Global Drug Development
Advanced Exploratory Analytics

Go fastR – how to make R code fast(er) and run it on high performance compute (HPC) clusters

Lukas A. Widmer, Novartis Pharma AG
Michael Mayer, Posit PBC

CEN 2023 Conference,
3rd of September 2023

NOVARTIS | Reimagining Medicine

Who we are

Michael Mayer, PhD
Posit

Scientist by training turned IT and now solutions engineer

Lukas A. Widmer, Dr. sc. ETH Zürich
Novartis

Computer Scientist / Computational Biologist turned Statistical Consultant

Some housekeeping before we start

- This is an interactive workshop ☺
  If you have a question, stop us / raise your hand during the talk!
- This course on R high performance computing is open-source
  - Course content is licensed under CC-BY 4.0, example code under the MIT license
- We collect feedback on the course and make the material available
- We will work with a web-based Posit Workbench cluster in the cloud

Learning goals

1. Debugging R code and identifying & optimizing bottlenecks locally
2. Basics of R parallelization on high performance compute environments
   - Understand limits of achievable performance (Amdahl's law)
   - Parallelize R code on compute clusters via {clusterMQ} and {batchtools}
   - Understand how to generate uncorrelated random numbers in parallel R code
   - Debug remote R code in {batchtools} and {clusterMQ} jobs
3. Know how to apply this knowledge on relevant case studies
   - Simulation studies, bootstrapping, cross-validation, parallel Stan models, ...
   - Your case study, if you brought one along with you ☺

Part I

Debugging R code and identifying & optimizing bottlenecks locally

Before we start: do not sacrifice correctness in the name of performance

The workflow should roughly be the following:

1. First of all, focus on *correctness* of your code before performance
   → Debugging & Testing
2. If your code is too slow, (always!) measure where it spends the most time
   → Profiling
3. With the information from step 2, optimize the bottlenecks
   → Local optimization
4. Only if the code is still too slow in step 3, go to the HPC (if possible)
   → Parallelization

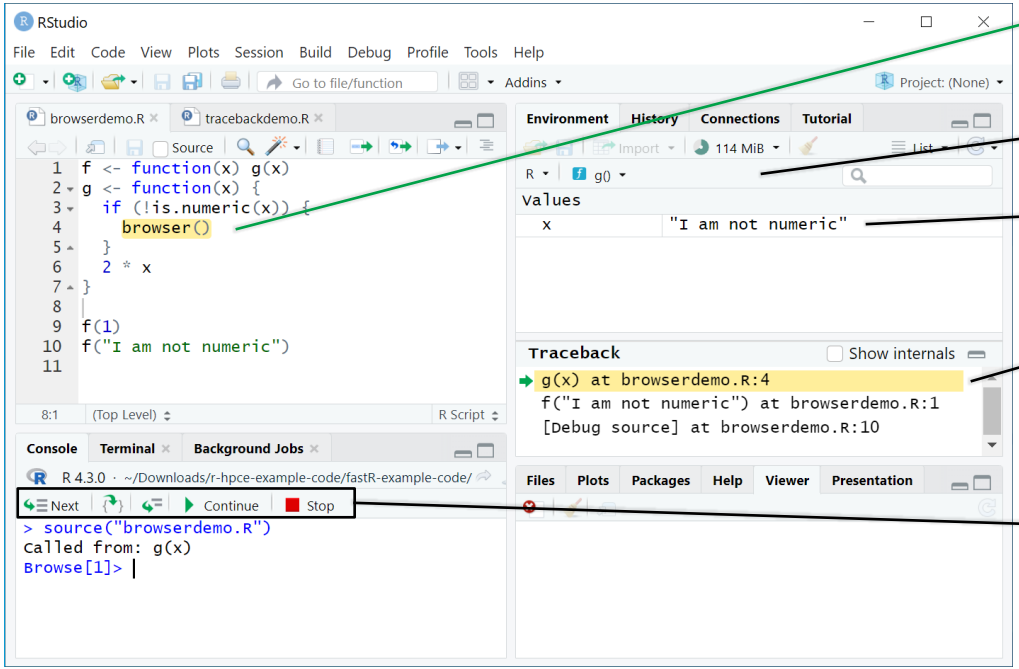
My R code is not behaving as expected… how to find the problem?

Overall approach we suggest:

1. If you get a non-obvious error message, use internet search
   – Chances are someone has already asked about it on StackOverflow
2. Make it repeatable
   – Simplify the example by removing code not needed to trigger the issue
   – The {reprex} package can help you with this (also for submitting bugs to Github!)
3. Figure out where it is
   – See the next slides for some helpful commands
4. Fix it and test it ☺

Helpful commands for debugging: `browser()` interactive debugger

Try it yourself:
→ browserdemo.R



Location in program

Environment explorer

Values in environment `g()`

Call stack and line numbers

Next	Execute next line of code
	Step into function / expression
	Step out of function
Continue	Continue executing (no stopping at next line)
Stop	Stop (drop to R console)

Helpful commands for debugging: traceback(), rlang::last_error() and rlang::last_trace()

tracebackdemo.R

Source on Save

```
1  f <- function(x) g(x)
2  g <- function(x) {
3    if (!is.numeric(x)) {
4      stop("`x` must be numeric")
5    }
6    2 * x
7  }
8
```

Calling `f()` with `x = "I am not numeric"` obviously errors:

```
> f("I am not numeric")
Error in g(x) : `x` must be numeric
```

Show Traceback
Rerun with Debug

Try it yourself:
→ tracebackdemo.R

```
> f("I am not numeric")
Error in g(x) : `x` must be numeric
```

Show Traceback
Rerun with Debug

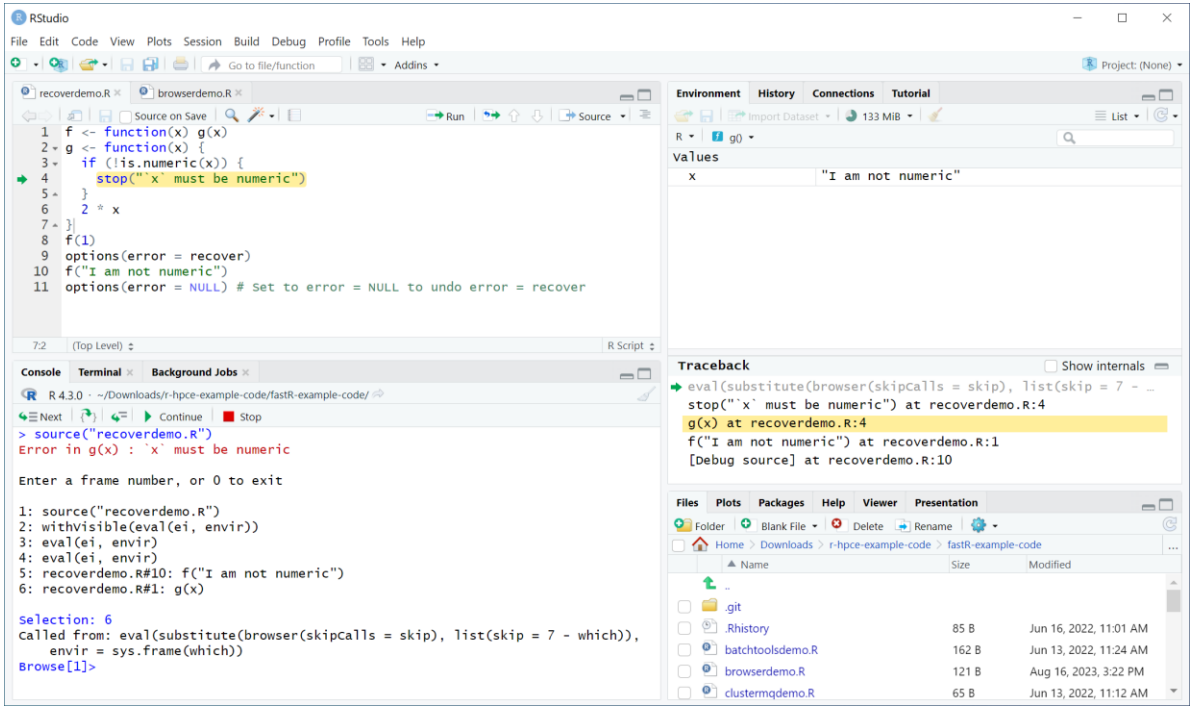
```
3. stop("`x` must be numeric") at tracebackdemo.R#4
2. g(x) at tracebackdemo.R#1
1. f("I am not numeric")
```

```
> traceback()
3: stop("`x` must be numeric") at tracebackdemo.R#4
2: g(x) at tracebackdemo.R#1
1: f("I am not numeric")
```

"Show Traceback" in RStudio or `traceback()` from base R show the call stack and code lines where the error occurred.

`last_error()` and `last_trace()` from {rlang} are more modern variants, however, they by default only cover `rlang::abort()`, not `base::stop()` errors. See `rlang::global_entrace()` for details.

Helpful commands for debugging:
`options(error = recover)`



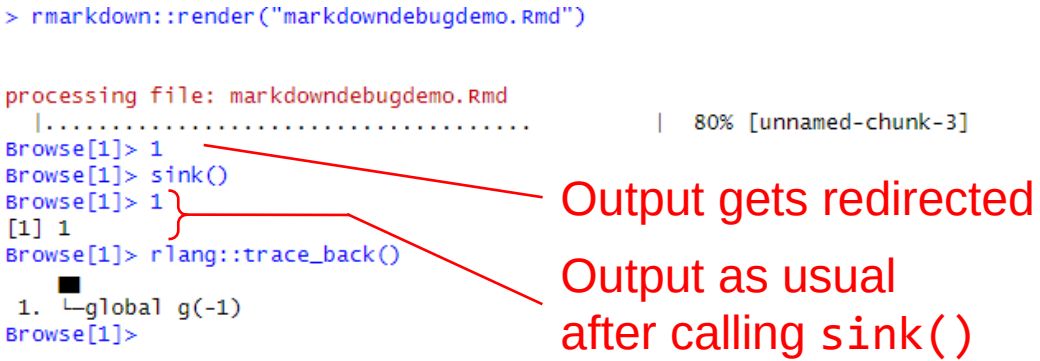
You can enter the interactive `browser()` debugger when the error occurs, too!

