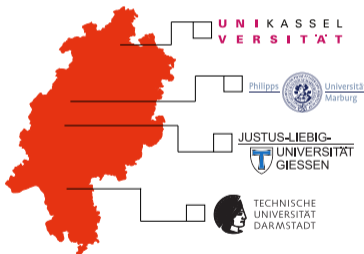


Using R on HPC Systems

Hessisches Kompetenzzentrum für Hochleistungsrechnen (HKHLR)

René Sitt

25.02.2025



HKHLR is funded by the Hessian Ministry of Sciences and Arts



Today's Agenda

First Part: Serial Performance and Language Features

- ▶ R Paradigms and Pitfalls
- ▶ Vectorized Functions
- ▶ Memory Management
- ▶ Using R in a Cluster Environment

Second Part: Parallel Programming with R

- ▶ General Hints on Parallel R
- ▶ Multicore Functions
- ▶ Cluster Functions
- ▶ Rmpi

Motivation: R and HPC

Reasons for using R

- ▶ Rich package ecosystem
- ▶ Strong support especially for statistical analysis, machine learning, data visualization
- ▶ Dynamic and flexible language
- ▶ Fast development cycles due to being a script language
- ▶ Powerful IDE options available (RStudio, Jupyter)
- ▶ **Question to the audience:** Why do *you* use R, and what do you use it for?



Reasons for using HPC Clusters

- ▶ Provides hardware resources far beyond workstation capabilities (Memory, CPU cores, Storage)
- ▶ Enables running computations that would take weeks or years to complete locally: *capability computing*
- ▶ Enables running 100's or 1000's of small computations concurrently: *capacity computing*
- ▶ Allows automating and managing large calculation campaigns: Arrays, dependencies, scripting
- ▶ Decouples setting up a calculation and running it
- ▶ **Question to the audience:** What do *you* expect to get out of HPC with your R usage scenario?



R features

- ▶ Script language - interpreted at runtime
- ▶ Interactive workflow
- ▶ Dynamic resource usage (memory, CPU cores, ...)

HPC features

- ▶ Compiled languages preferred (optimized for performance)
- ▶ Batch-/non-interactive workflow
- ▶ Static resource contingent - requested at submit

How to resolve these conflicts?

It's not as bad as it looks (and e.g. Python has the same problems)!

- ▶ **script vs. compiled code:** R packages that handle numerics often use optimized libraries as backend (e.g. LAPACK and BLAS for vector and matrix operations)
- ▶ **interactive vs. non-interactive workflow:** Control via commandline parameters and control constructs
- ▶ **dynamic vs. static resources:** Amount of processes and threads can be controlled manually; memory footprint can be kept manageable with the right coding choices

Two things are needed to make running R on a cluster worthwhile:

- ▶ **Correct installation and configuration** (done by cluster administrators, R installation maintainers, R package maintainers)
- ▶ **Writing efficient R code** (done by the R users) → **Today's agenda!**

What about interactive Runtimes such as RStudio or Jupyter (with R kernel)?

- ▶ Some clusters offer one or both
- ▶ Interactive == portion of the runtime is spent waiting for input → 'inefficient' from an HPC perspective
- ▶ Consequence: If offered, RStudio and/or Jupyter will be restricted w.r.t. maximum resource usage (especially memory and runtime)
- ▶ Result: Useful for development and testing, but **long and/or intensive calculations should be submitted as 'proper' batch jobs!**

R Packages

- ▶ The two largest package databases are CRAN (<https://cran.r-project.org/>) and Bioconductor (<https://www.bioconductor.org/>)
- ▶ Packages can be installed either for all users or on a per-user basis → On HPC clusters, the central package base is often quite minimal, and each user is expected to install the packages they need
- ▶ Using a (well-maintained) package is almost always faster and less error-prone than writing a functionality by yourself → **No need to 'reinvent the wheel'!**



CRAN
 Mirror
 What's new?
 Task View
 Search
 About R
 R Packages
 The R Journal
 Software
 R Security
 R Graphics
 R Package
 Other
 Documentation
 Manuals
 FAQ
 Contributing

The Comprehensive R Archive Network

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows** and **Mac** users most likely want one of these versions of R:

- [Download R for Linux](#)
- [Download R for MacOS X](#)
- [Download R for Windows](#)

It is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source code for all platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2019-07-05, Action of the Tiers) [R 3.6.1 for Linux](#), read [this page](#) in the latest version.
- Sources of [R alpha](#) and [beta releases](#) (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are [available here](#). Please read about [new features and bug fixes](#) before filing corresponding feature requests or bug reports.
- Source code of older versions of R is [available here](#).
- Contributed extension [packages](#).

Sections about R

- If you have questions about R like how to download and install the software, or what the license terms are, please read our [answers to frequently asked questions](#) before you send an email.

