Journal of Data Science 24 (4), 985–1006
October 2026
DOI: 10.6339/24-JDS1159
Data Science Reviews

High Performance Computing Cluster Setup: A Tutorial

Marius Hofert[1]

[1]*The University of Hong Kong, Department of Statistics and Actuarial Science, Hong Kong*

Abstract

When computations such as statistical simulations need to be carried out on a high performance computing (HPC) cluster, typical questions arise among researchers or practitioners. How do I interact with a HPC cluster? Do I need to type a long host name and also a password on every single login or file transfer? Why does my locally working code not run anymore on the HPC cluster? How can I install the latest versions of software on a HPC cluster to match my local setup? How can I submit a job and monitor its progress? This tutorial provides answers to such questions with experiments on an example HPC cluster.

Keywords *cluster computing; connection; setup; Slurm Workload Manager; software installation; tips*

1 Introduction

Multi-core computations on central processing units (CPUs) of local machines such as laptops or desktop computers have recently provided quite some computational power to conduct statistical computations in parallel, while allowing us to use our own installed software or work environment with superuser rights. There are various scenarios under which it is required to adapt code to run larger-scale computations on a computer cluster, though. For example, if more CPUs are required for an embarrassingly parallel simulation, so a simulation that can easily be split into parallel tasks with minimal or no interaction necessary between the parallel task; a classical example in this regard is Monte Carlo simulation with large inputs. Or a cluster becomes necessary if each CPU's computation takes more run time than suitable for a laptop, be it because its power supply needs to be switched to battery for commuting or because other intensive tasks such as video editing need to be run, which can heavily affect performance and thus run-time measurements. Or, especially, if graphics processing units (GPUs) are required, for example for machine learning applications.

The step from one's own comfort zone (superuser rights, locally installed software and graphical user interface) to a remote, cluster-based one in the command line under limited permissions is often somewhat of a cultural shock and frustrating to work with for the first time, especially for students whose senior supervisors believe this should all work with the push of a button nowadays and that software development and administration bears no intellectual contribution, a common misunderstanding of the notion of scaling of software among people who rarely, if ever, program.

This tutorial aims to provide help how to get started and to provide insight what to watch out for. Although single topics addressed can be found in greater technical detail online (for example, e.g. HPC cluster websites, manual pages, Stack Exchange or Wikipedia pages; see also the references for more details), we aim for a broader and more readable introduction to the topic, bridging the gap between elementary concepts and technical manuals. The reader should

© 2026 The Author(s). Published by the School of Statistics and the Center for Applied Statistics, Renmin University of China. Open access article under the CC BY license.
Received May 26, 2024; Accepted October 30, 2024

be familiar with basic hardware concepts such as CPU and GPU, as well as basic shell commands, to be able to follow. We will think of computations as (CPU-intensive) embarrassingly parallel simulations (often of Monte Carlo type) or as (GPU-intensive) machine learning computations. As a disclaimer, one often encounters different problems on different HPC clusters. However, this tutorial can still be helpful for several reasons: First, the methods presented here were tested and in use at various locations over the past 15 years: At the (former) Brutus cluster of ETH Zurich in Switzerland, at LRZ of Technical University of Munich in Germany, at Compute Canada's Graham cluster in Canada and on Microsoft Azure and Google Cloud Platform servers. And second, the concepts behind the presented ideas did not change over the past 15 years (even though commands for achieving certain tasks or also software versions did change). As such, this tutorial can also act as inspiration for how to overcome certain computational hurdles one faces on the HPC cluster of choice.

In Section 2 we present necessary preliminaries when working on an HPC cluster. Section 3 addresses how to make command-line logins to a cluster significantly more convenient. Section 4 covers aspects of a user's home directory on a cluster such as shell setup files, using already installed software and how users can install their own software on the cluster. Section 5 addresses how the widely used Slurm Workload Manager software can be used to submit and monitor compute jobs on HPC clusters. Finally, Section 7 provides general tips when developing code for or work with HPC clusters. As a running example throughout this tutorial, an HPC cluster named HPC2021 (maintained by The University of Hong Kong) will be used.

2 Preliminaries

First, we need to get the permission to access the HPC cluster. As this is cluster-specific, we need to navigate to the cluster's website, find out the requirements for obtaining an account, and then apply for one. There are often two types of cluster accounts, group and individual accounts. The principal investigator (PI) of a research group (for example, a professor, typically also charged for the cluster usage of her/his group members) would apply for a group account, and a researcher in this group (for example, a PhD student) would apply for an individual account within this group. Note that if the PI also wants to run jobs on the cluster, (s)he typically also needs to apply for an individual account (under her/his own group account). As individual account name, we use `saas_mhofert` throughout this tutorial.

Second, we should find out the contact details of the cluster's support team in case help is required, for example for installing specific types of software; see also later. You should always read all available documentation first before bothering the support team, otherwise you will be the support's 'favorite' user in no time, without having run a single job on the cluster yet.

Third, it is important to familiarize oneself with the cluster's available hardware and software. Ideally both the hardware and the software used should be reported in publications, for reproducibility and comparability. The precise hardware we use must also be specified when run-time measurement of a job is crucial; otherwise different replications may be run on CPUs of different speeds, say. A total run time of 1 min on a single core of a two year old laptop has an entirely different meaning than on a fast GPU cluster; note in passing that oftentimes CPUs in newer laptops are actually more powerful than those used on clusters, but of course one has much less cores in a laptop than on a cluster (see also Section 6.3 later). Pre-installed software indicates what we can (and should) already rely on when running a job on the cluster; we will cover later how to install software not already available if needed. Important information

to know about a cluster also includes limits for group or individual accounts, such as maximal storage space in your home directory (here `diskquota` and other shell commands like `du -h` can be of help; on HPC2021 the limit is 100 GB per user) or the maximal run time of a job before it is canceled; on HPC the limit of CPU jobs up to 1024 cores or GPU jobs up to 4 GPUs is one week.

All these preliminaries are cluster-specific, so will not be considered in more detail in what follows.

3 Login to a Cluster

3.1 Establishing the First Connection

After our application for an account was granted, we can log in to the cluster. Most HPC clusters only provide a command line interface (CLI) access, so no graphical user interface (GUI) based on mouse support and a windowing system. To use the CLI, one typically uses a shell program; see Wikipedia (2024d). Unix-like operating systems (OSes) such as Linux or macOS already come with a terminal (the application) which runs a shell (the program) to execute commands. On Windows, the Windows Subsystem for Linux (WSL) provides a capable such tool nowadays. From now on, we assume a shell is available on our local laptop and an internet connection is established.