This is undone with `options(error = NULL)`

Try it yourself:
→ recoverdemo.R

Advanced Debugging: R Markdown

- R Markdown redirects output, so if we put a `browser()` statement, the interactive console output is invisible – use `sink()` to stop the redirect:



Try it yourself: → markdowndebugdemo.Rmd

- See Debugging with the RStudio IDE – Posit Support for details.

Advanced Debugging: R Markdown

- We can also combine the `sink()` function with `trace_back()` from {rlang} and `recover()` for a powerful combo that prints where the error occurred, and allows us to interactively debug the R Markdown:

```
options(error = function() {
  sink()
  print(rlang::trace_back(bottom = sys.frame(-1)))
  recover()
})
```

Try it yourself: → markdowndebugdemo.Rmd

```
> rmarkdown::render("markdowndebugdemo.Rmd")


processing file: markdowndebugdemo.Rmd
  |.....................................................................................................| 100% [unnamed-chunk-3]
Quitting from lines 51-58 [unnamed-chunk-3] (markdowndebugdemo.Rmd)
Error in `f()`:
! x should not be < 0
Backtrace:
 1. global f(-1)
    ■
 1. ├─global f(-1)
 2. │ └─base::stop("x should not be < 0")
 3. └─base::.handleSimpleError(`<fn>`, "x should not be < 0", base::quote(f(-1)))
 4.   └─knitr (local) h(simpleError(msg, call))
 5.     └─rlang::entrace(e)
 6.       └─rlang::cnd_signal(entraced)
 7.         └─rlang:::signal_abort(cnd)
 8.           └─base::stop(fallback)

Enter a frame number, or 0 to exit
```

Advanced Debugging: Remote Sessions

- **Post-mortem debugging – resurrecting a session for debugging:**

dumpdemo1.R | dumpdemo2.R

Source on Save

```
1  options(error = function() {
2    if(!interactive()) {
3      dump.frames(to.file = TRUE, include.GlobalEnv = TRUE) # Save dump
4      q(status = 1) # Quit with error status
5    } else {
6      recover() # If we're working interactively, recover instead
7    }
8  })
9  a <- 1
10 if (a == 1) {
11   stop("I am a really bad error!")
12 }
```

```
$ Rscript dumpdemo1.R
Error: I am a really bad error!
```

dump.frames() → 

last.dump.rda

dumpdemo1.R | dumpdemo2.R

Source on Save

```
1  load("last.dump.rda")
2  debugger()
3
```

```
> source("dumpdemo2.R")
Message:  Error: I am a really bad error!
Available environments had calls:
1: .GlobalEnv
2: stop("I am a really bad error!")
3: (function ()
{
    if (!interactive()) {
        dump.frames(to.file = TRUE,

Enter an environment number, or 0 to exit
Selection: 1
Browsing in the environment with call:
   .GlobalEnv
Called from: debugger.look(ind)
Browse[1]> ls()
[1] "a"
Browse[1]> a
[1] 1
```

Try it yourself: → dumpdemo1/2.R

- **For Shiny apps,** see Debugging Shiny applications (rstudio.com)

My R code is slow… what can I do?

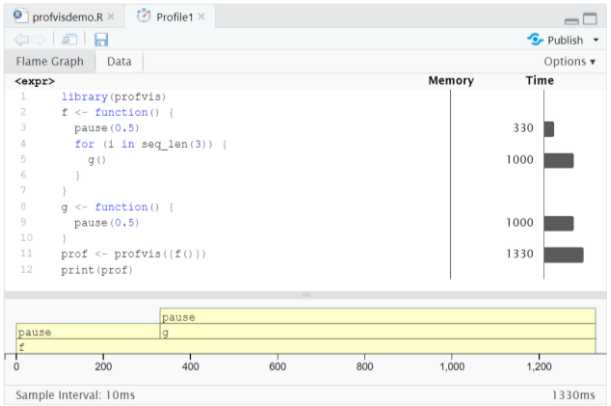
> “R is a language optimized for human performance, not computer performance”
>
> Hadley Wickham, New York R Conference 2018

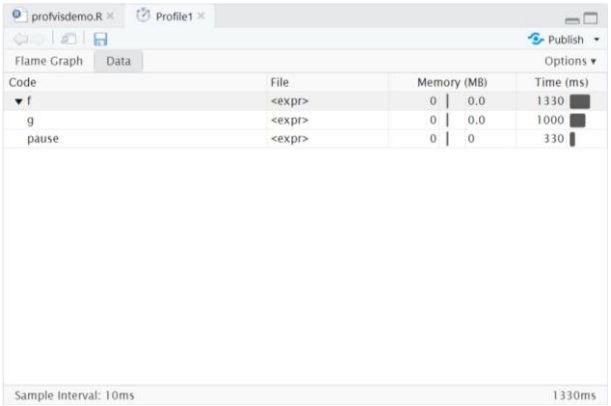
1. **Measure / “profile”:**
   Where is the code slow?
   – Avoid the trap of prematurely optimizing the part you “think” is slow.
2. **Then optimize** (once you have data!).

Which parts of the code are slow?

Profiling the code can tell you! In R, this is done using the profvis package (or in RStudio using the Profiling menu):

```
library(profvis)
f <- function() {
  pause(0.5)
  for (i in seq_len(3)) {
    g()
  }
}
g <- function() {
  pause(0.5)
}
prof <- profvis({f()})
print(prof)
```





Try it yourself:
→ profvisdemo.R

I have found a bottleneck... what now?
1. Check for existing solutions

If the slow function is from a package, search for a faster one!

Runtime complexity (runtime as a function of data size) of different algorithms can be wildly different – some work well on small data but take forever on large data!

Examples:

- For data frames, use {data.table} and base R instead of {tidyverse}
  - `dplyr::filter()` in a loop is *slow* (but nice to read, so only optimize if needed). If you need to filter in a loop, use base R logical indexing or {data.table} instead
- To read/write CSV data, use {vroom} instead of base R, {readr} or {data.table}
- To (de)serialize data, use `qread()` and `qsave()` from the {qs} package instead of `readRDS()` and `saveRDS()`

I have found a bottleneck… what now?
2. Do as little as possible…

… and compute things only once, if possible (and reasonable).
→ See the DRY (Don't Repeat Yourself) principle

Examples:

- When testing for the existence of a condition over data frame rows, use `any(x$condition)` rather than `nrow(filter(x, condition)) > 0`.
- Assemble a data frame / tibble / data table once, rather than creating it and appending to it over and over again.
- When subsetting in a data frame, don't subset the entire data frame, only the column needed for the computation (SELECT before FILTER).

I have found a bottleneck... what now?
3. Vectorize

Specialized vectorized functions will still be substantially faster than `apply/lapply/sapply()` or `for` loops, see for instance:

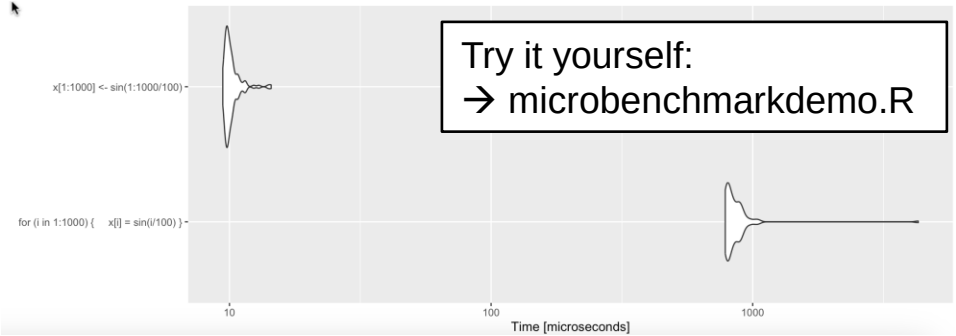
- `rowSums()`, `colSums()`, `rowMeans()`, `colMeans()` in base R
- The {matrixStats} package:
  - `anyMissing(), colQuantiles(), rowQuantiles()` and many, many more
- The {Rfast} package: A Collection of Efficient and Extremely Fast R Functions
- The {collapse} package: Advanced and Fast Data Transformation

Example: vectorize for loop

```
> library(microbenchmark)
> x<-c()
> microbenchmark(for(i in 1:1000) {x[i]=sin(i/100)})
Unit: microseconds
                                    expr     min      lq     mean   median       uq      max neval
 for (i in 1:1000) {     x[i] = sin(i/100) } 776.294 787.569 841.1576 792.2635 809.9755 4237.719   100
> microbenchmark(x[1:1000]<-sin(1:1000/100))
Unit: microseconds
                        expr   min    lq    mean median     uq    max neval
 x[1:1000] <- sin(1:1000/100) 9.184 9.635 9.92036 9.8605 10.127 17.794   100
> microbenchmark(for(i in 1:1000) {x[i]=sin(i/100)}, x[1:1000]<-sin(1:1000/100))
Unit: microseconds
                                    expr     min      lq      mean   median       uq      max neval
 for (i in 1:1000) {     x[i] = sin(i/100) } 784.535 799.459 882.28843 837.3225 883.7755 4211.684   100
                x[1:1000] <- sin(1:1000/100)   8.979   9.799  10.45541  10.1885  10.5780   16.687   100
> ggplot2::autoplot(microbenchmark(for(i in 1:1000) {x[i]=sin(i/100)}, x[1:1000]<-sin(1:1000/100)),times=1000)
```

The R package {microbenchmark} is a good tool to benchmark given parts of code. It will run the same code chunk n times (default 100) to get a "good" result.