Course Materials

- ▶ The materials for this course (slides, exercise sheet, exercise code) are available as a public git repository
- ▶ You can either clone the repository (requires git being installed):

```
git clone https://git.rwth-aachen.de/hkhlr/using-r-on-hpc.git
```

- ▶ Or download the material as an archive file:
Point your browser to

<https://git.rwth-aachen.de/hkhlr/using-r-on-hpc>
and click the download button (between
'Web IDE' and 'Clone')

The screenshot shows the GitHub repository page for 'Using R on HPC'. The repository is owned by 'HKHLR' and has 1 commit, 1 branch, 0 tags, and 0 bytes of project storage. The repository is public. The page shows the repository name, project ID, and a list of files. The files listed are: LICENSES, exercises, HKHLR_R_on_HPC_Handout.pdf, LICENSE.md, and README.md. The last commit for each file is 'Add current course material' by 'Sitt, René' 3 days ago. The README.md file was created 3 days ago.

Name	Last commit	Last update
LICENSES	Add current course material	3 days ago
exercises	Add current course material	3 days ago
HKHLR_R_on_HPC_Handout.pdf	Add current course material	3 days ago
LICENSE.md	Add current course material	3 days ago
README.md	Created README	3 days ago

Hands on!



Guided Exercise: 1.1 Package installation

Getting Help in R

- ▶ R has an extensive online help system
- ▶ Help on a specific topic, package, or function is invoked with `?<topic>` or `help(<topic>)`
- ▶ If the topic is vague or does not have its own help page, `??<topic>` or `help.search(<topic>)` searches for a term within the documentation texts
- ▶ You get back to the commandline by typing `q`

R Language Concepts and Paradigms

Concepts and Paradigms of R

"To understand computations in R, two slogans are helpful: Everything that exists is an object. Everything that happens is a function call."

- John M. Chambers, R developer

- ▶ The meaning of "object" and "function" depends on the programming language
- ▶ Probably meant here: *R does **not** distinguish between "primitive" types / operators and "complex" types (i.e. objects) / functions*
- ▶ Even a simple number is actually a **vector of length 1!**

listings/r_objects.R

```
1 > x <- 2
2 > x
3 [1] 2
4 > str(x)
5   num 2
6 > x <- c(x,3)
7 > x
8 [1] 2 3
9 > str(x)
10  num [1:2] 2 3
```

Concepts and Paradigms of R (cont.)

R as a language of objects and functions

- ▶ R functions are *also* objects. They may be assigned to a variable and act as arguments for other functions.
- ▶ Most R functions (including "+", "-", etc.) are *designed to act on whole vectors of data* instead of single elements.

What does that mean for R programmers?

- ▶ Whenever possible, make use of working on *whole vectors of data* instead of single values, and avoid explicit loops
- ▶ Whenever possible, write code that utilizes *pure functions* (see next slides), and avoid relying on side effects

Intermission: Side Effects and Idempotency

Side Effects, Idempotency, Pure Functions

Side Effects

A function is said to have a **side effect** if, besides returning a value, it has any effect on the state outside its scope, i.e. changing a global variable or producing output.

listings/sideeffect.R

```
1 increment.with.SE <- function(n) {  
2   n <- n+1 # global assignment  
3 }  
4 n <- 0  
5 print(n) # n is 0  
6 increment.with.SE(n)  
7 print(n) # n is 1  
8 increment.with.SE(n)  
9 print(n) # n is 2
```

listings/no_sideeffect.R

```
1 increment.no.SE <- function(n) {  
2   n <- n+1 # local assignment  
3 }  
4 n <- 0  
5 print(n) # n is 0  
6 increment.no.SE(n)  
7 print(n) # n is still 0  
8 m <- increment.no.SE(n)  
9 print(m) # m is n+1 = 1
```

Side Effects, Idempotency, Pure Functions

Idempotency

A function is said to be **idempotent** if its output depends solely on its input and not on any additional external state.

listings/non_idempotent.R

```
1 add.not.IP <- function(n) {  
2   n <- n+m # m is a global var  
3 }  
4 n <- 1  
5 m <- 3  
6 sum <- add.not.IP(n)  
7 print(sum) # sum is n+m  
8 m <- 4  
9 sum2 <- add.not.IP(n) # same command  
10 print(sum2) # same input, different output
```

listings/idempotent.R

```
1 add.IP <- function(n,m) {  
2   n <- n+m  
3 }  
4 n <- 1  
5 m <- 3  
6 sum <- add.IP(n,m)  
7 print(sum) # sum is n+m  
8 # ...[changes in the global environment]...  
9 sum2 <- add.IP(n,m) # same command  
10 print(sum2) # sum2 is still n+m
```



