Journal of Data Science 0 (0), 1–9
2022 DOI: 0000

Supplementary Material for "High performance computing cluster setup: A tutorial"

Marius Hofert[1],*

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

Supplementary material 1: Installing your own software

We now demonstrate how to install R on a remote cluster where we typically do not have the permission to write to system directories.

Why

```
module load R/4.3.2
```

```
Lmod has detected the following error: Local R library directory "/home/mhofert/R_libs
    /4.3.2" does not exist. Please create
by command: mkdir -p /home/mhofert/R_libs/4.3.2
While processing the following module(s):
    Module fullname  Module Filename
    ---------------  ---------------
    R/4.3.2          /share1/modulefiles/Core/R/4.3.2.lua
```

As we can see, even the available R version cannot easily be loaded as the directory `/home/mhofert/R_libs/4.3.2` is missing. This is the so-called *version-dependent* library where R wants to install packages in. The problem with this approach is that every other version of R needs its own package versions, further cluttering our home directory. Also for such reasons can it be of interest to install one's own software, with a version-independent library of software packages.

In order to not clutter our home directory, we create a directory in which our own software should be installed, say `~/soft`, and in there a subdirectory for R.

```
cd
mkdir -p soft/R # directly generates ~/soft and, in there, ./R
```

The installation now (more or less) follows the classical steps on Unix-like OSes via `configure`, `make` and `make install`; see Wikipedia (2024a) and Wikipedia (2024b) for more information. For R we skip `make install` as we do not have the permission to write to system directories. We later demonstrate the installation of Texinfo with `make install`, which then requires to provide a location in which we have the permission to write.

Getting the sources

Source code of R can be found on CRAN Team (2024). Copy the link address of the R version you want to install, so for example right-click the version `R-4.4.1.tar.gz` and select "Copy Link Address" or the like. The link is of the form `https://cran.r-project.org/src/base/R-4/R-4`

* Email: mhofert@hku.hk.

.4.1.tar.gz. Now, on the cluster, download this compressed file and unpack it, for example as follows.

```
cd ~/soft/R
wget https://cran.r-project.org/src/base/R-4/R-4.4.1.tar.gz # download R sources
tar -xzf R-4.4.1.tar.gz # unpack the compressed file
mv R-4.4.1 R-4.4.1_source # move sources to a new directory (to keep them clean)
mkdir R-4.4.1_build # create a build directory in which R will be built
cd R-4.4.1_build # change to the build directory
```

The `configure` step

We are now ready to prepare the installation of R, the **`configure`** step. This step includes determining the configuration of the current machine (which compilers are available, which dependencies that R may need, etc.). If you have specific requirements for your version of R, you should add respective flags here; we add the flag **`--enable-R-shlib`** as an example to build R as a shared library.

```
../R-4.4.1_source/configure --enable-R-shlib # configure step
```

It is important to note that the configure step will fail if software that R needs for its installation is not found. In particular, on HPC2021 we obtain the following.

```
[...]
checking for shmat... yes
checking for IceConnectionNumber in -lICE... no
checking for X11/Intrinsic.h... no
configure: error: --with-x=yes (default) and X11 headers/libs are not available
```

So the configure step reports an error related to the windowing system X11 which is not available on the cluster. This is not a surprise as HPC2021 does not support GUI access. Instead, let us try to configure the R installation without X11 support since we do not need GUI access anyway.

```
../R-4.4.1_source/configure --enable-R-shlib --with-x=no # configure step
```

```
[...]
R is now configured for x86_64-pc-linux-gnu

  Source directory:            ../R-4.4.1_source
  Installation directory:      /usr/local

  C compiler:                  gcc  -g -O2
  Fortran fixed-form compiler: gfortran  -g -O2

  Default C++ compiler:        g++ -std=gnu++17  -g -O2
  C++11 compiler:              g++ -std=gnu++11  -g -O2
  C++14 compiler:              g++ -std=gnu++14  -g -O2
  C++17 compiler:              g++ -std=gnu++17  -g -O2
  C++20 compiler:
  C++23 compiler:
  Fortran free-form compiler:  gfortran  -g -O2
  Obj-C compiler:

  Interfaces supported:        tcltk
  External libraries:          pcre2, readline, curl
  Additional capabilities:     PNG, NLS, ICU
  Options enabled:             shared R library, shared BLAS, R profiling

  Capabilities skipped:        JPEG, TIFF, cairo
  Options not enabled:         memory profiling

  Recommended packages:        yes

configure: WARNING: you cannot build info or HTML versions of the R manuals
configure: WARNING: you cannot build PDF versions of the R manuals
configure: WARNING: you cannot build PDF versions of vignettes and help pages
```

Now the configure step finished; we address the three remaining warnings later. Note that if certain compilers are not found and are also not available via `module`, you can try to install them in your directory `~/soft` first, but you would then need to provide the exact installation directory of these compilers to the above `configure` command so that the R installation knows where to look for them. This can be a hassle. Alternatively, with such elementary and important tools as compilers, you can also contact the cluster's support in the hope that they can – with superuser access and in default directories – install such software for you. Oftentimes, dependencies are not necessarily required in their latest versions either and are also important for multiple users, so that should increase your chances that the cluster's support team is willing to help you out there.