Speed-up 80x



Try it yourself:
→ microbenchmarkdemo.R

Remember: R is an interpreted language. Vectorization ensures that data is operated on in chunks by native (C/C++) code rather than element by element in R code.

How much memory am I using?

profvis also can visualize memory (de-)allocations!

```
library(profvis)
prof <- profvis({
  x <- integer()
  for (i in 1:1e4) {
    x <- c(x, i)
  }
})
print(prof)
```

Memory

de-allocations (Mb)

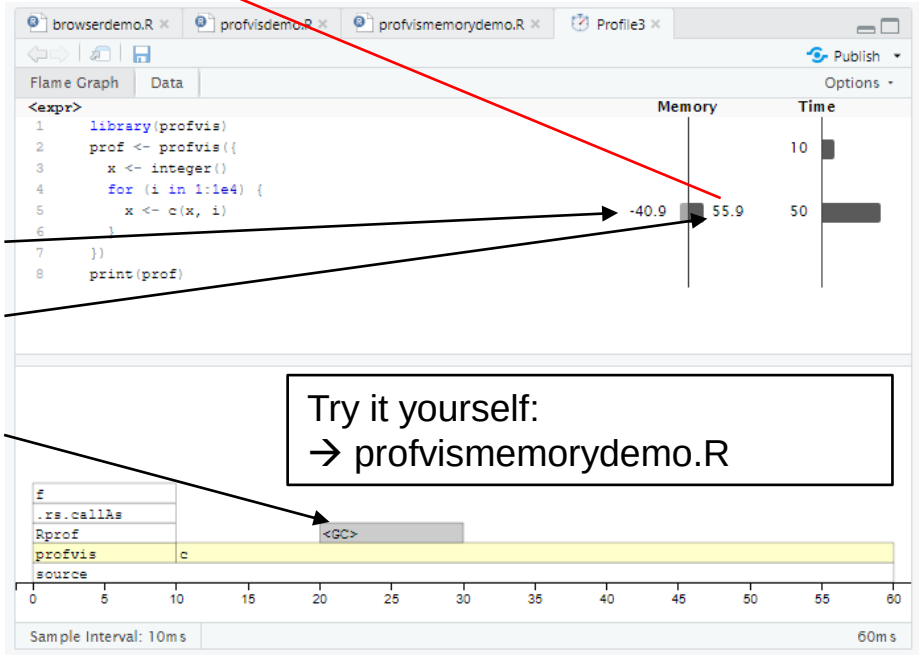
allocations (Mb)

Garbage collection*

* R manages memory for you (you don't have to explicitly allocate / free memory) by garbage collection. If this takes a lot of time, you might be creating a lot of short-lived objects (or in this case, copies)!

```
R 4.1.0> library(lobstr)
R 4.1.0> obj_size(integer(1e4))
40,048 B
```

1000x size of object!



Avoid making object copies if possible

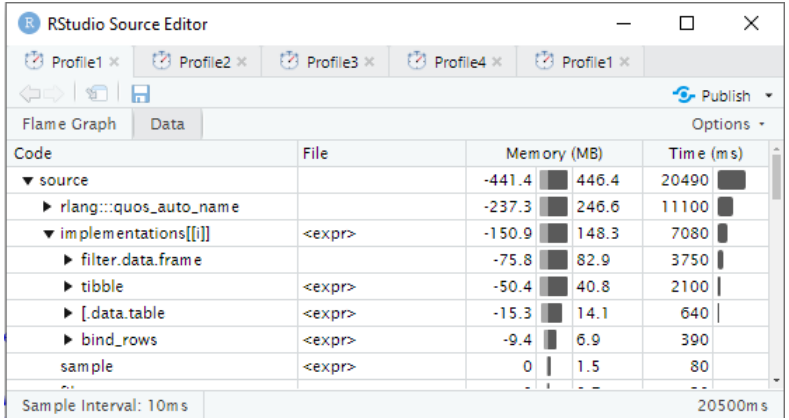
Or, if you must make copies, copy as little as possible.

Examples: if you are

- creating a vector, pre-allocate it (e.g. `x <- numeric(N)`) then fill it, rather than iteratively grow `x` with the `c()` function,
- creating a data frame, create it once from vectors rather than appending rows,
- subsetting a data frame, try subsetting only the column(s) you need for downstream analysis (resulting in vectors rather than data frames).

Demo: 5k bootstraps on 1k patients*

Try it yourself:
→ optimizebootstrap.R



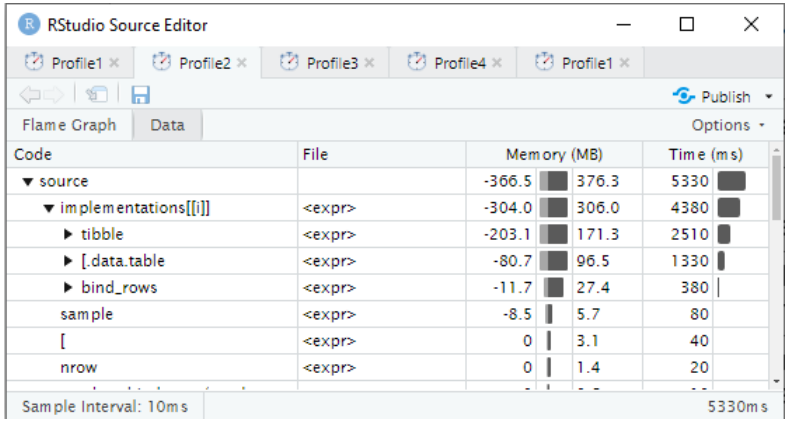

```r
impl_1 = function(population) {
    result <- NULL
    for (i in seq_len(bootstrap_n)) {
      bootstrap_data_rows <- sample(
        x = seq_len(nrow(population)),
        size = bootstrap_size,
        replace = TRUE
      )
      current_bootstrap <- population[bootstrap_data_rows, ]
      analysis_pop <- filter(current_bootstrap, analysis_flag == T)
      current_result <- tibble(
        bootstrap_index = i,
        computed_output = median(analysis_pop$dummy_measurement)
      )
      result <- bind_rows(result, current_result)
    }
    return(result)
  }
```

```
> population[1:10]
    patient_id dummy_measurement analysis_flag
 1:          1        0.57632155          TRUE
 2:          2        0.56474213          TRUE
 3:          3        0.07399023          TRUE
 4:          4        0.45386562          TRUE
 5:          5        0.37327926          TRUE
 6:          6        0.33131745          TRUE
 7:          7        0.94763002          TRUE
 8:          8        0.28111731          TRUE
 9:          9        0.24540405         FALSE
10:         10        0.14604362          TRUE
```

* Realistically, 20 seconds is okay, but in the context of this seminar, anything longer would have been too tedious to demo.

Demo: 5k bootstraps on 1k patients

Try it yourself:
→ optimizebootstrap.R

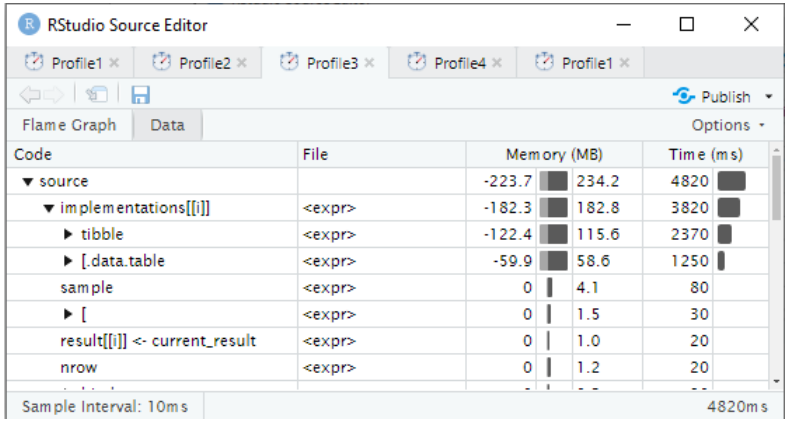



```
impl_2 = function(population) {
    result <- NULL
    for (i in seq_len(bootstrap_n)) {
      bootstrap_data_rows <- sample(
        x = seq_len(nrow(population)),
        size = bootstrap_size,
        replace = TRUE
      )
      current_boot <-
        population[bootstrap_data_rows][(analysis_flag)]
      current_result <- tibble(
        bootstrap_index = i,
        computed_output = median(current_boot$dummy_measurement)
      )
      result <- bind_rows(result, current_result)
    }
    return(result)
  }
```