Side Effects, Idempotency, Pure Functions

Pure Functions

- ▶ A function is said to be a **pure function** if it is *both* idempotent *and* has no side effects.
- ▶ While “function” in *programming* normally implies nothing about the existence of side effects or idempotency, a “function” as defined by *mathematics* implies a **pure function**.
- ▶ It is not strictly enforced to write pure functions in R, but it often makes vectorization and parallelization easier
- ▶ The "apply" functions, their parallel equivalents, and the "foreach" package rely on using pure functions

Common Pitfalls

Pitfalls in R Programming: Growing Objects

- ▶ Dynamically growing objects in a loop can have a huge performance impact
- ▶ Functions that do this are e.g. `c()`, `cbind()`, and `rbind()`

listings/growing_object.R

```
1 concat <- function(n) {  
2   vec <- numeric(0) # Create a vector of length 0  
3   for(i in 1:n) vec <- c(vec,i) # Concatenate 1,2,...,n  
4   return(vec) }  
5  
6 fill <- function(n) {  
7   vec <- numeric(n) # Create a vector of length n  
8   for(i in 1:n) vec[i] <- i # Fill vector with 1,2,...,n  
9   return(vec) }  
10  
11 assign <- function(n) {  
12   vec <- 1:n # Assign a vector with length n containing 1,2,...,n  
13   return(vec) }
```

Pitfalls in R Programming: Growing Objects (cont.)

- ▶ Runtimes of the functions from previous slide in milliseconds, median of 100 runs, for different vector sizes

n	concat	fill	assign
100	0.075 ms	0.023 ms	0.002 ms
1000	2.045 ms	0.165 ms	0.003 ms
10000	152.476 ms	1.776 ms	0.016 ms

Possible strategies to avoid growing objects

- ▶ Avoid loops if vectorized value assignment is possible
- ▶ If the final size of an object is known, construct it prior to the loop
- ▶ Collect the loop results in a list and construct the object after the loop

Hands on!



Guided Exercise: 1.2 Microbenchmark

Pitfalls in R Programming: Rounding and Numerical Error

- ▶ Floating point math means that accuracy is limited to the precision of a number's *internal representation*
- ▶ Comparing floating point numbers might yield surprising results
- ▶ To test floating point (near-)equality, `all.equal(target, current, ...)` can be used
- ▶ R hides a lot of these imprecisions for the sake of readability (default print precision: 7 digits)
- ▶ You can change that default by setting `options(digits=<x>)` (max. 22)

listings/fpmath.r

```
1 0.3/3
2 [1] 0.1
3 0.1 == 0.3/3
4 [1] FALSE
5 0.1 - (0.3/3)
6 [1] 1.387779e-17
```

listings/fpdisplay.r

```
1 print(7/13 - 3/31)
2 [1] 0.4416873
3 print(7/13 - 3/31,digits=16)
4 [1] 0.4416873449131513
```

Pitfalls in R Programming: Rounding and Numerical Error (cont.)

- ▶ The R `round()` function uses “*round half to even*” instead of the more commonly known “round half up”
- ▶ Rounding half to even is the standard procedure for floating point math (conforming to IEEE 754)
- ▶ A .5 value is always rounded towards nearest *even* integer, i.e. `<even_value>.5` is always rounded down and `<odd_value>.5` is always rounded up
- ▶ **Question to the audience:** What might be the statistical reasoning for preferring round half to even instead of round half up?

listings/rounding.R

```
1 > round(4.5)
2 [1] 4
3 > round(3.5)
4 [1] 4
5 > round(2.5)
6 [1] 2
7 > round(1.5)
8 [1] 2
9 > round(0.5)
10 [1] 0
```

Round half up vs. round half to even demo

listings/rounding_demo.R

```
1 # Define a round-half-up function
2 # taken from: https://gist.github.com/sotoattanito/8e6fad4b7322ceae9f14f342985f1681
3 round.off <- function (x, digits=0)
4 {
5   posneg = sign(x)
6   z = trunc(abs(x) * 10 ^ (digits + 1)) / 10
7   z = floor(z * posneg + 0.5) / 10 ^ digits
8   return(z)
9 }
10 vec1 <- rnorm(1000000) # Create a large normal distribution (centered on 0)
11 vec2 <- round.off(vec1) # Apply round half up
12 vec3 <- round(vec1) # Apply round half to even
13 # Compare mean values - round half up will deviate further from 0 than round half to even
14 mean(vec1)
15 mean(vec2)
16 mean(vec3)
```

Vectorized Functions

Writing Efficient R Code: Vectorized Functions

Vectorization

- ▶ “Vectorized” in an R sense means that a function acts on a whole vector (or matrix) of data at once
 - ▶ It does *not* necessarily mean that the code uses “vector instructions” (i.e. SSE, AVX, etc.) or even parallelization
-
- ▶ R has an extensive library of vectorized functions which are generally *much* faster than doing an element-wise calculation (e.g. `sum()`, `mean()`, ...)
 - ▶ Before you write a `for` or `while` loop, always check whether there is a function that can do the operation on the whole data at once

Writing Efficient R Code: Vectorized Functions (cont.)