The `make` step

As the configure step seems to have worked, we now find the generated `Makefile` in the current directory `~/soft/R/R-4.4.1_build`. We can use it to build R from its sources via the command `make`, so `make` turns R from its sources into an executable program (a binary) based on the specifications determined during the `configure` step.

```
make # make step (can take a while, just let it run until the end)
```

Basically, the R binary can now already be found, it is located in `~/soft/R/R-4.4.1_build/bin/R`. Additionally, we can run tests to check the installation.

```
make check # run checks
```

Creating PDF help files fails, though, as the following output shows.

```
make pdf # creating PDF help files
```

```
make[1]: Entering directory '/home/mhofert/soft/R/R-4.4.1_build/doc'
make[2]: Entering directory '/home/mhofert/soft/R/R-4.4.1_build/doc/manual'
ERROR: 'pdflatex' needed but missing on your system.
make[2]: *** [Makefile:270: fullrefman.pdf] Error 1
make[2]: Leaving directory '/home/mhofert/soft/R/R-4.4.1_build/doc/manual'
make[1]: *** [Makefile:175: pdf] Error 2
make[1]: Leaving directory '/home/mhofert/soft/R/R-4.4.1_build/doc'
make: [Makefile:295: pdf] Error 2 (ignored)
```

This is because the `pdflatex` command (to generate PDF files via LaTeX) is not found. Had we *first* executed `module load texlive/20220503` *before* the above `configure` step, `pdflatex` would have been found. Similarly for calling `make info` for creating R help files (for example the help page of optim when you call ?optim in an R session). However, on HPC2021, none of the three available Texinfo modules led to the required program texi2any to be found.

Installing a dependency

In this case, we install texinfo ourselves, say, in ~/soft/texinfo. The link to the latest source code (.tar.gz) can be found on GNU project (2024).

```
cd ~/soft
mkdir texinfo
cd texinfo
wget https://ftp.gnu.org/gnu/texinfo/texinfo-7.1.tar.gz
tar -xzf texinfo-7.1.tar.gz
mv texinfo-7.1 texinfo-7.1_source
mkdir texinfo-7.1_build
mkdir texinfo # to install texinfo in with 'make install'
cd texinfo-7.1_build
../texinfo-7.1_source/configure --prefix=/home/mhofert/soft/texinfo/texinfo
make
make install # installs in /home/mhofert/soft/texinfo/texinfo
```

Here we provide the configure command with a path where to install texinfo in when make install is called. Otherwise, make install will result in errors when trying to write to default system directories we do not have the permission to write to on the cluster (such as the already mentioned /usr/bin or /usr/local/bin). Is texi2any now found? Not quite yet.

```
texi2any
```

```
bash: texi2any: command not found...
```

The problem is that the Bash shell does not know that texi2any is located in ~/soft/texinfo /texinfo/bin/. How can we tell the Bash to look for software in this location?

The environment variable PATH

This can be done with the environment variable PATH, which contains a sequence of directories in which the Bash looks for (in this order) to find texi2any. In the shell, we can show the value of PATH as follows.

```
echo $PATH
```

```
/home/mhofert/.local/bin:/home/mhofert/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/
    usr/sbin:/share1/bin
```

We see that /home/mhofert/soft/texinfo/texinfo/bin/ is not part of PATH. We can add this directory to PATH by adding the following lines to ~/.bashrc.

```
PATH="$HOME/soft/texinfo/texinfo/bin:$PATH"
export PATH
```

In a new shell process so that ~/.bashrc is executed and PATH updated, we now obtain the updated PATH.

```
echo $PATH
```

```
/home/mhofert/soft/texinfo/texinfo/bin:/home/mhofert/.local/bin:/home/mhofert/bin:/usr
    /local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/share1/bin
```

We see that `/home/mhofert/soft/texinfo/texinfo/bin` was prepended to the former value of `PATH`. Even if `texi2any` was already available on the machine, the *first* match in this list of directories in `PATH` is the version of `texi2any` that will be executed then, so this is the version that we installed. As `texi2any` was not already found in any other directory of `PATH`, the order of the directories listed in `PATH` does not matter in this example, but it may matter in case you need to work with a more up to date version of a software that is already available on the machine. As the following line demonstrates, our installation of `texi2any` is now indeed found.

```
texi2any --version
```

```
texi2any (GNU texinfo) 7.1
[...]
```