An HPC cluster typically consists of a *login node*, a computer one can log in to via the shell, in contrast to the actual *compute nodes* which will run computationally intensive jobs. The login node is reserved for login, basic file editing, compiling programs, running them for very short amounts of time (seconds) in order to check whether they work and do not miss any dependencies (other programs, software or input files) during run time, and, mainly, to submit the actual compute job. On HPC2021, the login node is reachable via `hpc2021.hku.hk` and for file transfer or data visualization, one should use `hpc2021-io1.hku.hk` or `hpc2021-io2.hku.hk`, so different nodes. We should thus be able to log in to HPC2021 by executing the following `ssh` (Secure Shell (SSH)) command in our local shell; for file transfer to or from the cluster, one would use `scp` (Secure Copy (SCP)) instead of `ssh` (see later for examples).

```
ssh mhofert@hpc2021.hku.hk
```

Now we run into our first problem. The command seems to hang. After a while, the error `ssh: connect to host hpc2021.hku.hk port 22: Operation timed out` appears. This happens, for example, when the HPC cluster requires a running virtual private network (VPN) connection. For security reasons, most clusters nowadays require a VPN connection unless one is already connected to the cluster's own wireless network, for example by connecting to a university's HPC cluster from on-campus via the university's provided wireless network. After having established the VPN connection, executing the above command and then typing our HPC account's password, we finally see the following greeting message from the login node.

```
******************************************************************************
Welcome to HPC2021 cluster (hpc2021.hku.hk)
------------------------------------------------------------------------------
This frontend node is reserved for program modification, compilation and job
queue submission/manipulation.

User Guide: https://hpc.hku.hk/hpc/hpc2021/userguide
```

```
Please acknowledge the support of HKU Research Computing facilities(HPC/HTC)
in any research report, journal, or publications. This information would be
important for us to acquire funding for new resources.

Our suggested acknowledgement for academic publications is:

The computations were performed using research computing facilities offered
by Information Technology Services, the University of Hong Kong.

For technical issues, you are welcomed to contact us at GROUP-ITS-HPC@hku.hk.

Lilian Chan
Information Technology Services, HKU                          1 July 2021
********************************************************************************
Activate the web console with: systemctl enable --now cockpit.socket

Last login: Sat May 18 08:55:59 2024 from 10.64.196.66
```

Via a VPN connection, a shell, our account name, the login node address and our password, we finally established a connection to the HPC cluster. Now that we are on the login node, we can already look a bit around, for example to check what OS this machine runs, so that we know which commands are available to us by default; in what follows we hide irrelevant output via `[...]` to save space.

```
cat /etc/*release
```

```
Rocky Linux release 8.7 (Green Obsidian)
[...]
```

As we see for HPC2021 here, most clusters use some variation of Linux as OS; see also Wikipedia (2024f).

3.2 Workflow Improvement

Besides local shortcuts to our VPN client and to our terminal application, we can improve the above workflow in at least two ways when connecting to a HPC cluster. First, we can define an SSH and SCP shortcut identifying the HPC cluster in order to not having to type its full address `hpc2021.hku.hk` (for SSH) or `hpc2021-io1.hku.hk` (for SCP) all the time. Second, we can set up an RSA key pair to avoid being asked for the cluster's password on every login or file transfer.

3.2.1 The `~/.ssh/config` File

A central role plays the directory `~/.ssh` on our local Unix-like OS; on Windows ~ is typically a folder of the form `C:\Users\your_username`. If it does not exist yet, we can generate it with `mkdir ~/.ssh`. The tilde indicates that `.ssh` is a subdirectory of our home directory. And the fact that this directory starts with a dot means it is hidden by default, so a simple directory listing command like `ls` in the home directory will not reveal it, only `ls -al` and other options of `ls` allow to reveal it. Let us see how `.ssh` looks like on our local machine.

```
cd ~/.ssh
ls -lh # list directories in long format and with more readable file size units
```

```
total 60K
-rw------- 1 mhofert staff 2.8K 2023-11-07 18:34 config
-rw------- 1 mhofert staff 1.8K 2022-11-24 12:38 id_rsa
-rw------- 1 mhofert staff  396 2022-11-24 12:38 id_rsa.pub
-rw------- 1 mhofert staff  24K 2024-05-16 13:07 known_hosts
```

As the name suggestions, the file `known_hosts` contains information about previous connections to other computers (known hosts). And we will come back to the two `id_rsa*` files soon. For now we consider the file `config`, which can contain configurations of computers we want to connect to.

```
cat config # shows the content of the file config
```

```
[...]
Host hpc
HostName hpc2021.hku.hk
User mhofert
ForwardX11 true
ForwardX11Trusted true
[...]
```

Here we learn how to define the shortcut `hpc` for the machine `hpc2021.hku.hk`; the provided `mhofert` is our username on HPC2021. With this setup, the following much shorter command already logs us in to HPC2021.

```
ssh hpc
```

A similar entry in `config` should be generated for file transfers to `hpc2021-io1.hku.hk` via `scp`; we choose `Host hpcio` as first line in the corresponding `config` entry and see a usage of this access later.

3.2.2 The ~/.ssh/id_rsa* Files

We are still asked for a password on every connection. To avoid this, we can generate an RSA key pair; see Wikipedia (2024c). It consists of a public key contained in the file ~`/.ssh/id_rsa.pub` and a private key contained in the file ~`/.ssh/id_rsa`. If it does not already exist, we can generate an RSA key pair as follows, where `myemail@myinstitution.com` should of course be replaced by your real email address.

```
ssh-keygen -t rsa -C "myemail@myinstitution.com" -b 4096
```

When executing this shell command and then following its instructions, users are often confused by the fact that after setting up the RSA key pair, they are *still* asked for a password on every login to the HPC cluster. This password is precisely the one that we provided when generating the RSA key pair (not the password of our HPC cluster account, unless the two are identical). So, when asked for a password on execution of the above `ssh-keygen` command, it is important *not* to provide any password (when prompted, just hit the return key) as providing one would make the setup of an RSA key pair obsolete for our purpose of a more convenient passwordless login in the first place. Using no RSA key pair password is still sufficiently safe; for more information about `ssh-keygen`, see `man ssh-keygen`, which also explains the options used in our `ssh-keygen` call before.

Now we locally generated the RSA key pair, but how do we tell the HPC cluster to use "it", and what is "it" actually here? For password-free access to the cluster, we need to save the

public part of the RSA key pair (the contents of the file ~/.ssh/id_rsa.pub) on the cluster, namely in the file ~/.ssh/authorized_keys on the cluster. For this we can use scp via hpcio.

```
scp ~/.ssh/id_rsa.pub hpcio:~/.ssh/authorized_keys # scp <what> <server>:<where>
```

If ~/.ssh/authorized_keys already exists on the cluster, this scp command overwrites the server's current authorized_keys. To append to the cluster's ~/.ssh/authorized_keys, just save the public key under a different file name on the server first and use a pre-installed text editor (nano and vi are typically available, the former being easy to use for basic tasks) to append your public key to ~/.ssh/authorized_keys. Alternatively, and often easier, one could use scp to download authorized_keys to our local machine (the corresponding scp command to copy authorized_keys from the server to our local home directory would be scp hpcio:~/.ssh/authorized_keys ~ executed from the local machine), modify the file locally under the convenience of a local text editor and then scp it back to the remote cluster. Note that during these operations, we are *still* prompted for the cluster's account password. Once the user's public key is saved under ~/.ssh/authorized_keys on the cluster, we can open a new shell on the local machine (important as otherwise the new settings may not become active) and try the following.

```
ssh hpc
```

This should log us in to the cluster without being prompted for a password (if not, something went wrong in the setup of the RSA key pair or we did not execute ssh hpc in a new shell). The same applies to file transfers with scp. We can also share the very same public key (or different public keys, as preferred) with more than one cluster or remote computer. Just our *private* key contained in the file ~/.ssh/id_rsa must *never* be shared with anybody and should remain in our local file ~/.ssh/id_rsa.