- ▶ Technically, there are two different types of vectorized functions: Those that apply a function element-wise, and those that combine vector elements into a result
- ▶ The former case's result and input keep the same *shape* while the latter's result has a *different* shape than its input
- ▶ An example for the first type is vector addition, or the `log()` function
- ▶ An example for the second type are the `mean()` and `sum()` functions

$$\begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline 3 \\ \hline \end{array} + \begin{array}{|c|} \hline 4 \\ \hline 5 \\ \hline 6 \\ \hline \end{array} = \begin{array}{|c|} \hline 5 \\ \hline 7 \\ \hline 9 \\ \hline \end{array}$$

$$\Sigma \begin{array}{|c|} \hline 1 \\ \hline 2 \\ \hline 3 \\ \hline \end{array} = 1+2+3=6$$

Vectorized Functions: `apply()`

- ▶ The `apply()` function family applies a specified function to a list, vector, matrix, or array of inputs
- ▶ It is not (much) faster than a loop, but often much more efficient to write
- ▶ The applied function is *generally assumed to be a pure function*
- ▶ Many parallelization functions are using the same principle as `apply()`, so using it from the start facilitates later parallelization

Name	Parameters	Output Type	Comment
<code>apply</code>	X, MARGIN, FUN	vector, array, or list	most general
<code>lapply</code>	X, FUN	list	'L'='List'
<code>sapply</code>	X, FUN	vector, matrix, or array	'S'='Simplify'
<code>mapply</code>	FUN, X, Y, ...	vector, matrix, or array	'M'='Multivariate'

Vector Recycling

- ▶ If the length of two vector arguments does not match, the shorter vector's entries will be repeated to match the longer vector's length
- ▶ **There is no warning message to indicate that vector recycling has happened and this can lead to unexpected results**

listings/recycling.R

```
1 > vec_1 <- 1:10
2 > vec_1
3 [1] 1 2 3 4 5 6 7 8 9 10
4 > vec_2 <- 1:5
5 > vec_2
6 [1] 1 2 3 4 5
7 > vec_1 + vec_2
8 [1] 2 4 6 8 10 7 9 11 13 15
```



Hands on!



Guided Exercise: 1.3 Vectorized Functions

Memory Management

Memory Management in R

- ▶ R holds all currently active objects in memory, which can fill up available RAM space very fast
- ▶ When working with large data sets, it is not uncommon to hit the memory ceiling
- ▶ R has a built-in garbage collector that will periodically clean up unused objects

Checking Memory Usage in R

- ▶ There are several ways to check current memory usage in R:
 - ▶ Invoking the garbage collector with `gc()`
 - ▶ Overall memory usage: On Windows `memory.size()` works; on Linux, the function `mem_used()` from package `pryr` has to be used
 - ▶ The size of a single object can be displayed with `object.size(<object>)`

listings/mem_display.R

```
1 > library("pryr")
2 > mat <- matrix(rnorm(1000),nrow=10)
3 > gc()
4      used (Mb) gc trigger (Mb) max used (Mb)
5 Ncells 370576 19.8      641837 34.3    641837 34.3
6 Vcells 705637  5.4     8388608 64.0   1754294 13.4
7 > mem_used()
8 26.4 MB
9 > object.size(mat)
10 8216 bytes
```

Strategies for Managing Memory

- ▶ Only use global objects (i.e. objects and data structures that exist outside of a function) when it is necessary
- ▶ Delete objects that are no longer needed with `rm(<object>)`
- ▶ For large data sets, check if they can be partitioned into several independent tasks
- ▶ Partitioning and gathering data from tasks can often be done by caching R objects on disk (see next slide)



Human-readable formats: Table and CSV

- ▶ `write.table()`, `write.csv()` and their counterparts `read.table()` and `read.csv()` produce and read from text files
- ▶ Mainly used to output structured data that is to be read either by a human or e.g. a spreadsheet program
- ▶ Saving and restoring complex objects might be difficult

Compressed formats: `save` and `saveRDS`

- ▶ `save()`, `saveRDS()` and their counterparts `load()` and `readRDS()` produce and read from compressed R object files
- ▶ Complex objects are kept exactly as-is, as long as all packages they may depend on are also loaded

Hands on!



