GooseSLURM Documentation

Tom de Geus

Jan 25, 2021

BASICS

Note: This library is free to use under the MIT license. Any additions are very much appreciated, in terms of suggested functionality, code, documentation, testimonials, word of mouth advertisement, ... Bugs or feature requests can be filed on GitHub. As always, the code comes with no guarantee. None of the developers can be held responsible for possible mistakes.

CHAPTER 1

Clusters

1.1 Basic cluster layout

The basic layout of a cluster is shown in the image below. The cluster consists of a *head-node* and several *compute-nodes*. The head-node is the "computer" you log in to from your own PC. From the head-node, jobs are distributed to the compute-nodes by a scheduler: the *queuing system*. The scheduler sends the job to a compute-node when a free spot with the requested resources is available. Running a job thus corresponds to submitting a special (Bash-)script to the scheduler. You don't know a priori when, or on which compute-node, your job will be handled.

1.2 Connecting, File Transfer and Editing

The clusters run on Linux and you interface to them using a service called "ssh", which is short for Secure SHell. This service can present a remote Command Line Interface (CLI) over a network. In Layman's terms, *it allows you to type commands from your own computer which are executed on the cluster*. An ssh-client is therefore needed, which can be obtained for any operating system.

Additionally, before and after running a job, the relevant files have to be transferred to and from the cluster respectively. Below we discuss several interfaces for either Linux/macOS or Windows users. It is remarked that generally copying actions/commands run on the users' computer, not on the cluster.

1.2.1 Connecting and File Transfer from Windows

To connect to the cluster from Windows, first a ssh-client is required. Several options exist, for example:

- SSH Secure Shell (shareware).
- PuTTy (free).
- cygwin (free): provides a Linux-like environment on Windows.
- git (free). Git is actually a version management system, but it also includes a BASH-shell on Windows.

- FileZilla (free) can be used to transfer files to and from the clusters.

1.2.2 Connecting and File Transfer from Linux/macOS

Connecting

Almost all Linux/macOS distributions have a ssh-client installed. To log in to the *furnace* cluster, open a terminal window on your own computer, and execute:

```
[username@mypc ~]$  ssh myclusteraccount@clustername
```

If this is the first time, ssh will ask you to confirm the identity of the computer you are connecting to. Confirm the question, and enter your password to log in on the cluster.

To avoid the need to provide the user's password on each login, a key can be generated on the host computer and provided to the cluster. If still logged in on the *furnace*, hit `Ctrl-D` or type `exit` to log out and return to your own computer. Then follow these steps:

1. Generate a key on your own computer:

   ```
   [username@mypc ~]$  ssh-keygen -t dsa
   ```

   confirm each question with `Return`.

2. Copy the key to the cluster with:

   ```
   [username@mypc ~]$  ssh-copy-id myclusteraccount@clustername
   ```

If done right, from now on logging in will be password-less.

File transfer

There are several file transfer options, partially depending on the Linux/macOS distribution used:

- Using a file browser (e.g. Nautilus for Gnome, or Dolphin for KDE). Open the browser and type `sftp://myclusteraccount@clustername` in the address bar (or location bar).
- Using `scp` from the command line. This command is completely analogous to the `cp` command (see *The BASH-shell*). To copy files to the cluster (e.g. *furnace*) type in your local prompt:

  ```
  [username@mypc ~]$  scp somepath myclusteraccount@clustername:someclusterpath
  ```

  where `somepath` (and `myclusteraccount` and `clustername`) should be replaced. To copy files from the cluster back to the users' computer, the source and the destination should be reversed:

  ```
  [username@mypc ~]$  scp myclusteraccount@clustername:someclusterpath somepath
  ```

  Note that to copy folders `-r` should be added after `scp`.

- The more advanced user may want to take a look at the `rsync` command, which can perform fast transfer/synchronization.

1.2.3 Editing Scripts

Here, some basic examples are given to edit files/scripts. There are basically two ways of doing this

1. Copy the file to your own computer, and then edit it with your favourite text/script editor (e.g. with code highlighting), and finally copy the file back to the cluster. This option is recommended for large modifications to a script, and usually is easier to manage in terms of versions.
2. Edit the file on the cluster with an editor that runs in the command line from a shell on the cluster. This option is recommended for small script modifications.

   There are several command line based text editors available on the clusters (and on most other *Linux* machines) named `vi`, `emacs`, and `nano`. The first two are advanced and powerful, but are extremely beginner unfriendly.

CHAPTER 2

Linux

Most clusters run on a Linux Operating System (OS). Other examples of operating systems are Windows or Mac OSx. Linux operating systems are also known as "Distributions" because they basically are a collection of software packages, which are "distributed" to the users. Popular Linux distributions are: Ubuntu, Fedora, RedHat, CentOS and openSuse, but there are many more (see for example distrowatch for an overview of all distributions).

2.1 The (mis)use of the name

Within this software collection, almost always a package called Linux is found, which is the "kernel". This particular piece of software is very important, because it handles the connection between the hardware and all the other pieces of software. However, when people speak about Linux, they typically mean a distribution containing Linux. It is like calling an airplane by the name of its engine, which is a bit awkward, but this is just how it is.

Most users never interact with the "kernel", they experience the pieces of software that provide the user interface (UI). UIs come in two flavours, the Graphical User Interface (GUI) and the Command Line Interface (CLI). When you install a Linux Distribution on your own computer, it typically comes with a GUI or desktop environment, e.g. Gnome and KDE. Typically clusters only offer a CLI, which is basically a terminal ("window") which presents you a prompt, where you can type a command.

2.2 Linux file structure

- Directories (folders) are delimited with a `/` (instead of a `\` in Windows).
- The top most directory (or root) is called `/`. Hard-drives, other media, or even remote file systems can be mounted anywhere. For example a USB drive is commonly mounted at `/media/mystick`. In contrast to Windows where each drive has a different name in the file tree (e.g. `C:\`).
- All characters can be used in directory and file names, but it is best not to use exotic characters (e.g. `*`, `"`, `'`).
- File (and directory) names starting with a `.` are hidden files, and are not visible by default.
- Files (and directories) have owners and permissions, preventing misuse or accidental removal.

- Each user has his or her own home-folder which is typically located at `/home/myusername`.

2.3 The BASH-shell

To interact with the Linux operating system a Shell is used. In this command line environment, commands given by the user are interpreted by the Shell. Several Shells exist, each with its own syntax and built-in commands. One of the most popular is the Bash-shell.

Before introducing several features of the Bash-shell it is useful to discuss the basic controls. In principle the only form of control is through the keyboard. The exception is copying and pasting (parts) of commands, which is **exclusively** done using the mouse. `Crtl+c`, `Crtl+v`, etc. have a different meaning (see below). Specifically, highlighted text is automatically copied to the clipboard. It is pasted using the middle mouse button. Alternatively, the `copy` and `paste` command can be reached through the right mouse button. Several other basic controls are listed below.

- Each command follows a prompt that is displayed by the terminal. For example:

```
[username@hostname ~]$
```

- A command is followed by `return` to execute the command.
- By pressing `TAB`, Bash will try to auto-complete your typed command, pressing `TAB` twice will print auto-complete suggestions.
- To stop a command `Ctrl+c` is used. It is advised **not** to use `Ctrl+z` or `Ctrl+s` which, respectively, sleeps or freezes a command.
- To move the location of the cursor (by definition only possible inside the current command) the keys ←, →, `Home`, and `End` can be used.
- To move through the history the keys ↑ and ↓ are used. Alternatively, the history can be searched using `Crtl+r` followed by keywords. To progress through the selection, use `Crtl+r`. It is noticed that when pressing `return` the selected command is directly executed. Use the → to edit the selected command in stead.
- To log-out `Crtl+d` is used, this is equivalent of typing `exit`.