4 HPC Cluster Home Directory

Once we are on the cluster, we can also make our lives easier. In contrast to our own OS, on the cluster we are not *superuser* anymore, so we cannot install software on the cluster as we are used to on our local machine. However, we can define shortcuts, work with the already installed software or even install our own software (but only) in the user's home directory. These are the three points we address next.

4.1 The .bashrc File

One of the most important hidden shell configuration files on Linux clusters is ~/.bashrc; as already mentioned before, you can use ls -al to list all files in the current directory including hidden files. ~/.bashrc stands for "bash read command", with Bash being a popular type of shell program; another one is the Z shell for which there would be the configuration file .zshrc. It is read every time you open a new (Bash) shell on the cluster; one can typically use echo $0 to find out the shell we are running. After navigating to your home directory via cd (also the default directory you land in when logging in to the cluster), you can see the contents of .bashrc via cat .bashrc and modify this file with a text editor (such as nano or vi already mentioned before); alternatively scp the file to a directory of your local machine (but do not overwrite your own ~/.bashrc if you have one!), modify it there and scp it back to the server (this procedure is now much easier due to the more convenient login setup). Two important types of modifications one may want to add to .bashrc concern aliases introduced next and the environment variable PATH (introduced later).

We can define the following aliases in ~/.bashrc.

```
alias ls="ls -lh"
alias ll="ls -alh"
alias R="R --no-restore-history --no-save"
alias sb="sbatch"
alias sq="squeue -u $USER"
alias sc="scancel"
```

After saving ~/.bashrc and opening a new shell (for example, by exiting the cluster via `exit` and logging in again via `ssh hpc`), so that these newly defined aliases are known to the shell, we can simply type `ls` and obtain the output generated by `ls -lh`. Aliases can also be used to make the remote machine behave more like your local machine. The third alias will be useful later when we have installed the statistical software R (The R Foundation, 2024) as it avoids being prompted to store the workspace every time we quit R. The last three aliases are there for convenience when working with the Slurm Workload Manager (see later) to submit (via `sbatch` or its alias `sb`), monitor (via `squeue` or its alias `sq`) or cancel (via `scancel` or its alias `sc`) compute jobs.

4.2 Using Already Installed Software

Oftentimes, already installed software on the cluster can be accessed via the Environment Modules system; see Wikipedia (2024a). This software is available for roughly 30 years now and is used at some of the largest computer clusters. For example, executing the following command lists all available software on the cluster.

```
module avail
```

```
[...]
R/4.0.4
R/4.1.0_test
R/4.1.2-G
R/4.1.2-gcc
R/4.1.2-one
R/4.1.2
R/4.2.1
R/4.3.2
[...]
python/3.9.2
python/3.9.7
python/3.12.1
```

Here we list only a minor fraction of the available software, namely the different versions of R and Python available on HPC2021; one can also search available software via `module avail mysoftware`, for example.

However, typing `python --version` just generates an error.

```
python --version
```

```
bash: python: command not found...
```

This is because we first need to load the software for it to be available to us, which can be done as follows.

```
module load python/3.12.1
```

Now we can use this Python version.

```
python --version
```

```
Python 3.12.1
```

We can also unload software with the `module` command.

```
module unload python/3.12.1
python --version
```

```
bash: python: command not found...
```

These commands are important to know, since when we run a compute job on the cluster, the `module load` or `module unload` commands need to be part of the starter script (more on that later) so that the software is available to our program at run time.

It seems to be a good idea to put `module load` commands for frequently used software in `~/.bashrc`, however it is not. It is preferred and more fail-safe to load software only when needed by a job specifically in its starter script (and thus also in a single place). Otherwise you may have software loaded by default, but the starter script of your job loads software in clash with the already loaded one (for example different versions of R, etc.), which can produce cryptic error messages that can be hard to solve, especially if you are not aware anymore that your `~/.bashrc` already loads software (as such, the error would also not be reproducible if others on the cluster would submit your compute job). So only load software in one place, namely your compute job's starter script where you actually need it, which then also only contains the software necessary to run this particular compute job; similar to the good programming practice of defining variables in programs when they are needed, not all in the very beginning of the program.

4.3 Installing Your Own Software

Software available on a server is typically picked for its stability rather than whether it is the latest version. For example, at the time of writing in July 2024, the latest version of R available on HPC2021 already came out 2023-10-31. It is often required to run a program under the latest version of a software, such as R 4.4.1 (from 2024-06-14), a development version or even your own newly developed R package. As such we need to know how we can install such software on a cluster where we typically do not have the permission to write to `/usr/bin` or `/usr/local/bin`, the directories where software is normally installed in. We demonstrate this in detail in the Supplementary Material using R itself as an example (with related appearing issues) of a more complex piece of software and also demonstrate how to install contributed R packages. In what follows we assume this has been done.

5 Slurm Workload Manager

As we already mentioned, if you log in to HPC2021 with `ssh hpc`, you land on the so-called login node, on which you are only allowed to do basic tasks, in particular not running demanding programs of any sorts. Also, if this node was already a compute node of the HPC cluster, then you would waste others' resources (such as memory) by occupying the machine with installing

software. So how can you submit a job to run on a compute node then? Via a job scheduler. Popular job schedulers are Slurm Workload Manager (Slurm; Yoo et al., 2003 and Slurm Workload Manager, 2024), TORQUE (Adaptive Computing, 2024) and HTCondor (Thain et al., 2005; HTCondor, 2024). We focus on Slurm (which stands for Simple Linux Utility for Resource Management), the scheduler also available on HPC2021 and, according to Wikipedia (2024e), used by about 60% of the supercomputers on Wikipedia (2024f); see The National Radio Astronomy Observatory (2024) for help on how to translate various commands between these three job schedulers.

Slurm takes a shell script as input, say `starter.sh`, which specifies all required resources of your compute job (CPUs, GPUs, memory, maximal run time, etc.). Submitted to the job scheduler, your compute job specified in `starter.sh` then starts to run once the requested resources are available on the cluster; if not available yet, your job is pending until the requested resources are available.

