Applied HPC with R

George G. Vega Yon

Contents

[1] Code copied verbatim from the `profvis` R package here.

Preface

The R programming language [1] can be fantastic for most daily tasks. But as soon as you start dealing with more complicated problems, you may face the for-loop bottle-neck. If you ever encounter such a problem, this book is for you. Applied HPC with R is a collection of talks and lectures I have given about speeding up your R code using parallel computing and other resources such as C++. The contents have been primarily developed during my time at USC and UofU.[2]

The book was written using quarto and is hosted on GitHub, where you can access all the source code.

About the Author

I am a Research Assistant Professor at the **University of Utah's Division of Epidemiology**, where I work on studying Complex Systems using Statistical Computing. I was born and raised in Chile. I have over ten years of experience developing scientific software focusing on high-performance computing, data visualization, and social network analysis. My training is in Public Policy (M.A. UAI, 2011), Economics (M.Sc. Caltech, 2015), and Biostatistics (Ph.D. USC, 2020).

I obtained my Ph.D. in Biostatistics under the supervision of **Prof. Paul Marjoram** and **Prof. Kayla de la Haye**, with my dissertation titled "*Essays on Bioinformatics and Social Network Analysis: Statistical and Computational Methods for Complex Systems.*"

If you'd like to learn more about me, please visit my website at https://ggvy.cl.

Spanish Version Available

A Spanish translation of this book is available at es/. The Spanish version was created using AI translation and may require human review for technical accuracy.

AI Disclosure

Starting in mid-2023, I have been using AI to help me write this book. Mainly, I use a combination of GitHub co-pilot, which aids with code and text. AI's role has been to help me write faster and more accurately, but it has not been involved in the book's conceptualization or the development of the methods presented here.

About the cover

The cover image was created using ChatGPT/Dall-E version 5 using the prompt "I need you to draw a picture representing high-performance computing using the R programming language (which is mostly used for statistics). The picture will go as the cover of the book"Applied HPC with R". It should feature components related to C++, parallel computing, and high-performance computing"

[2] With many to thank, including Paul Marjoram, Zhi Yang, Emil Hvitfeldt, Malcolm Barrett, Garrett Weaver, USC's IMAGE P01 research group, and my students both at USC and UoU.

I Fundamentals

[3]Code copied verbatim from the `profvis` R package here.

1. Profiling

Code profiling is a fundamental tool for programmers. With profiling, it is possible to **identify potential bottle necks in your code**, as well as exessive memory usage. Some important things to consider when profiling your code:

- **It has a stochastic component**: Unrelated to computational complexity, a program can exhibit different performance characteristics across different runs. This means that you should always profile your code multiple times and take the average performance as the final result.
- **It is worthless if the program is too quick**: Most of the time, code profiling must be done on programs that take a noticeable amount of time to run. If a program runs too quickly, the overhead of the profiling itself can skew the results. If you still need to address this, you can, for example, do multiple runs or use a larger dataset to increase the run time of the program.
- **Don't do it too early**: Most of the time, developers tend to optimizing (and thus profiling) their code too early. You don't want to spend time doing code profiling on something that could quickly change. So it is better to do it on a working code rather than pieces of it.

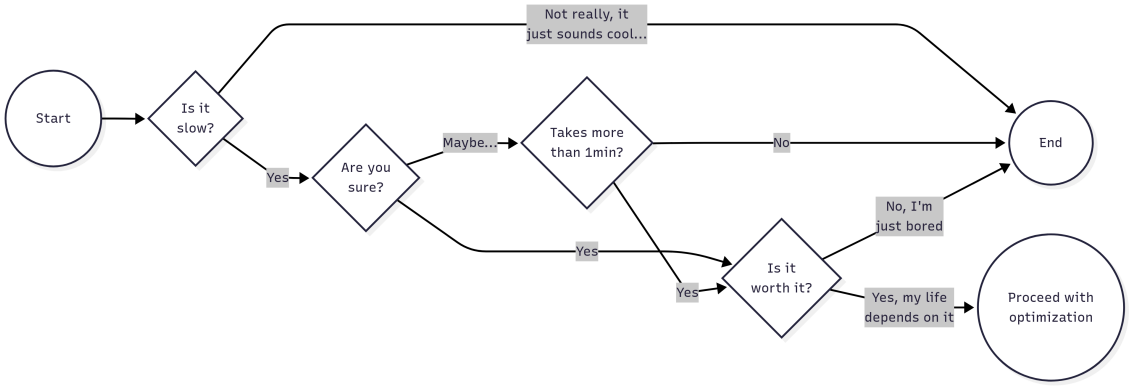
  Instead of profiling the code too early, make sure that you have a good design and implementation plan.
- **Be strategic**: Not everything needs to be optimized. Focus on the parts of the code that are critical for performance and user experience. You can think of it in terms of how much time the program/user spends on each bit.
- **AI can give you good tips**: In my personal experience, AI can be useful doing code profiling too. This works better if you are using an agentic AI, as it is more likely to get good feedback from it (it will test the code for you).

Here is a proposed workflow for profiling and optimizing your code in general

1.1 A proposed workflow

Here is a formula to follow when doing code profiling/optimization:

1. Ask yourself these questions before you jump into profiling:



2. If you succeed, then ensure that the profiling is done in a finite time, this is, use a subset of the data to avoid having long waits. **Running the profiler will add overhead computing time, so try to keep it short (e.g., 1 minute)**.
3. There may be many things that could be optimized, focus on what would deliver the highest impact. Could be a function that is only called once but takes a long time to execute, or a function that is called multiple times but is relatively fast.

4. Before making any changes, ensure that you have a backup of your original code, as well as a copy of the current profiling results. **You should also ensure to save (if possible) the outcome of the code.**
5. Re-run the profiler and compare performance. If no changes are observed, then go back to step 2. **Ensure the new version of the code maintains the same functionality as the original (check the results).**

1.2 Profiling code in R

In the R programming language, the most used profiling tool comes with the `profvis` package. The package provides a wrapper of the `Rprof` function, which is a built-in R function for profiling code. The `profvis` package makes it easier to visualize the profiling results in a web-based interface.

To use `profvis`, you first need to install it from CRAN:

```
install.packages("profvis")
```

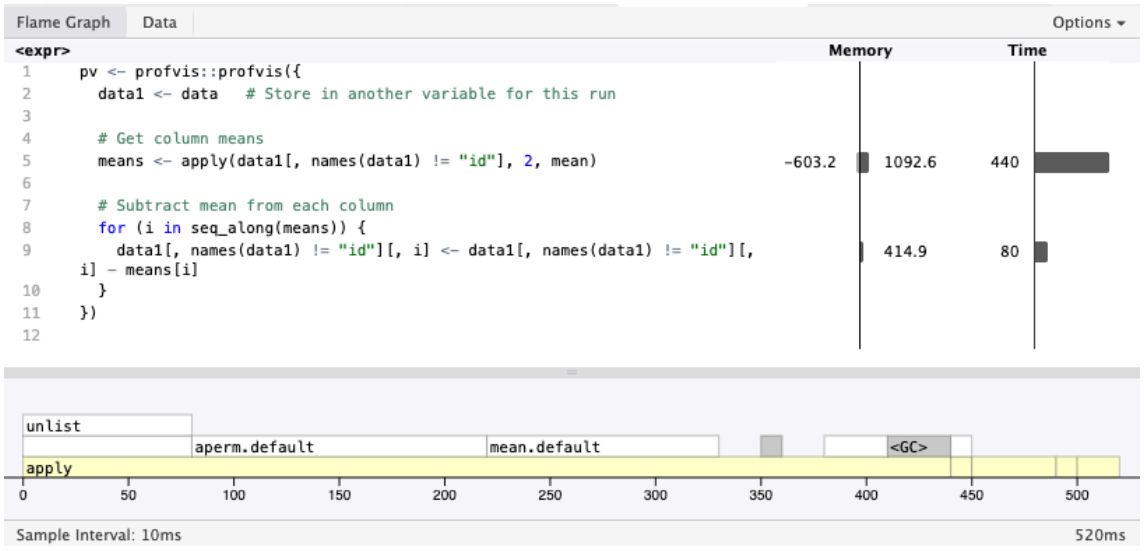
Then, you can use it to profile your R code like this:

```
library(profvis)

profvis({
  # Your R code here
})
```

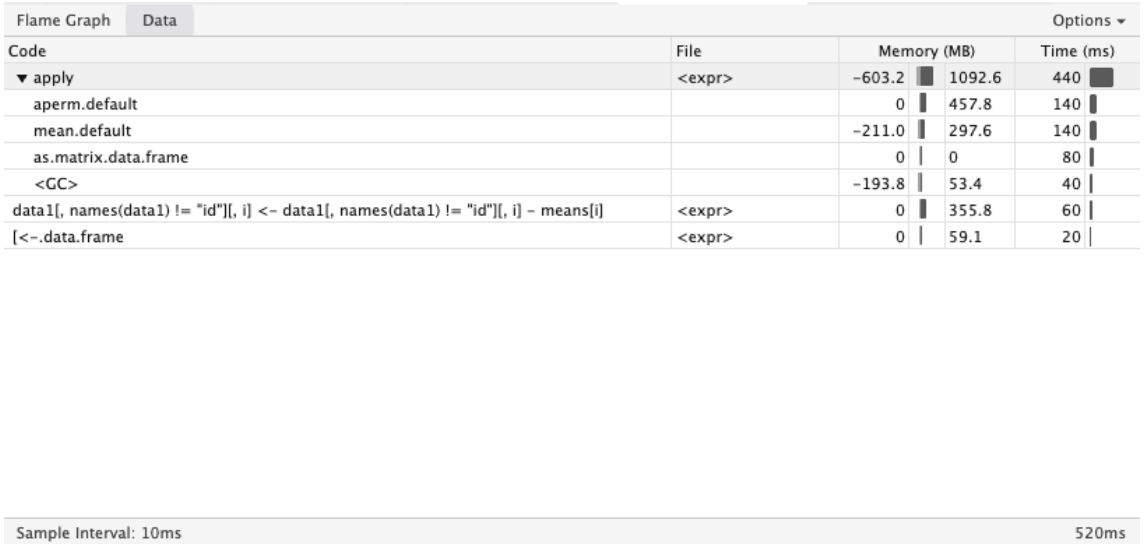
This will execute the code and generate a visualization of the profiling results in a new browser window. You can also save the output using the `htmlwidgets` package:

```
pv <- profvis({
  # Your R code here
})

htmlwidgets::saveWidget(pv, "profvis.html")
```

Once open, you will see two visualizations: the flamegraph and the data. The flamegraph is one of the most useful visualizations. It directly maps the time and memory used by each line of code



The data visualization shows the distribution of time spent in each function. Using a tree structure, which allows taking deep dives into the call stack.