2.3.1 Commands

Overview

In general a command consists of three parts: the command, options, and input arguments. Without going into detail, we consider an example. The command

```
[username@hostname ~]$  tar -czp -f outputname.tar.gz foldername
```

creates a compressed archive. This command can be divided as follows

```
prompt $ command <options> arguments

# prompt:   [username@hostname ~]$
# command:  tar
# options:  -czp -f outputname.tar.gz
# argument: foldername
```

From this, we observe that different parts of the command are separated by spaces. Also, we observe that options begin with a "-". Furthermore some options require an argument. As is observed for the `-f` option, the argument directly follows the option. Finally, it is remarked that options are commonly combined. In the command above the options `-c`, `-z` and `-p` are grouped to `-czp`.

Most commands have a manual page. This page is found using

```
[username@hostname ~]$  man commandname
```

This opens a simple text-viewer. Using the ↓ / ↑, `PageUp` / `PageDown`, and the scroll wheel on the mouse one can scroll through the manual page. To search the manual use `/` followed by your query, and `n` to progress through the search results. To close the editor type `q`. The `man` command prompts accept the same commands as the `less`-viewer.

Alternatively (or sometimes exclusively), a (short) manual page can often be printed to the screen. This is provided by the command itself, i.e.

```
[username@hostname ~]$  commandname -h
[username@hostname ~]$  commandname --help
```

Several useful commands are listed, the most important ones are elaborated in the following sections.

Command	Description
pwd	print the current working directory
ls	list directory contents
du	report disk usage of files
find	search and find files
cd	change directory
mkdir	make a directory
cp	copy files (and directories with the `-r` option)
mv	move (rename) files and directories
rm	remove files (and directories with the `-r` option)
cat	concatenate files and print on the standard output
head	print the first few lines of a file
tail	print the last few lines of a file
grep	Globally search a Regular Expression and Print, use this for simple output filtering
less	a text-file viewer
vi	a text-file editor
top	display Linux tasks
ps	report a process status list
which	shows the full path of (shell) commands
chmod	change file's permissions

`cd` – Change directory

The change directory (**cd**) command can be used to navigate through the file tree by changing the current directory. Let us use an example of a file tree such as displayed above. Typically the terminal will start in the user's home folder:

```
[username@hostname ~]$
```

where the current directory is indicated between brackets: `[ ... ]`. Notice that `[ ~ ]` is the abbreviation of `[ /home/username ]`. We can now change directory by typing

```
[username@hostname ~]$ cd ~/sim/sub1
[username@hostname sub1]$
```

where the change of directory is specified in absolute sense. Alternatively, we can use a relative file path to do the same. In a relative file path definition use

- `./` to denote the current directory
- `../` to denote the one directory up
- `../../` to denote the two directories up

The previous command could therefore also be specified as follows

```
[username@hostname ~]$ cd ./sim/sub1
[username@hostname sub1]$
```

where `./` is not strictly necessary, i.e.

```
[username@hostname ~]$ cd sim/sub1
[username@hostname sub1]$
```

is equivalent. If we would now like to change the directory to `~/sim/sub2` we could use a relative path definition:

```
[username@hostname sub1]$ cd ../sub2
[username@hostname sub2]$
```

Notice that it is convenient to use relative file definitions inside code, as they are not dependent on the file structure. For example if `../sub2/` would have been included in a code, the code is not influenced by changing `sim` to `test`. In contrast, if we had used an absolute path, the code would fail. This is particularly important when running the same code or script on different machines (running on different platforms), such as in the case of a desktop computer and a cluster.

`ls` – List directory contents

The contents (files and directories) of the current directory are listed in "matrix" format using

```
[username@hostname ~]$ ls
```

Depending on the shell and the terminal that are used, executable files, files, and folders are highlighted differently. By specifying (optional) input arguments, the contents of directories other than the current directory are listed. For the example above

```
[username@hostname ~]$  ls ~/sim/sub1
```

would list one file, `output.log`.

More detailed file information can be obtained using the `-lh` option. For example

```
[username@hostname ~]$  ls -lh ~/sim/sub1
```

would output for example

```
-rw-rw-r-- 1 exuser exgroup 26K Sep 18 11:57 output.log
```

whereby the columns indicate:

1. permissions
2. count
3. owner
4. size
5. time/data modified
6. name

Or more specifically

1. In Linux each file/directory/link has permissions. In the output of **`ls -l`** these permissions break down as follows:

   ```
   a. -      -/d/l
   b. rw-    user
   c. rw-    group
   d. r--    other
   ```

   Herein, the first item specifies if the item is a file (`-`), a directory (`d`), or link (`l`). The next three group specify the permissions of the file's owner, its group (both specified in 3.), and other users. Herein `r` corresponds to read permission, `w` to write permission, and `x` to execute permission. In this case the user `exuser` is allowed to

read and write the file. The same permission resides with users in the group `exgroup`, while other users may only read the file.

From this it follows that an executable in Linux is nothing more than a file (e.g. plain text) with the right permissions. The `extension` is in principle meaningless. The file can be made executable using the command chmod, e.g.

```
[myname@hostname ~] $ chmod u + x output . log
```

More information is found online.

Note: The permissions can be directly specified (instead of added or removed) using a numerical notation:

- 4 = r (read)
- 2 = w (write)
- 1 = x (execute)

The desired permissions are set by adding the numerical value of those permissions you would like to allow. For example:

```
[username@hostname ~]$  chmod 764 output.log

[username@hostname ~]$  ls -lh output.log
-rwxrw-r-- 1 exuser exgroup 26K Sep 18 11:57 output.log
```

---

2. The number of directories and links inside the item. For a file the counter is always equal to one.
3. The user and group name to which the file belongs.
4. The size of the file. Because we have used the `-h` option, this is in human readable format (i.e. kilo-, mega-, giga-, or terabytes).
5. The time and date of the last modification to the file.
6. The file name

`cp`, `rm`, `mv` – File operations

The copy (**`cp`**), remove (**`rm`**), and move (**`mv`**) commands are used to do file operations, directories are created using **`mkdir`**.

Copy

To copy a file:

```
[myname@hostname ~] $ cp source destination
```

For example to make a backup of the `output.log` file, used as an example in the previous section, in the same folder:

```
[myname@hostname ~] $ cp ~/ sim / sub1 / output . log ~/ sim / sub1 / output . bak
```

If this command is issued from the `~/sim/sub1` directory, the relative command

```
[myname@hostname sub1] $cp output . log output . bak
```

is sufficient.

If a directory is copied, the `-r` (recursive) options should be specified to also copy all the content of the directory. For example:

```
[myname@hostname ~] $ cp -r ~/ sim / sub2 ~/ sim / sub3
```

Remove

Analogous to the copy command, a file is removed using

```
[myname@hostname ~] $ rm filename
```

To remove a directory use

```
[myname@hostname ~] $ rm -r directoryname
```

Notice that, in principle, removed files cannot be recovered, i.e. there is no such thing as a recycle bin when removing files from the command line. For convenience, wild cards can be used. One example of a wild card is `*`. Simply said, the `*` replaces zero or more characters. For example to remove all `.log` files in the `~/sim/sub1` folder:

```
[myname@hostname sub1] $ rm *. log
```

which in this case would remove only `output.log`. In contrast, the command

```
[myname@hostname ~] $ rm -r ~/ sim / sub *
```

would remove all the directories beginning with `sub`, which, in this case would be both the directories `sub1` and `sub2` including all their content.