Finishing the installation of R

With `texi2any` now available we can load `pdflatex` via `module load` and then configure R again.

```
module load texlive/20220503
cd ~/soft/R/R-4.4.1_build
../R-4.4.1_source/configure --enable-R-shlib --with-x=no
```

We now obtain the same output as before (omitted here), just without the three warnings at the end. We can thus continue with the `make` commands, which now all run flawlessly.

```
make
make check
make pdf
make info
```

In contrast to `texinfo` before, for installing R we omitted `make install` as we technically do not need it. The executables `R` and `Rscript` can be found in `/home/mhofert/soft/R/R-4.4.1_build/bin` and we set symbolic links to them from ∼`/sort/R` as follows.

```
cd ~/soft/R
ln -s ~/soft/R/R-4.4.1_build/bin/R R-4.4.1 # now R can be called via ./R-4.4.1
ln -s ~/soft/R/R-4.4.1_build/bin/R R # now R points to the latest version of R
ln -s ~/soft/R/R-4.4.1_build/bin/Rscript Rscript # makes 'Rscript' available
```

The first of the three `ln -s` commands is not needed, but if additional versions of R are installed (for example, for testing purposes), one can call each by using the respective `R-*` command; the symbolic link `R` itself we always use to point to the current default R version we work with (if there are several).

Also note that `Rscript` and `R CMD BATCH` both allow to run R scripts in batch mode (non-interactively), which is what we need for compute jobs. `Rscript` behaves more like a Unix command in that it writes output to the shell. `R CMD BATCH` is preferred for larger compute jobs. If our code is in `myscript.R` and we run it with `R CMD BATCH myscript.R`, then `R CMD BATCH` creates the file `myscript.Rout` that contains the output of the R session that runs `myscript.R` as soon as the job starts. We can then check the output `myscript.R` generated in `myscript.Rout` after the job finished, but we can even use `cat myscript.Rout` to monitor our job during run time (to roughly determine where it is at).

Let us call R now.

```
~/soft/R/R
```

```
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.
```

In the starter script of our compute job we will discuss later, we can call R via `~/soft/R/R`. This guarantees that our own installation of R is found and used. We also add `PATH="$HOME/soft/R:$PATH"` to `~/.bashrc` to guarantee that this version of R is found from anywhere in the shell (as before, only in a newly opened shell).

Version-independent library

As mentioned, R itself consists base and recommended packages. To install a contributed package `mypackage` from within R we can use `install.packages("mypackage")`. Where does R install and find these packages? In the above setup, this would be `/home/mhofert/soft/R/R-4.4.1_build/library` which can be seen when opening R and executing `.libPaths()`. The problem is that this is a version-dependent library, so with every new version of R we install and use, we would need to install the (possibly unchanged) packages in such a version-dependent library. To create a version-independent library of contributed packages, we first generate a directory.

```
~/soft/R
mkdir library
```

R needs to be informed to install contributed packages in this directory by default when we call `install.packages()`. This information is specified in the environment variable `R_LIBS_USER` that we can define in `~/.Renviron` as follows (obviously, as before, adjust the provided directory path to match your home directory, `pwd` for printing your current working directory can be helpful in this regard).

```
R_LIBS_USER=/home/mhofert/soft/R/library # version-independent library
```

If you now open a new R session and call `.libPaths()` you should see two paths, the first being `/home/mhofert/soft/R/library` and the second being `/home/mhofert/soft/R/R-4.4.1_build/library`. This order is important as R will use the first of the two as default installation directory for contributed packages if it exists (and has permissions to write to), which is just what we want.

Further useful settings in ~/.Renviron are the following; note that the already defined alias in ~/.bashrc to run R with --no-restore-history --no-save only applies to R being run interactively, the below R_BATCH_OPTIONS variable applies to the case where R is run in batch mode.

```
R_BATCH_OPTIONS=--no-restore-history --no-save # do not ask whether to save on q()
R_ENCODING_LOCALES="UTF-8=en_US.UTF-8" # specify default locale
LANGUAGE=en_US.UTF-8 # specify default language
```