Flame Graph | Data | Options

Code	File	Memory (MB)		Time (ms)
▼ apply	<expr>	-603.2	1092.6	440
aperm.default		0	457.8	140
mean.default		-211.0	297.6	140
as.matrix.data.frame		0	0	80
<GC>		-193.8	53.4	40
data1[, names(data1) != "id"][, i] <- data1[, names(data1) != "id"][, i] - means[i]	<expr>	0	355.8	60
[<-.data.frame	<expr>	0	59.1	20

Sample Interval: 10ms 520ms

> **Tip**
>
> When developing R packages, it is a good idea to pair your `profvis::profvis` call with `devtools::load_all()` to ensure that all source code is available to the profiler. Otherwise, the flamegraph won't show your code (it will say "unavailable").

1.3 Exercise: Identifying the bottle neck[4]

```
# Generate data
times <- 4e5
cols <- 150
data <- as.data.frame(x = matrix(rnorm(times * cols, mean = 5), ncol = cols))
data <- cbind(id = paste0("g", seq_len(times)), data)

pv <- profvis::profvis({
  data1 <- data   # Store in another variable for this run

  # Get column means
  means <- apply(data1[, names(data1) != "id"], 2, mean)

  # Subtract mean from each column
  for (i in seq_along(means)) {
    data1[, names(data1) != "id"][, i] <- data1[, names(data1) != "id"][, i] - means[i]
  }
})

htmlwidgets::saveWidget(pv, "profvis-slow-code.html")

# In interactive mode, we can directly view the profiling results
if (interactive())
  print(pv)
```

Can you identify where is the bottleneck in the code? What would you do to speed it up?

[4]Code copied verbatim from the `profvis` R package here.

2. Writing efficient code

- R code can be very efficient for typical tasks, but, as the code starts to increase in complexity, it is easy for it to become inefficient.
- Some quick R tips for efficient computing code:
  - Use vectorized operations instead of loops.
  - Try to use caching to avoid repeated calculations. Caching can also be done out of memory!
  - Avoid unnecessary steps/data processing.
  - Reduce the number of copy operations.

2.1 Vectorized Operations

Vectorization can mean many things in programming, but in R, vectorization refers to using functions over vectors. For instance, instead of using a loop to add two vectors together, you can use the + operator directly on the vectors:

```
# Using a loop
set.seed(331)
a <- runif(1e3)
b <- runif(1e3)
result <- numeric(length(a))
for (i in seq_along(a)) {
  result[i] <- a[i] + b[i]
}

# Using vectorized operation
result <- a + b
```

We can even bechmark the performance of these two approaches:

```
library(microbenchmark)
microbenchmark(
  loop = {
    result <- numeric(length(a))
    for (i in seq_along(a)) {
      result[i] <- a[i] + b[i]
    }
  },
  vectorized = {
    result <- a + b
  },
  unit = "relative"
)
```

```
Unit: relative
       expr      min       lq     mean   median       uq      max neval
       loop 2046.729 1462.041 718.6657 841.2948 640.4706 274.9027   100
 vectorized    1.000    1.000   1.0000   1.0000   1.0000   1.0000   100
```

> Tip
>
> For-loops are not always bad. The main issue is with the code inside of the for-loop. If the code is already vectorized, then there's no need to remove the for-loop (unless you can vectorize the for-loop itself).

2.2 Caching calculations

Many times, it is useful to cache calculations that are expensive to compute. For instance, if you have a function that takes a long time to run, you can store the result in a variable and reuse it later instead of recalculating it.

Here is a bad example using the Fibonacci sequence:

```
fibonacci <- function(n) {
  if (n <= 1) {
    return(n)
  }
  return(fibonacci(n - 1) + fibonacci(n - 2))
}

fibonacci_cached <- function(n) {
  prev <- numeric(n + 1)
  for (i in seq_len(n)) {
    if (i <= 1) {
      prev[i + 1] <- i
    } else {
      prev[i + 1] <- prev[i] + prev[i - 1]
    }
  }

  return(prev[n + 1])
}
```

Both of these functions should return the same result, but the second is significantly faster as it avoids calling the function recursively:

```
microbenchmark(
  fibonacci(10),
  fibonacci_cached(10),
  times = 10,
  unit = "relative",
  check = "equal"
)
```

```
Unit: relative
                expr      min       lq     mean   median       uq      max
       fibonacci(10) 25.20665 24.63582 17.86991 24.92241 23.30907 6.338251
 fibonacci_cached(10)  1.00000  1.00000  1.00000  1.00000  1.00000 1.000000
 neval
    10
    10
```

2.3 Caching calculations (bis)

In the case of large calculations, we can also save results to the disk. For example, if we are running a simulation/computation, one per city/scenario, we can save the results to a file and read them later. Here is how to do it:

For each value of `i`, do the following:

1. Check if the file `result_i.rds` exists.
2. If it does not exist, run the computation and save the result to `result_i.rds`.
3. If it does exist, read the result from `result_i.rds`.

As simple as that! Here is an example using R code:

```r
# A complicated simulation function
simulate <- function(i, seed) {
  set.seed(seed)
  rnorm(1e5)
}

# Generating seeds for each iteration
set.seed(331)
nsims <- 100
seeds <- sample.int(.Machine$integer.max, nsims)

# Just for this example, we will use a tempfile
res_0 <- vector("list", length = nsims)
for (i in seq_len(nsims)) {

  # Creating the filename
  fn <- file.path(tempdir(), paste0(i, ".rds"))

  # Does the result already exist?
  if (file.exists(fn))
    res_0[[i]] <- readRDS(fn)
  else {
    # If not, run the simulation and save the result
    res_0[[i]] <- simulate(i, seed = i)
    saveRDS(res_0[[i]], fn)
  }

}
```

> **Tip**
>
> When running simulations, it is a good pracitice to set individual seeds for each simulation (if these are individually complex). That way, if the code fails, you can rerun only the failed simulations without having to redo all of them.

Furthermore, it is a good idea to wrap your code in a `tryCatch()` call to handle errors gracefully. This way, if a simulation fails, you can log the error and continue with the next simulation without stopping the entire process.

```r
# Just for this example, we will use a tempfile
res_0 <- vector("list", length = nsims)
for (i in seq_len(nsims)) {

  # Creating the filename
  fn <- file.path(tempdir(), paste0(i, ".rds"))

  # Does the result already exist?
  res <- tryCatch({
    if (file.exists(fn))
      readRDS(fn)
    else {
      # If not, run the simulation and save the result
      ans_i <- simulate(i, seed = i)
      saveRDS(ans_i, fn)
      ans_i
```

```
    }
  }, error = function(e) e)

  if (inherits(res, "error")) {
    message("Simulation ", i, " failed: ", res$message)
    next  # Skip to the next iteration
  }

  # We still store it, even if it failed
  res_0[[i]] <- res

}
```

> Tip
>
> The `saveRDS` function in R uses the `compress = TRUE` argument as default. Compressing the data for saving space is generally a good idea, but not if you need to read data fast. So, if space is not a constraint, you can set `compress = FALSE` when saving the RDS file to accelerate the reading process.

2.4 Caching calculations in a ShinyApp

Below is an example of a plotly figure that is pre-recorded for a shiny app. The idea is that, if the figure does not need to be reactive, you can always pre-compute the results and store them on a file, in this case, as an HTML file:

```
library(shiny)
library(bslib)
library(plotly)

# Like we did with the simulations, we have a default filename
fn <- "plotly.html"

# Notice I'm adding the www because, outside of the
# server call, this writes directly to the top level.
# Once reading, it will read from www.
if (!file.exists(file.path("www", fn))) {

  message("Creating the file...")

  # if it doesn't exist, then it creates it and saves it
  p <- plot_ly(x = 1:10, y = 1:10) %>% add_lines()
  htmlwidgets::saveWidget(
    p,
    file = "www/plotly.html",
    selfcontained = TRUE
    )
} else {
  message("The file already exists!")
}


# Define UI for app that draws a histogram ----
ui <- page_sidebar(
  # App title ----
  title = "Hello Shiny!",
  # Sidebar panel for inputs ----
  sidebar = sidebar(
    # Input: Slider for the number of bins ----
```

```
    sliderInput(
      inputId = "bins",
      label = "Number of bins:",
      min = 1,
      max = 50,
      value = 30
    )
  ),
  htmlOutput(outputId = "plotlyOutput")
)

server <- function(input, output) {

  output$plotlyOutput <- renderUI({
    tags$iframe(
      src = "plotly.html"
    )
  })

}

shinyApp(ui = ui, server = server)
```

2.5 Avoiding unnecessary steps

Many times, we can find shortcuts to reduce the amount of data processing we need to do. A great example is in the linear regression function `lm()`. The `lm()` function will go beyond finding the coefficients in a linear model, it will also compute residuals, fitted values, and more. Instead, we can use the function `lm.fit()` which only computes the coefficients:

```
set.seed(331)
x <- rnorm(2e3)
y <- 2 + 3 * x + rnorm(2e3)

# Comparing
microbenchmark(
  lm = coef(lm(y ~ x)),
  lm_fit = coef(lm.fit(cbind(1, x), y)),
  times = 10,
  unit = "relative"
)
```

```
Unit: relative
   expr      min       lq     mean   median       uq      max neval
     lm 6.514466 6.246501 7.065777 5.948742 5.615682 14.36015    10
 lm_fit 1.000000 1.000000 1.000000 1.000000 1.000000  1.00000    10
```

2.6 Reducing copy operations

Like in any programming language, copy operations in R can be expensive. Beyond increasing the amount of memory used, copy operations require tme to allocate memory and then copy the data. Modern R minimizes these by using copy-on-modify. This means that R will not copy an object until it is modified. For example, the following code makes multiple copies of `X`, but it is until the last line that R actually makes a copy of `X`:

```
set.seed(331)
X <- runif(1e4)
Y <- X
```

```
Z <- X

# Checking the address of the objects
library(lobstr)
obj_addr(X)
## [1] "0x562534a87650"
obj_addr(Y)
## [1] "0x562534a87650"
obj_addr(Z)
## [1] "0x562534a87650"
```

Modifying `X` will trigger a copy operation, and the addresses of `Y` and `Z` will remain the same, while `X` will have a new address:

```
# Modifying X
X[1] <- 100  # This is when R makes a copy of X
obj_addr(X)
## [1] "0x562535670250"
obj_addr(Y)
## [1] "0x562534a87650"
obj_addr(Z)
## [1] "0x562534a87650"
```

II Parallel computing

3. Introduction

While most people see R as a slow programming language, it has powerful features that dramatically accelerate your code[5]. Although R wasn't necessarily built for speed, there are some tools and ways in which we can accelerate R. This chapter introduces what we will understand as High-performance computing in R.

