatia, Cyprus (co-funded by the EU within the framework of the Cohesion Policy Programme "THALIA 2021-2027"), Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Türkiye, Republic of North Macedonia, Iceland, Montenegro, Serbia under grant agreement No 101101903.