Danger: Never use the command

```
[myname@hostname ~] $ rm -r *.*
```

since it removes all files and directories up and down the file tree (including those that are hidden) to which the user has permissions. Thus, all your files on the computer are permanently lost. The `.*` in the wild card string also matches `..` which causes the remove command to also remove higher directories. This mistake is typically made by DOS users, where it has a different meaning. In a Linux environment, **`rm -r *`** is usually the intended command, i.e. empty the current directory.

Move

To move a file to a different location (or to rename a file) the following command is used (for files and directories)

```
[myname@hostname ~] $ mv source destination
```

For example to rename the `output.log` file:

```
[myname@hostname sub1] $ mv output . log output . txt
```

To move this file to the `~/sim/sub2` directory:

```
[myname@hostname sub1] $ mv output . log ../ sub2 / output . txt
```

Make a directory

To create a directory, use the command

```
[myname@hostname sub1] $ mkdir dirname
```

2.3.2 Redirecting output

Redirecting output is a powerful capability of (among others) Bash. This way the output that is printed to standard Input/Output (i.e. the screen) can be intercepted and used differently. The output can be transferred to another command using `|`, or it can be stored to a file using > or appended to a file using >>.

For example to find the lines in which error messages are included in the file `output.log`, we could use:

```
[username@hostname sub1]$  cat output.log | grep -n "error"
```

The **`cat`** command outputs the contents of the `output.log` file. The `|` intercepts this output and forwards it to the The **`grep`** command, which and prints the lines matching the pattern `error` (including the line numbers, because of the `-n` option).

These lines can be stored to a file `error.log` using the command

```
[username@hostname sub1]$  cat output.log | grep -n "error" > error.log
```

To get the current directory as the top line of the file, we do

```
[username@hostname sub1]$  pwd > error.log
```

which empties or creates the file `error.log` and prints the current working directory. The file is now appended with the error lines by

```
[username@hostname sub1]$  cat output.log | grep -n "error" >> error.log
```

As a final note, the Bash shell considers two outputs, the `stdout` and the `stderr`. Any program can write to these outputs, and typically both are shown in the terminal window. It is possible to redirect each output differently, but this is considered outside the scope of this document.

2.3.3 Basic scripting

Bash commands, some of which are introduced above, can be combined in a script. Such a script is an executable plain-text file. Below, we consider a very simple script **`myscript`**. We first make the file and give the user executable permissions, e.g. by

```
[username@hostname ~]$  touch myscript
[username@hostname ~]$  chmod u+x myscript
```

We then edit the file's contents to

```
#!/bin/bash
#
# This is a very simple script

varname="Hello world"
echo $varname
```

In this script, the first line selects the environment in which the script is programmed, in this case the bash environment. Except for the shell-definition on the first line, any statement that follows a # is a comment and is not evaluated. The last two lines are the only lines of code, in which the string "Hello world" is assigned to the variable varname. In the second line, the **echo** command prints the variable varname, and thus "Hello world", to the screen. the variable name is preceded by a $, to get the value of a variable.

2.3.4 Environment settings

If a script is often used, it can be useful to make it a "global" script, such that it can be used in the same way as for example **cd**. To this end, it is common to create a directory bin in the home folder:

```
[username@hostname ~]$  mkdir ~/bin
```

Next, Bash has to look for executable files in this directory. To this end, we add the new directory to the PATH variable:

```
[username@hostname ~]$  export PATH=$HOME/bin:$PATH
```

where $HOME is equivalent to ~.

Warning: Beware that copy/pasting code from this page may not transfer correctly.

To avoid having to specify this after every new login, this (and other commands) can be added to the file ~/.bashrc. This file is evaluated at the beginning of each login. This file is commonly of the following format:

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
   . /etc/bashrc
fi

export PATH=$HOME/bin:$PATH
```

CHAPTER 3

Cheat-sheet

3.1 SLURM commands

Command	Description
`sbatch "SLURM-file"`	submit a job, controlled using the script in `"SLURM-file"`
`scancel "job-id"`	delete the job with identifier `"job-id"`
`squeue`	list basic information of all jobs
`sinfo`	list basic information of all nodes

3.2 GooseSLURM wrapper functions

Command	Description
`Gsub [...]`	submit a (batch of) job(s), controlled using the provided scripts
`Gdel [...]`	delete a (batch of) job(s)
`Gstat`	list basic information of jobs
`Ginfo`	list basic information of all nodes
`Gps`	list basic information of all running processes (on the system that you are logged onto)

See *Command-line tools*

3.3 Monitor processes and resources

Command	Description
`top`	live-monitor of current running processes
`ps`	show snap-shot of processes
`ps -aux`	show snap-shot of all processes
`du -h`	size of directories
`df -h`	total, used, and available disk-space

3.4 Directory operations

Command	Description
`pwd`	print working (current) directory
`mkdir "dir"`	make new directory `"dir"`
`cd "dir"`	go to directory `"dir"`
`cd ..`	go up one directory
`ls`	list files
`ls -lh`	list files, with detailed file information

3.5 File-operations

Command	Description
`cat "file"`	print file content to the screen
`head "file"`	show first 10 line of the file content
`tail "file"`	show last 10 line of the file content
`cp "file1" "file2"`	copy `"file1"` to `"file2"`
`mv "file1" "file2"`	move (rename) `"file1"` to `"file2"`
`rm "file"`	remove `"file"`
`rm -r "dir"`	remove `"dir"`

3.6 Bash commands

Command	Description
`whoami`	show your username
`man command`	show manual of a `command` (sometimes: `command -h` or `command --help`

3.7 Search files

Command	Description
`find`	find files
`grep`	show matched pattern in file content

3.8 Keyboard shortcuts

Command	Description
`Ctrl+c`	abort command
`Ctrl+r`	search command history (use `Ctrl+r` to proceed to next match, and arrows to modify the command)
`Ctrl+d`	exit terminal

3.9 Further reading

- Linux cheat sheet

CHAPTER 4

Command-line tools

4.1 Tools

4.1.1 Gstat

Gstat provides wrapper around *squeue* resulting in more intuitively formatted output.

4.1.2 Ginfo

Ginfo provides wrapper around *sinfo* resulting in more intuitively formatted output.

4.1.3 Gps

Gps provides wrapper around *ps* resulting in more intuitively formatted output.

4.1.4 Gsub

Gsub submits jobs from the folder of the job-script (essential for jobs that use a relative path directory). It therefore wraps *sbatch*.

4.1.5 Gdel

Gdel deletes some or all of the user's jobs.

CHAPTER 5

Python module

5.1 Installation

5.1.1 pip

Installing GooseSLUM proceeds easily with `pip`:

```
cd /path/to/GooseSLURM
pip install .
```

Note:

- Be sure to use the proper executable for `pip` (it could for example be `pip3`).
- Add the `--user` option to the `pip`-command to install in the user's home-folder.

5.2 Overview

5.2.1 Write job scripts

GooseSLURM.scripts.plain([filename, ...])	Return simple SBATCH-file (as text).
GooseSLURM.scripts.tempdir([filename, ...])	Return SBATCH-file (as text) that uses a temporary working directory on the compute node.
GooseSLURM.files.cmake()	Return a list of typical build files/folders generated by CMake.

5.2.2 Parse ps

GooseSLURM.ps.read_interpret([data, theme])	Read and interpret `ps -eo pid,user,rss,%cpu,command`.
GooseSLURM.ps.read([data])	Read `ps -eo pid,user,rss,%cpu,command`.
GooseSLURM.ps.interpret(lines[, theme])	Interpret the output of `GooseSLURM.ps.read`.
GooseSLURM.ps.colors([theme])	Return dictionary of colors.

5.2.3 Parse squeue

GooseSLURM.squeue.read_interpret([data, ...])	Read and interpret `squeue -o "%all"`.
GooseSLURM.squeue.read([data])	Read `squeue -o "%all"`.
GooseSLURM.squeue.interpret(lines[, now, theme])	Interpret the output of `GooseSLURM.squeue.read`.
GooseSLURM.squeue.colors([theme])	Return dictionary of colors.

5.2.4 Parse sinfo

GooseSLURM.sinfo.read_interpret([data, theme])	Read and interpret `sinfo -o "%all"`.
GooseSLURM.sinfo.read([data])	Read `sinfo -o "%all"`.
GooseSLURM.sinfo.interpret(lines[, theme])	Interpret the output of `GooseSLURM.sinfo.read`.
GooseSLURM.sinfo.colors([theme])	Return dictionary of colors.

5.2.5 Rich strings

GooseSLURM.rich.String(data[, width, align, ...])	Rich string.
GooseSLURM.rich.Integer(data, **kwargs)	Rich integer.
GooseSLURM.rich.Float(data, **kwargs)	Rich float.
GooseSLURM.rich.Duration(data, **kwargs)	Rich duration (seconds).
GooseSLURM.rich.Memory(data, **kwargs)	Rich memory (bytes).

5.2.6 Print

GooseSLURM.table.print_long(lines)	Print full data without much formatting.
GooseSLURM.table.print_columns(lines, ...[, ...])	Print table to fit the screen.
GooseSLURM.table.print_list(lines, key[, sep])	Print a single column as a list.

5.2.7 Duration

GooseSLURM.duration.asSeconds(data[, default])	Convert string to seconds.
GooseSLURM.duration.asUnit(data, unit, precision)	Convert to rich-string with a certain unit and precision.
GooseSLURM.duration.asHuman(data[, precision])	Convert to string that has the biggest possible unit.
GooseSLURM.duration.asSlurm(data)	Convert to a SLURM time string.

5.2.8 Memory

GooseSLURM.memory.asBytes(data[, default, ...])	Convert string to bytes.
GooseSLURM.memory.asUnit(data, unit, precision)	Convert to rich-string with a certain unit and precision.
GooseSLURM.memory.asHuman(data[, precision])	Convert to string that has the biggest possible unit.
GooseSLURM.memory.asSlurm(data)	Convert to a SLURM string.

5.3 Documentation

5.3.1 GooseSLURM.scripts

GooseSLURM.scripts.**plain**(*filename='job.slurm'*, *command=[]*, *cd_submitdir=True*, ***sbatch*)

Return simple SBATCH-file (as text).

Options

filename (<str> | ['job.slurm']) The filename to assume to construct default paths for the out file.

command (<str> | <list>) Command(s) to execute. If the input is a list each entry is included as an individual line.

cd_submitdir ([True] | False) Include `cd "${SLURM_SUBMIT_DIR}"` at the beginning of the script.

SBATCH options

mem (<int> | <str>) Memory claim (may be human readable, see `GooseSLUM.memory.asSlurm`).

time (<str>) Wall-time claim (may be human readable, see `GooseSLUM.duration.asSlurm`).

out ([filename+'.out'] | <str>) Name of the output file.

...

GooseSLURM.scripts.**tempdir**(*filename='job.slurm'*, *remove=[]*, *command=[]*, ***sbatch*)

Return SBATCH-file (as text) that uses a temporary working directory on the compute node.

Options

filename (['job.slurm'] | <str>) The filename to assume to construct default paths for the out- and JSON files.

remove (<list>) List with files/folders to remove from the temporary directory before copying.

command (<str> | <list>) Command(s) to execute. If the input is a list each entry is included as an individual line.

SBATCH options

mem (<int> | <str>) Memory claim (may be human readable, see `GooseSLUM.memory.asSlurm`).

time (<str>) Wall-time claim (may be human readable, see `GooseSLUM.duration.asSlurm`).

out ([filename+'.out'] | <str>) Name of the output file.

...

5.3.2 GooseSLURM.files

`GooseSLURM.files.`**`cmake`**`()`
Return a list of typical build files/folders generated by CMake.

5.3.3 GooseSLURM.ps

`GooseSLURM.ps.`**`colors`**`(`*theme=None*`)`
Return dictionary of colors.

```
{
  'selection' : '...',
}
```

Options

theme (['dark'] | <str>) Select color-theme.

`GooseSLURM.ps.`**`interpret`**`(`*lines*, *theme={'selection': ''}*`)`
Interpret the output of `GooseSLURM.ps.read`. All fields are converted to the `GooseSLURM.rich` classes adding useful colors in the process.

Arguments

lines <list<dict>> The output of `GooseSLURM.ps.read`

Options

theme (<dict>) The color-theme, as selected by `GooseSLURM.ps.colors`.

Returns

lines (<list<dict>>) A list of dictionaries, that contain the different fields. All data are `GooseSLURM.rich.String` or derived types.

`GooseSLURM.ps.`**`read`**`(`*data=None*`)`
Read `ps -eo pid,user,rss,%cpu,command`.

Options

data (<str>) For debugging: specify the output of `ps -eo pid,user,rss,%cpu,command` as string.

Returns

lines <list<dict>> A list of dictionaries, that contain the different fields. All data are strings.

GooseSLURM.ps.**read_interpret**(*data=None*, *theme={'selection': ''}*)
Read and interpret `ps -eo pid,user,rss,%cpu,command`.

Returns

lines (<list<dict>>) A list of dictionaries, that contain the different fields. All data are `GooseSLURM.rich.String` or derived types.

5.3.4 GooseSLURM.squeue

GooseSLURM.squeue.**colors**(*theme=None*)
Return dictionary of colors.

```
{
  'selection' : '...',
  'queued'    : '...',
}
```

Options

theme (['dark'] | <str>) Select color-theme.

GooseSLURM.squeue.**interpret**(*lines*, *now=None*, *theme={'queued': '', 'selection': ''}*)
Interpret the output of `GooseSLURM.squeue.read`. All fields are converted to the `GooseSLURM.rich` classes adding useful colors in the process.

Arguments

lines <list<dict>> The output of `GooseSLURM.squeue.read`

Options

theme (<dict>) The color-theme, as selected by `GooseSLURM.squeue.colors`.

Returns

lines (<list<dict>>) A list of dictionaries, that contain the different fields. All data are `GooseSLURM.rich.String` or derived types.

GooseSLURM.squeue.**read**(*data=None*)
Read `squeue -o "%all"`.

Options

data (<str>) For debugging: specify the output of `squeue -o "%all"` as string.

Returns

lines <list<dict>> A list of dictionaries, that contain the different fields. All data are strings.

GooseSLURM.squeue.**read_interpret**(*data=None*, *now=None*, *theme={'queued': '', 'selection': ''}*)
Read and interpret `squeue -o "%all"`.

Returns

lines (<list<dict>>) A list of dictionaries, that contain the different fields. All data are `GooseSLURM.rich.String` or derived types.

5.3.5 GooseSLURM.sinfo

GooseSLURM.sinfo.**colors**(*theme=None*)
Return dictionary of colors.

