

1 Exercises Set 1: Serial Programming

Exercise code files, course slides and this exercise sheet are available for download at <https://git.rwth-aachen.de/hkhlr/using-r-on-hpc>

1.1 Package installation

Installing and using packages is fundamental for working with R.

1. Make sure all necessary environment modules are loaded. E.g. on the MaRC3a Cluster in Marburg, we would do the following to load the default gcc, openmpi, and R modules into a clean environment:

```
module purge
module load gnu9 openmpi4 R
module list
```

```
1 Derzeit geladene Module:
2 1) gnu9/9.4.0          5) openmpi4/4.1.1 9) gdal-hdf5/3.5.2
3 2) hwloc/2.5.0         6) hdf5/1.10.8   10) openblas/0.3.7
4 3) ucx/1.11.2          7) proj/9.1.0   11) R/4.1.2
5 4) libfabric/1.13.0    8) geos/3.11.0
```

2. Simply type 'R' to start R's interactive mode. You should see a start message that lists version, license information, and some hints to get started.
3. We will now install some packages that are necessary for later exercises. Note that **R package names (and everything else in R) are case sensitive!** To start, type:

```
install.packages(c("microbenchmark","foreach","doParallel",
"snow","pryr"))
```

4. You will be asked if you want to use a personal library and if you want to create that library now. Answer both questions by typing 'yes'.
5. Then, you will have to choose a download mirror. Preferably choose one that is close to your location (Göttingen or Erlangen for example). If the download should fail, it is always a good idea to try another mirror before searching for an error elsewhere.
6. The last package, Rmpi, needs an additional parameter to link to the correct MPI version:

```
install.packages("Rmpi",configure.args="--with-
mpi=/opt/ohpc/pub/mpi/openmpi4-gnu9/4.1.1/")
```

The paths displayed here are dependent on the system or cluster you are working on!

To find the location of the currently loaded MPI module, you can for example try:

```
which mpicc
```

On MaRC3a, the output will look like this:

```
/opt/ohpc/pub/mpi/openmpi4-gnu9/4.1.1/bin/mpicc
```

7. You should now be able to load and use all of the above packages by typing

```
library("<package_name>")
```

8. If you want to quit R, type "q()". You can choose whether to save your workspace environment or discard it.

1.2 Microbenchmark

When trying to write well-performing R code, the "microbenchmark" package offers a great interface to compare different implementations of the same functionality.

1. Change into the exercise directory:

```
cd exercises/microbenchmark
```

2. Open the file `growing_objects.R` in an editor:

```
nano growing_objects.R
```

3. Take note of the `check()` function, which uses the built-in functionality of `microbenchmark` to keep a list of all results for comparison.
4. Close the file by hitting CTRL+X.

5. Execute the script from the commandline with

```
Rscript growing_objects.R
```

or simply

```
./growing_objects.R
```

(this is possible because of the file has been made executable with `chmod u+x <file_name>` and includes the "shebang" line `#!/usr/bin/env Rscript`).

6. Change the values for `vec_length` and the `times` parameter and observe how the output changes (do not set the numbers too high, though!).

1.3 Vectorized Functions

Using vectorized functions is essential for writing well-performing and maintainable R code.

1. Change into the exercise directory:

```
cd exercises/vectorized_functions
```

2. Open the file `vecsum.R` in an editor:

```
nano vecsum.R
```

3. Inspect the code, then close the file and run the script. Check that both functions return the same result.
4. Modify the script so that `vec_2` is only half the length of `vec_1`.
5. Run the script again. In the direct sum function, you can see vector recycling in action; `vec_2`'s contents are automatically repeated so that its length matches `vec_1`.
6. By yourself: Try to modify the script into a microbenchmark for comparing the two functions `vecsum_loop()` and `vecsum_direct()`. Use the `growing_objects.R` script from the previous exercise as a template (implementing the `check()` function is optional).

1.4 Saving and Loading Objects

For this exercise, we will use the interactive shell of R.