Subset with data.table instead of filter

Demo: 5k bootstraps on 1k patients

Try it yourself:
→ optimizebootstrap.R

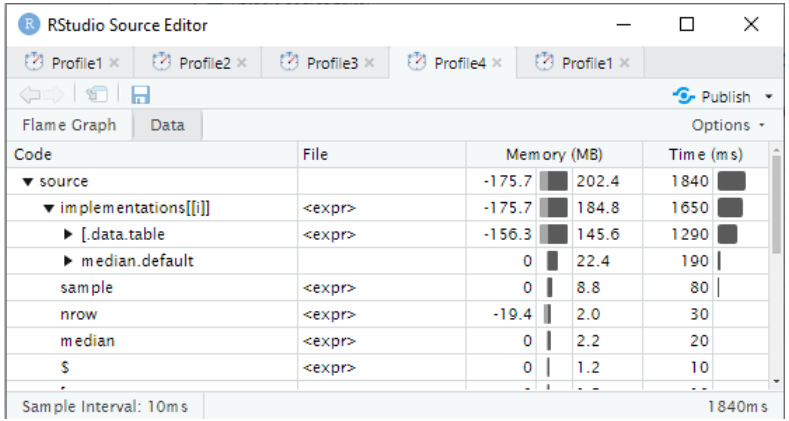



```r
impl_3 = function(population) {
    result <- list()
    for (i in seq_len(bootstrap_n)) {
      bootstrap_data_rows <- sample(
        x = seq_len(nrow(population)),
        size = bootstrap_size,
        replace = TRUE
      )
      current_boot <-
        population[bootstrap_data_rows][(analysis_flag)]
      current_result <- tibble(
        bootstrap_index = i,
        computed_output = median(current_boot$dummy_measurement)
      )
      result[[i]] <- current_result
    }
    return(bind_rows(result))
  }
```

Create list of tibbles, then bind_rows on list instead of iterative bind_rows

Demo: 5k bootstraps on 1k patients

Try it yourself:
→ optimizebootstrap.R

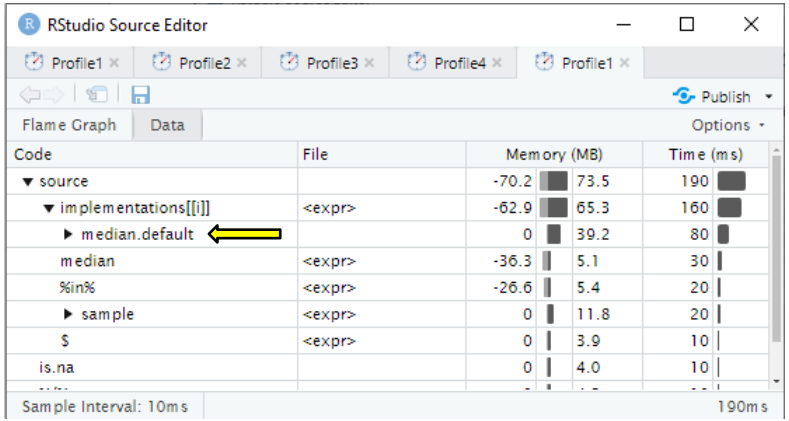



```r
impl_4 = function(population) {
    computed_output <- numeric(bootstrap_n)
    for (i in seq_len(bootstrap_n)) {
      bootstrap_data_rows <- sample(
        x = seq_len(nrow(population)),
        size = bootstrap_size,
        replace = TRUE
      )
      current_boot <-
        population[bootstrap_data_rows][(analysis_flag)]
      computed_output[i] <- median(current_boot$dummy_measurement)
    }
    return(
      tibble(
        bootstrap_index = seq_len(bootstrap_n),
        computed_output = computed_output
      )
    )
  }
```

Create the results tibble only at the end from vectors (and only once)

Demo: 5k bootstraps on 1k patients

Try it yourself:
→ optimizebootstrap.R




```
impl_5 = function(population) {
    computed_output <- numeric(bootstrap_n)
    analysis_indices <- which(population$analysis_flag)
    for (i in seq_len(bootstrap_n)) {
      bootstrap_data_rows <- sample(
        x = seq_len(nrow(population)),
        size = bootstrap_size,
        replace = TRUE
      )
      current_bootstrap_indices <-
bootstrap_data_rows[bootstrap_data_rows %in% analysis_indices]
      computed_output[[i]] <-
median(population$dummy_measurement[current_bootstrap_indices])
    }
    return(
      tibble(
        bootstrap_index = seq_len(bootstrap_n),
        computed_output = computed_output
      )
    )
  }
```

Subset the column of data needed for analysis only (rather than the data frame)

Demo: 5k bootstraps on 1k patients

Try it yourself:
→ optimizebootstrap.R

Variant	Change	Runtime
1	(baseline)	~ 20 s
2	Subset with data.table instead of filter	~ 5.3 s
3	Create list of tibbles, then bind_rows on list instead of iterative bind_rows	~ 4.8 s
4	Create the results tibble only at the end from vectors (and only once)	~ 1.8 s
5	Subset the column of data needed for analysis only (rather than the data frame)	~ 200 ms

100x speedup!
Identical result!

I know which part of my code is slow and cannot make it faster... what now?

This is the point where you should consider parallelizing on the HPC cluster:

If your time-consuming step is a loop, does the next iteration depend on the results of the last one?

- If yes, parallelization will likely be more difficult
  Example: Stan within-chain parallelization, ...
- If not, you can probably run each iteration on a different CPU core on the cluster
  Example: bootstrapping, cross-validation, simulation studies under replication, ...

These cases are the focus of the next part of this seminar: so-called «embarassingly parallel» problems ☺

Amdahl's Law

Given code where a fraction p can be parallelized, the speedup on s processors can be calculated as

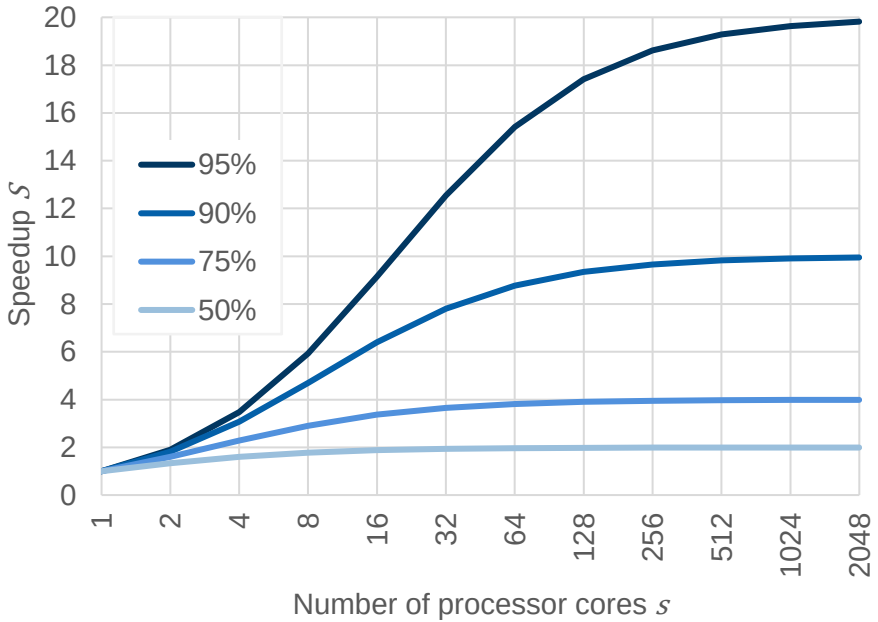
$$S(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

The more processors s, the faster the code runs. The maximum speedup is determined by the fraction of the code that cannot be parallelized:

$$S(\infty) = \frac{1}{1-p}$$

Big speedups are only possible if a large portion of the program can be parallelized!

→ Parallelizing is not magic.



Rodgers, D. P. (1985). Improvements in multiprocessor system design. ACM SIGARCH Computer Architecture News, 13(3), 225–231.

Part II

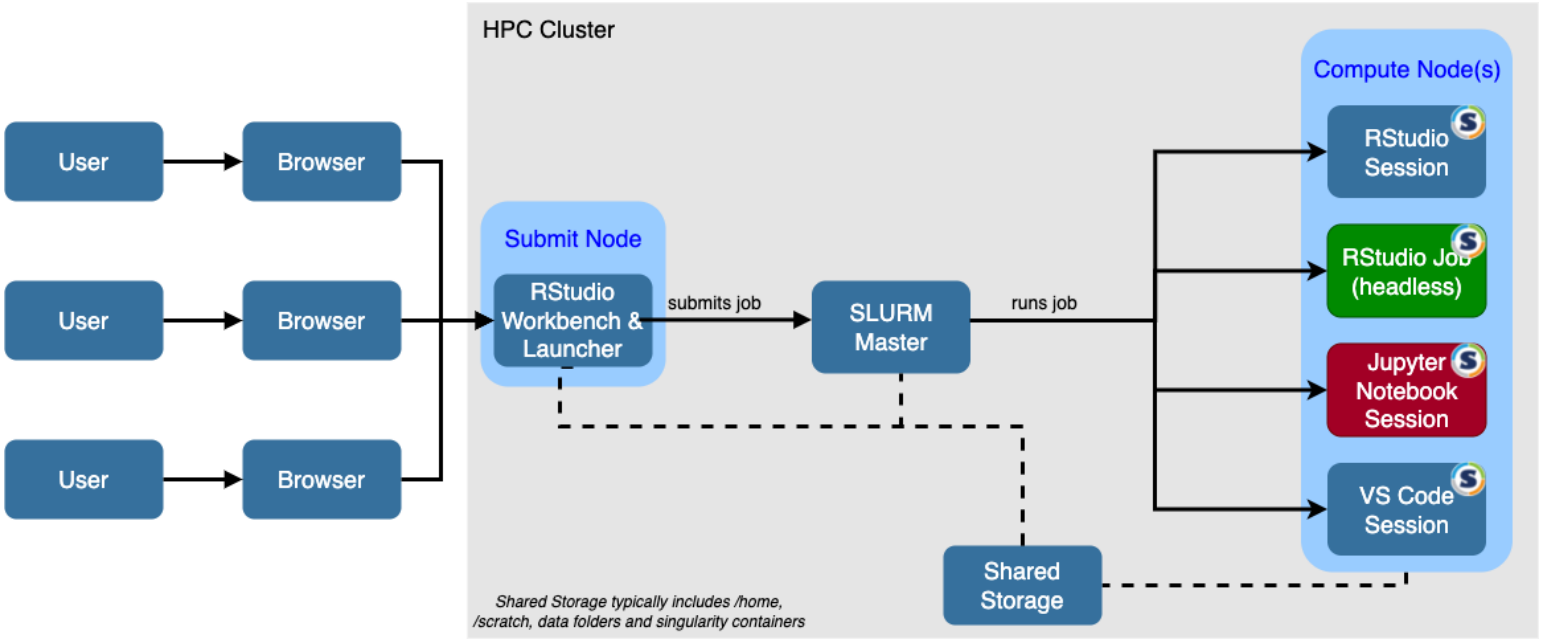
R parallelization on high performance computing environments (HPC)

R parallelization on the HPC: Background information

It is good to know some basics to help you get going on the HPC:

- (Rough) architecture of the system
- How to access the HPC
- What storage locations you can use
- How to start your jobs
- Fair use of the system

Rstudio in the cloud – overview



Rstudio in the cloud – getting access

- Username and password will be shared by the instructors
- Log into

URL

Storage locations

/data/home/<youruser>/...
Your user home directory

/scratch/...
Fast shared temporary space (files will typically deleted after X days without accessing them – your files are not safe here!)