5.1 The Starter Script

The following shows an example of a Slurm `starter.sh` shell script intended to start a job requiring GPU usage.

```
#!/bin/bash

## Job specification
#SBATCH --account=saas_mhofert
#SBATCH --job-name=myjob
#SBATCH --nodes=1 # number of compute nodes
#SBATCH --ntasks=1 # number of parallel computing tasks of your job
#SBATCH --cpus-per-task=1 # number of CPUs used by each task
#SBATCH --qos=gpu # quality of service limits ('gpu' = GPU, 'normal' = CPU)
#SBATCH --partition=gpu # request the GPU partition on the node(s)
#SBATCH --gres=gpu:1 # specifies 1 graphics card
#SBATCH --constraint="GPU_GEN:VLT" # specifies NVIDIA Volta V100 GPU(s)
#SBATCH --mem=32G # total amount of memory per node (units K/M/G/T)
#SBATCH --time=0-00:02:00 # max. wall-clock time in format D-HH:MM:SS
#SBATCH --output=%x_%j_stdout.out # standard output log with job name and job ID
#SBATCH --error=%x_%j_stderr.err # standard error log with job name and job ID
#SBATCH --mail-user=myemail@myinstitution.com # adapt to your email address
#SBATCH --mail-type=END,FAIL # can also be NONE, BEGIN, ALL

## Call Python
module load python/3.12.1 # load Python
python3.12 -c 'print("Hello, World!")' # typically 'python myjob.py' (batch mode)
module unload python/3.12.1 # unload Python

## Call Conda
module load anaconda # load Anaconda and provide 'source' command
source activate # provides 'activate' and activates default Conda environment 'base'
conda activate pytorch-gpu-2.1.0-cuda120 # Conda activates PyTorch virtual env
python --version # example call in the PyTorch environment (typically NN training)

## Call R script in batch mode
$HOME/soft/R/R CMD BATCH myscript.R
```

The first line specifies that `starter.sh` is a Bash script, which is also the reason for the file ending `.sh`. Next come the job options. They can vary a bit from cluster to cluster which would be addressed in the cluster's user guide; for HPC2021, see HPC (2024). We left short descriptions after each option, more details can be found in the Supplementary Material.

Next, the script `starter.sh` loads the latest Python module and then prints the string "Hello, World!"; as indicated in the script, we would typically have (more) Python code saved in a separate script, say `myjob.py`, and then run Python in batch mode. Note that just calling `python -c 'print("Hello, World!")'` or `python3 -c 'print("Hello, World!")'` would fail with `'-bash: syntax error near unexpected token `('` on HPC2021, using `python2` works, though. As already mentioned, these things are best sorted out with such a minimal reproducible example (MRE) as considered here (printing a string). After this call we explicitly unload Python here via `module unload python/3.12.1`. This is normally not required but we use it here for demonstration purposes, we come back to this point later.

Training of neural networks with PyTorch (The PyTorch Foundation, 2024) or TensorFlow (Abadi et al., 2024) requires running Python in a virtual environment. We consider PyTorch here. First, we load the Python distribution Anaconda (Anaconda Inc, 2024) to obtain access to the `source` command. We can then load the package manager and environment management system Conda (Conda, 2024) with the `activate` command, which also puts us in the default Conda environment `base`, so changes the prompt from `[mhofert@hpc2021 ~]$` to `(base) [mhofert@hpc2021 ~]$` when executed interactively in the shell. Second, we activate via Conda the PyTorch virtual environment that is already setup on HPC2021; see `conda env list` for available options on your cluster. Interactively, this changes the prompt from `(base) [mhofert@hpc2021 ~]$` to `(pytorch-gpu-2.1.0-cuda120) [mhofert@hpc2021 ~]$`. Finally, as a minimal reproducible example (Wikipedia, 2024b) call, we let Python print its version number.

In the last part of `starter.sh`, we call our previously installed version of R in batch mode to execute the following R script called `myscript.R`. It merely prints the version of R and the package version of `copula` we installed, with a 60 s wait time in-between.

```
getRversion()
Sys.sleep(60) # 1min wait time
packageVersion("copula")
```

The shell command `date` in `starter.sh` prints the start and end date of running `myscript.R`. It conveniently provides us with a way to assess the overall run time of the compute job. However, if you instruct Slurm to send you email notifications when the job ended, that email already reports used wall-clock time and memory used, from which it is easy to adjust the job's requested resources more accurately for the next run; see later. Finally, note that more accurate run-time measurements of specific function calls should be conducted tightly around the actual computations inside `myscript.R` and then be stored and reported as part of the computation results.

5.2 Submitting and Monitoring the Job

To keep our home directory clean, we generate a project directory `~/pro` and copy `starter.sh` and `myscript.R` there. Then we log in to HPC2021 and change to `pro`.

```
scp starter.sh myscript.R hpcio:~/pro
ssh hpc
```

```
cd ~/pro
```

We are now ready to submit our job specified in `starter.sh` to Slurm as follows; note that we use the previously defined alias `sb` for that.

```
sb starter.sh # submits starter.sh as a job to Slurm
```

With our alias `sq`, we can see the queue of our submitted jobs.

```
[mhofert@hpc2021 pro]$ sq
```

```
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           1656516       gpu myscript  mhofert  R       0:00      1 SPG-1-2
```

The `R` for "running" means that the job already started (otherwise it would typically show `PD` for "pending"). If we realized, for example, that the currently submitted job will fail (for example due to a syntax error), we would want to cancel it. This could then be done with our alias `sc` via `sc 1656516`, where the number is the job's identification number as shown in the output of `sq` before. In our MRE, we simply wait until the job is done, which can be seen from an empty queue.

```
sq
```

```
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
```

We then also receive an email containing the following.

```
Job ID: 1656516
Cluster: hpc2021
User/Group: mhofert/staff
State: COMPLETED (exit code 0)
Cores: 1
CPU Utilized: 00:00:01
CPU Efficiency: 1.59% of 00:01:03 core-walltime
Job Wall-clock time: 00:01:03
Memory Utilized: 43.11 MB
Memory Efficiency: 0.13% of 32.00 GB
```

If the job was terminated due to running out of time, `State: COMPLETED (exit code 0)` would be `State: TIMEOUT (exit code 0)`, for example.