[5]Nonetheless, this claim can be said about almost any programming language; there are notable examples like the R package `data.table` [2] which has been demonstrated to out-perform most data wrangling tools.

4. High-Performance Computing: An overview

From R's perspective, we can think of HPC in terms of two or three things:[6] Big data, parallel computing, and compiled code.

4.1 Big Data

When we talk about big data, we refer to cases where your computer struggles to handle a dataset. A typical example of the latter is when the number of observations (rows) in your data frame is too many to fit a linear regression model. Instead of buying a bigger computer, there are many good solutions to solve memory-related problems:

- **Out-of-memory storage**. The idea is simple, instead of using your RAM to load the data, use other methods to load the data. Two notewirthy alternatives are the bigmemory and implyr R packages. The `bigmemory` package provides methods for using "*file-backed*" matrices. On the other hand, `implyr` implements a wrapper to access Apache Impala, an SQL query engine for a cluster running Apache Hadoop.
- **Efficient algorithms for big data**: To avoid running out of memory with your regression analysis, the R packages biglm and biglasso deliver highly-efficient alternatives to `glm` and `glmnet`, respectively. Now, if your data fits your RAM, but you still struggle with data wrangling, the data.table package is the solution.
- **Store it more efficiently**: Finally, when it comes to linear algebra, the Matrix R package shines with its formal classes and methods for managing Sparse Matrices, *i.e.*, big matrices whose entries are primarily zeros; for example, the `dgCMatrix` objects. Furthermore, `Matrix` comes shipped with R, which makes it even more appealing.

[6] Make sure to check out CRAN Task View on HPC.

4.2 Parallel computing

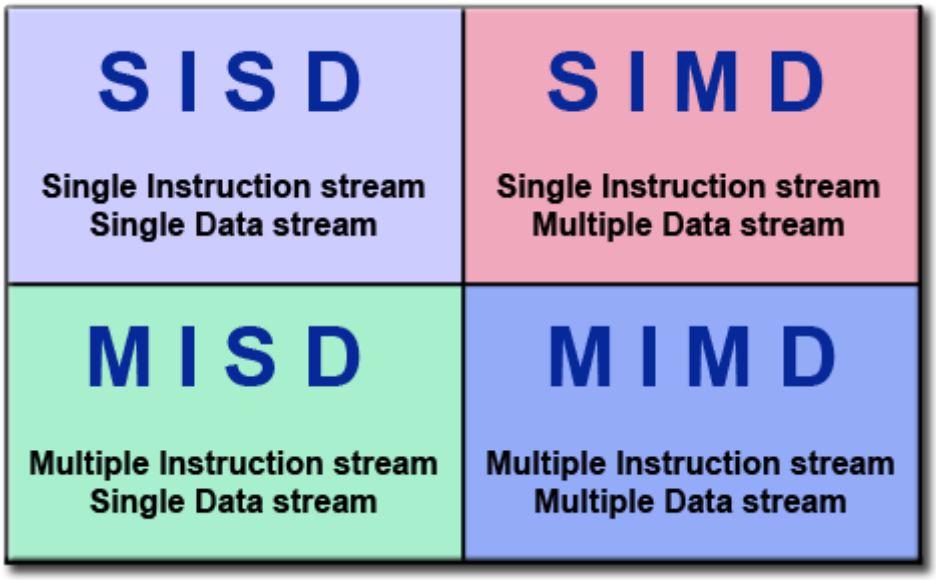


Figure 4.1: Flynn's Classical Taxonomy (Blaise Barney, **Introduction to Parallel Computing**, Lawrence Livermore National Laboratory)

We will focus on the **S**ingle **I**nstruction stream **M**ultiple **D**ata stream.

In general terms, a parallel computing program is one in which we use two or more *computational threads* simultaneously. Although computational thread usually means core, there are multiple levels at which a computer program can be parallelized. To understand this, we first need to see what composes a modern computer:

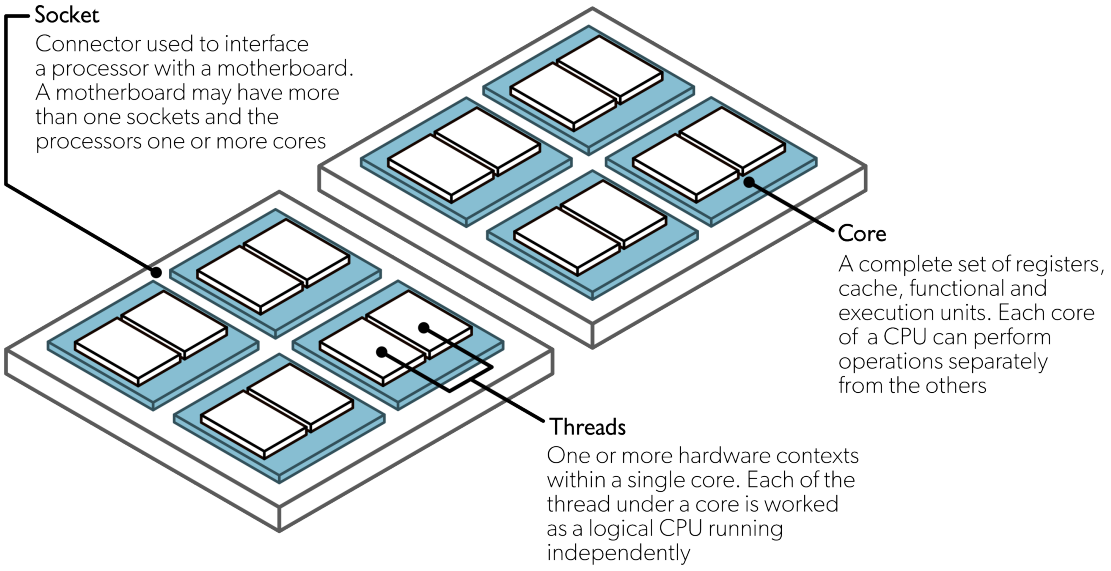
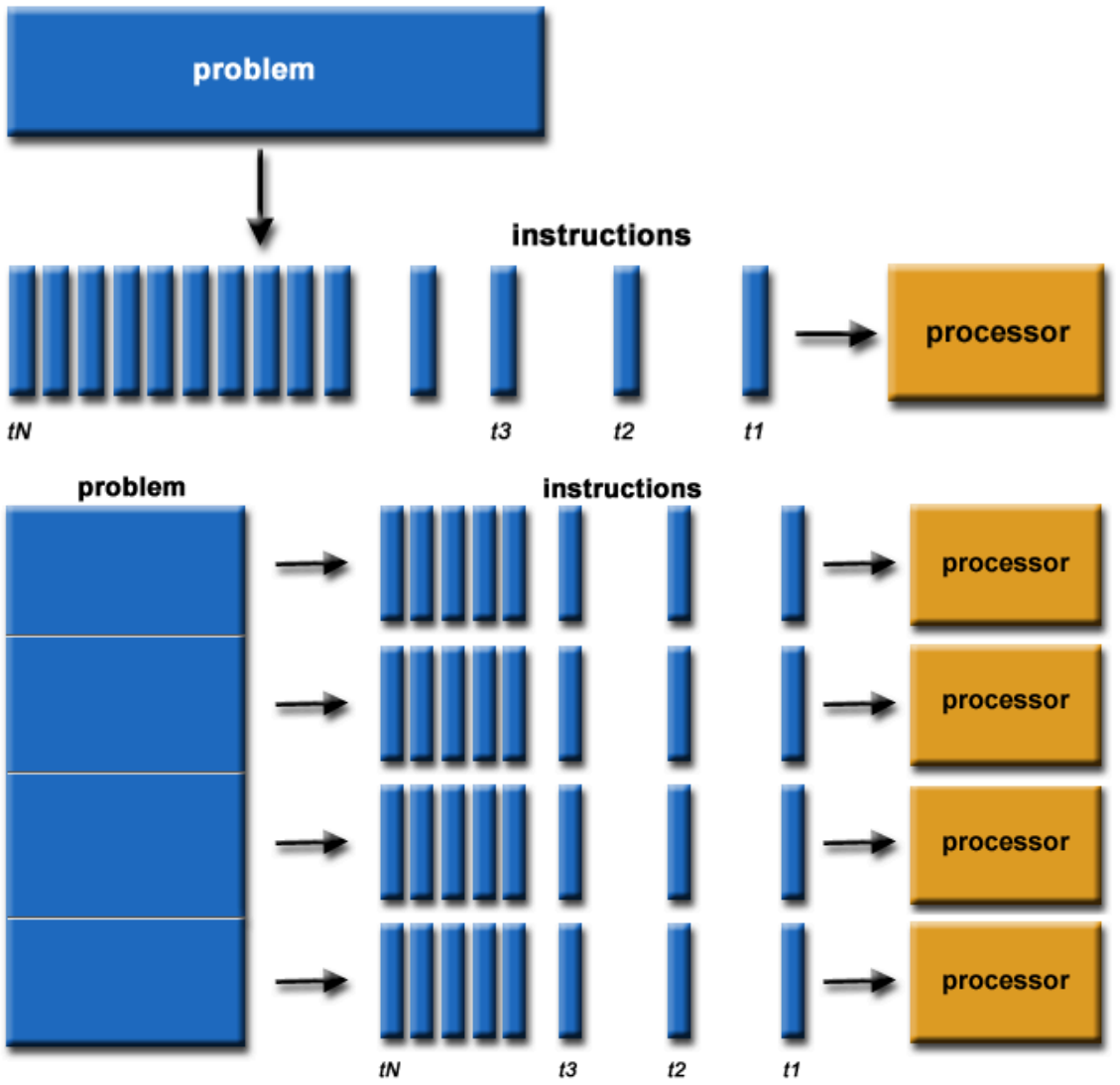


Figure 4.2: Source: Original figure from LUMI consortium documentation [3]

Streaming SIMD Extensions [SSE] and Advanced Vector Extensions [AVX]

4.2.1 Serial vs. Parallel



Source: Blaise Barney, **Introduction to Parallel Computing**, Lawrence Livermore National Laboratory

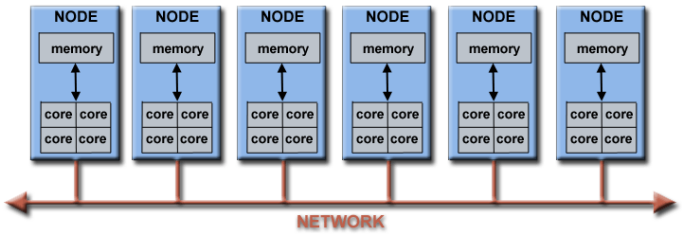


Figure 4.3: source: Blaise Barney, **Introduction to Parallel Computing**, Lawrence Livermore National Laboratory

4.3 High-performance computing in R

4.3.1 Some vocabulary for HPC

In raw terms

- Supercomputer: A **single** big machine with thousands of cores/GPGPUs.