/tmp/...
Local machine temporary directory (typically ~ a few GB, cleared at reboot, no executables)

/opt/R/...
Location of R installation

HPC – Schedulers

- High Performance Computing environments (HPCs) typically use a *scheduler* to manage batch or interactive jobs.
  - Batch: non-interactive – Interactive: you get a console where you can type commands
  - Typical examples:
    IBM Load Sharing Facility (LSF), SLURM, PBS/Torque, Altair Grid Engine
- The process works as follows:
  - Jobs first enter a queue and will be distributed to worker nodes depending on hardware availability and the specified requirements
  - Typically, different queues exist (e.g., for short/long jobs, jobs requiring GPUs, ...)
- Schedulers can be used in conjunction with R packages such as {clustermq} and {batchtools}. If you have access to an HPC, typically, sensible, pre-defined defaults exist (only customize if needed or setting up your own).

HPC – Fair use

Only **a very small set of restrictions** exist with regards to total number of running jobs and resources occupied by one user. While this allows for maximum potential speedups, since total capacity is capped, this means that **one user can potentially negatively impact the performance for all other users**.

Stakes are low in this training environment. At your institution, when submitting jobs, ensure that your resource request is meaningful and does not harm other users. **BE FAIR**.

Check available resources
(this will depend on the cluster at your institution)

Things to consider when using an HPC

- Wait times on a HPC cluster are normal.
  - Jobs are processed according to the assigned priority.
- SLURM commands for
  - Currently-used and available CPU cores: `sinfo -o "%20P %20n %10e %10m %5a %4c %20C"`
  - Running and pending jobs: `squeue`
- Non-interactive and interactive jobs:
  - Interactive jobs have a GUI or console (à la ssh) session on a cluster node.
  - If the cluster is full and you want to start an interactive session, this can cause waiting times – keep this in mind. Typically HPC admins configure different partitions for interactive and non-interactive work to optimize for better user experience.

The {clusterMQ} and {batchtools} R packages can submit to different backends

  - Your laptop (multi-process or local session)!
  - Remote computers via SSH
  - HPCs via a scheduler

- Backends can easily be substituted (often without changes to the R user code)
  - Backend logic is hidden in templates
  - This makes moving code to a compute cluster easy ☺

- {clusterMQ} and {batchtools} interface with the scheduler to submit jobs from R directly – no need to use `bsub` or `sbatch` (SLURM commands) in the terminal.

Parallelizing with {clusterMQ} locally

Changing backends is easy – e.g.:

Try it yourself:
→ localclustermqdemo.R

localclustermqdemo.R

Source on Save

```
1  # Do local debugging for 2 values of x -------
2  options(clustermq.scheduler = "LOCAL")
3  library(clustermq)
4
5  fx_debug <- function(x) {browser(); x * 2}
6  Q(fx_debug, x = 1:2)
7
8  # Run on 3 cores locally via 3 R processes ---
9  fx <- function(x) {x * 2}
10 options(clustermq.scheduler = "multiprocess")
11 Q(fx, x = 1:6, n_jobs = 3)
```

locally debugging jobs sequentially in the main R session

locally running with 3 R workers

Parallelizing with {clusterMQ} and {batchtools} on a laptop vs HPC cluster

Changing backends is easy – reference for {clustermq} and {batchtools}:

		{clusterMQ}
		`options(clustermq.scheduler=)`
			{batchtools}
			`reg = makeRegistry(NA)`
			`reg$cluster.functions =`

Local	**Local (main) R session: very useful for debugging code interactively**	`LOCAL`	`makeClusterFunctionsInteractive()`
	Multiple R processes on a single machine (e.g., a laptop)	`multiprocess`	`makeClusterFunctionsSocket(N)`
HPC Cluster	LSF	`lsf`	`makeClusterFunctionsLSF()`
	SLURM	`slurm`	`makeClusterFunctionsSlurm()`
	PBS	`pbs`	`makeClusterFunctionsTORQUE()`
	TORQUE	`Torque`	
	Grid Engine	`sge`	`makeClusterFunctionsSGE()`

The {clustermq} and {batchtools} R packages submit HPC jobs for you

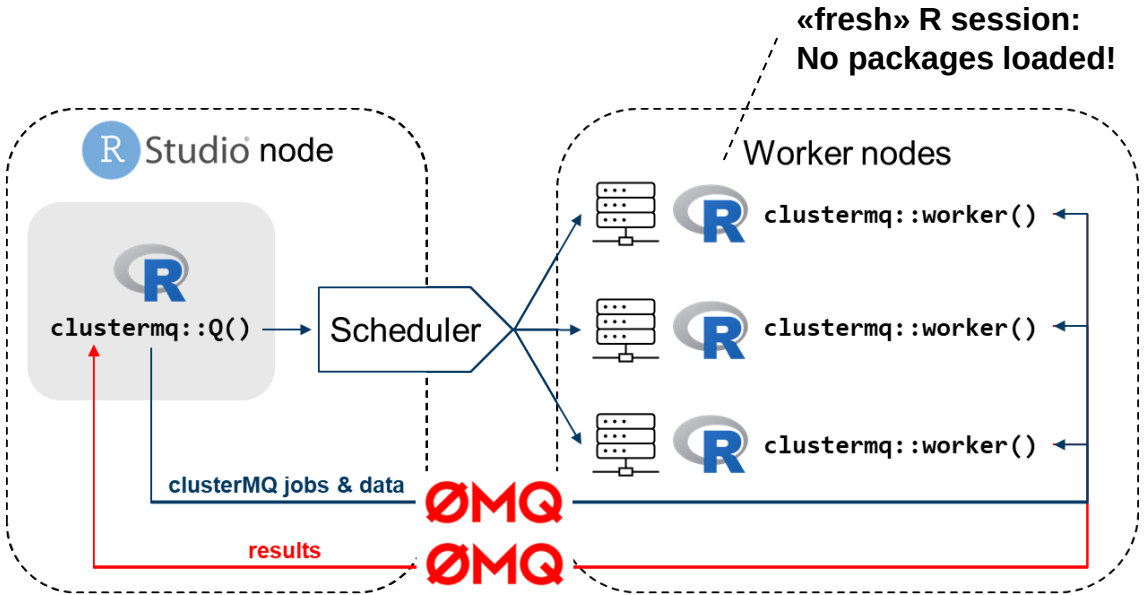
- {clustermq} and {batchtools} have two different philosophies:
  - {clustermq}:
    1. Submit one job per CPU core (or per x CPU cores that are needed for your program) that starts an R session and runs `clustermq::worker`
    2. The head node (e.g., through a web-based RStudio session) then sends jobs to the workers & receives results, and sends new jobs as long as there are unfinished ones.
    3. When all the clusterMQ jobs are done, shuts down workers & returns the results.

Understanding {clustermq} basics:

```
library(clustermq)
fx = function(x) x * 2
Q(fx, x=1:6, n_jobs=3)
```

`Q()` does the following:

1. Submit `n_jobs` R workers (via scheduler)
2. Connect to the node that called `Q()`, get clusterMQ jobs* & data
3. Receive & aggregate results
4. Shut down workers



* here we have 6 clusterMQ jobs, i.e., 2 clusterMQ jobs per *persistent* worker

Michael Schubert, clustermq enables efficient parallelization of genomic analyses, Bioinformatics, Volume 35, Issue 21, 1 November 2019, Pages 4493–4495 and https://cran.r-project.org/package=clustermq

Understanding {clustermq} basics:

```
library(clustermq)
fx = function(x) x * 2
Q(fx, x=1:6, n_jobs=3)
```

The R output
then looks like this*:

```
R 4.1.0> library(clustermq)
R 4.1.0> fx = function(x) {Sys.sleep(1); x * 2}
R 4.1.0> Q(fx, x=1:6, n_jobs=3)
Submitting 3 worker jobs (ID: cmq9706) ...
Running 6 calculations (0 objs/0 Mb common; 1 calls/chunk) ...
Master: [4.0s 1.1% CPU]; Worker: [avg 5.1% CPU, max 249.5 Mb]    17% (3/3 wrk) eta:  9s
[[1]]
[1] 2

[[2]]
[1] 4

[[3]]
[1] 6

[[4]]
[1] 8

[[5]]
[1] 10

[[6]]
[1] 12
```

Try it yourself:
→ clustermqdemo.R

How to set the number of jobs?
Typically, use as many jobs as there are values for x, up to the maximum responsibly usable on the cluster