Guided Exercise: 1.4 Saving and Loading Objects

Using R in a Cluster Environment

Running R in a Cluster Environment

- ▶ Most work on HPC clusters is done with *batch jobs*, i.e. scripts that run your code with no "live" user input
- ▶ Old R scripts sometimes use `R CMD BATCH`, which is **deprecated and should not be used anymore**
- ▶ Instead, to run R code via script, use
`Rscript <your_R_code>`
- ▶ `Rscript` also allows setting additional parameters to be read in by the R script



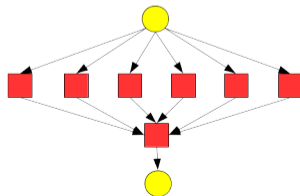
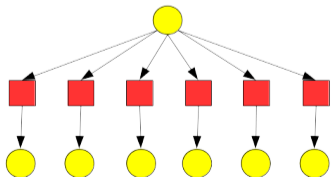
Running R in a Cluster Environment (cont.)

- ▶ R packages can either be installed into the global library (done by someone with root access) or into a user's local library
- ▶ The decision on when to do which is dependent on:
 - ▶ The preference of the local cluster maintainers (some might prefer global installations while others might not)
 - ▶ The availability of disk space (local installations might take a significant portion of the /home directory)
 - ▶ The complexity and - occasionally - external dependencies of a package installation

If you are maintaining an R installation yourself, **make sure that R is properly linked to a fast, optimized BLAS and LAPACK library** or overall performance might be significantly worse!

Structured Job Organization on a Cluster

- ▶ The scheduling software allows to submit several similar jobs at once, and defining dependencies between jobs
- ▶ This may be used for a "coarse-grained" parallelism
- ▶ Cluster jobs often involve doing the same calculation on a wide range of parameters (parameter sweep)
- ▶ Another use case is having each task working on a portion of a very large data set (and possibly having a single job collecting the results afterwards)



Using Commandline Arguments

- ▶ Having to slightly change the code whenever making a calculation that requires different parameters is not very efficient
- ▶ Providing the parameters via the commandline allows for code flexibility
- ▶ The simplest way in R is using `commandArgs()`
- ▶ `commandArgs()` returns an array of all provided commandline arguments, in order of input

- ▶ Arguments are provided as a character vector; if you want to use numerical values, you will need to convert them first (e.g. using `as.integer(<value>)`)!
- ▶ Checking the correct amount and order of arguments, as well as providing default values for missing arguments, must be done manually!

Using Commandline Arguments - Simple Example

exercises/submit_serial/growing_objects_param.R

```
1 args <- commandArgs(trailingOnly=TRUE)
2
3 if (length(args) < 1) {
4   vec_length <- 10000L
5 } else {
6   vec_length <- as.integer(args[1])*1000L
7 }
```

- ▶ `trailingOnly=TRUE` means we are getting *only* the arguments we explicitly put in

An alternative to `commandArgs`: `optparse`

- ▶ The alternative package `optparse` offers a more detailed and feature-rich interface for commandline argument parsing that is modeled after Python's `optparse` module

Hands on!



Guided Exercise: 1.5 Running R Jobs on a Cluster

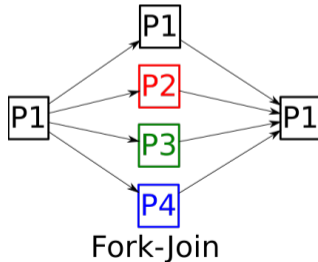
Parallel Programming in R

General Hints and Tips on Parallelization

- ▶ R calculations are often more a problem of *HTC* (*High Throughput Computing*) than *HPC* (*High Performance Computing*)
- ▶ Size of a single job often ranges from 1 to 32 cores
- ▶ Several hundreds or thousands of similar jobs/tasks
- ▶ Parallel scalability is limited by the amount of serial code (cf. *Amdahl's Law*)
- ▶ *Good data partitioning and data organization are often more effective than just "making it parallel somehow"!*

R's parallelism model

- ▶ Most of R's parallel functionality is based on *Fork-Join-Parallelism*
- ▶ More specifically, the parallel part(s) of R code are mostly limited to single functions
- ▶ General scheme: Serial setup (read in/prepare data) → Parallel computation (process data) → Serial finalization (output/plot results)



Levels of Parallelization

- ▶ A code can be parallelized on multiple levels
- ▶ All kinds of parallelization have their own use cases, advantages and challenges
- ▶ Rule of thumb: As process coupling (i.e. amount of possible communication between units) decreases, the parallelization scope (and possible scalability) increases

Type	Parallelized Unit	Parallel Scope	Communication
Job Array	Job	Cluster	None
Socket/MPI Cluster	Process	Cluster	Messages
Fork Cluster	Process	Node	Shared Memory
OpenMP/Multithread	Thread	Node	Shared Memory
Vectorization	CPU Instruction	CPU Core	Shared Registers

Parallel Programming with R

- ▶ There are multiple packages for the parallelization of R code
- ▶ There is no single "right way" to parallelize; there are different techniques and packages that are well-suited for different tasks
- ▶ We will give an overview over of the most common packages and how they are used, in order from most simple to most complex