- High-Performance Computing (HPC): **Multiple** machines within a **single** network.
- High Throughput Computing (HTC): **Multiple** machines across **multiple** networks.

You may not have access to a supercomputer, but certainly, HPC/HTC clusters are more accessible these days, *e.g.*, AWS provides a service to create HPC clusters at a low cost (allegedly, since nobody understands how pricing works)

4.4 GPU vs. CPU

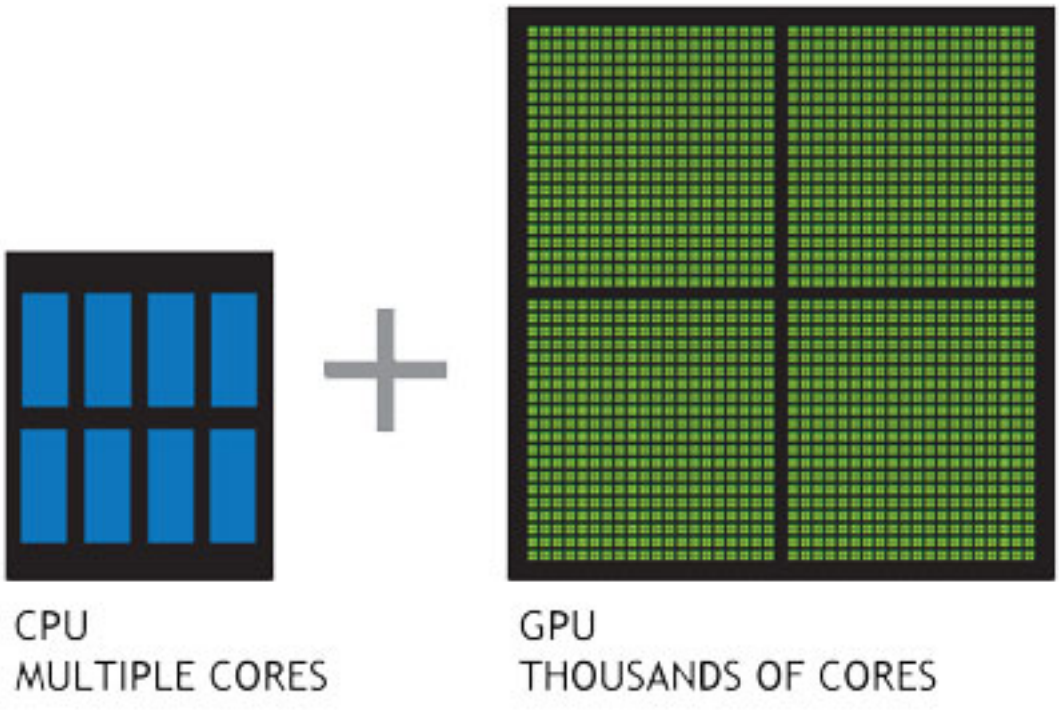


Figure 4.4: NVIDIA Blog

- Why use OpenMP if GPU is *suited to compute-intensive operations*? Well, mostly because OpenMP is **VERY** easy to implement (easier than CUDA, which is the easiest way to use GPU).[7]

[7] Sadia National Laboratories started the Kokkos project, which provides a one-fits-all C++ library for parallel programming. More information on the Kokkos's wiki site.

4.5 When is it a good idea?

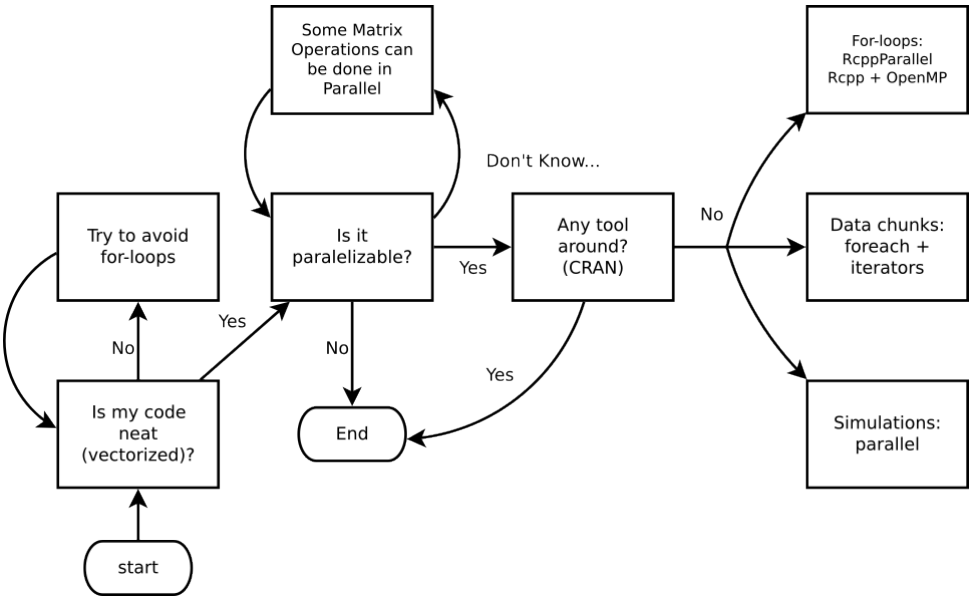


Figure 4.5: Ask yourself these questions before jumping into HPC!

4.6 Parallel computing in R

While there are several alternatives (just take a look at the High-Performance Computing Task View), we'll focus on the following R-packages for **explicit parallelism**:

- **parallel**: R package that provides '[s]upport for parallel computation, including random-number generation'.
- **future**: '[A] lightweight and unified Future API for sequential and parallel processing of R expression via futures.'
- **Rcpp** + OpenMP: Rcpp is an R package for integrating R with C++ and OpenMP is a library for high-level parallelism for C/C++ and FORTRAN.

Others but not used here

- **foreach** for iterating through lists in parallel.
- **Rmpi** for creating MPI clusters.

And tools for implicit parallelism (out-of-the-box tools that allow the programmer not to worry about parallelization):

- **gpuR** for Matrix manipulation using GPU
- **tensorflow** an R interface to TensorFlow.

A ton of other types of resources, notably the tools for working with batch schedulers such as Slurm, and HTCondor.

5. Parallel workflow

Although R was not built for parallel computing, multiple ways of parallelizing your R code exist. One of these is the `parallel` package. This R package, shipped with base R, provides various functions to parallelize R code using embarrassingly parallel computing, i.e., a divide-and-conquer-type strategy. The basic idea is to start multiple R sessions (usually called child processes), connect the main session with those, and send them instructions. This section goes over a common workflow to work with R's parallel.
(Usually) We do the following:

1. Create a `PSOCK/FORK` (or other) cluster using `makePSOCKCluster`/`makeForkCluster` (or `makeCluster`). How many child processes will depend on how many threads your computer has. A rule of thumb is to use `parallel::detectCores() - 1` cores (so you leave one free for the rest of your computer).
2. Copy/prepare each R session (if you are using a `PSOCK` cluster):
   a. Copy objects with `clusterExport`. This would be all the objects that you need in the child sessions.
   b. Pass expressions with `clusterEvalQ`. This would include loading R packages and other code into the other sessions.
   c. Set a seed (if you are doing something that involves randomness)
3. Do your call: `parApply`, `parLapply`, etc.
4. Stop the cluster with `clusterStop`

As we mention later, step 2 will depend on the type of cluster you are using. If you are using a Socket connection (`PSOCK` cluster), then the spawned R sessions will be completely, fresh (no data or R packages pre-loaded); whereas using a Fork connection (`FORK` cluster) will copy the current R session, including all objects and loaded packages.

5.1 Types of clusters: PSOCK

- Can be created with `makePSOCKCluster`
- Creates brand new R Sessions (so nothing is inherited from the master), e.g.

```
# This creates a cluster with 4 R sessions
cl <- makePSOCKCluster(4)
```

- Child sessions are connected to the master session via Socket connections
- Can be created outside the current computer, **i.e.**, across multiple computers!

5.2 Types of clusters: Fork

- Fork Cluster `makeForkCluster`:
- Uses OS Forking,
- Copies the current R session locally (so everything is inherited from the master up to that point).
- Data is only duplicated if altered (need to double check when this happens!)
- Not available on Windows.

Other types are available via the function `makeCluster` from the **snow** R package (Simple Network of Workstations). These include MPI (Message Passing Interface) clusters and Slurm (Socket) clusters.

5.3 A template program

The following code chunk shows a template for using the `parallel` package in R. You can copy this and comment the bits that you don't need:

```
library(parallel)

# 1. CREATING A CLUSTER ----------------
nnodes <- 4L # Could be less or more!
cl <- makePSOCKcluster(nnodes)

# 2. PREPARING THE CLUSTER -------------

# Mostly if using PSOCK
clusterEvalQ(cl, {
  library(...) # Loading the necesary packages
  source(...) # Source additional scripts
})

# Always if you are doing random numbers
clusterSetRNGStream(cl, 123)

# 3. DO YOUR CALL ----------------------
ans <- parLapply(
  cl,
  ... long list to iterate ...,
  function(x) {
    ...
  },
  ... further arguments ...
  )

# 4. STOP THE CLUSTER
stopCluster(cl)
```

Generally, the `... long list to iterate ...` will be a vector or another list that contains either data (e.g., individual datasets), a sequence of numbers (e.g., from 1 to 1000), a list of file paths (if you were processing files individually), or directly a short sequence with numbers from 1 to the number of nodes (least common application).

When calling `parLapply` or `parSapply` (the parallel versions of `lapply` and `sapply` respectively), the function call will automatically split the iterations across nodes using the `splitIndices` function. Here is an example of what happens under the hood:

```
# Distributing 9 iterations across two cores
(n_iterations <- parallel::splitIndices(nx = 9, ncl = 2))
```

```
[[1]]
[1] 1 2 3 4

[[2]]
[1] 5 6 7 8 9
```

Which means that the first R session will get 4 jobs, wereas the second R session will get 5 jobs. This way, each spawned R session (child session) gets to do a similiar number of iterations.