* Function `fx` slowed down with `Sys.sleep` for demo purposes

Michael Schubert, clustermq enables efficient parallelization of genomic analyses, Bioinformatics, Volume 35, Issue 21, 1 November 2019, Pages 4493–4495 and https://cran.r-project.org/package=clustermq

The {clustermq} and {batchtools} R packages submit batch jobs for you

- {clusterMQ} and {batchtools} have two different philosophies:
  - {batchtools}:
    1. Save one job file per job into a shared directory (typically somewhere in /scratch)
    2. Schedule one* batch job that runs R on the job file for each batchtools job
    3. Each batch job saves an output file
    4. Wait for all the batch jobs to complete
    5. Aggregate / process results

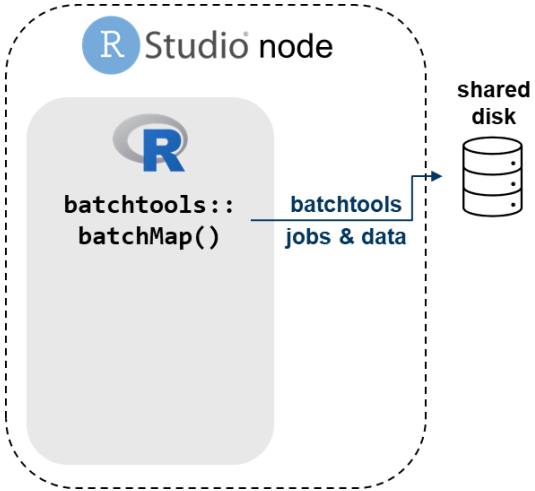
    * by default, one batch job can execute multiple batchtools jobs through chunking

Understanding {batchtools} basics

This creates a temporary registry

```
library(batchtools)
makeRegistry(file.dir=NA)
fx = function(x) x * 2
batchMap(fun = fx, x = 1:6)
submitJobs(); waitForJobs()
reduceResultsList()
removeRegistry()
```

1. `makeRegistry()` creates a folder on a shared disk
2. `batchMap()` writes jobs to that folder



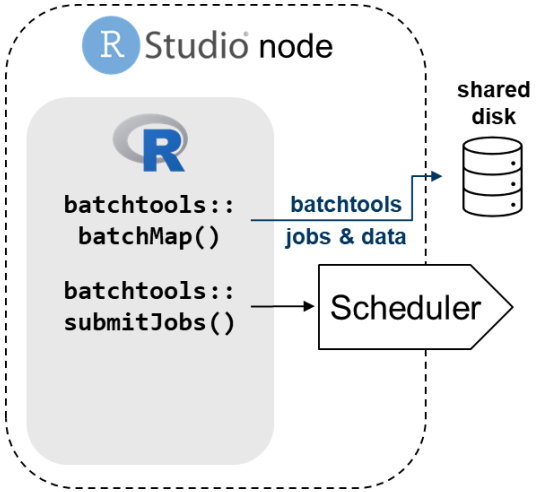
Try it yourself:
→ batchtoolsdemo.R

Lang et al, (2017), batchtools: Tools for R to work on batch systems, Journal of Open Source Software, 2(10), 135. and https://cran.r-project.org/package=batchtools

Understanding {batchtools} basics

```
library(batchtools)
makeRegistry(file.dir=NA)
fx = function(x) x * 2
batchMap(fun = fx, x = 1:6)
submitJobs(); waitForJobs()
reduceResultsList()
removeRegistry()
```

1. `makeRegistry()` creates a folder on a shared disk
2. `batchMap()` writes jobs to that folder
3. `submitJobs()` submits the jobs and `waitForJobs()` waits for them to complete



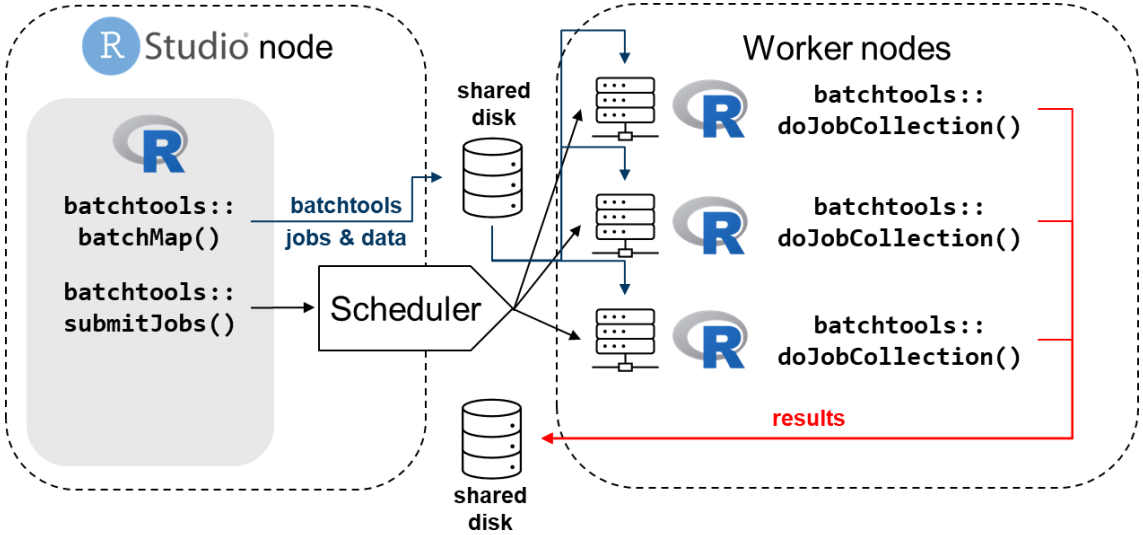
Try it yourself:
→ batchtoolsdemo.R

Lang et al, (2017), batchtools: Tools for R to work on batch systems, Journal of Open Source Software, 2(10), 135. and https://cran.r-project.org/package=batchtools

Understanding {batchtools} basics

```
library(batchtools)
makeRegistry(file.dir=NA)
fx = function(x) x * 2
batchMap(fun = fx, x = 1:6)
submitJobs(); waitForJobs()
reduceResultsList()
removeRegistry()
```

1. `makeRegistry()` creates a folder on a shared disk
2. `batchMap()` writes jobs to that folder
3. `submitJobs()` submits the jobs and `waitForJobs()` waits for them to complete

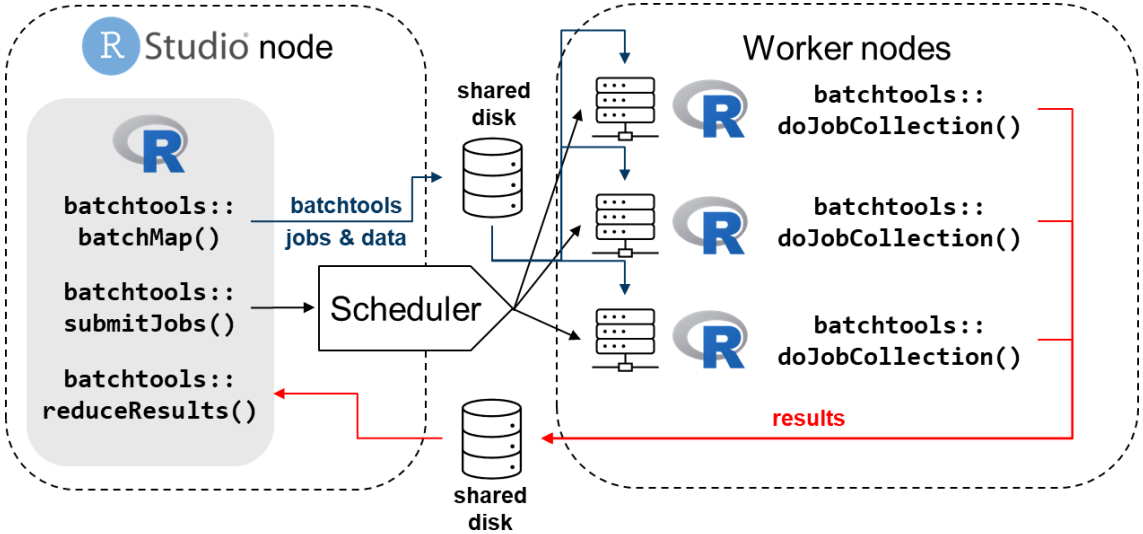


Lang et al, (2017), batchtools: Tools for R to work on batch systems, Journal of Open Source Software, 2(10), 135. and https://cran.r-project.org/package=batchtools

Understanding {batchtools} basics

```
library(batchtools)
makeRegistry(file.dir=NA)
fx = function(x) x * 2
batchMap(fun = fx, x = 1:6)
submitJobs(); waitForJobs()
reduceResultsList()
removeRegistry()
```

1. `makeRegistry()` creates a folder on a shared disk
2. `batchMap()` writes jobs to that folder
3. `submitJobs()` submits the jobs and `waitForJobs()` waits for them to complete
4. `reduceResultsList()` loads the results from disk into a list
5. `removeRegistry()` deletes the folder



Lang et al, (2017), batchtools: Tools for R to work on batch systems, Journal of Open Source Software, 2(10), 135. and https://cran.r-project.org/package=batchtools

Why is {clustermq} so much faster?

- {batchtools} saves job & results files to network-shared storage (slow!)
  - {clustermq} does not, and has load balancing over *persistent* workers
  - ➢ Much lower overhead!
- There is a tradeoff between speed and stability:
  - {batchtools} allows to restart specific jobs that crashed (e.g., due to memory constraints) – {clustermq} would not return any results / require a re-run.