Type	Parallelized Unit	Parallel Scope	Communication
Job Array	Job	Cluster	None
Socket/MPI Cluster	Process	Cluster	Messages
Fork Cluster	Process	Node	Shared Memory
OpenMP/Multithread	Thread	Node	Shared Memory
Vectorization	CPU Instruction	CPU Core	Shared Registers

OpenMP Multithreading

- ▶ Some of R's backend libraries are multithreading-capable without additional programming input
- ▶ This especially includes libraries handling vector- and matrix-operations or numerics (BLAS, LAPACK, FFTW, ...)
- ▶ OpenMP multithreading is generally on a lower parallelism level (read: Better than serial, but often less efficient than explicitly parallel routines) and needs to be controlled/switched off when parallelizing on a higher level!
- ▶ Otherwise, each parallel process might start multiple threads, which leads to CPU core oversubscription and slowdown

Recommendation

Always set `export OMP_NUM_THREADS=1` in R job scripts!

Package- /Function-level parallelism

- ▶ Some packages already offer parallelized functions out-of-the-box
- ▶ A well-tested, prebuilt parallelization is preferable to a 'roll-your-own' solution in most cases!
- ▶ Check for function parameters like `num.threads`
- ▶ Many functions will default to using all CPU cores that they can 'see' - *make sure that the number of threads or processes is explicitly set to the amount of cores that the job script asks for!*
- ▶ Example: The `ranger` package (Random Forest implementation) is parallelized and will use all available CPU cores by default
 - ▶ On the cluster: Set `--nodes=1`, `--ntasks=1`, and `--cpus-per-task` and the `ranger` functions' `num.threads` both to the *same* value

Random Numbers, Parallelization, and Reproduceability

- ▶ R code often requires random numbers, e.g. for initializing model weights with a random component, or randomly selecting samples
- ▶ The `parallel` package provides an RNG implementation that ensures that all worker processes get an independent sequence of random numbers: The *L'Ecuyer CMRG*
- ▶ With a given seed, every substream will produce the same (independent!) number sequence on every run
- ▶ For the `cluster` interface, use `clusterSetRNGStream(cluster_object, iseed)`
- ▶ For `mc*apply`, it be selected with `set.seed(iseed, kind="L'Ecuyer-CMRG")`
- ▶ Setting `iseed` to `NULL` will initialize the RNG differently on every run
- ▶ We will use a different technique for reproduceability to be able to compare serial with parallel runs, but occasions where you would normally use the L'Ecuyer-CMRG are marked and commented out in the exercise code

Code for parallel R exercises: `glm_iris.R`

- ▶ The basic function for all following parallel demos/exercises will be always the same
- ▶ We do this to maintain comparability between the different examples and to show how the approaches differ from each other
- ▶ The base setup looks like this:
 - ▶ We start with the well-known iris dataset, which contains 150 entries in total
 - ▶ Each entry has four measurements (sepal length, sepal width, petal length, petal width) and a subspecies label (setosa, versicolor, virginica)
 - ▶ We generate a generalized linear model describing the relation between sepal length and species for versicolor and virginica species while resampling the dataset a large number of times
 - ▶ Since each sampled run is independent, we can parallelize these runs and have a multitude of options to do so

Iris data generalized linear model

listings/glm_iris.R

```
1 library(stats)
2
3 # Get iris dataset, but only columns 'Sepal.length' and 'Species',
4 # We filter out 'setosa' so we only have two species to fit to
5 get_data <- function() {
6   iris[which(iris[,5] != "setosa"), c(1,5)]
7 }
8
9 # Fit a generalized linear model relating iris sepal length with iris species
10 boot_fx <- function(trial,data) {
11   ## Normally we wouldn't hard-set a seed here, but we do so
12   ## for comparing results from different methods!
13   set.seed(trial)
14   ind <- sample(100,100,replace=TRUE) # Every call resamples the data indices randomly
15   result1 <- glm(data[ind,2]~data[ind,1],family=binomial(logit)) # Fit the model
16   r <- coefficients(result1) # Get model coefficients
17   ## We would usually create a dataframe row; we omit it for easier result comparison
18   #res <- rbind(data.frame(),r)
19 }
```



Multicore Apply

The `mc*apply()` Functions

- ▶ The `mc*apply()` function family is a multicore wrapper around the `apply()` functions
- ▶ It applies a function to a vector of values and takes the amount of cores it should use as additional argument `mc.cores`
- ▶ `mclapply`, `mcmapply`, and `mcMap` are the parallel versions of `lapply`, `mapply`, and `Map`
- ▶ Advantages:
 - ▶ Does not need additional setup besides `library('parallel')`
 - ▶ Relatively small overhead
- ▶ Disadvantages:
 - ▶ Parallelizes only a single function at a time
 - ▶ Function has to be in (or has to be transformable to) a form that works with `apply()`
 - ▶ Does not work on Windows

Hands on!