Let us now install a contributed R package, say copula; see Hofert et al. (2020). Within an R session, we do the following.

```
install.packages("copula")
```

```
[...]
** byte-compile and prepare package for lazy loading
Error in dyn.load(file, DLLpath = DLLpath, ...) :
  unable to load shared object '/home/mhofert/soft/R/library/gsl/libs/gsl.so':
  libgsl.so.25: cannot open shared object file: No such file or directory
Calls: <Anonymous> ... asNamespace -> loadNamespace -> library.dynam -> dyn.load
Execution halted
ERROR: lazy loading failed for package ‘’copula
* removing ‘/home/mhofert/soft/R/library/’copula
* restoring previous ‘/home/mhofert/soft/R/library/’copula

The downloaded source packages are in‘
    /tmp/RtmpCvqkPq/’downloaded_packages
Warning message:
In install.packages("copula") :
  installation of package ‘’copula had non-zero exit status
```

Clearly, the R package gsl is missing. After loading gsl via module load gsl/gcc/2.7.1, we first install the R package gsl via install.packages("gsl") and then install copula again via install.packages("copula"), which now works flawlessly.

```
install.packages("copula")
```

```
[...]
* DONE (copula)
[...]
```

Default mirror

Finally, note that every time you install an R package, you are prompted for the CRAN mirror the contributed package is installed to. This can be avoided by specifying a corresponding option in the file ~/.Rprofile. The following line specifies that contributed packages are always downloaded from https://cran.r-project.org, so for example https://cran.r-project.org/src/contrib/copula_1.1-3.tar.gz for the version of copula we installed before.

```
options(repos = c(CRAN="https://cran.r-project.org"))
```

Supplementary Material 2: Detailed explanation of the job options in the starter script

In what follows, we provide more details about the options in our starter script:

`--account=saas_mhofert` Specifies the PI's account (here: `saas_mhofert`); typically this account is charged for the resources used.

`--job-name=myjob` The basename of the job (here: myjob).

`--nodes=1` The number of compute nodes your job should be run on (here: 1).

`--ntasks=1` The number of parallel computing tasks of your job (here: 1), that is sub-jobs of a job run on a single node. One can also combine `--nodes` with `--ntasks-per-node` to guarantee a certain number of tasks be run per node.

`--cpus-per-task=1` The number of CPUs used per task (here: 1).

`--qos=gpu` Specifies the 'quality of service', a set of limits that apply to the job on HPC2021. For example, `normal` (the default) implies up to 1024 cores for up to one week of run time on HPC2021, `gpu` implies maximal one node with 4 GPUs for up to one week of run time. Note that this parameter most likely differs for different HPC systems.

`--partition=gpu` Specifies the type of compute nodes where the job is to be executed (here: on GPUs).

`--gres=gpu:1` For GPU usage ("generic resource"), this option specifies that the job is run on one graphics card.

`--constraint="GPU_GEN:VLT"` Specifies that the job is only run on NVIDIA Volta V100 GPU(s).

`--mem=32G` Amount of memory per node (here: 32 GB).

`--time=0-00:02:00` Maximal number of wall-clock time your job will run (here: 2 min), for example `1-06:30:00` specifies 1 day, 6 hours and 30 minutes. The chosen wall-clock time should be sufficiently large to allow your job to finish, but not much larger as your job may then have to wait longer until it starts to run since Slurm may have a harder time finding a suitable slot for your job if the machine is busy. It is generally a good idea to run a smaller simulation first, just to see if the job runs without syntax errors, files not found, etc. Submitting a several day job and waiting for more than a week until it starts, just to fail within seconds because of a syntax error, say, is lost time.

`--output=%x_%j_stdout.out` Name of the standard output log of your job. In this file Slurm will write output about your job. Note that this is not the output that your actual compute job writes, it is rather what Slurm sees your compute job is doing/not doing. For example, if your job is killed because it ran longer than the wall-clock time specified with `--time`, then Slurm will report on that in this output file (the same applies if your job runs out of the requested memory, say). Note that `%x` will be replaced by the job's name and `%j` by the job's ID, so that one can get output files for each combination of these two job specifications.

`--error=%x_%j_stderr.err` Name of the standard error log file of your job.

`--mail-type=END,FAIL` This option allows you to specify when email notifications are sent (here: when your compute job finished or failed to run).

`--mail-user=myemail@myinstitution.com` Your email address to send email notifications to (obviously, adapt to yours).

References

CRAN Team (2024). The Comprehensive R Archive Network. URL: `cran.r-project.org`.

GNU project (2024). Index of /gnu/texinfo. URL: `ftp.gnu.org/gnu/texinfo`.

Hofert M, Kojadinovic I, Mächler M, Yan J (2020). *copula: Multivariate Dependence with Copulas*. R package version 1.0.0.

Wikipedia (2024a). configure script. URL: `en.wikipedia.org/wiki/Configure_script`.

Wikipedia (2024b). Make (software). URL: `en.wikipedia.org/wiki/Make_(software)`.