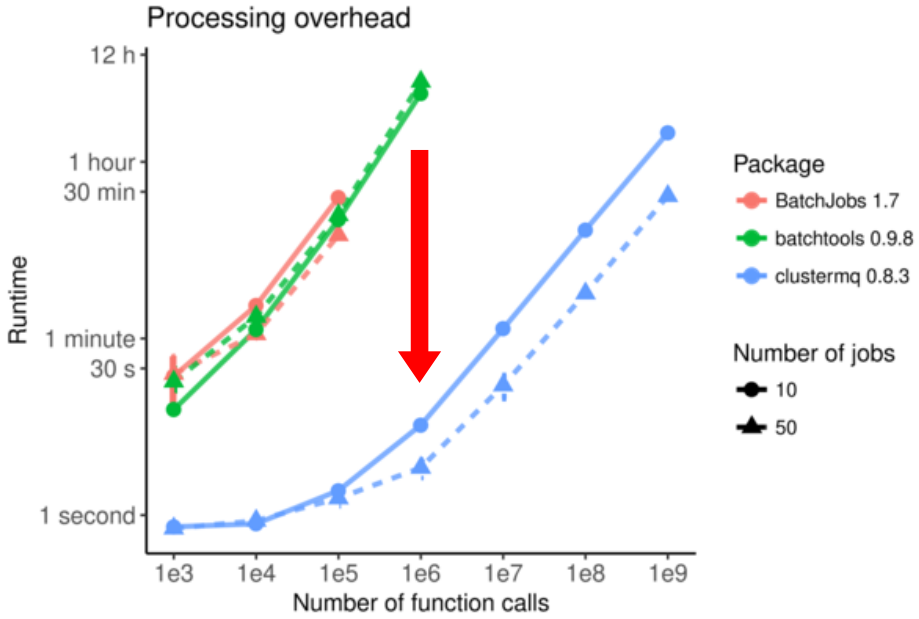


Figure: https://mschubert.github.io/clustermq/

Michael Schubert, clustermq enables efficient parallelization of genomic analyses, Bioinformatics, Volume 35, Issue 21, 1 November 2019, Pages 4493–4495 and https://cran.r-project.org/package=clustermq

Configuration file locations & resource settings

Default configuration file locations:

- {batchtools}: location determined by **`batchtools::findConfFile()`**
- {clustermq}: location defined by R option: **`getOption("clustermq.template")`**

These files set the job parameters for the cluster scheduler, e.g., resources:

Resource	{clustermq}	{batchtools}	Our defaults
Cores	`cores`: #	`ncpus`: #	1 core
Wall time	`walltime`: minutes	`walltime`: seconds	1 hour
Memory	`memory`: megabytes	`memory`: megabytes	1 GB

Setting specific resource requirements for {batchtools} and {clustermq}:

```
submitJobs(...,
  resources = list(
    walltime = 3600,
    memory = 3072,
    ncpus = 1,
    max.concurrent.jobs = N
))
```

```
Q(..., n_jobs = N,
  timeout = 180,
  template = list(
    walltime = 60,
    memory = 3072,
    cores = 1
))
```

→Unless you know better, parallelize the outermost loop, and use 1 core per job.

- A subtlety with {clustermq} is to set a per-clustermq job timeout (on top of the walltime of the R session for the cluster job scheduler; in seconds).
  - If you don't set this, {clustermq} in the main R session may hang *forever* if an R worker crashes!

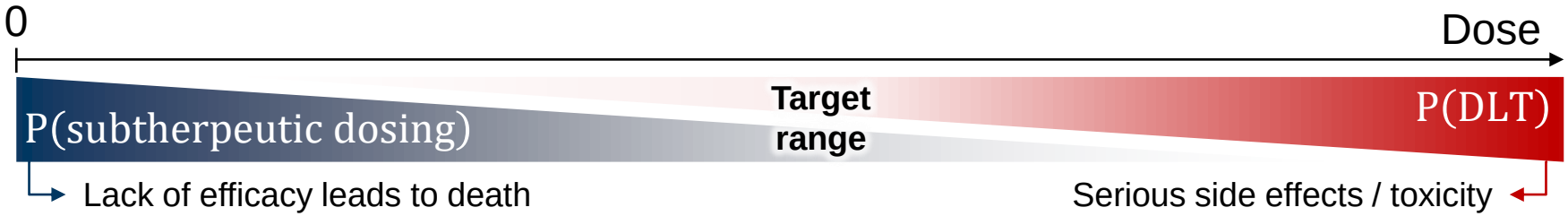
Parallelization workflow

1. Optimize your code locally
2. Run the code through clustermq (or batchtools), but using the local backend rather than on the cluster.
   → This allows you to locally debug (e.g., using `browser()`)
3. Run a small (2-5 replications) test on the cluster
   – Errors commonly occur here because local debugging uses the same interactive R session (with the same loaded packages, etc.), whereas cluster R jobs will not.
4. Once all of this works, run your full workload

Part III

Case studies & best practices

Oncology phase I dose escalation: a delicate balance between sub-therapeutic & toxic dosing



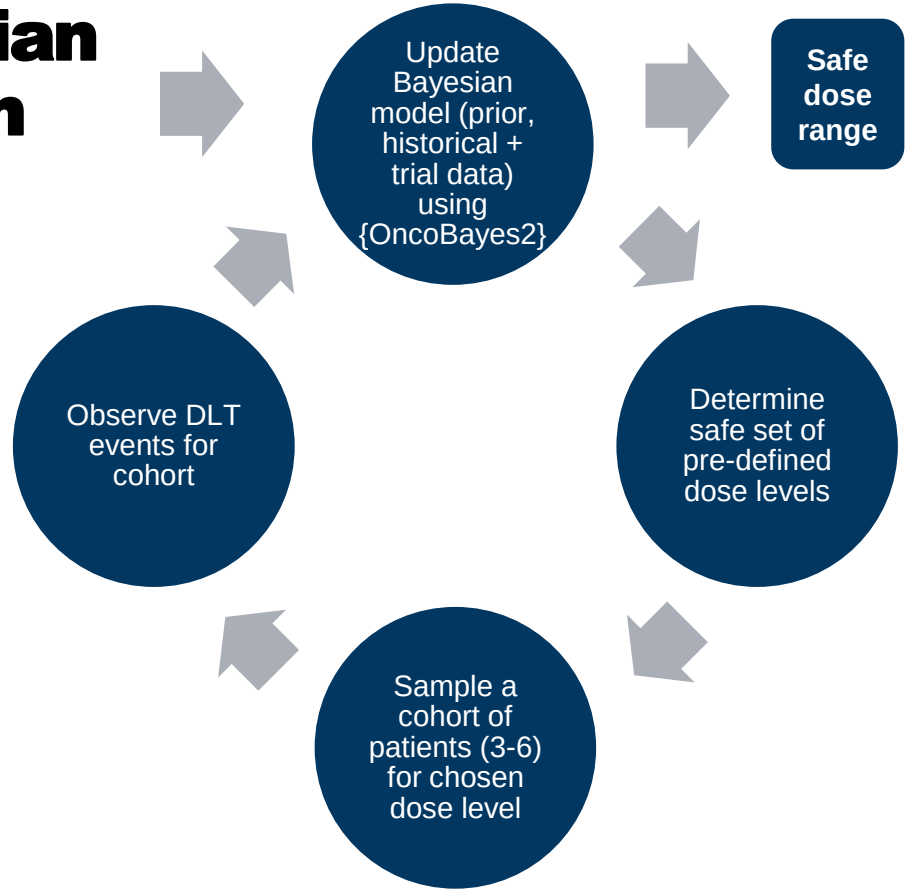
- Volunteers are “patients ... whose cancers progressed despite standard treatments”[1]
  - Initially: limited knowledge on toxicity: phase I trial to determine safety of new drug.
  - Need to limit risk to current and future patients[2] ⇒ can initially only use small cohorts.
- Goal: systematically increase the dose as quickly & safely as possible to determine the safe dose range for further study.

1. Le Tourneau, C. *et al*. Dose escalation methods in phase I cancer clinical trials. *J. Natl. Cancer Inst.* **101**, 708–720 (2009).
2. Babb, J *et al*. Cancer phase I clinical trials: Efficient dose escalation with overdose control. *Stat. Med.* **17**, 1103–1120 (1998).

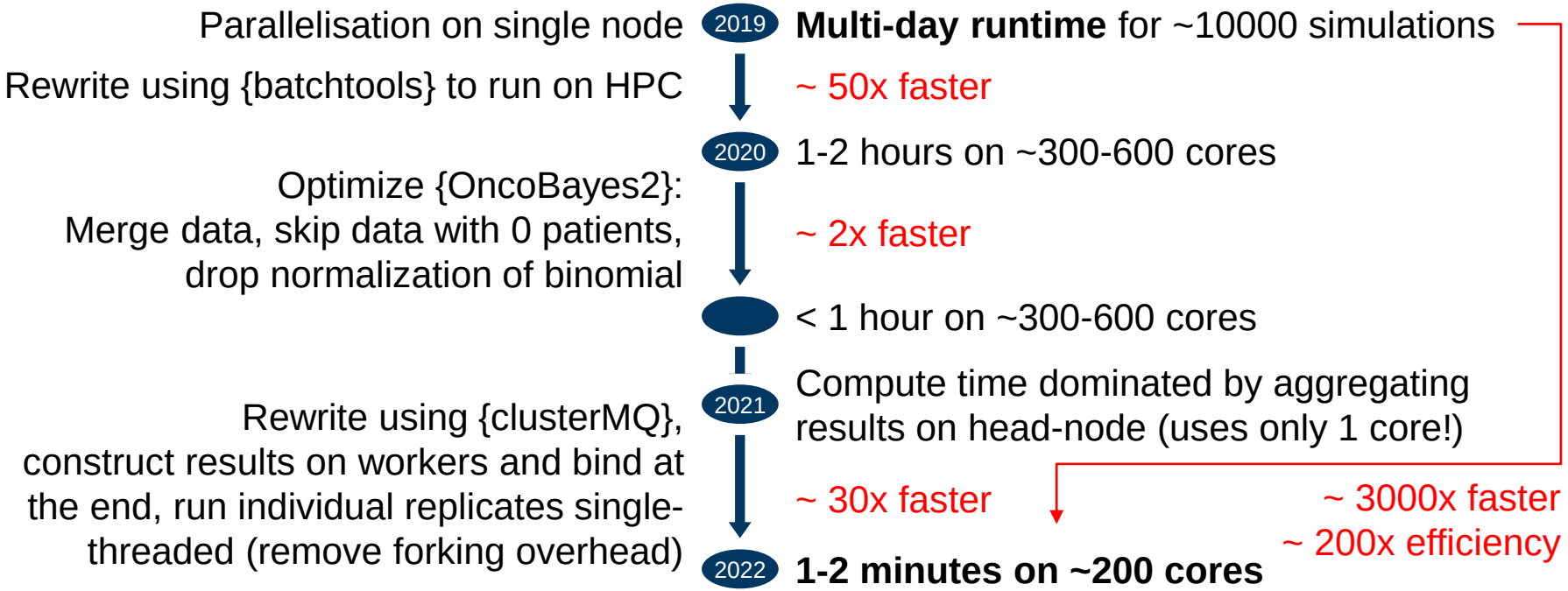
Oncology phase I Bayesian adaptive dose escalation