Our example job produces three output files in ∼`/pro`, namely `myjob_1656516_stderr.err` , `myjob_1656516_stdout.out` and `myscript.Rout`. As there were no errors, the first is empty. The second, Slurm's file `myjob_1656516_stdout.out`, contains the following.

```
cat myjob_1656516_stdout.out
```

```
========================================
SLURM_JOB_ID        = 1656516
SLURM_NODELIST      = SPG-1-2
SLURM_NTASKS        = 1
SLURM_CPUS_PER_TASK = 1
This SLURM script is running on host SPG-1-2
Working directory is /home/mhofert/pro
========================================
Hello, World!
Python 3.10.13
```

We see information from Slurm about the job and then, at the bottom, the output produced by our Python calls in `starter.sh`; as one would normally call jobs in batch mode, the standard output file of Slurm would typically not contain job-related output. Something we can learn here is that "Hello, World!" was printed by Python 3.12.1 that we loaded (but then also unloaded) and the string "Python 3.10.13" shows that Anaconda's default Python version is 3.10.13. Also, as already mentioned, the file `myscript.Rout`, shown below, contains the output of the whole R session used when running `myscript.R`.

```
cat myscript.Rout
```

```
R version 4.4.1 (2024-06-14) -- "Race for Your Life"
Copyright (C) 2024 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> getRversion()
[1] `4.4.1'
> Sys.sleep(60) # 1min wait time
> packageVersion("copula")
[1] `1.1.3'
>
> proc.time()
   user  system elapsed
  0.118   0.030  60.231
```

In particular, we conveniently see all the input (content of `myscript.R`) and output together, including a time measurement; if one doesn't work with R, one could call any other shell command by prepending the shell command `time` which then provides a run-time measurement of the shell command. Note that `myscript.Rout` is created as soon as the job runs. If the job was killed because of running out of time, say, one can see in `myscript.Rout` how far the job got and can make adjustments. Also, as already mentioned before, if a longer job runs, one can use `cat myscript.Rout` to roughly see what it currently computes. From our `myscript.Rout` here we can see that both R and the package `copula` used were indeed those we installed previously.

Those are the basics. Of course, there are more involved commands to monitor jobs available, for example the following (again based on our aliases).

```
sc -u $USER # cancels all your jobs
sc -t PENDING -u $USER # cancels all your pending jobs
```

6 A Multi-core Computational Example

We now consider a rather minimal demonstration for how to run embarrassingly parallel computations on multiple cores in different ways on a HPC cluster, and also what we can learn from this demonstration about HPC; in particular the same R script can also be run locally for comparison. For a larger-scale example, see Hofert and Mächler (2016a) and the R package `simsalapar` (Hofert and Mächler, 2016b), or crank up the numbers in the below example.

As example, we use R's `mclapply()` function to parallelize multiple replications of estimating the exceedance probability $\mathbb{P}(\boldsymbol{X} > \boldsymbol{x})$ by Monte Carlo simulation, where $\boldsymbol{X}$ follows a multivariate Student's t distribution with ν degrees of freedom, zero location vector and where the scale matrix is an equicorrelation matrix $\boldsymbol{P}$. Such problems are typical in embarrassingly parallel computations when assessing the variance of estimators; one often includes multiple variance-reduced versions to compare their performance. As exceedance threshold $\boldsymbol{x}$ we use the 0.95-quantile of the (marginal) univariate standard Student's t distribution with ν degrees of freedom as x_j for each j.

6.1 The Job Script

We now describe the script `myscript.R` for this job in several parts to explain its purpose. We first attach the R packages `mvtnorm` (for simulating the multivariate Student's t) and `parallel` for `mclapply()`, a parallel version of `lapply()`. Then we define the function we call to estimate $\mathbb{P}(\boldsymbol{X} > \boldsymbol{x})$ by Monte Carlo and measure the corresponding run time.

```
library(mvtnorm)
library(parallel)

##' @title Estimating P(X > threshold) for X from a 2-Parametric Equicorrelated
##'        Multivariate t Distribution
##' @param n Monte Carlo sample size (integer > 0)
##' @param d dimension of X (integer >= 2)
##' @param rho number in [0,1] providing the off-diagonal entry of the
##'        equicorrelation matrix in the standardized multivariate t_nu
##'        distribution (location vector is the 0 vector)
##' @param nu degrees of freedom parameter (> 0)
##' @param threshold (numeric) threshold to consider exceedances of. Either
##'        of length d or 1 (in which case its value is repeated d times)
##' @return 2-vector containing the estimated probability and run time
##' @author Marius Hofert
tail_prob <- function(n, d, rho, nu, threshold = qt(0.95, df = nu))
{
    ## Checks
    if(length(threshold) == 1) threshold <- rep(threshold, d)
    stopifnot(n >= 1, d >= 2, 0 <= rho, rho <= 1, nu > 0, length(threshold) == d)

    ## Build ingredients
    threshold.mat <- rep(threshold, each = n) # d * n vector
    P <- matrix(rho, nrow = d, ncol = d) # correlation matrix of the multivariate t
    diag(P) <- 1 # force 1s on the diagonal

    ## Pseudo-random sampling
    start.time <- proc.time() # start watch
```

```
    X <- rmvt(n, sigma = P, df = nu) # sampling t_{\nu} with t_{\nu}(0,1) margins
    prob <- mean(rowSums(X > threshold.mat) == d) # estimate exceedance probability
    end.time <- proc.time() # stop watch
    rt <- (end.time - start.time)[["elapsed"]] # elapsed time in s

    ## Return
    c(prob = prob,  time = rt)
}
```

Next, we fix the parameters of the simulation and run it sequentially on our local laptop (Apple M1 Max with 10-core CPU, 32-core GPU, 64 G memory). This is still easily possible, the whole script runs locally in about 64 s (one can adjust the Monte Carlo sample size `n` to make it faster or slower if necessary).

```
## Setup
N <- 1000 # number of replications
n <- 1e5 # Monte Carlo sample size
d <- 7 # dimension
rho <- 0.5 # off-diagonal entry of the correlation matrix in the t_nu distribution
nu <- 3.5 # degrees of freedom
dimnms <- list("Type" = c("Probability", "Time"), "Replication" = 1:N) # for output

## Sequential simulation (with machine-dependent caching)
OS <- Sys.info()[["sysname"]] # OS type ("Darwin" = macOS; "Linux" = Linux)
file <- if(grepl("Darwin", OS)) { # file for storing result object
            "res_seq_sim_prob_local.rda"
        } else if(grepl("Linux", OS)) {
            "res_seq_sim_prob_HPC2021.rda"
        } else "res_seq_sim_prob.rda"
if(file.exists(file)) { # output file exists, so we load the result object from it
    load(file) # makes previously stored object 'res.seq' available
} else { # output file does not exist, so run the simulation and save it
    set.seed(271) # for reproducibility
    print(system.time(res.seq. <- lapply(1:N, function(N.) {
        tail_prob(n, d = d, rho = rho, nu = nu)
    })))
    res.seq <- array(unlist(res.seq.), dim = c(2, N), dimnames = dimnms)
    save(res.seq, file = file, compress = "xz") # save the computed result
}
```

As the sequential calculation is already taking a bit longer (about 42 s), we decide to cache its result as an `.rda` object. If the file `res_seq_sim_prob_local.rda` does not exist yet, the simulation (reproducible by the provided seed) is run, the run time is displayed (the additional `print()` is there so that the run time is displayed in `myscript.Rout` produced by `R CMD BATCH`) and the result is saved in `res_seq_sim_prob_local.rda`. And if `res_seq_sim_prob_local.rda` already exists, the previously saved object is loaded. Such conditional statements are helpful when long calculations are run (on a cluster or locally) to be able to quickly load them to analyze the result objects in a second run of the script (typically locally) afterwards. However, one should make sure that older result files are not lying around (for example, from test runs with much smaller Monte Carlo sample sizes, say, or before the code was changed somewhere) as executing the above code will then simply load the outdated result object then. Instead, just delete `res_seq_sim_prob_local.rda` first and then run the code again to produce new output

objects as intended.