Guided Exercise: 2.1 Using mclapply

R Cluster Functions

R Cluster Interface

- ▶ The R `cluster` functions provide a more detailed interface to multicore programming
- ▶ There are multiple "parallel backends" can be used in exactly the same way
- ▶ Most commonly used backends are FORK (on Linux only) and PSOCK (on Windows and Linux)
- ▶ PSOCK handles process communication via sockets and is available on both Windows and Linux systems
- ▶ FORK works by spawning child processes through Linux's `fork()` function; this reduces memory consumption but does not work on Windows machines
- ▶ We will concentrate on the FORK variant in this course

R Cluster Interface (cont.)

▶ Advantages:

- ▶ Provides a common interface for different parallel backends
- ▶ Possibility to parallelize is not restricted to functions that can be `apply()`ed

▶ Disadvantages:

- ▶ Larger overhead and more package dependencies
- ▶ Needs more code to work correctly; more possibilities to make errors
- ▶ Master process is not included in the worker process group (i.e. a cluster with `n_cores=8` runs 8 workers **and** one master process)
- ▶ Any global objects may need to be explicitly exported to be available for worker processes

R Cluster Interface (cont.)

par*apply

- ▶ Yet another parallel version of `apply()`
- ▶ As long as the declared cluster is of `PSOCK` type, it also works on Windows

Parallel foreach

- ▶ Looks almost like a regular `for` loop, but works more like a function
- ▶ More flexible than `apply()`
- ▶ Partitioning of iterations between processes has to be done by the developer!

Structure of an R Cluster Call

1 Initialization

- ▶ Set up the cluster and specify amount of cores, parallel backend, etc.
- ▶ Register the parallel backend for foreach
- ▶ Export needed global objects into the cluster environment

listings/cluster_init.R

```
1 cl <- makeCluster(<cores>, type=<backend>)  
2 registerDoParallel(cl)  
3 clusterExport(cl, <global_vars>)
```

2 Parallel function calls

- ▶ Foreach, par*apply, and cluster* functions

listings/cluster_call.R

```
1 ret <- parLapply(cl, <range>, <function>)  
2 ret <- foreach(<range>) %dopar% {<function>}
```

3 Finalization

- ▶ Cleanup and stopping the cluster

listings/cluster_finalize.R

```
1 stopCluster(cl)
```

Foreach

- ▶ Although `foreach` looks similar to a `for` loop, it behaves much more like a function
- ▶ Regular loops do work through *side effects*, but `foreach` in general *does not* interact with global program state
- ▶ The result of a `foreach` is not its side effects, but its return value
- ▶ If multiple values should be returned by `foreach`, they have to be organized in a vector, list, array, or dataframe
- ▶ Dataframes are able to hold data of different types simultaneously, thus allowing for some flexibility

Hands on!



Guided Exercise: 2.2 Using the R Cluster Functions

Rmpi

- ▶ *Another* parallelization mechanism?!
- ▶ All previous techniques are mostly confined to a single cluster node
- ▶ In theory, a PSOCK cluster can utilize multiple nodes, but it is not well-integrated into the cluster environment:
 - ▶ Needs info which hosts to use, which has to be extracted manually
 - ▶ Compute nodes might not be set up to allow socket communication
- ▶ By contrast, virtually every HPC cluster's MPI integration handles process distribution, opening of communication channels, etc. mostly automatically
- ▶ MPI is by far the most popular and convenient way to manage calculations that span multiple nodes on HPC clusters

Type	Parallelized Unit	Parallel Scope	Communication
Job Array	Job	Cluster	None
Socket/MPI Cluster	Process	Cluster	Messages
Fork Cluster	Process	Node	Shared Memory
OpenMP/Multithread	Thread	Node	Shared Memory
Vectorization	CPU Instruction	CPU Core	Shared Registers



The Rmpi Package

- ▶ Rmpi is an R wrapper for the MPI (*Message Passing Interface*) library
- ▶ It can be used in two different ways:
 - ▶ As yet another backend for the `cluster` package, utilizing the `RMPIsnow` wrapper
 - ▶ As a simple wrapper for using MPI functions inside R code, analogous to how it is used in C/C++ or FORTRAN code
- ▶ The `snow` package functions look very similar to the `parallel` interface, but have a slightly different syntax in some cases
- ▶ The simple wrapper variant is based on a somewhat different idea of parallelism than what we have seen until now

Rmpi + snow

- ▶ The name `snow` stands for *Simple Network of Workstations*
- ▶ We will restrict ourselves to snow's MPI backend in this course

`snow` **overrides** several functions of the `parallel` packages and vice versa

- ▶ If you want to make sure that the right implementation is used, add an explicit package prefix (`snow::<function>` or `parallel::<function>`)!