Model-based escalation with small sample sizes ⇒ busy design stage:

- Need to assess short- and long-term operating characteristics through simulation – lots of it:
  - Many different trajectories possible: Cohort sizes & events are sampled from different possible scenarios
  - **Originally, this needed days of compute time!**



Making the submission workload *fast*



Typical parallel workloads follow a pattern

- Simulation studies under replication, bootstrapping and cross validation workloads all follow a similar pattern of computation:

Step	Simulation study	Bootstrapping	Cross validation
1. Preparation	Definition of ground truth «scenarios»	Prepare data	Partition data into cross validation folds
2. Parallel computation	Simulate (independent) trials	Resample data & perform analysis (independent of other resampled datasets)	Perform analysis on training folds, evaluate on validation fold (independent of other training/validation splits)
3. Results aggregation	Compute metrics, e.g., trial operating characteristics	Compute metrics (e.g., confidence intervals) across bootstraps	Compute performance across data splits

Typical parallel workloads follow a pattern

- Simulation studies under replication, bootstrapping and cross validation workloads all follow a similar pattern of computation:

Step	Simulation study	Bootstrapping	Cross validation
1. Preparation	Main R session (e.g., RStudio IDE)		
2. Parallel computation	R workers (e.g., via HPC jobs)		
3. Results aggregation	Compute metrics on R workers, aggregate results in main R session		

Dragons await: R workers need to be set up so they execute work as the main R session would

In particular, library locations, loaded libraries and options set in the main R session must *also* be set on the R workers **on startup**:

- If library locations are set by `.libPaths()`, this should be done consistently!
- R Packages loaded with `library()` or `require()` must also be loaded on the R workers, not just in the main R session!
- When specifying `options()` such as the number of digits to display, etc., again, this must also be done on the R workers on startup, otherwise there will be inconsistencies with the main R session!

Remember this when locally debugging jobs in the main R session!

Random numbers for parallel R jobs – how not to do it

- Each parallel job will need a pseudorandom number generator (PRNG).
- Can I just `set.seed(i)` for $i = 0,1,\ldots,N-1$ in each of my parallel R jobs?
  - **No!** Why?

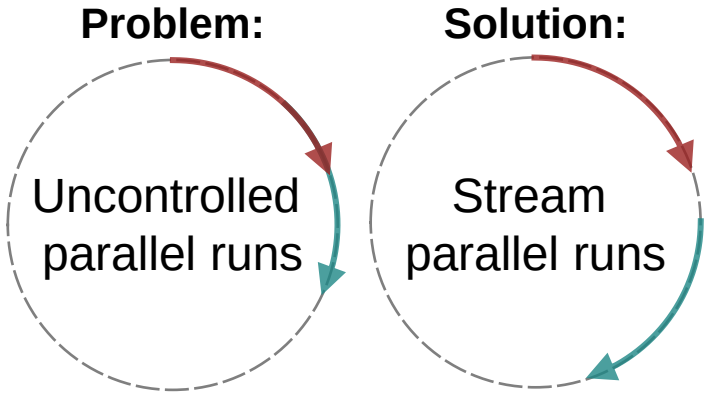
```
> set.seed(0)
> x <- runif(10)
> set.seed(1)
> y <- runif(10)
> x
 [1] 0.897 0.266 0.372 0.573 0.908 0.202 0.898 0.945 0.661 0.629
> y
 [1] 0.2655 0.3721 0.5729 0.9082 0.2017 0.8984 0.9447 0.6608 0.6291 0.0618
> x[2:10] == y[1:9]
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

The first two jobs will use the same «random» numbers (except for one)!
→ Adjacent seeds do not guarantee uncorrelated PRNG streams

Random numbers for parallel R jobs – a better way

1. Set the seed once in the main job via `set.seed`
2. Derive **uncorrelated random number streams** that are guaranteed to not overlap for each parallel job
   - **E.g., L'Ecuyer's Random Number Generator** (RNG) in the parallel package is designed for this.
     Each 'stream' is a subsequence of the period of length 2^{127}
     by construction! R example for `s1` and `s2` below:

```
library(parallel)
RNGkind("L'Ecuyer-CMRG")
set.seed(384634)
s1 <- nextRNGStream(.Random.seed)
s2 <- nextRNGStream(s1)
```




```
RNGkind("L'Ecuyer-CMRG")
.Random.seed <<- s1
```
Job 1

```
RNGkind("L'Ecuyer-CMRG")
.Random.seed <<- s2
```
Job 2

For an overview of best practices (ADEMP) and common pitfalls, see
Morris *et al* (2019). Using simulation studies to evaluate statistical methods. *SIM*, *38*(11), 2074–2102.

We provide some template code with "batteries included", in particular:

- `.libPaths()` and `options()` will be transferred from the main R session to the R workers
- They also load all the packages in `load_packages.R` consistently both in the main R session and on the R workers on the HPC (only once per HPC job on worker startup, not for each {clustermq} job).
- The R pseudorandom number generator is set up to generate uncorrelated random numbers between clustermq jobs.
- Options are provided to first locally test and debug single jobs before firing off a lot of jobs to the cluster

Template for more complex workloads

Template structure:

`main.R`: Main file, defines what to run and how many replications / bootstraps

`cluster_engine.R`:

- Provides infrastructure to aggregate results fast (also into a data frame)
- Wraps clustermq to provide additional features (next slide)

`load_packages.R`: Defines which packages to load (and `.libPaths` if needed)

`simulate_trial.R` / `bootstrap.R` / `cv.R`: Code for actual workload

Debugging remote R jobs

- If a remote job fails, the templates provide a call stack on top of the clustermq error to help you locate the issue:

```
Clustermq run - enumerating clustermq experiments...
Clustermq run - submitting and running experiments...
Clustermq run - submitting jobs at 2022-06-09 10:32:03
Running sequentially ('LOCAL') ...
Current job:  1
Calling setup function:
Error in job_id 1: This is a simulated error
Call stack:
1: simulate_error()
2: stop("This is a simulated error")
```

- The `run_batch` function in `cluster_engine.R` supplies two arguments to assist in debugging by running an offending job locally instead (so you can e.g., use `browser()` and investigate):
  - `test_single_job_index = c(11,12)`
    → Use this to test single jobs (e.g. if job_id 11 and 12 crashes, set this to `c(11,12)`)
  - `test_single_job_per_experiment = FALSE`
    → Use `TRUE` to test 1 replication per experiment

Your own case study

Let us know if you have questions or need help!

You can also try the example code in case you don’t have a case study with you. Available in the *fastR-example-code* folder of your home-directory

Summary of resources & further reading

All the course material is available online.

Key papers:

- Morris *et al* (2019). Using simulation studies to evaluate statistical methods. SIM.
- Schubert (2019), clustermq enables efficient parallelization of genomic analyses, Bioinformatics.
- Lang *et al* (2017), batchtools: Tools for R to work on batch systems, JOSS.

New packages to keep tabs on for building fast R pipelines:

- {mirai}: NNG protocol (successor to zeroMQ) via {nanonext}
- {crew} and {crew.cluster}: distributed launcher using {mirai}, backend for {targets}

MCMC within-chain parallelization in Stan:

- Running brms models with within-chain parallelization (r-project.org)

Further example for use of {clustermq}

Related conference sessions

- Monday, September 4th
  - 11:00 – 12:40: **Neutral comparison studies in methodological research**
  - From 14:00: **Statistics in Practice:** Simulation studies as a tool to assess and compare the properties of statistical methods – an overview
- Tuesday, September 5th
  - 16:10 – 17:50: **Simulation studies**
- Wednesday, September 6th
  - 10:40 – 12:20: **Software engineering**
    - 11:00: An overview of R software tools to support simulation studies: towards standardizing coding practices
- Thursday, September 7th
  - From 14:35: **STRATOS Satellite Symposium** (STRengthening Analytical Thinking for Observational Studies)

Thank you for participating!

Please take 5 minutes to fill out a brief feedback survey about this seminar.

Thank you for joining today and for sharing your thoughts!

<Feedback Form Link>

Feel free to reach out to us if you have additional questions or suggestions:

✉ lukas_andreas.widmer@novartis.com

✉ michael.mayer@posit.co