1. On the command line, enter 'R' to start the interactive shell and load the 'pryr' package.

```
R  
(...some lines of text from R startup...)  
library("pryr")
```

2. We can now check the currently used memory of our R environment:

```
mem_used()
```

or

```
gc()
```

3. Now, we will create two large matrices and check their memory footprint.

```
m_1 <- matrix(rnorm(1000000),nrow=1000)
m_2 <- matrix(rnorm(1000000),nrow=1000)
mem_used()
```

4. If we want to see all currently active objects, we can use `ls()`

```
ls()
```

Take the time in between the following steps using `ls()` and `mem_used()` to observe how the environment changes.

5. We will now save `m_1` as an RDS file and then throw away the object:

```
saveRDS(m_1,file="m_1.rds")
rm(m_1)
```

6. Then, we load it back into a new variable:

```
m_3 <- readRDS(file="m_1.rds")
```

7. To save multiple objects in an Rdata file, we do:

```
save(m_2,m_3,file="objects.Rdata")
```

8. We remove the objects from the environment and load them again:

```
rm(m_2,m_3)
load(file="objects.Rdata")
```

Take note that `load()` does not need a variable, but just dumps the objects back into the environment with the names they had before.

9. Once you quit the R interactive shell with `q()`, take note of the two created files `m_1.rds` and `objects.Rdata`. These are a persistent store of your objects and may be moved, copied, and opened elsewhere.

1.5 Running R Jobs on a Cluster

In this exercise, we will look at submitting R jobs to a cluster scheduler and also how to set and read R script parameters.

1. Change into the exercise directory:

```
cd exercises/submit_serial
```

2. Open the file `simple_R_job.sh`.

```
nano simple_R_job.sh
```

3. This is a simple, but typical SLURM submit script. Take note that we load the same modules that we also loaded to work with R locally. The executed code is the same as the one from the "Microbenchmark" exercise. Submit it with:

```
sbatch simple_R_job.sh
```

4. You can check on your job's progress with the `squeue` command.
5. After your job has finished, look inside the exercise directory; there should be files called `simple_R_job.<jobID>.err` and `simple_R_job.<jobID>.out`. Inspect these files using `nano` or `less`.
6. What if we wanted to do several tests with different vector lengths? The modified code in `growing_objects_param.R` takes a command-line argument that sets the vector length and saves the benchmark results into an RDS file.
7. We could submit several jobs for different lengths, or we could submit a task array. The submit script `tasked_R_job.sh` does the latter and will emit 4 tasks that will run the benchmark for lengths 4000, 6000, 8000, and 10000.

- Inspect the input files, then submit the task array:

```
sbatch tasked_R_job.sh
```

- Check the job output and verify all files have been created. You could now, at any time, read these objects back into an R script, for example in a following job that further operates on and/or combines the results. **Note: To get back the original formatting, the "microbenchmark" package has to be loaded when reading the objects.**
- With the information on how to read RDS files from the previous exercise, can you write an R script that displays all results from the RDS files produced by the task array? Bonus points if you can do it **without** hard-coding the file names!

2 Exercises Set 2: Parallel Programming

2.1 Using mclapply

The `mclapply` and `mcmapply` functions are the easiest way to parallelize an R function on Linux.

- Change into the exercise directory:

```
cd exercises/mclapply
```

- Open the file `mclapply.R` and compare the serial and parallel function calls.
- There is also a SLURM submit script called `mclapply_R_job.sh`, which will run the R code on the cluster. Take a look at its content and then submit it:

```
sbatch mclapply_R_job.sh
```

- Check the job output and verify that the parallel routine was faster.
- Augment the code in `mclapply.R` so that the amount of CPU cores can be set as a script parameter (e.g. so that the command `Rscript mclapply.R 8` would use 8 CPU cores for `mclapply`). Use the file `growing_objects_param.R` from the previous exercise as a template on how to handle command-line parameters in R.

2.2 Using the R Cluster Functions

The `cluster` function family offers a flexible interface to parallelize R code.