```
{
  'selection' : '...',
  'free'      : '...',
  'error'     : '...',
  'warning'   : '...',
  'low'       : '...',
}
```

Options

theme (['dark']|<str>) Select color-theme.

GooseSLURM.sinfo.**interpret**(*lines*, *theme={'error': '', 'free': '', 'low': '', 'selection': '', 'warning': ''}*)
Interpret the output of GooseSLURM.sinfo.read. All fields are converted to the GooseSLURM.rich classes adding useful colors in the process.

Arguments

lines <list<dict>> The output of GooseSLURM.sinfo.read

Options

theme (<dict>) The color-theme, as selected by GooseSLURM.sinfo.colors.

Returns

lines (<list<dict>>) A list of dictionaries, that contain the different fields. All data are GooseSLURM.rich.String or derived types.

GooseSLURM.sinfo.**read**(*data=None*)
Read `sinfo -o "%all"`.

Options

data (<str>) For debugging: specify the output of `sinfo -o "%all"` as string.

Returns

lines <list<dict>> A list of dictionaries, that contain the different fields. All data are strings.

GooseSLURM.sinfo.**read_interpret**(*data=None*, *theme={'error': '', 'free': '', 'low': '', 'selection': '', 'warning': ''}*)
Read and interpret `sinfo -o "%all"`.

Returns

lines (<list<dict>>) A list of dictionaries, that contain the different fields. All data are GooseSLURM.rich.String or derived types.

5.3.6 GooseSLURM.rich

class GooseSLURM.rich.**Duration**(*data*, ***kwargs*)
Rich duration (seconds).

Note: All options are attributes, that can be modified at all times.

Options

data (<str> | None) The data.

width ([None] | <int>) Print width (formatted print only).

color ([None] | <str>) Print color, e.g. "1;32" for bold green (formatted print only).

align (['<'] | '>') Print alignment (formatted print only).

precision ([None] | <int>) Print precision to use for the conversion (formatted print only). `None` means automatic precision. See `GooseSLURM.duration.asHuman`.

dummy ([0] | <int> | <float>) Dummy numerical value, used in case of non-numerical `data`.

Methods

A.format() Formatted string, after unit conversion.

str(A) Unformatted string, after unit conversion.

A.isnumeric() Return if the "data" is numeric.

int(A) Return `data` as integer (`dummy` is returned if `data` is not numeric).

float(A) Return `data` as float (`dummy` is returned if `data` is not numeric).

class `GooseSLURM.rich.`**Float**(*data*, ***kwargs*)

Rich float.

Note: All options are attributes, that can be modified at all times.

Options

data (<str> | None) The data.

width ([None] | <int>) Print width (formatted print only).

color ([None] | <str>) Print color, e.g. "1;32" for bold green (formatted print only).

align (['<'] | '>') Print alignment (formatted print only).

precision ([2] | <int>) Print precision (formatted print only).

dummy ([0] | <int> | <float>) Dummy numerical value, used in case of non-numerical `data`.

Methods

A.format() Formatted string.

str(A) Unformatted string.

A.isnumeric() Return if the "data" is numeric.

int(A) Return `data` as integer (`dummy` is returned if `data` is not numeric).

float(A) Return `data` as float (`dummy` is returned if `data` is not numeric).

isnumeric()
Return if the "data" is numeric : always zero for this class.

class GooseSLURM.rich.**Integer**(*data*, ***kwargs*)
Rich integer.

Note: All options are attributes, that can be modified at all times.

Options

data (<str> | None) The data.

width ([None] | <int>) Print width (formatted print only).

color ([None] | <str>) Print color, e.g. "1;32" for bold green (formatted print only).

align (['<'] | '>') Print alignment (formatted print only).

dummy ([0] | <int> | <float>) Dummy numerical value, used in case of non-numerical `data`.

Methods

A.format() Formatted string.

str(A) Unformatted string.

A.isnumeric() Return if the "data" is numeric.

int(A) Return `data` as integer (`dummy` is returned if `data` is not numeric).

float(A) Return `data` as float (`dummy` is returned if `data` is not numeric).

isnumeric()
Return if the "data" is numeric : always zero for this class.

class GooseSLURM.rich.**Memory**(*data*, ***kwargs*)
Rich memory (bytes).

Note: All options are attributes, that can be modified at all times.

Options

data (<str> | None) The data.

width ([None] | <int>) Print width (formatted print only).

color ([None] | <str>) Print color, e.g. "1;32" for bold green (formatted print only).

align (['<'] | '>') Print alignment (formatted print only).

precision ([None] | <int>) Print precision to use for the conversion (formatted print only). `None` means automatic precision. See `GooseSLURM.memory.asHuman`.

dummy ([0] | <int> | <float>) Dummy numerical value, used in case of non-numerical `data`.

default_unit (int) The unit to assume if no unit if specified (specify the number of bytes).

Methods

A.format() Formatted string, after unit conversion.

str(A) Unformatted string, after unit conversion.

A.isnumeric() Return if the "data" is numeric.

int(A) Return `data` as integer (`dummy` is returned if `data` is not numeric).

float(A) Return `data` as float (`dummy` is returned if `data` is not numeric).

class GooseSLURM.rich.**String**(*data*, *width=None*, *align='<'*, *color=None*, *dummy=0*)
Rich string.

Note: All options are attributes, that can be modified at all times.

Options

data (<str> | None) The data.

width ([None] | <int>) Print width (formatted print only).

color ([None] | <str>) Print color, e.g. "1;32" for bold green (formatted print only).

align (['<'] | '>') Print alignment (formatted print only).

dummy ([0] | <int> | <float>) Dummy numerical value.

Methods

A.format() Formatted string.

str(A) Unformatted string.

A.isnumeric() Return if the "data" is numeric.

int(A) Dummy integer.

float(A) Dummy float.

format()
Return formatted string: align/width/color are applied.

isnumeric()
Return if the "data" is numeric : always zero for this class.

5.3.7 GooseSLURM.table

GooseSLURM.table.**print_columns**(*lines*, *columns*, *header*, *no_truncate=False*, *sep=' '*, *cols=None*, *print_header=True*)
Print table to fit the screen. This function can show data truncated, or even suppress columns if there is insufficient room.

Arguments

lines ([{'JOBID': '1234', ...}, ...]) List of lines, with each line stored as a dictionary. Note that all data has to be stored as one of the GooseSLURM.rich classes (to customize the color, precision, ...) or as string.

columns ([{'key': 'JOBID', 'width': 7, 'align': '>', 'priority': True}, ...]) List with print settings of each column: - 'key' : the key-name used to store each line (see `lines` below) - 'width' : minimum print width (expanded as much as possible to fit the data) - 'align' : alignment of the column - 'priority': priority of column expansion, columns marked `True` are expanded first

header (**{'JOBID': 'JobID', ...}**) Header name for each column.

Options

no_truncate (**[False]|True**) Disable truncation of columns. In this case each column is expanded to fit the data.

sep (**[', ']|<str>**) Separator between columns.

cols (**[None]|<int>**) Number of characters on one line. If `None` the current terminal's width is used.

print_header (**[True]|False**) Optionally skip printing of header.

`GooseSLURM.table.`**print_list** (*lines*, *key*, *sep=' '*)

Print a single column as a list.

Arguments

lines (**[{'JOBID': '1234', ...}, ...]**) List of lines, with each line stored as a dictionary. Note that all data has to be stored as one as string, or as one the GooseSLURM.rich classes (no rich printing is used though).

key (**'JOBID'**) Column to print.

Options

sep (**[' ']|<str>**) Separator between columns.

`GooseSLURM.table.`**print_long** (*lines*)

Print full data without much formatting. The output looks as follows:

```
+------------------+
| line 1, column 1 |
| line 1, column 2 |
| ...              |
+------------------+
| line 2, column 1 |
| line 2, column 2 |
| ...              |
+------------------+
```

Arguments

lines (**[{'JOBID': '1234', ...}, ...]**) List of lines, with each line stored as a dictionary. Note that all data has to be stored as one as string, or as one the GooseSLURM.rich classes (no rich printing is used though).

5.3.8 GooseSLURM.duration

`GooseSLURM.duration.`**asHuman** (*data*, *precision=None*)

Convert to string that has the biggest possible unit. For example: `100` (seconds) -> `"1.7m"`.

Arguments

data (**<str>|<float>|<int>**) A time, see `GooseSLURM.duration.asSeconds` for conversion.

precision (**<int>**) The precision with which to print. By default a precision of one is used for `0 < value < 10`, while a precision of zero is used otherwise.

Returns

<str> The rich-string.

GooseSLURM.duration.**asSeconds**(*data*, *default=None*)
Convert string to seconds. The following input is accepted:

- A humanly readable time (e.g. “1d”).
- A SLURM time string (e.g. “1-00:00:00”).
- A time string (e.g. “24:00:00”).
- `int` or `float`: interpreted as seconds.

Arguments

data (<str> | <float> | <int>) The input string (number are equally accepted; they are directly interpreted as seconds).

Options

default ([None] | <int>) Value to return if the conversion fails.

Returns

<int> Number of seconds as integer (or default value if the conversion fails).

GooseSLURM.duration.**asSlurm**(*data*)
Convert to a SLURM time string. For example `"1d"` -> `"1-00:00:00"`.

Arguments

data (<str> | <float> | <int>) A time, see `GooseSLURM.duration.asSeconds` for conversion.

Returns

<str> The rich-string.

GooseSLURM.duration.**asUnit**(*data*, *unit*, *precision*)
Convert to rich-string with a certain unit and precision. The output is e.g. `"1.1d"`.

Arguments

data (<int> | <float>) Numerical value (e.g. `1.1`).

unit (<str>) The unit (e.g. `"d"`).

precision (<int>) The precision with which to print (e.g. `1`).

Returns

<str> The rich-string.

5.3.9 GooseSLURM.memory

GooseSLURM.memory.**asBytes**(*data*, *default=None*, *default_unit=1*)
Convert string to bytes. The following input is accepted:

- A humanly readable string (e.g. “1G”).
- `int` or `float`: interpreted as bytes.

Arguments

data (<str> | <float> | <int>) The input string (number are equally accepted; they are directly interpreted as bytes).

Options

default ([None] | <int>) Value to return if the conversion fails.

default_unit (int) The unit to assume if no unit if specified (specify the number of bytes).

Returns

<int> Number of bytes as integer (or default value if the conversion fails).

GooseSLURM.memory.**asHuman**(*data*, *precision=None*)

Convert to string that has the biggest possible unit. For example `1e6` (bytes) -> `"1.0M"`.

Arguments

data (<str> | <float> | <int>) An amount of memory, see `GooseSLURM.duration.asBytes` for conversion.

precision (<int>) The precision with which to print. By default a precision of one is used for `0 < value < 10`, while a precision of zero is used otherwise.

Returns

<str> The rich-string.

GooseSLURM.memory.**asSlurm**(*data*)

Convert to a SLURM string. For example `"1G"`.

Arguments

data (<str> | <float> | <int>) An amount of memory, see `GooseSLURM.duration.asBytes` for conversion.

Returns

<str> The rich-string.

GooseSLURM.memory.**asUnit**(*data*, *unit*, *precision*)

Convert to rich-string with a certain unit and precision. The output is e.g. `"1.1G"`.

Arguments

data (<int> | <float>) Numerical value (e.g. `1.1`).

unit (<str>) The unit (e.g. `"G"`).

precision (<int>) The precision with which to print (e.g. `1`).

Returns

<str> The rich-string.

CHAPTER 6

Tips

6.1 Use conda to pre-build

When using a compiled code it can be useful to build once (on the appropriate hardware) and simply use the executable in a job. If you want to avoid using the path of the executable, you can install the executable. A nice way to do this is to install it in a virtual *conda* environment. This was you can create a *conda* environment relevant to your batch of jobs without having to worry too much about naming: You can have many different *conda* environments to overcome potential naming conflicts, and, moreover, you can easily throw them away and start over.

6.1.1 CMake

To install inside a *conda* environment add the following to your `CMakeLists.txt`

```
if (APPLE)
    set_target_properties(${PROJECT_NAME} PROPERTIES MACOSX_RPATH ON)
else()
    set_target_properties(${PROJECT_NAME} PROPERTIES
        BUILD_WITH_INSTALL_RPATH 1
        SKIP_BUILD_RPATH FALSE)
endif()

set_target_properties(${PROJECT_NAME} PROPERTIES
    INSTALL_RPATH_USE_LINK_PATH TRUE)

install(TARGETS ${PROJECT_NAME} DESTINATION bin)
```

Then, before building, activate your *conda* environment:

```
conda activate myenv
```

Then build with the following

```
cd /path/to/your/CMakeLists
mkdir build
cd build
cmake .. -DCMAKE_INSTALL_PREFIX:PATH="${CONDA_PREFIX}"
make
make install
```

whereby the `-DCMAKE_INSTALL_PREFIX:PATH="${CONDA_PREFIX}"` will make sure that the executable is stored in the active *conda* environment.

6.1.2 With hardware optimisation & different hardware configurations

Suppose that you have multiple hardware layouts of compute nodes, and that your are building with hardware-specific optimisations. In that case a nice option is to create (identical) *conda* environments for each hardware layout, and build and install the executable for the different hardware layouts. Then, in the job, you simply activate the right *conda* environment based on the hardware on which the job runs.

Consider having two types of hardware configurations `E5v4` and `s6g1`, then we will create two *conda* environments, `myenv_E5v4` and `myenv_s6g1`, and build (on the respective hardware layouts) and install to the relevant environment:

```
cd /path/to/your/CMakeLists
mkdir build
cd build
```

For the first environment:

```
conda activate myenv_E5v4
cmake .. -DCMAKE_INSTALL_PREFIX:PATH="${CONDA_PREFIX}"
srun -C E5v4 -p build -c 16 make -j16
make install
```

and similarly for the second environment:

```
conda activate myenv_s6g1
cmake .. -DCMAKE_INSTALL_PREFIX:PATH="${CONDA_PREFIX}"
srun -C s6g1 -p build -c 16 make -j16
make install
```

Then, in the job-script include the following:

```
source ~/miniconda3/etc/profile.d/conda.sh

if [[ "${SYS_TYPE}" == *E5v4* ]]; then
    conda activate myenv_E5v4
elif [[ "${SYS_TYPE}" == *s6g1* ]]; then
    conda activate myenv_s6g1
fi
```

CHAPTER 7

GNU parallel

7.1 Overview

This example focuses on a case in which you claim (or are given) more than one CPU on one node, which will be used to run several instances of a serial processes in parallel (for example using different input parameters, or for different statistical realizations). This case is relevant for example when the queuing system restricts allocation to entire nodes. In that case, no matter how many CPUs-per-node you would demand you would be given all CPUs on the node, so you should better make good use of it.

The idea is to use GNU Parallel to do all the work for you. It is designed to run several serial processes in parallel. It is a simple `perl` script which is easy to 'install' (see *Local installation of parallel*). What it basically does is to run some (nested) loop in parallel. Each iteration of this loop contains a (long) process (e.g. Matlab, Python, your own executable, ...). Parallel then runs `N` processes at the same time (where `N` is the number of CPUs you have available). As soon as one process finishes, the next process is submitted, and so on until your entire loop is finished. It is perfectly fine that not all processes take the same amount of time, as soon as one CPU is freed, another process is started. And the great thing is: its syntax is extremely easy.

Note: GNU Parallel also allows parallelization one several nodes. In the present context this in not very relevant as you can just submit several single node jobs. Should you be interested consult the manual.

7.2 Job script

[source: job.slurm]

```
#!/bin/bash
#SBATCH --job-name parallel
#SBATCH --out job.slurm.out
#SBATCH --nodes 1
#SBATCH --ntasks 1
#SBATCH --cpus-per-task 5
```

(continues on next page)

(continued from previous page)

```
#SBATCH --partition debug

parallel --max-procs=${SLURM_CPUS_PER_TASK} 'echo {1} {2}' ::: {1..12} ::: summer␣
↪winter
```

Note: To facilitate writing job-scripts, it can be created using the GooseSLURM Python module: [`source: writeJob.py`].

7.2.1 Code explained

The job script has to following parts:

```
#!/bin/bash
```

Definition of the language that this script is written in (BASH in this case).

```
#SBATCH --job-name parallel
#SBATCH --out job.slurm.out
#SBATCH --nodes 1
#SBATCH --ntasks 1
#SBATCH --cpus-per-task 5
#SBATCH --partition debug
```

Allocation of resources. These lines are interpreted by the `sbatch` command, but are then ordinary comments when the script run on the compute node. Notice that in this case 5 CPUs on one node are demanded.

```
parallel --max-procs=${SLURM_CPUS_PER_TASK} 'echo {1} {2}' ::: {1..12} ::: summer␣
↪winter
```

Run your executable in parallel. In this case the executable is mimicked using the *echo* command, which provided with two arguments. As an example a nested loop is run with the first argument an index in the range 1-12 and the second argument a name which gets the values 'summer' or 'winter'.

7.3 Behaviour

The job is submitted using

```
$ sbatch job.slurm
```

Once the job is completed, the output can be inspected:

```
$ cat job.slurm.out

1 summer
1 winter
2 summer
2 winter
3 summer
3 winter
4 summer
```

(continues on next page)

(continued from previous page)

```
4 winter
5 summer
5 winter
6 summer
6 winter
7 summer
7 winter
8 summer
8 winter
9 summer
9 winter
10 summer
10 winter
11 summer
11 winter
12 summer
12 winter
```

where we clearly see the effect of the nested loop.

To check that the processes actually are run in parallel one can:

1. Log in to the node

   ```
   $ ssh ...
   ```

   where the host name of the compute node that job runs on, is listed in the output of the `squeue` command.

2. Use `top` or `htop` to inspect the running processes:

   ```
   $ htop
   ```

   (You can limit the output to your account by typing `u` and then your user name.)

Note: This example runs too fast to really see anything. It becomes obvious for a more realistic (slower) process.

7.4 Other usage

7.4.1 Read commands from text file

An easy option is to store a list of commands in a plain text file:

```
somoprogram 1 1 1
somoprogram 1 1 2
somoprogram 1 1 3
somoprogram 1 2 1
somoprogram 1 2 2
somoprogram 1 2 3
...
```

These commands can then be executed in parallel using

```
parallel :::: example.txt
```

Note: Don't forget to specify the number of processors to use:

```
parallel --max-procs=${SLURM_CPUS_PER_TASK} :::: example.txt
```

7.5 Local installation of parallel

To 'install' GNU parallel into your home folder, for example under `~/opt`.

1. Download the latest version of GNU Parallel. *You'll find 'parallel-latest.tar.bz2' on the bottom of the page.*
2. Make the directory `~/opt` if it does not yet exist

   ```
   $ mkdir $HOME/opt
   ```

3. 'Install' GNU Parallel:

   ```
   $ tar jxvf parallel-latest.tar.bz2
   $ cd parallel-XXXXXXXX
   $ ./configure --prefix=$HOME/opt
   $ make
   $ make install
   ```

   Note that the relevant files will be installed in `~/opt/bin`.

4. Add `~/opt/bin` to your path:

   ```
   $ export PATH=$HOME/opt/bin:$PATH
   ```

   To make this permanent, add this line to your `~/.bashrc` (or some equivalent configuration file), see for example this page.

CHAPTER 8

Temporary working directory

8.1 Overview

This script exploits a temporary directory on the compute node to avoid costly read/write actions over the network.

In particular it does the following:

1. Copy files to a temporary directory on the compute node (which is in this case automatically provided by the queuing system).
2. Run the job (while reading and writing locally on the compute node).
3. Copy all files back to the directory from which the job is submitted (on the head node).

Here only items 1 and 3 use the network. During the computation all actions are local on the compute node. Often, quite some efficiency can be gained by doing this.

8.2 File structure

This example assumes a file structure where (almost) everything that a simulation needs, and all its output, are located in a single directory (which may have arbitrary sub-directories). For example:

```
/home/user/...
| - simulation
  | - job.slurm
  | - ... (code, input)
```

It is vital that you submit from this directory, to copy only the files relevant to this simulation:

```
$ cd /home/user/.../simulation
$ sbatch job.slurm
```

Note: *Gsub* can be called from everywhere but guarantees this behaviour.

If the job terminates (the simulation is finished, or it is cut by the queuing system) all files that are in the *simulation* on the compute node are copied back to the directory in the home folder (on the head node). After that the folder looks like

```
/home/user/...
| - simulation
  | - job.slurm
  | - ... (code, input)
  | - ... (output)
```

8.3 Job script

[source: job.slurm]

```
#!/bin/bash
#SBATCH --job-name tempdir
#SBATCH --out job.slurm.out
#SBATCH --nodes 1
#SBATCH --ntasks 1
#SBATCH --cpus-per-task 1
#SBATCH --partition debug

# I. Define directory names [DO NOT CHANGE]
# =========================================

# get name of the temporary directory working directory, physically on the compute-
↪node
workdir="${TMPDIR}"

# get submit directory
# (every file/folder below this directory is copied to the compute node)
submitdir="${SLURM_SUBMIT_DIR}"

# 1. Transfer to node [DO NOT CHANGE]
# ===================================

# create/empty the temporary directory on the compute node
if [ ! -d "${workdir}" ]; then
  mkdir -p "${workdir}"
else
  rm -rf "${workdir}"/*
fi