In this rather small simulation, we also demonstrated how machine information via `Sys.info()` can be used to name the result object's file according to which machine the code ran on (in particular, on which OS). We could have, additionally, made the Monte Carlo sample size depend on this information, so that a much larger sample size could have been used on the cluster than locally; for pedagogic reasons in terms of comparability with the locally run results later, we do not do that here. Note that we still would only need to run the same (single) script, thanks to such conditional statements.

Now let us turn to a multi-core parallel version. We first detect the number of CPU cores, which is 10 on our laptop (each single threaded). We then proceed as we did sequentially, but with `lapply()` replaced by its multi-core version `mclapply()` with argument `mc.cores` specifying the number of cores to be used. When working with `mclapply()`, one decision one typically has to make is whether to work without load balancing (argument `mc.preschedule = TRUE`, the default) or with load balancing (argument `mc.preschedule = FALSE`). Without load balancing, the `N` replications are divided into (at most) `ncores`-many cores, which then start to work on the about `N/ncores`-many jobs. This is advantageous if all computations roughly take the same time (as is the case for our `N` replications here). With load balancing, the `ncores`-many cores start to work on the first `ncores`-many replications until the first core is done and is then fed with the next replication, etc. This is advantageous if the tasks (largely) differ in run time. Load balancing would be advantageous here if we parallelized over different dimensions rather than different replications. As this is not the case in our example, and to save space, we focus on the case without load balancing. As in the sequential case, we set a seed and then run the computations. We also run them a second time with the same seed and compare the results.

```
## Setup
(ncores <- detectCores()) # number of cores we use (all available)

## Run 1
set.seed(271) # to increase the chance of reproducibility (but not guaranteed)
system.time(res1. <- mclapply(1:N, function(N.) {
    tail_prob(n, d = d, rho = rho, nu = nu)
}, mc.cores = ncores)) # no load balancing (default)
res1 <- array(unlist(res1.), dim = c(2, N), dimnames = dimnms)

## Run 2
set.seed(271) # as before
system.time(res2. <- mclapply(1:N, function(N.) {
    tail_prob(n, d = d, rho = rho, nu = nu)
}, mc.cores = ncores))
res2 <- array(unlist(res2.), dim = c(2, N), dimnames = dimnms)

## Compare both runs
all.equal(res1["Probability",], res2["Probability",])
## => Not the same! set.seed() is not respected
```

Even though the two runs were produced with the same seed, the estimated exceedance probabilities differ; they also differ from the sequentially estimated exceedance probabilities (but all are equal in distribution, of course). In other words, `set.seed()` is not respected here (R's default random number generator Mersenne Twister is not suitable for parallel computations as one cannot efficiently compute sufficiently apart streams). To get reproducible results, one can

switch to L'Ecuyer's combined multiple recursive random number generator (CMRG), which (via `nextRNGStream()`) allows to generate sufficiently apart streams and thus allows reproducible parallelism. Conducting the same two runs again then provides us with equal (and thus reproducible) estimated exceedance probabilities; the same would apply if load balancing was specified.

```
## Switch RNG to CMRG
RNGkind("L'Ecuyer-CMRG")

## Run 1
set.seed(271)
system.time(res1. <- mclapply(1:N, function(N.) {
    tail_prob(n, d = d, rho = rho, nu = nu)
}, mc.cores = ncores))
res1 <- array(unlist(res1.), dim = c(2, N), dimnames = dimnms)

## Run 2
set.seed(271)
system.time(res2. <- mclapply(1:N, function(N.) {
    tail_prob(n, d = d, rho = rho, nu = nu)
}, mc.cores = ncores))
res2 <- array(unlist(res2.), dim = c(2, N), dimnames = dimnms)

## Compare both runs
all.equal(res1["Probability",], res2["Probability",])
## => Now the same!

## Switch RNG back to MT (R's default)
RNGkind("Mersenne-Twister")

## Session info
sessionInfo()
```

As done here, it is typically advisable to include `sessionInfo()` at the end of the R script. It prints the version of R, the OS and the packages used; see later for an example. This way one can also see this information in the `.Rout` file generated by `R CMD BATCH`, which is important for reproducibility in case future versions of R, the OS or packages prevent your code from running.

6.2 The Multi-core Starter Script

To start the computation, we use the following starter script, now with multi-core computations in mind. In comparison to our GPU-requesting previous `mystarter.sh`, this version of the starter script uses `--cpus-per-task=32`, `--qos=normal` paired with `--partition=intel` (specifying Dual Intel Xeon Gold 6226R CPUs with 32 cores and 192 G RAM) and omits `--gres=gpu:1` and `--constraint="GPU_GEN:VLT"`. For pedagogical reasons, we run the same `myscript.R` on HPC2021 (no altered input parameters) even though the local run of `myscript.R` was already sufficiently fast to solve this computational problem for our specifications.

```
#!/bin/bash

## Job specification
#SBATCH --account=saas_mhofert
```

```
#SBATCH --job-name=myjob
#SBATCH --nodes=1 # number of compute nodes
#SBATCH --ntasks=1 # number of parallel computing tasks of your job
#SBATCH --cpus-per-task=32 # number of CPUs used by each task
#SBATCH --qos=normal # quality of service limits ('gpu' = GPU, 'normal' = CPU)
#SBATCH --partition=intel # request the CPU partition on the node(s)
#SBATCH --mem=32G # total amount of memory per node (units K/M/G/T)
#SBATCH --time=0-00:02:00 # max. wall-clock time in format D-HH:MM:SS
#SBATCH --output=%x_%j_stdout.out # standard output log with job name and job ID
#SBATCH --error=%x_%j_stderr.err # standard error log with job name and job ID
#SBATCH --mail-user=myemail@myinstitution.com # adapt to your email address
#SBATCH --mail-type=END,FAIL # can also be NONE, BEGIN, ALL

## Call R script in batch mode
$HOME/soft/R/R CMD BATCH myscript.R
```

6.3 Interpretation of the Results

The actual computed probabilities or their individual run time measurements are not of particular importance here, so we focus on the obtained overall run time results (whole script, sequential computation part, parallel computation part) and related lessons to learn.