- Change into the exercise directory:

```
cd exercises/cluster
```

- This is the same function as in the `mclapply` exercise, but this time, we will use both `parLapply` and `foreach` to parallelize it. Check the content of the file `cluster_1.R` and take note of the additional function calls.

- We will first submit the job by using the provided job script:

```
sbatch cluster_1_R_job.sh
```

- `Foreach` seems to be significantly slower than `parLapply`. On the slides, there is a hint on what is the issue here.
- Splitting the input in as much chunks as there are cores can be done via `clusterSplit()`. Add an additional entry to the microbenchmark where `foreach` splits the input into appropriate chunks. Caution: The function call also has to be modified (why?).
- We can also add a call to `mclapply` to the microbenchmark, to compare the two methods.
- `foreach` can be much more flexible than `*apply` since it can execute whole code blocks (including variable assignments, if-else-constructs, loops, etc.)! We can demonstrate that by implementing the code of `boot_fx()` directly in `foreach` instead of calling it.
- We have to add some more code if we want to process whole chunks of data (i.e. using `clusterSplit()`), since the original implementation of `boot_fx()` only processes a single value on each call!

2.3 Using RMPISNOW

The `snow` package for R includes a parallel function interface for MPI that shares most functionality with the `cluster` package that we have already seen. Calling conventions differ a bit, since `snow` includes a framework of scripts that takes care of abstracting away the technicalities of properly starting R from within an MPI context.

1. Change into the exercise directory:

```
cd exercises/Rmpi
```

2. Look at the files `rmpisnow.R` and `rmpisnow_R_job.sh`. Note the differences to the "cluster.R" example from the previous exercise, especially how the R processes are started through `mpiexec`. We also use `--no-echo` here because RMPISNOW does *not* use `Rscript` and would otherwise echo all code lines to output.
3. The advantage of using MPI compared to the FORK cluster from the previous exercise is that we are no longer limited to a single node. Whether our code and problem size is scaling so well that it's worth using more than a single node, however, is a different question!
4. We will submit the job to the cluster by using the provided script:

```
sbatch rmpisnow_R_job.sh
```

5. Once the job has completed, check its output to confirm that the parallel call completed faster than the serial version.

2.4 An Rmpi Wrapper Example

We will look at an example of how the RMPI Wrapper works. MPI is a big and complex topic that cannot be covered fully in this course; however, `Rmpi` allows that already obtained or newly acquired skills in using MPI's C interface can be implemented almost one-to-one in R code.

1. Change into the exercise directory:

```
cd exercises/Rmpi
```

2. The MPI code is found in the file `mpi.R`. Open the file and try to figure out what the code will do. Keep in mind that all processes will execute the same code, the only difference is their rank number (from 0 to `<number_of_processes>-1`).
3. There is also a SLURM submit script named `mpi_R_job.sh`. Open it and check how the R code is invoked with `mpiexec`.
4. Submit the job to the cluster, and check the output once it has finished.

```
sbatch mpi_R_job.sh
```

5. In the `.err` file, you'll find an (approximate) timing for the run. Most likely, it will be a bit slower than the other parallel methods. What could be the reasons? What could be done to improve performance?
6. The code in `mpi.R` is a naive implementation meant to showcase MPI's most basic point-to-point communication. A more 'serious' version that uses appropriate collective communications (in this case `scatter` and `gather`) can be found in the file `mpi_collective.R`. It has an associated job script that we can run:

```
sbatch mpi_R_job_collective.sh
```

7. The output of `mpi_collective.R` is transformed (additional one-level `unlist()`) so that it has the same format as the output from `mpi.R`. Thus, you can verify that both implementations have the same result by using the shell command `diff`.

Online Resources

- "The R Inferno" (in-depth analysis of R programming pitfalls):
https://www.burns-stat.com/pages/Tutor/R_inferno.pdf

- “R-Bloggers” (great articles about R parallelization):
<https://www.r-bloggers.com>
- The R Manual (online R package help files):
<https://stat.ethz.ch/R-manual/R-devel/doc/html/index.html>
- SLURM documentation:
<https://slurm.schedmd.com>