5.4 Example: Replacing a for-loop with `parLapply`

One of the most common use cases for parallel computing is replacing a for-loop with a parallel version. In this example, we will call `runif(10)` 1,000 times, comparing a simple for-loop with `parLapply`.

5.4.1 Serial version using a for-loop

The serial version iterates over the simulations using a for-loop:

```
nsims <- 1000

# First, create seeds for individual runs
set.seed(1231)
seeds <- sample.int(1e7, nsims)

# Serial version using a for-loop
result_serial <- vector("list", nsims)
for (i in seq_len(nsims)) {
  set.seed(seeds[i])
  result_serial[[i]] <- runif(10)
}

# Looking at the first few results
head(result_serial, 2)
```

```
[[1]]
 [1] 0.38201608 0.30000010 0.03890397 0.51085385 0.63979380 0.26806211
 [7] 0.86383563 0.64311030 0.17286217 0.89551119

[[2]]
 [1] 0.23208922 0.65968122 0.55613334 0.75545780 0.25803644 0.89517280
 [7] 0.90952159 0.69711625 0.02608031 0.60768982
```

One thing to note here is that we are pre-generating a vector of seeds and setting the seed inside each simulation. While this is not a common practice for running things serially, as you will see, it is a common strategy for parallel computing that allows you to get the same results regardless of the number of cores/threads used.

5.4.2 Parallel version using `parLapply`

We can replace the for-loop with `parLapply` following the template we described earlier. Note the seeding strategy: instead of using `clusterSetRNGStream`, we are pre-generating a vector of seeds and setting the seed inside each simulation. This makes the results 100% reproducible regardless of the number of cores/threads used.

```
library(parallel)

# 1. CREATING A CLUSTER ---------------
cl <- makePSOCKcluster(4L)

# 2. PREPARING THE CLUSTER -------------
# We export the seeds vector to all workers
clusterExport(cl, "seeds")

# 3. DO YOUR CALL ----------------------
result_parallel <- parLapply(cl, seq_len(nsims), \(i) {
  # This makes it 100% reproducible, regardless of the
  # number of cores/threads we are using
  set.seed(seeds[i])
  runif(10)
})

# 4. STOP THE CLUSTER
stopCluster(cl)
```

```
# Results are identical
all.equal(result_serial, result_parallel)
```

```
[1] TRUE
```

> **i** Per-simulation seeding vs `clusterSetRNGStream`
>
> The approach used here—generating seeds beforehand and calling `set.seed()` inside each simulation—is an alternative to `clusterSetRNGStream`. The key differences are:
>
> - **`clusterSetRNGStream`** initializes the L'Ecuyer-CMRG random number generator across the cluster. The streams depend on the number of workers, so results may change if you change the number of cores.
> - **Per-simulation seeding** (as shown here) assigns a unique seed to each iteration. Because the seed is tied to the simulation index rather than the worker, the results are identical no matter how many cores are used.
>
> Both approaches are valid. Per-simulation seeding is especially useful when you want results that are invariant to the number of threads, for example, when debugging or comparing runs across different machines. Note that combining the two strategies is generally redundant: once you call `set.seed()` inside each iteration, the worker's RNG stream initialized by `clusterSetRNGStream` no longer affects the draws in that iteration. Using `clusterSetRNGStream` alongside per-simulation seeding would only matter if you also rely on RNG outside the per-iteration `set.seed()` call, for example, in setup code running on the workers.

5.5 Example: Running a linear regression across multiple columns

In genomics, it is common to analyze genomic data at the gene level comparing expression levels against some phenotype/disease. A simple analysis consists of running a linear regression across multiple columns (genes) of a data frame. The following code-block generates some artificial data we can use for this example:

```
set.seed(331)
n_genes <- 10000
n_obs <- 1000

# A random matrix of omics
X_genes <- rnorm(n_obs * n_genes) |>
  matrix(nrow = n_obs)

# A random phenotype (completely unrelated for this example)
Y <- rnorm(n_obs) |> cbind()
```

We will wrap the analysis into a function so we can do benchmarking. We will use the `lapply` function to iterate over the columns of `X_genes`

```
ols_serial <- function(X, Y) {
  lapply(
    X = seq_len(n_genes),
    FUN = \(i) {lm.fit(X[, i, drop = FALSE], Y) |> coef()}
  ) |> do.call(what = rbind)
}

# Calling the function and looking at the first few rows
ols_serial(X_genes, Y) |> head()
```

```
              x1
[1,]  0.029403088
```

```
[2,]  0.008907854
[3,] -0.027246099
[4,] -0.031280262
[5,] -0.001309752
[6,]  0.066971469
```

> Tip
>
> Like we did in the efficient programming section, instead of using `lm()` or `glm()`, we can use `lm.fit()` for better performance. The `lm.fit()` function does less than the `lm()` function by skipping computing residuals and other overhead, making it faster for large datasets.

Using parallel computing (and following the template we presented earlier), this could be done in the following way with the parallel package:

```
library(parallel)

ols_parallel <- function(X, Y, ncores) {
  # 1. CREATING A CLUSTER ----------------
  cl <- makePSOCKcluster(ncores)

  # This will be called when exiting the function
  on.exit(stopCluster(cl))

  # 2. PREPARING THE CLUSTER -------------
  # We copy the data over
  clusterExport(cl, c("X", "Y"), envir = environment())

  # 3. DO YOUR CALL ----------------------
  parLapply(
    cl,
    seq_len(n_genes),
    function(i) {
      lm.fit(X[, i, drop = FALSE], Y) |> coef()
    }
  ) |> do.call(what = rbind)
}

# Checking it works
ols_parallel(X_genes, Y, ncores = 4L) |> head()
```

```
              x1
[1,]  0.029403088
[2,]  0.008907854
[3,] -0.027246099
[4,] -0.031280262
[5,] -0.001309752
[6,]  0.066971469
```

> Tip
>
> Just like the `return()`, `on.exit()` can only be used within a function call. We could have used `stopCluster(cl)` at the end as we do in our template example, but the benefit of using `on.exit()` is that it will be called automatically when the function exits, even if an error occurs. This helps to ensure that the cluster is always stopped properly.

Now that we have the function implemented, we can go ahead and (1) compare results and (2) measure performance.

```
library(microbenchmark)

microbenchmark(
  serial = ols_serial(X_genes, Y),
  parallel = ols_parallel(X_genes, Y, ncores = 4L),
  times = 10L,
  check = "identical"
)
```

```
Unit: milliseconds
     expr       min        lq      mean    median        uq       max neval
   serial  581.6953  615.4718  645.6976  644.6076  694.5022  709.0955    10
 parallel 1841.2412 1851.4243 1871.6257 1874.8970 1883.3980 1906.7394    10
```

From the comparison, we can see that the parallel version is significantly slower than the serial version. Two things to note here are (a) the task we are running is already fast (about 0.3 seconds on average for the serial run) and (b) there is an overhead cost associated with creating, preparing, and stopping the cluster. As we mentioned earlier, parallel optimizations only make sense if your code is already talking a significant amount of time, making the overhead cost associated with the setup relatively small. The following implementation of the function should make it significantly faster:

```
ols_parallel2 <- function(cl) {
  # 1. CREATING A CLUSTER ----------------
  # 2. PREPARING THE CLUSTER -------------
  # No need anymore as we are handling the core outside

  # 3. DO YOUR CALL ----------------------
  parLapply(
    cl,
    seq_len(n_genes),
    function(i) {
      lm.fit(X_genes[, i, drop = FALSE], Y) |> coef()
    }
  ) |> do.call(what = rbind)
}

# Checking it works
cl <- makePSOCKcluster(4)
clusterExport(cl, c("X_genes", "Y"))
ols_parallel2(cl) |> head()
```

```
               x1
[1,]  0.029403088
[2,]  0.008907854
[3,] -0.027246099
[4,] -0.031280262
[5,] -0.001309752
[6,]  0.066971469
```

```
# 4. STOP THE CLUSTER
stopCluster(cl)
```

The main differences from the previous version of the function are:

1. We are creating the cluster outside of the function and passing it as an argument.
2. We are exporting the `X_genes` and `Y` variables to the cluster only once, which should also reduce overhead significantly.
3. Because of the previous step, we are now calling `X_genes` directly in the main function.

4. The cluster is stopped outside of the function call (since the function no longer manages the cluster object).

Let's measure the performance to see how much faster the parallel version is.

```
library(microbenchmark)

# We need to prepare the cluster before hand
cl <- makePSOCKcluster(4)
clusterExport(cl, c("X_genes", "Y"))

microbenchmark(
  serial = ols_serial(X_genes, Y),
  parallel = ols_parallel2(cl),
  times = 10L,
  check = "identical"
)
```

```
Unit: milliseconds
     expr      min       lq     mean   median       uq      max neval
   serial 579.3778 619.3939 645.1135 645.5034 667.2292 715.6528    10
 parallel 283.8007 309.1091 315.6120 315.8411 327.5199 345.7187    10
```

```
# We need to stop the cluster
stopCluster(cl)
```

Now, the parallel version is significantly faster than the serial version. Just using the parallel package (or any other package that can be used for parallel computing) does not guarantee improved performance.

5.6 More examples

The following three examples are a simple application of the package in which we are explicitly running as many replications as threads the cluster has. Generally, the number of replicates will be a function of the data.