The following output is from the `.Rout` file generated by `R CMD BATCH` on HPC2021. It shows the output of `sessionInfo()` (including our version of R we installed and the version numbers of the contributed packages used). At the end we see the overall run time of the script, split in *user time* (CPU time spent on the user's R session), *system time* (CPU time spent by OS on behalf of the R session) and *elapsed time* (wall-clock time) in seconds; typically the user and elapsed time are of particular interest.

```
> sessionInfo()
R version 4.4.1 (2024-06-14)
Platform: x86_64-pc-linux-gnu
Running under: Rocky Linux 8.7 (Green Obsidian)

Matrix products: default
BLAS:   /home/mhofert/soft/R/R-4.4.1_build/lib/libRblas.so
LAPACK: /home/mhofert/soft/R/R-4.4.1_build/lib/libRlapack.so;  LAPACK version 3.12.0

locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

time zone: Asia/Hong_Kong
tzcode source: system (glibc)

attached base packages:
[1] parallel  stats     graphics  grDevices utils     datasets  methods
[8] base
```

```
other attached packages:
[1] mvtnorm_1.2-5

loaded via a namespace (and not attached):
[1] compiler_4.4.1
>
> proc.time()
   user  system elapsed
490.966  26.635  94.868
```

The overall run times for the local run were 230.798 s, 8.238 s and 63.742 s, respectively. In particular, we see that the local run was faster than the one on the cluster. To understand why, we have to consider the run times of the sequential and parallel parts.

For fixed machine (local Apple M1 Max or HPC2021) and our simulation setup, run times are almost identical across different runs and whether the Mersenne Twister or L'Ecuyer's CMRG are used as random number generators. For the parallelized runs, we thus only consider the run times obtained from the first run with L'Ecuyer's CMRG. The results are displayed in Table 1.

For the sequential run, we see that the elapsed (and user) times are much larger on HPC2021 than when run locally. As already mentioned in Section 2, this comes from the fact that a single Apple M1 Max core is faster than those used on HPC2021; HPC clusters mainly provide an advantage due to their sheer number of cores. Note in passing that chips (here: Apple M1 Max) nowadays come with efficiency (here: 2) and performance (here: 8) cores, so there are two notions of local cores. The first two cores displayed with the shell command `htop`, for example, are the efficiency cores and the sequential run of `myscript.R` triggered the usage of a performance core; several of the cores 3 to 10 were active during the run, switching between cores is common to avoid overheating, for example.

We also see from Table 1 that the parallel run on HPC2021 was indeed faster than that conducted locally, so a sufficiently large number of cores leads to a faster performance than locally (this would have been more impressive with a larger example and larger number of cores on HPC2021). However, the cluster run was not faster than the local one by a factor of $32/10 = 3.2$ (as would be expected by the difference in number of cores alone), which again can be attributed to the local cores being faster than the remote ones.

Comparing, for each machine, the elapsed times of the sequential with the parallel runs (factor $5.4/42.196 \approx 0.13$ for the local machine and $3.892/80.043 \approx 0.05$ for HPC2021) reveals that the run time improvements are not quite $1/10 = 0.1$ (local) and $1/32 \approx 0.03$ (HPC2021) as one would expect. This comes from the overhead in multi-core computing (communication between the cores, potentially switching of cores, etc.).

Furthermore, for sequential runs, the elapsed time is roughly the user time plus the system time. For parallel runs, however, the user time is much larger than the elapsed time. This comes from the fact that user time is CPU time spent and so the run times spent on all cores need to

Table 1: Run time measurements.

Run times (s)	Sequential `lapply()`			Parallel `mclapply()`		
Machine	User	System	Elapsed	User	System	Elapsed
Local (10 cores)	38.948	3.022	42.196	44.989	1.148	5.400
HPC2021 (32 cores)	78.585	1.286	80.043	109.419	6.011	3.892

be aggregated. More accurate for the local machine than HPC2021, we see that user time spent on the parallel computation is roughly 10 times (the number of cores) the elapsed time. We also see that the user time spent on the parallel computation is roughly equal to the elapsed time of the sequential computation, again in line with the argument just given.

7 General Tips

Besides the already provided tips for working on HPC clusters throughout this tutorial, we now close with some more general ones; see also Hofert and Schepsmeier (2016), Hofert and Schepsmeier (2017a) and Hofert and Schepsmeier (2017b) for other general tips.

1. Develop your code locally on your desktop or laptop in your preferred environment (OS, text editor, etc.). This is the most efficient way and you do not lose unsaved code or certain characters due to a dropped internet connection, for example.
2. Modify your script locally for running both locally and on the cluster. This way we can work with the same script and do not need to copy-paste code into a new R script just to adapt some lines to make it run on the cluster, which would require us to maintain and remember to update most of the code in two places (also, for scientific publications, oftentimes single reproducible scripts are required).
3. After development, test your code locally and try to run it locally with a small number of replications or small number of input parameters (a *pilot job*). In embarrassingly parallel computations with, say, 1000 replications, this number could be set very small, for example, to 2 to 5; note that 1 often leads to errors due to computed unbiased sample variances or, in R, due to subsetting a dimension of length 1 of a matrix leading to a vector (unless `drop = FALSE` is used).
4. Also use the locally run pilot study to already check whether the job produces the right outputs you would like to have, for example, are output files actually written, are they named correctly, etc. Here it is advisable to work with the ISO 8601 date format YYYY-MM-DD if dates are required, to replace spaces in output files by underscores or to add leading zeros in file names (for example via `sprintf("%02d", 1:10)` in R), so that files appear correctly ordered in your file system (without leading zeros, the order would typically be 1, 10, 2, 3, ...).
5. If scaling-up to the actual simulation in mind is not doable anymore run-time-wise (and only then), use an HPC cluster. However, do not try to submit a full-blown compute job right away. Instead, start with the aforementioned small/pilot job to check basic job specifications such as whether the job also runs and finishes properly on the cluster (for example, whether all dependencies are found), whether parts of the script need to be adjusted to work on both the HPC cluster and locally, etc.
6. Once the pilot job runs on the cluster, compare the job's local performance with that on the cluster. This allows one to assess the efficiency gain when running on the cluster. Also, if run time scales in a predictable manner, this often allows one to adjust the full-blown job's resource requirements (degree of parallelism, memory, run time) rather accurately (with some safety margin in mind, you do not want your job to be canceled after 999 replications).
7. Make sure that your code is set up to be convenient to run more than once. Oftentimes, supervisors, reviewers or readers of your work ask for different sample sizes or other changes once they see the results of the first larger run. Be prepared to having to run your code again at that stage (so automate your code as much as reasonable to simplify incorporating requested changes). Unlike the belief of many senior supervisors, this indeed requires some

thought, careful design and planning (some of the related ideas we have seen in this tutorial).

