CSCS-USI Summer School 2020

1.1. Clusters

Definition 1.1 ELA: Is a front-end/getway system that is accessible via ssh over
Hostname: `username@ela.cscs.ch`

Definition 1.2 Daint: Is the real computing system that is accessible via ssh through ela[def. 1.1]:
Hostname: `daint`

2. File System

Definition 1.3 General Parallel File System (GPFS): is a high-performance clustered file system that allows us to concurrently access data that is on multiple cluster nodes.

Definition 1.4 User `/users/<username>` **or** `$HOME`**:** GPFS filesystem for the users' homes. It is intended for reliability over performance: all home directories are backed

Warning: Users are NOT *supposed* to run jobs from this file system because of the low performance

Definition 1.5 Scratch `~/scratch` **or** `$SCRATCH`**:** High performance cluster filesystem accessible from the computing nodes that is designed for performance rather than reliability, as a fast workspace for temporary storage.
- Total capacity: 6.2 PB
- Must be used for heavy I/O

Note

Piz Daint shares a Lustre scratch file system mounted on `/scratch/snx3000`, the other clusters share the GPFS?? scratch file system under `/scratch/shared`.

Warning: All files older than 30 days will be deleted

Definition 1.6 Project
`/project/<goup-id>/<username>` **or** `$PROJECT`**:**
Is a shared-parallel file system based on the IBM GPFS software that points to the shared folder.
It provides intermediate storage space for datasets, shared code or configuration scripts that need to be accessed from different platforms.

Warning: Users are NOT *allowed* to run jobs from this file system because of the low performance

Definition 1.7 Store `/store`**:** This is a shared-parallel filesystem based on the IBM GPFS software. It provides long term storage for large amount of datasets, code or scripts that need to be accessed from different platforms. It is also intended for large files.

Warning: Users are NOT *allowed* to run jobs from this file system because of the low performance

2.1. Moving Files from/to the Cluster

Is more efficient than `scp` as it only transfers files that have change and uses an optimized data transfer algorithm:

```
rsyc [optional] source destination
```

`-r, -recursive:` sync files and directories recursively

`-z, -compress:` compress file data during the transfer

`-a, -archive:` achieve files and directories while synchronizing (`-a` equal to following options `-rlptgoD`)

`-v, -verbose:` verbose output.

`-q, -quiet:` suppress message output.

`-n, -dry-run:` perform a trial run without synchronization

`-h, -human-readable:` display the output numbers in a human-readable format

`-p, -progress:` show sync progress during transfer

Good standard combination — **Listening 1.1**

```
rsync -avrzp source destination
```

Note: use of / at the end of paths

- source:
  - /: `rsync` will copy the content of the last folder.
  - no slash: `rsync` will copy the last folder and the content of the folder.
- destination:
  - /: `rsync` will paste the data inside the last folder.
  - no slash: `rsync` will create a folder with the last destination folder name and paste the data inside that folder.

In order to copy files from/to the server we may use `scp`:

```
scp [optional] source destination
```

`-r:` Allows to copy entire directories

bashExample scp -r source/folder username@host:destination

Note: graphical user interfaces

we may use graphical interfaces s.a. MacFUSE, Macfusion, Cyberduck, Filezilla.

3. Modules

Definition 1.8 Modules: All systems at CSCS use a modules framework to simplify access to various compiler suites and libraries:

```
module command [args]
```

3.1. Commands

`list`: list installed modules

`help [MODULE_NAME]`: get help about a module

`show [MODULE_NAME]`: get information i.e. about environment variables about a module

`avail [MODULE_NAME]`: check available versions of a module

`load [MODULE_NAME]`: load a module into our current working environment

`unload [MODULE_NAME]`: unload a module from our current working environment

`list [MODULE_NAME]`: list the currently loaded modules

`whatis [MODULE_NAME]`: show short information about module if `whatis` is defined for it

3.2. Programming Enviroments

There exist predefined programming environments that can be loaded using:

```
module switch PrgEnv-envname
```

to get the current programming environment we can look at:

```
echo $PE_ENV
```

3.2.1. Enviroments

envname
- cray (default)
- gnu
- intel
- pgi

Notes

We may load multiple enviroments simultanously `switch`

3.2.2. Programming Enviroments and Compilers

For a given programming environment there may be several different versions of the base compiler available, one of which will be tagged as "(default)".
Compilter version can be switches using:

```
module switch compiler compiler/version
```

4. Working with the Slurm Batch System

There exist three different ways of ruining jobs:
- either we use `salloc` to allocate an interactive session
- or we use `srun` command to submit jobs to the queue.
- in order to store job settings and have more precise control over the parameters we can use `sbatch` in order to submit a *SLURM* bash script. This is the preferred way of submitting jobs.