5.6.1 Ex 1: Parallel RNG with `makePSOCKCluster`

> **Caution**
>
> Using more threads than cores available on your computer is never a good idea. As a rule of thumb, clusters should be created using `parallel::detectCores() - 1` cores (so you leave one free for the rest of your computer.)

```
# 1. CREATING A CLUSTER
library(parallel)
nnodes <- 4L
cl     <- makePSOCKcluster(nnodes)
# 2. PREPARING THE CLUSTER
clusterSetRNGStream(cl, 123) # Equivalent to `set.seed(123)`
# 3. DO YOUR CALL
ans <- parSapply(cl, 1:nnodes, function(x) runif(1e3))
(ans0 <- var(ans))
```

```
              [,1]          [,2]          [,3]          [,4]
[1,]  0.0861888293 -0.0001633431  5.939143e-04 -3.672845e-04
[2,] -0.0001633431  0.0853841838  2.390790e-03 -1.462154e-04
[3,]  0.0005939143  0.0023907904  8.114219e-02 -4.714618e-06
[4,] -0.0003672845 -0.0001462154 -4.714618e-06  8.467722e-02
```

Making sure it is reproducible

```
# I want to get the same!
clusterSetRNGStream(cl, 123)
ans1 <- var(parSapply(cl, 1:nnodes, function(x) runif(1e3)))
# 4. STOP THE CLUSTER
stopCluster(cl)
all.equal(ans0, ans1) # All equal!
```

```
[1] TRUE
```

5.6.2 Ex 2: Parallel RNG with `makeForkCluster`

In the case of `makeForkCluster`

```
# 1. CREATING A CLUSTER
library(parallel)
# The fork cluster will copy the -nsims- object
nsims  <- 1e3
nnodes <- 4L
cl     <- makeForkCluster(nnodes)
# 2. PREPARING THE CLUSTER
clusterSetRNGStream(cl, 123)
# 3. DO YOUR CALL
ans <- do.call(cbind, parLapply(cl, 1:nnodes, function(x) {
  runif(nsims) # Look! we use the nsims object!
               # This would have fail in makePSOCKCluster
               # if we didn't copy -nsims- first.
  }))
(ans0 <- var(ans))
```

```
              [,1]          [,2]          [,3]          [,4]
[1,]  0.0861888293 -0.0001633431  5.939143e-04 -3.672845e-04
[2,] -0.0001633431  0.0853841838  2.390790e-03 -1.462154e-04
[3,]  0.0005939143  0.0023907904  8.114219e-02 -4.714618e-06
[4,] -0.0003672845 -0.0001462154 -4.714618e-06  8.467722e-02
```

Again, we want to make sure this is reproducible

```
# Same sequence with same seed
clusterSetRNGStream(cl, 123)
ans1 <- var(do.call(cbind, parLapply(cl, 1:nnodes, function(x) runif(nsims))))
ans0 - ans1 # A matrix of zeros
```

```
     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0
[4,]    0    0    0    0
```

```
# 4. STOP THE CLUSTER
stopCluster(cl)
```

Well, if you are a Mac-OS/Linux user, there's a more straightforward way of doing this...

5.6.3 Ex 3: Parallel RNG with `mclapply` (Forking on the fly)

In the case of `mclapply`, the forking (cluster creation) is done on the fly!

```
# 1. CREATING A CLUSTER
library(parallel)
# The fork cluster will copy the -nsims- object
nsims  <- 1e3
nnodes <- 4L
# cl     <- makeForkCluster(nnodes) # mclapply does it on the fly
# 2. PREPARING THE CLUSTER
set.seed(123)
# 3. DO YOUR CALL
ans <- do.call(cbind, mclapply(1:nnodes, function(x) runif(nsims)))
(ans0 <- var(ans))
```

```
              [,1]          [,2]          [,3]         [,4]
[1,]  0.0827634110 -0.0008389458  0.0002376533 -0.003362618
[2,] -0.0008389458  0.0830645850  0.0003492542  0.005660360
[3,]  0.0002376533  0.0003492542  0.0807240638 -0.001233359
[4,] -0.0033626176  0.0056603604 -0.0012333590  0.081660443
```

Once more, we want to make sure this is reproducible

```
# Same sequence with same seed
set.seed(123)
ans1 <- var(do.call(cbind, mclapply(1:nnodes, function(x) runif(nsims))))
ans0 - ans1 # A matrix of zeros
```

```
             [,1]         [,2]          [,3]          [,4]
[1,] -0.002165207 -0.004371033  0.0048714820 -0.0033968054
[2,] -0.004371033 -0.004178137 -0.0011647031  0.0056419520
[3,]  0.004871482 -0.001164703  0.0001829237 -0.0003086966
[4,] -0.003396805  0.005641952 -0.0003086966  0.0002262708
```

```
# 4. STOP THE CLUSTER
# stopCluster(cl) no need of doing this anymore
```

5.7 Exercise: Overhead costs

Compare the timing of taking the sum of 100 numbers when parallelized versus not. For the unparallized (serialized) version, use the following:

```
set.seed(123)
x <- runif(n=100)

serial_sum <- function(x){
  x_sum <- sum(x)
  return(x_sum)
}
```

For the parallized version, follow this outline

```
library(parallel)
```

```
set.seed(123)
x <- runif(n=100)

parallel_sum <- function(){


  # Set number of cores to use
  # make cluster and export to the cluster the x variable
```

```
  # Use "split function to divide x up into as many chunks as the number of cores

  # Calculate partial sums doing something like:

  partial_sums <- parallel::parSapply(cl, x_split, sum)

  # Stop the cluster

  # Add and return the partial sums

}
```

Compare the timing of the two approaches:

```
microbenchmark::microbenchmark(
  serial   = serial_sum(x),
  parallel = parallel_sum(x),
  times    = 10,
  unit     = "relative"
)
```

III Working with a Cluster

6. What is Slurm

> **Note**
>
> Most of this section was extracted from the `slurmR` R package's vignette "Working with Slurm."

Nowadays, high-performance-computing (HPC) clusters are commonly available tools for either in or out of cloud settings. Slurm Work Manager (formerly *Simple Linux Utility for Resource Manager*) is a program written in C that is used to efficiently manage resources in HPC clusters. The slurmR R package–which we will be using in this book–provides tools for using R in HPC settings that work with Slurm. It provides wrappers and functions that allow the user to seamlessly integrate their analysis pipeline with HPC clusters, emphasizing on providing the user with a family of functions similar to those that the parallel R package provides.

6.1 Definitions

First, some important discussion points within the context of Slurm+R that users, in general, will find useful. Most of the points have to do with options available for Slurm, and in particular, with the `sbatch` command with is used to submit batch jobs to Slurm. Users who have used Slurm in the past may wish to skip this and continue reading the following section.

- **Node** A single computer in the HPC: A lot of times jobs will be submitted to a single node. The simplest way of using R+Slurm is submitting a single job and requesting multiple CPUs to use, for example, `parallel::parLapply` or `parallel::mclapply`. Usually, users do not need to request a specific number of nodes to be used as Slurm will allocate the resources as needed.

  A common mistake of R users is to specify the number of nodes and expect that their script will be parallelized. This won't happen unless the user explicitly writes a parallel computing script.

  The relevant flag for `sbatch` is `--nodes`.

- **Partition** A group of nodes in HPC. Generally large nodes may have multiple partitions, meaning that nodes may be grouped in various ways. For example, nodes belonging to a single group of users may be in a single partition, and nodes dedicated to working with large data may be in another partition. Usually, partitions are associated with account privileges, so users may need to specify which account are they using when telling Slurm what partition they plan to use.

  The relevant flag for `sbatch` is `--partition`.

- **Account** Accounts may be associated with partitions. Accounts can have privileges to use a partition or set of nodes. Often, users need to specify the account when submitting jobs to a particular partition.

  The relevant flag for `sbatch` is `--account`.

- **Task** A step within a job. A particular job can have multiple tasks. tasks may span multiple nodes, so if the user wants to submit a multicore job, this option may not be the right one.

  The relevant flag for `sbatch` is `--ntasks`

- **CPU** generally this refers to core or thread (which may be different in systems supporting multithreaded cores). Users may want to specify how many CPUs they want to use for a task. And this is the relevant option when using things like OpenMP or functions that allow creating cluster objects in R (e.g. `makePSOCKcluster`, `makeForkCluster`).

  The relevant option in `sbatch` is `--cpus-per-task`. More information regarding CPUs in Slurm can be found here. Information regarding how Slurm counts CPUs/cores/threads can be found here.

- **Job Array** Slurm supports job arrays. A job array is in simple terms a job that is repeated multiple times by Slurm, this is, replicates a single job as requested per the user. In the case of R, when using this

option, a single R script is spanned in multiple jobs, so the user can take advantage of this and parallelize jobs across multiple nodes. Besides from the fact that jobs within a Job Array may be spanned across multiple nodes, each job in that array has a unique ID that is available to the user via environment variables, in particular `SLURM_ARRAY_TASK_ID`.

Within R, and hence the Rscript submitted to Slurm, users can access this environment variable with `Sys.getenv("SLURM_ARRAY_TASK_ID")`. Some of the functionalities of `slurmR` rely on Job Arrays.

More information on Job Arrays can be found here. The relevant option for this in `sbatch` is `--array`.

More information about Slurm can be found their official website here. A tutorial about how to use Slurm with R can be found here.

7. A brief intro to Slurm

For a quick-n-dirty intro to Slurm [4], we will start with a simple "Hello world" using Slurm + R. For this, we need to go through the following steps:

1. Copy a Slurm script to HPC,
2. Logging to HPC, and
3. Submit the job using `sbatch`.

7.1 Step 1: Copy the Slurm script to HPC

We need to copy the following Slurm script to HPC (00-hello-world.slurm):

```
#!/bin/sh
#SBATCH --output=00-hello-world.out
module load R/4.2.2
Rscript -e "paste('Hello from node', Sys.getenv('SLURMD_NODENAME'))"
```

Which has four lines:

1. `#!/bin/sh`: The **shebang** (**shewhat?**)
2. `#SBATCH --output=00-hello-world.out`: An option to be passed to `sbatch`, in this case, the name of the output file to which **stdout and stderr** will go.
3. `module load R/4.2.2`: Uses **Lmod** to load the `R` module.
4. `Rscript ...`: A call to R to evaluate the expression `paste(...)`. This will get the environment variable `SLURMD_NODENAME` (which `sbatch` creates) and print it on a message.

To do so, we will use **Secure copy protocol (scp)**, which allows us to copy data to and fro computers. In this case, we should do something like the following

```
scp  00-hello-world.slurm  [userid]@notchpeak.chpc.utah.edu:/path/to/a/place/you/can/
access
```

In words, "Using the username `[userid]`, connect to `notchpeak.chpc.utah.edu`, take the file `00-hello-world.slurm` and copy it to `/path/to/a/place/you/can/access`. With the file now available in the cluster, we can submit this job using Slurm.

7.2 Step 2: Logging to HPC

1. Log in using ssh. In the case of Windows users, download the **Putty** client.
2. To log in, you will need to use your organization ID. Usually, if your email is something like `myemailuser@school.edu`, your ID is `myemailuser`. Then:

   ```
   ssh myemailuser@notchpeak.chpc.utah.edu
   ```

7.3 Step 3: Submitting the job

Overall, there are two ways to use the compute nodes: interactively (`salloc`) and in batch mode (`sbatch`). In this case, since we have a Slurm script, we will use the latter.