8. Once all previous points are fulfilled, we can run the full-blown job. If possible, `scp` all output files back to your local machine for analysis of the computed results. If done right, this can and should be done with the very same script, as mentioned before.
9. Always separate computation from visualization. Do not just produce plots as outputs of large computations, but first save the computed results and then analyze them (which includes creating plots). If we saved only the plots and wanted to change a tiny detail (such as line colors, for example), we would need to run the whole simulation again. Rather think about what are suitable output objects (typically arrays, lists or specific data objects such as `.rda` in R for example) and then save the computed results as those objects. Only then load them (locally) and create plots for the analysis. Remember, computations are expensive (cost energy and money), do not waste them (another reason for pilot studies). Another example comes from optimization in, say, R. Do not just save the argmax or argmin of your objective function passed on to `optim()` in R, save the whole return object as it contains important information such as whether the optimizer actually converged. Otherwise, if a significant proportion of your computed optima were actually not even local optima, your results and their interpretation may be off. If storing such results takes too much space, then make a careful selection of what information to store. One can also selectively store results, so store the results only if `optim()` did not converge (see the return value `convergence`), as those may be the runs you want to debug, etc. As such, ideally also make sure you know the seed for each run, so that you can directly reproduce replication 917 out of 1000 in case that this replication is one for which `optim()` fails.
10. A mix of top-down and bottom-up strategies is often the right way to go when implementing larger computations. First think about the bigger picture (top-down), what does the code get as inputs, what should it compute as outputs, what should the main functions involved compute, how can you structure and store the computed results, what do you want to produce from these results (types of graphics, their axes, curves displayed, etc.) and more. Once that is settled, start writing (bottom-up) the functions to help you compute the results. Document and test those functions while developing them (in line with the *divide-and-conquer* principle) instead of putting together thousands of lines of code without a single test run and then needing to find the – most likely many – needles in the haystack when trying to get that code to run. There is also an important psychological difference between these two approaches. If you start with smaller auxiliary functions, each tested after its implementation, then your code should be much more reliable in the end. In contrast, if you write all functions/code in one go without documentation or tests *and* your code actually runs (perhaps after fixing minor bugs) *and* produces somewhat expected results, you would not suspect anything going wrong. However, your code may still have many bugs you did not catch and you would also, at that point, not be too keen on finding as the obtained results seem reasonable and in line with what you expected them to be. You'd rather start analyzing the produced results. Similarly hard-to-find bugs are those that appear (randomly) only in specific simulation replications but remain undetected when aggregating results across all replications.
11. When contacting HPC cluster staff, be as short – but precise – as possible. Instead of "My code did not work" and sending thousands of lines of code (which will probably lead to no answer), do your own investigation first. Create a MRE by omitting everything unnecessary to reproduce your problem; a lot of problems can already be solved at this stage. Provide staff with this MRE, your account name, how you tried to run the example (the starter script) and short descriptions including outputs of anything you already tried to solve the

problem. Do not just paste code in the email, attach the actual scripts instead (sometimes staff also has access to your home folder on the cluster and can directly conduct tests there, possibly after asking for your permission first). And, as already mentioned, read all available documentation first before contacting the support staff.

12. When publishing results obtained from a cluster, cite the cluster (also acknowledge its staff in case they helped you), its hardware and, if important, also the software versions used. In order to keep this brief, something along the following lines seems reasonable given that a reproducible script `myscript.R` is also submitted: "The computations in this publication can be reproduced with the R script `myscript.R` which was run on a single NVIDIA Volta V100 GPU with 32 GB RAM as part of the HPC cluster HPC2021 offered by Information Technology Services, The University of Hong Kong." Also pay attention to how software is written. For example, R is not written as R but as the sans serif R.

Acknowledgement

I would like to thank Gan Yao (Department of Statistics and Actuarial Science, The University of Hong Kong) for testing the presented methods and more insights on Conda. I would also like to thank Haim Bar and the editor Jun Yan (both Department of Statistics, University of Connecticut) for valuable feedback on this paper.

Supplementary Material

The Supplementary Material contains detailed information on how R can be installed entirely in one's home directory without the permission to write to system directories. We also provide more details about the options specified in our `starter.sh` Bash script.

References

Abadi M, et al. (2024). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. URL: tensorflow.org.

Adaptive Computing (2024). TORQUE Resource Manager. URL: adaptivecomputing.com/cherry-services/torque-resource-manager.

Anaconda Inc (2024). The Operating System for AI. URL: anaconda.com.

Conda (2024). Conda. URL: docs.conda.io/en/latest/.

Hofert M, Mächler M (2016a). Parallel and other simulations in R made easy: An end-to-end study. *Journal of Statistical Software*, 69(4). https://doi.org/10.18637/jss.v069.i04

Hofert M, Mächler M (2016b). simsalapar: Tools for Simulation Studies in Parallel with R. CRAN.R-project.org/package=simsalapar.

Hofert M, Schepsmeier U (2016). Guidelines for statistical projects: General aspects (part I). *International Chinese Statistical Association Bulletin*, 28(2): 110–116.

Hofert M, Schepsmeier U (2017a). Guidelines for statistical projects: Coding and typography (part II). *International Chinese Statistical Association Bulletin*, 29(1): 52–58.

Hofert M, Schepsmeier U (2017b). Guidelines for statistical projects: Coding and typography (part III). *International Chinese Statistical Association Bulletin*, 29(2): 113–122.

HPC (2024). SLURM Job Scheduler. URL: hpc.hku.hk/guide/slurm-guide.

HTCondor (2024). HTCondor Software Suite. URL: htcondor.org.

Slurm Workload Manager (2024). Documentation. URL: slurm.schedmd.com.

Thain D, Tannenbaum T, Livny M (2005). Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2–4): 323–356. https://doi.org/10.1002/cpe.938

The National Radio Astronomy Observatory (2024). Translating between Torque, Slurm, and HTCondor. URL: info.nrao.edu/computing/guide/cluster-processing/appendix/translating-between-torque-htcondor-and-slurm.

The PyTorch Foundation (2024). PyTorch. URL: pytorch.org.

The R Foundation (2024). *The R Project for Statistical Computing*. URL: r-project.org.

Wikipedia (2024a). Environment Modules (software). URL: en.wikipedia.org/wiki/Environment_Modules_(software).

Wikipedia (2024b). Minimal reproducible example. URL: en.wikipedia.org/wiki/Minimal_reproducible_example.

Wikipedia (2024c). RSA (cryptosystem). URL: en.wikipedia.org/wiki/RSA_(cryptosystem).

Wikipedia (2024d). Shell (computing). URL: en.wikipedia.org/wiki/Shell_(computing).

Wikipedia (2024e). Slurm Workload Manager. URL: en.wikipedia.org/wiki/Slurm_Workload_Manager.

Wikipedia (2024f). TOP500. URL: en.wikipedia.org/wiki/TOP500.

Yoo AB, Jette MA, Grondona M (2003). SLURM: Simple Linux Utility for Resource Management. Lecture Notes in Computer Science, vol. 2862.