Warning: Do not run executables without these commands on the login nodes!

4.1. Srun

Definition 1.1 `srun`**:** Run a parallel jobs:

```
srun options ./executable
```

4.2. Salloc – Interactive Jobs

Definition 1.2 `salloc`**:** Allocate an interactive session on a compute node:

```
salloc options
```

Attention: Interactive jobs have limited time wall time duration and are generally meant for debugging purposes.

4.3. Sbatch – Submitting SLURM Bash Scripts

Definition 1.3 `sbatch`**:** Run a SLURM bash script??:

```
sbatch script.sh
```

where scripts have the form:

```
#!/bin/bash -l
#
#SBATCH option1=val1
#SBATCH option2=val2
        ...

srun ./executable
```

Options

`--reservation=summer school`**:** request job under summer school queue (210 nodes valid until 24.7 - 20:30h)

`(--constrain=options|-Coptions)`**:** Allows us to constraint jobs to nodes with desired features. This can be single features or list, where we can make use of the ampersand `&` and or symbol `|`:

```
-constraint="gpu&intel"
```

Attention: the `-C gpu` option is required in order to run GPU jobs.

`(--nodes=|-N)1`**:** number of compute nodes that we want to a acquire

`(--ntasks=|-n)1`**:** Specify the number of tasks to run. Request that srun allocate resources for ntasks tasks.

`-t 30`: maximum duration of the job, the shorter the job – the faster the resource acquisition.
Allowed formats are:
`(min|min:sec|days-hours|days-hours:min|days-hours:min:sec)`

CUDA

The GPU programming environment is not readily available when you log in if we want to compile CUDA code, we should load the module `craype-accel-nvidia60`:

```
module load craype-accel-nvidia60
```

This module will load in turn the `cudatoolkit` module, making available
- the nvcc compiler
- the CUDA runtime environment
- all the CUDA programming tools (profilers, debuggers etc.)
- In addtion, it will also set the target accelerator architecture to the Tesla P100 GPU (`CRAY_ACCEL_TARGET` environment variable) and allow you to compile OpenACC code with the Cray compiler.

Note

If not loaded, Cray compiler will ignore the `OpenACC directives`, because a target accelerator could not be found. For compiling OpenACC code with the `PGI compiler`, the `CRAY_ACCEL_TARGET` variable is not necessary, but the `cudatoolkit` module must be loaded.

Piz Daint is a hybrid system featuring two types of nodes:
1. one Intel Haswell processor and one NVIDIA Tesla P100 GPU
2. two Intel Broadwell processors

Depending on the architecture that you are addressing, you should compile for a different target and possibly link against different libraries. To make this process easy, we provide two modules that set up the environment correctly for cross-compiling.

4.4. Intel Haswell

this module addresses the gpu architecture. It will set the target processor to Intel Haswell (craype-haswell) and will also make available the software stack compiled for the gpu nodes of Piz Daint:

```
module load daint-gpu
```

GPU Architecture
Please note that when building your CUDA code you should address the NVIDIA Tesla P100 architecture, by setting the nvcc flag: `(--gpu-architecture/-arch)`

```
nvcc -arch=sm_60 options main.cu
```

4.5. Intel Broadwell

this module addresses the multicore architecture. It will set the target processor to Intel Broadwell (craype-broadwell) and will also make available the software stack compiled for the multicore nodes of Piz Daint:

```
module load daint-mc
```

4.6. Profiling CUDA Programs

Definition 1.4 nvprof&nvvp:
1. Creating a profile:
   ```
   srun nvprof -o pname.nvvp poptions ./executable args
   ```
2. Launching graphical debugger:
   ```
   nvvp out_f_name.nvvp
   ```

Options

`(--force-overwrite|-f)`**:** overwrite profile output files

`--profile-from-start(on|off)`**:** Enable/disable profiling from the start of the application. If it's disabled, the application can use cu,cudaProfilerStart,Stop to turn on/off profiling.

4.7. CUDA Example

Example 1.1 Running Hello World:

```
module load daint-gpu cudatoolkit
salloc -Cgpu --reservation=summer_school -t60
nvcc -arch=sm_60 hello.cu -o hello
srun -Cgpu --reservation=summer_school ./hello_world
```

5. Managing Jobs

Definition 1.5 `scancel`**:** allows us to cancel jobs

```
scancel JOBID
```

5.1. Monitoring Jobs

Definition 1.6 `squeue`**:** Allows us to get information about our running jobs:

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

Options

`-j JOBID`: output only for the specified job

Warning: Do not run:
- `squeue` alone ⇒ prints all users=expensive
- `squeue | grep user`, also expensive

Definition 1.7 `scontrol`**:** detailed information about partitions, reservations, computing nodes etc.

Definition 1.8 `sinfo`**:** further information about partitions, list of queues,...