To submit the job, we can type the following:

```
sbatch 00-hello-world.slurm
```

And that's it! That said, it is often required to specify the account and partition the user will be submitting the job. For example, if you have the account `my-account` and partition `my-partition` associated with your user, you can incorporate that information as follows:

```
sbatch 00-hello-world.slurm --acount=my-account --partition=my-partition
```

In the case of interactive sessions, You can start one using the `salloc` command. For example, if you wanted to run R with 8 cores, using 16 Gigs of memory in total, you would need to do the following:

```
salloc -n1 --cpus-per-task=8 --mem-per-cpu=2G --time=01:00:00
```

Once your request is submitted, you will get access to a compute node. Within it, you can load the required modules and start R:

```
module load R/4.2.2
R
```

Interactive sessions are not recommended for long jobs. Instead, use this resource if you need to inspect some large dataset, debug your code, etc.

8. Simulating pi

The following is an example many people (including me) have used to illustrate parallel computing with R. The example is straightforward: we want to approximate pi by doing some Monte Carlo simulations.

We know that the area of a circle is $A = \pi r^2$, which is equivalent to $\pi = A/r^2$, so if we can approximate the Area of a circle, then we can approximate π. How do we do this?

Using Monte Carlo experiments, we can approximate the probability of a random point x falling within the unit circle using the following formula:

$$\hat{p} = \frac{1}{n} \sum_i \mathbf{1}(x \in \text{Circle}) \tag{8.1}$$

This approximation, $\hat{p}$, multiplied by the area of the square containing the circle, which has an area equal to $(2 \times r)^2$, thus, we can finally write

$$\hat{\pi} = \hat{p} \times (2 \times r)^2 / r^2 = 4\hat{p} \tag{8.2}$$

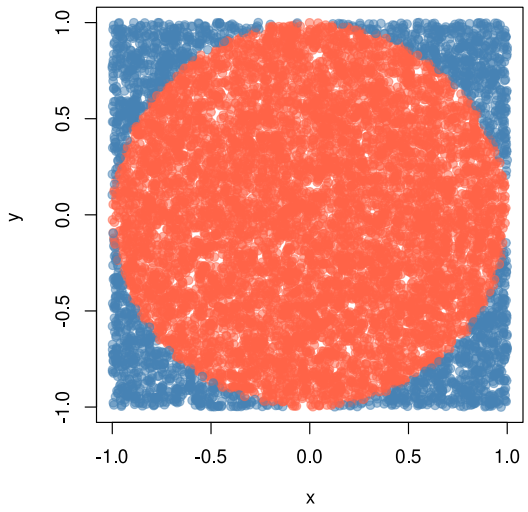


Figure 8.1: 10,000 random points drawn within the unit circle.

8.1 Submitting jobs to Slurm

We will primarily work by submitting jobs using the `sbatch` function. This function takes as its main argument a bash file with the program to execute. In the case of R, a regular bash file looks something like this:

```
#!/bin/sh
#SBATCH --job-name=sapply
#SBATCH --time=00:10:00

module load R/4.2.2
Rscript --vanilla 01-sapply.R
```

This file has three components:

- **The Slurm flags `#SBATCH`**: These comment-like entries pass Slurm options to the job. In this example, we only specify the options `job-name` and `time`. Other common options would include `account` and `partition`.
- **Loading R `module load R/4.2.2`**: Depending on your system's configuration, you may or may not need to load modules or run bash scripts before being able to run R. In this example, we are loading R version 4.2.2 using `LMod` (see previous section).
- **Executing the R script**: After specifying Slurm options and loading whatever needs to be loaded before executing R, we are using `RScript` to execute the program we wrote.

Submission is then made as follows:

```
sbatch 01-sapply.slurm
```

The following examples have two files, a bash script and an R script, to be called by Slurm.

8.1.1 Case 1: Single job, single core job

The most basic way is submitting a job using the `sbatch` command. In this case, you must have two files: (1) An R script and (2) a bash script. e.g.

The contents of the R script (01-sapply.R) are:

```
# Model parameters
nsims <- 1e3
n     <- 1e4

# Function to simulate pi
simpi <- function(i) {

  p <- matrix(runif(n*2, -1, 1), ncol = 2)
  mean(sqrt(rowSums(p^2)) <= 1) * 4

}

# Approximation
set.seed(12322)
ans <- sapply(1:nsims, simpi)

message("Pi: ", mean(ans))

saveRDS(ans, "01-sapply.rds")
```

The contents of the bashfile (01-sapply.slurm) are:

```
#!/bin/sh
#SBATCH --job-name=sapply
#SBATCH --time=00:10:00

module load R/4.2.2
Rscript --vanilla 01-sapply.R
```

8.1.2 Case 2: Single job, multicore job

Imagine that we would like to use more than one processor for this job, using the `parallel::mclapply` function from the parallel package.[8] Then, besides adapting the code, we need to tell Slurm that we are using more than one core per task, as in the following example:

R script (02-mclapply.R):

[8]This function is sort of a wrapper of `makeForkcluster`. Forking provides a way to duplicate a process in the OS without replicating the memory, which is both faster and efficient.

```
# Model parameters
nsims  <- 1e3
n      <- 1e4
ncores <- 4L

# Function to simulate pi
simpi <- function(i) {

  p <- matrix(runif(n*2, -1, 1), ncol = 2)
  mean(sqrt(rowSums(p^2)) <= 1) * 4

}

# Approximation
set.seed(12322)
ans <- parallel::mclapply(1:nsims, simpi, mc.cores = ncores)
ans <- unlist(ans)

message("Pi: ", mean(ans))

saveRDS(ans, "02-mclpply.rds")
```

Bashfile (02-mclapply.slurm):

```
#!/bin/sh
#SBATCH --job-name=mclapply
#SBATCH --time=00:10:00
#SBATCH --cpus-per-task=4

module load R/4.2.2
Rscript --vanilla 02-mclapply.R
```

8.2 Jobs with the slurmR package

The `slurmR` R package [5], [6] is a lightweight wrapper of Slurm. The package's primary functions are the `*apply` family–mainly through Slurm job arrays–and the `makeSlurmCluster()`–which is a wrapper of `makePSOCKcluster`.

This section will illustrate how to submit jobs using the `makeSlurmCluster()` function and `Slurm_sapply`. Furthermore, the last example demonstrates how we can skip writing Slurm scripts entirely using the `sourceSlurm()` function included in the package.

8.2.1 Case 3: Single job, multinode job

In this case, there is no simple way to submit a multinodal job to Slurm; unless you use the **slurmR** package.[9] In this example, we will combine `slurmR` with the parallel package's `parSapply` function to submit a multinodal job using the function `makeSlurmCluster()`. With it, `slurmR` will submit a job requesting `njobs` tasks (processors) that could span multiple nodes,[10] and create a Socket cluster out of it (like using `makePSOCKcluster`.) One thing to keep in mind is that Socket clusters are limited in the number of connections a single R session can span. You can read more about it here and here.

R script (03-parsapply-slurmr.R):

```
# Model parameters
nsims  <- 1e3
n      <- 1e4
ncores <- 4L
```

[9] See installation instructions **here**

[10] Although possible, most multinode jobs will be allocated if insufficient threads are within a single node. Remember Slurm does not run the jobs but rather reserves computational resources for you to run it.

```
# Function to simulate pi
simpi <- function(i) {

  p <- matrix(runif(n*2, -1, 1), ncol = 2)
  mean(sqrt(rowSums(p^2)) <= 1) * 4

}

# Setting up slurmR
library(slurmR) # This also loads the parallel package

# Making the cluster, and exporting the variables
cl <- makeSlurmCluster(ncores)

# Approximation
clusterExport(cl, c("n", "simpi"))
ans <- parSapply(cl, 1:nsims, simpi)

# Closing connection
stopCluster(cl)

message("Pi: ", mean(ans))

saveRDS(ans, "03-parsapply-slurmr.rds")
```

Bashfile (03-parsapply-slurmr.slurm):

```
#!/bin/sh
#SBATCH --job-name=parsapply
#SBATCH --time=00:10:00

module load R/4.2.2
Rscript --vanilla 03-parsapply-slurmr.R
```

8.2.2 Case 4: Multi job, single/multi-core

Another way to submit jobs is using **job arrays**. A job array is a job repeated `njobs` times with the same configuration. The main difference between replicates is what you do with the `SLURM_ARRAY_TASK_ID` environment variable. This variable is defined within each replicate and can be used to make the "subjob" depending on that.

Here is a quick example using R

```
ID <- Sys.getenv("SLURM_ARRAY_TASK_ID")
if (ID == 1) {
  ...[do this]...
} else if (ID == 2) {
  ...[do that]...
}
```

The `slurmR` R package makes submitting job arrays easy. Again, with the simulation of pi, we can do it in the following way:

R script (04-slurm_sapply.R):

```
# Model parameters
nsims  <- 1e3
n      <- 1e4
# ncores <- 4L
njobs  <- 4L

# Function to simulate pi
simpi <- function(i, n.) {
```

```
  p <- matrix(runif(n.*2, -1, 1), ncol = 2)
  mean(sqrt(rowSums(p^2)) <= 1) * 4

}

# Setting up slurmR
library(slurmR) # This also loads the parallel package

# Approximation
ans <- Slurm_sapply(
  1:nsims, simpi,
  n.       = n,
  njobs    = njobs,
  plan     = "collect",
  tmp_path = "/scratch/vegayon" # This is where all temp files will be exported
  )

message("Pi: ", mean(ans))

saveRDS(ans, "04-slurm_sapply.rds")
```

Bashfile (04-slurm_sapply.slurm):

```
#!/bin/sh
#SBATCH --job-name=slurm_sapply
#SBATCH --time=00:10:00

module load R/4.2.2
Rscript --vanilla 04-slurm_sapply.R
```

One of the main benefits of using this approach instead of the `makeSlurmCluster` function (and thus, working with a SOCK cluster) are:

- The number of jobs is not limited here (only by the admin, but not by R).
- If a job fails, then we can re-run it using `sbatch` once again (see example here).
- You can check the individual logs of each process using the function `Slurm_lob()`.
- You can submit the job and quit the R session without waiting for it to finalize. You can always read back the job using the function `read_slurm_job([path-to-the-temp])`

8.2.3 Case 5: Skipping the .slurm file

The `slurmR` package has a function named `sourceSlurm` that can be used to avoid creating the `.slurm` file. The user can add the SBATCH options to the top of the R script (including the `#!/bin/sh` line) and submit the job from within R as follows:

R script (05-sapply.R):

```
#!/bin/sh
#SBATCH --job-name=sapply-sourceSlurm
#SBATCH --time=00:10:00

# Model parameters
nsims <- 1e3
n     <- 1e4

# Function to simulate pi
simpi <- function(i) {

  p <- matrix(runif(n*2, -1, 1), ncol = 2)
  mean(sqrt(rowSums(p^2)) <= 1) * 4
```

```
}

# Approximation
set.seed(12322)
ans <- sapply(1:nsims, simpi)

message("Pi: ", mean(ans))

saveRDS(ans, "05-sapply.rds")
```

From the R console (is OK if you are in the Head node)

```
slurmR::sourceSlurm("05-sapply.R")
```

And voilá! A temporary bash file will be generated to submit the R script to the queue. The following video shows a possible output on the University of Utah's CHPC with slurmR version 0.5-3:
https://youtu.be/OasEla5EszI

IV Using C++

9. Rcpp

When parallel computing is not enough, you can boost your R code using a lower-level programming language[11] like C++, C, or Fortran. With R itself written in C, it provides access points (APIs) to connect C++/C/Fortran functions to R. Although not impossible, using lower-level languages to enhance R can be cumbersome; Rcpp [7], [8], [9] can make things **very** easy. This chapter shows you how to use Rcpp–the most popular way to connect C++ with R–to accelerate your R code.