# change current directory to the location of the sbatch command
# ("submitdir" is somewhere in the home directory on the head node)
cd "${submitdir}"
# copy all files/folders in "submitdir" to "workdir"
# ("workdir" == temporary directory on the compute node)
cp -prf * ${workdir}
# change directory to the temporary directory on the compute-node
cd ${workdir}
```

(continues on next page)

(continued from previous page)

```
# 3. Function to transfer back to the head node [DO NOT CHANGE]
# ============================================================

# define clean-up function
function clean_up {
  # - remove log-file on the compute-node, to avoid the one created by SLURM is␣
↪overwritten
  rm job.slurm.out
  # - delete temporary files from the compute-node, before copying
  # rm -r ...
  # - change directory to the location of the sbatch command (on the head node)
  cd "${submitdir}"
  # - copy everything from the temporary directory on the compute-node
  cp -prf "${workdir}"/* .
  # - erase the temporary directory from the compute-node
  rm -rf "${workdir}"/*
  rm -rf "${workdir}"
  # - exit the script
  exit
}

# call "clean_up" function when this script exits, it is run even if SLURM cancels␣
↪the job
trap 'clean_up' EXIT

# 2. Execute [MODIFY COMPLETELY TO YOUR NEEDS]
# ============================================

# simplest example in the world, sleep a bit to allow a bit of monitoring
echo "hello world" > "test.log"
sleep 10
```

Note: To facilitate writing job-scripts, it can be created using the GooseSLURM Python module: [`source: writeJob.py`].

8.3.1 Code explained

Language selection

```
#!/bin/bash
```

Resource allocation

Definition of the language that this script is written in (BASH in this case).

```
#SBATCH --job-name tempdir
#SBATCH --out job.slurm.out
#SBATCH --nodes 1
#SBATCH --ntasks 1
#SBATCH --cpus-per-task 1
#SBATCH --partition debug
```

Allocation of resources. These lines are interpreted by the `sbatch` command, but are then ordinary comments when the script run on the compute node.

Directory selection

```
# I. Define directory names [DO NOT CHANGE]
# =========================================

# get name of the temporary directory working directory, physically on the compute-
↪node
workdir="${TMPDIR}"

# get submit directory
# (every file/folder below this directory is copied to the compute node)
submitdir="${SLURM_SUBMIT_DIR}"
```

Definition of:

- `workdir`: the temporary directory on the compute node, here taken from the `${TMPDIR}` provided by SLURM. Reading from and writing to the `workdir` is local on the compute node, and does not involve the cluster's internal network.
- `submitdir`: the directory from which the `sbatch` command is run. It is assumed that this is the simulation directory (`/home/user/.../simulation` above). All files/directory in this folder are copied back and forth. **Be sure to select the correct directory here.**

Copy to the compute-node

```
# 1. Transfer to node [DO NOT CHANGE]
# ===================================

# create/empty the temporary directory on the compute node
if [ ! -d "${workdir}" ]; then
  mkdir -p "${workdir}"
else
  rm -rf "${workdir}"/*
fi

# change current directory to the location of the sbatch command
# ("submitdir" is somewhere in the home directory on the head node)
cd "${submitdir}"
# copy all files/folders in "submitdir" to "workdir"
# ("workdir" == temporary directory on the compute node)
cp -prf * ${workdir}
# change directory to the temporary directory on the compute-node
cd ${workdir}
```

1. Optionally create or clear the temporary directory on the compute node (`workdir`).
2. Copy all files/directory in `submitdir` to the temporary directory on the compute node (over the cluster's internal network).

Copy back to the head-node (when the job finishes)

```
# 3. Function to transfer back to the head node [DO NOT CHANGE]
# ============================================================

# define clean-up function
function clean_up {
  # - remove log-file on the compute-node, to avoid the one created by SLURM is␣
↪overwritten
  rm job.slurm.out
  # - delete temporary files from the compute-node, before copying
  # rm -r ...
  # - change directory to the location of the sbatch command (on the head node)
  cd "${submitdir}"
  # - copy everything from the temporary directory on the compute-node
  cp -prf "${workdir}"/* .
  # - erase the temporary directory from the compute-node
  rm -rf "${workdir}"/*
  rm -rf "${workdir}"
  # - exit the script
  exit
}

# call "clean_up" function when this script exits, it is run even if SLURM cancels␣
↪the job
trap 'clean_up' EXIT
```

Define a function that will be run when the job ends (exits normally, or is voluntarily or involuntarily terminated by the queuing system). This function will copy everything (including all the generated results) back to the `submitdir`, which again involves the cluster's internal network. Note that this may overwrite files in `submitdir`.

If temporary files are created that you do not need anymore (for example build files, executables, debug output, ...) it is wise to delete it by uncommenting and modifying line 46. This way these files are created before copying them over the network.

Actual job

```
  # - erase the temporary directory from the compute-node
  rm -rf "${workdir}"/*
  rm -rf "${workdir}"
  # - exit the script
  exit
}

# call "clean_up" function when this script exits, it is run even if SLURM cancels␣
↪the job
trap 'clean_up' EXIT

# 2. Execute [MODIFY COMPLETELY TO YOUR NEEDS]
# ============================================

# simplest example in the world, sleep a bit to allow a bit of monitoring
echo "hello world" > "test.log"
sleep 10
```

Here you can do what you want. Remember that all read and write operations in the current directory (i.e. all files like `./somepath`) are local on the compute node (which is as efficient as reading and writing gets). Avoid here to do

anything involving the home folder, as that is a network mount.

CHAPTER 9

JSON log-file

This example demonstrated how to manually write a log-file in the `SLURM_SUBMIT_DIR` (the directory, on the head node, from which the job is submitted) when the job is submitted. This file, job.slurm.json, is formatted using JSON to facilitate (automatic) readability.

9.1 Job script

[source: job.slurm]

```
#!/bin/bash
#SBATCH --job-name json
#SBATCH --out job.slurm.out
#SBATCH --nodes 1
#SBATCH --ntasks 1
#SBATCH --cpus-per-task 1
#SBATCH --partition debug

# Write job info to a JSON-log file
# ================================

# get hostname
myhost=`hostname`

# write log file
cat <<EOF > job.slurm.json
{
  "hostname"            : "${myhost}",
  "SLURM_SUBMIT_DIR"    : "${SLURM_SUBMIT_DIR}",
  "SLURM_JOB_ID"        : "${SLURM_JOB_ID}",
  "SLURM_JOB_NODELIST"  : "${SLURM_JOB_NODELIST}",
  "SLURM_SUBMIT_HOST"   : "${SLURM_SUBMIT_HOST}",
  "SLURM_JOB_NUM_NODES" : "${SLURM_JOB_NUM_NODES}",
  "SLURM_CPUS_PER_TASK" : "${SLURM_CPUS_PER_TASK}"
```

(continues on next page)

(continued from previous page)

```
}
EOF

# Execute
# =======

# simplest example in the world, sleep a bit to allow a bit of monitoring
echo "hello world" > "test.log"
sleep 10
```

Note: To facilitate writing job-scripts, it can be created using the GooseSLURM Python module: [`source: writeJob.py`].

CHAPTER 10

Chang-log

10.1 v0.3.0

- Gsub: adding status using YAML-file, allowing submit from YAML-file (#24)

10.2 v0.2.0

- Bugfix: implementing –no-header options
- Adding delay to Gsub
- Adding tip to build once (using hardware optimization) (#22)
- Updating gitignore

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index