listings/snow_parallel_override.R

```
1 > library(parallel)
2 > library(snow)
3
4 Attache Paket: 'snow'
5
6 The following objects are masked from 'package:parallel':
7
8   clusterApply, clusterApplyLB, clusterCall, clusterEvalQ, clusterExport, clusterMap, clusterSplit,
9   makeCluster, parApply, parCapply, parLapply, parRapply, parSapply, splitIndices, stopCluster
```



Rmpi + snow usage

- ▶ Similar to `parallel`, but `snow` gets some info (e.g. # of processes) through its `RMPISNOW` wrapper automatically
- ▶ `par*apply()` and `cluster*()` functions work like their `parallel` equivalents
- ▶ As long as `registerDoParallel(<cluster>)` is set, `foreach %dopar%` works as expected
- ▶ R code invocation: Instead of `Rscript`, we have to use `MPI execute command + RMPISNOW wrapper script + R code`

listings/cluster_usage_snow.R

```
1 cl <- snow::makeCluster()
2 registerDoParallel(cl)
3 snow::clusterExport(cl, <global_vars>)
4
5 ret <- snow::parLapply(cl, X, <function>)
6 ret <- foreach(snow::clusterSplit(cl, <range>)\
  %dopar% {<function>})
7
8 snow::stopCluster(cl)
```

listings/rmpisnow_invocation.sh

```
1 mpiexec -np <procs> RMPISNOW < <R_code>.R
```

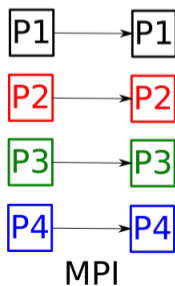
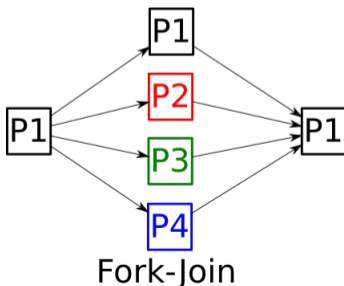
Hands on!



Guided Exercise: 2.3 Using RMPISNOW

The Rmpi Wrapper

- ▶ The parallelization techniques we have seen so far are mainly of the *fork-join* type
- ▶ With MPI, the processes run in parallel for the whole length of the program, while data distribution, data gathering, and process synchronization, are coordinated by messages between the processes (*MPI = Message Passing Interface*)
- ▶ These messages can be one-to-one (send/receive), one-to-all (broadcast, scatter), all-to-one (gather, reduce), and all-to-all



MPI send

- ▶ Invocation: `mpi.send(x, type, dest, tag, comm=0)`
- ▶ x: Data to be sent; type: 1=integer, 2=double, 3=character; dest: Rank of destination process; tag: (Integer) tag for the message, can be set to zero if not used; comm: Which communicator to use, the default communicator is zero.

MPI receive

- ▶ Invocation: `mpi.recv(x, type, source, tag, comm=0, status=0)`
- ▶ x: Buffer for the data to be received; type: 1=integer, 2=double, 3=character; source: Rank of the sender; tag: (Integer) tag for the message; comm: Which communicator to use; status: Status value (normally zero).

- ▶ Since we can only send integers, doubles, or characters, complex data structures would be excluded
- ▶ Rmpi also allows to send whole R objects instead, which is often more convenient
- ▶ The parameters are mostly the same as in the simple send and receive functions

MPI send R objects

- ▶ Invocation: `mpi.send.Robj(obj, dest, tag, comm=0)`

MPI receive R objects

- ▶ Invocation: `mpi.recv.Robj(source, tag, comm=0, status=0)`

Hands on!



Guided Exercise: 2.4 An Rmpi Wrapper Example

Final Remarks

- ▶ You should now have an overview over most of R's features related to HPC and a working knowlege of how to run R code on a cluster
- ▶ Each problem is unique and may require a unique solution; we cannot provide a catch-all manual, but we can enable you to *know where to start searching*
- ▶ R documentation is spread out onto many sites, with partially deprecated or conflicting information on the same problems; having some simple, but functional examples is often a good way to approach a complex task

Online Sources and References

- ▶ “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>
- ▶ Parallel exercises code adapted from:
<https://nceas.github.io/oss-lessons/parallel-computing-in-r/parallel-computing-in-r.html>

Image credits

- ▶ Slides 3, 7: The R logo is ©The R Foundation 2016, usage permitted by CC-BY-SA 4.0
<https://creativecommons.org/licenses/by-sa/4.0/>
- ▶ Slide 4: MaRC3a backplane photo used with permission from Author, ©Marcus Lechner 2020
- ▶ Slide 37: MaRC3a backplane photo used with permission from Author, ©Clemens Thölken 2022
- ▶ HKHLR logo and HKHLR map ©HKHLR 2014-2023
- ▶ Additional graphics created in Inkscape and Gnu Image Manipulation program

Thank you for your attention!