9.1 Before we start

1. You need to have Rcpp installed in your system:

```
install.packages("Rcpp")
```

2. You need to have a compiler
   - Windows: You can download Rtools from here.
   - MacOS: It is a bit complicated... Here are some options:
     - CRAN's manual to get the clang, clang++, and gfortran compilers here.
     - A great guide by the coatless professor here

And that's it!

9.2 R is great, but...

- The problem:
  - As we saw, R is very fast... once vectorized
  - What to do if your model cannot be vectorized?
- The solution: **Use C/C++/Fotran! It works with R!**
- The problem to the solution: **What R user knows any of those!?**
- R has had an API (application programming interface) for integrating C/C++ code with R for a long time.
- Unfortunately, it is not very straightforward

9.3 Enter Rcpp

- One of the **most important R packages on CRAN**.
- As of January 22, 2023, about 50% of CRAN packages depend on it (directly or not).
- From the package description:

> The 'Rcpp' package provides R functions as well as C++ classes which offer a seamless integration of R and C++

[11] In general, a low-level programming language is "*a programming language that provides little or no abstraction from a computer's set architecture [...]*" (wiki), yet, here we use that term to refer to programming languages that are closer to machine code than what R is.

9.4 Why bother?

- To draw ten numbers from a normal distribution with sd = 100.0 using R C API:

```
SEXP stats = PROTECT(R_FindNamespace(mkString("stats")));
SEXP rnorm = PROTECT(findVarInFrame(stats, install("rnorm")));
SEXP call = PROTECT(
  LCONS( rnorm, CONS(ScalarInteger(10), CONS(ScalarReal(100.0),
  R_NilValue))));
SET_TAG(CDDR(call),install("sd"));
SEXP res = PROTECT(eval(call, R_GlobalEnv));
UNPROTECT(4);
return res;
```

- Using Rcpp:

```
Environment stats("package:stats");
Function rnorm = stats["rnorm"];
return rnorm(10, Named("sd", 100.0));
```

9.5 Example 1: Looping over a vector

```
#include<Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
NumericVector add1(NumericVector x) {
  NumericVector ans(x.size());
  for (int i = 0; i < x.size(); ++i)
    ans[i] = x[i] + 1;
  return ans;
}
```

```
add1(1:10)
```

```
 [1]  2  3  4  5  6  7  8  9 10 11
```

Make it sweeter by adding some “sugar” (the Rcpp kind)

```
#include<Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
NumericVector add1Cpp(NumericVector x) {
  return x + 1;
}
```

```
add1Cpp(1:10)
```

```
 [1]  2  3  4  5  6  7  8  9 10 11
```

9.6 How much fast?

Compared to this:

```
add1R <- function(x) {
  for (i in 1:length(x))
    x[i] <- x[i] + 1
```

```
  x
}
microbenchmark::microbenchmark(add1R(1:1000), add1Cpp(1:1000))
```

```
Unit: microseconds
            expr    min     lq     mean  median      uq      max neval
   add1R(1:1000) 57.488 58.274 82.26190 58.6595 61.8105 2276.407   100
 add1Cpp(1:1000)  2.865  3.126 12.54477  3.5320  9.1820  688.414   100
```

9.7 Main differences between R and C++

1. One is compiled, and the other interpreted
2. Indexing objects: In C++ the indices range from 0 to `(n - 1)`, whereas in R is from 1 to `n`.
3. All expressions end with a `;` (optional in R).
4. In C++ object need to be declared, in R not (dynamic).

9.8 C++/Rcpp fundamentals: Types

Besides C-like data types (`double`, `int`, `char`, and `bool`), we can use the following types of objects with Rcpp:

- Matrices: `NumericMatrix`, `IntegerMatrix`, `LogicalMatrix`, `CharacterMatrix`
- Vectors: `NumericVector`, `IntegerVector`, `LogicalVector`, `CharacterVector`
- And more!: `DataFrame`, `List`, `Function`, `Environment`

9.9 Parts of "an Rcpp program"

```
#include<Rcpp.h>
using namespace Rcpp
// [[Rcpp::export]]
NumericVector add1(NumericVector x) {
  NumericVector ans(x.size());
  for (int i = 0; i < x.size(); ++i)
    ans[i] = x[i] + 1;
  return ans;
}
```

Line by line, we see the following:

1. The #include<Rcpp.h> is similar to `library(...)` in R, it brings in all that we need to write C++ code for Rcpp.
2. using namespace Rcpp is somewhat similar to `detach(...)`. This simplifies syntax. If we don't include this, all calls to Rcpp members need to be explicit, **e.g.**, instead of typing `NumericVector`, we would need to type `Rcpp::NumericVector`
3. The `//` starts a comment in C++, in this case, the // [[Rcpp::export]] comment is a flag Rcpp uses to "export" this C++ function to R.
4. It is the first part of the function definition. We are creating a function that returns a NumericVector, is called add1, has a single input element named x that is also a NumericVector.
5. Here, we are declaring an object called ans, which is a NumericVector with an initial size equal to the size of x. Notice that .size() is called a "member function" of the `x` object, which is of class `NumericVector`.
6. We are declaring a for-loop (three parts):
   a. int i = 0 We declare the variable `i`, an integer, and initialize it at 0.

   b. i < x.size() This loop will end when `i`'s value is at or above the length of `x`.

   c. ++i At each iteration, `i` will increment in one unit.

7. ans[i] = x[i] + 1 set the i-th element of `ans` equal to the i-th element of `x` plus 1.

8. return ans exists the function returning the vector `ans`.

Now, where to execute/run this?

- You can use the `sourceCpp` function from the `Rcpp` package to run .cpp scripts (this is what I do most of the time).
- There's also `cppFunction`, which allows compiling a single function.
- Write an R package that works with Rcpp.

For now, let's use the first option.

9.10 Example running .cpp file

Imagine that we have the following file named `norm.cpp`

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
double normRcpp(NumericVector x) {

  return sqrt(sum(pow(x, 2.0)));

}
```

We can compile and obtain this function using this line `Rcpp::sourceCpp("norm.cpp")`. Once compiled, a function called `normRcpp` will be available in the current R session.

9.11 Your turn

9.11.1 Problem 1: Adding vectors

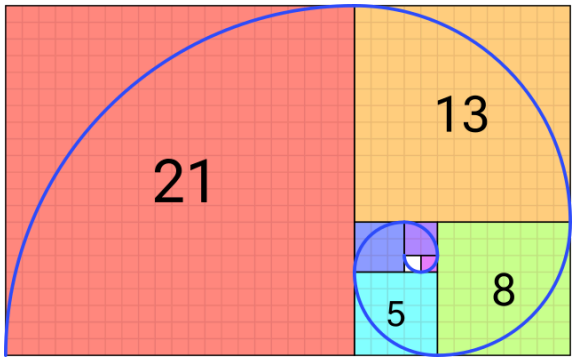
1. Using what you have just learned about Rcpp, write a function to add two vectors of the same length. Use the following template

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
NumericVector add_vectors([declare vector 1], [declare vector 2]) {

  ... magick ...

  return [something];
}
```

2. Now, we have to check for lengths. Use the `stop` function to make sure lengths match. Add the following lines in your code

```
if ([some condition])
  stop("an arbitrary error message :)");
```

9.11.2 Problem 2: Fibonacci series



Each element of the sequence is determined by the following:

$$F(n) = \begin{cases} n, & \text{if } n \leq 1 \\ F(n-1) + F(n-2), & \text{otherwise} \end{cases} \tag{9.1}$$

Using recursions, we can implement this algorithm in R as follows:

```
fibR <- function(n) {
  if (n <= 1)
    return(n)
  fibR(n - 1) + fibR(n - 2)
}
# Is it working?
c(
  fibR(0), fibR(1), fibR(2),
  fibR(3), fibR(4), fibR(5),
  fibR(6)
)
```

```
[1] 0 1 1 2 3 5 8
```

Now, let's translate this code into Rcpp and see how much speed boost we get!

9.11.3 Problem 2: Fibonacci series (solution)

```
#include <Rcpp.h>
// [[Rcpp::export]]
int fibCpp(int n) {
  if (n <= 1)
    return n;

  return fibCpp(n - 1) + fibCpp(n - 2);

}
```

```
microbenchmark::microbenchmark(fibR(20), fibCpp(20))
```

```
Unit: microseconds
       expr      min        lq       mean    median        uq       max neval
   fibR(20) 6200.508 6264.3175 6578.24197 6292.0240 6409.9530 11508.109   100
 fibCpp(20)   15.709   16.0295   26.23908   16.9265   18.1635   885.773   100
```

9.12 RcppArmadillo and OpenMP

- Friendlier than **RcppParallel**... at least for 'I-use-Rcpp-but-don't-actually-know-much-about-C++' users (like myself!).
- Must run only 'Thread-safe' calls, so calling R within parallel blocks can cause problems (almost all the time).
- Use `arma` objects, e.g. `arma::mat`, `arma::vec`, etc. Or, if you are used to them `std::vector` objects as these are thread-safe.
- Pseudo Random Number Generation is not very straightforward... But C++11 has a nice set of functions that can be used together with OpenMP
- Need to think about how processors work, cache memory, etc. Otherwise, you could get into trouble... if your code is slower when run in parallel, then you probably are facing false sharing
- If R crashes... try running R with a debugger (see Section 4.3 in Writing R extensions):

```
~$ R --debugger=valgrind
```

9.12.1 RcppArmadillo and OpenMP workflow

1. Tell Rcpp that you need to include that in the compiler:

```
#include <omp.h>
// [[Rcpp::plugins(openmp)]]
```

2. Within your function, set the number of cores, e.g

```
// Setting the cores
omp_set_num_threads(cores);
```

3. Tell the compiler that you'll be running a block in parallel with OpenMP

```
#pragma omp [directives] [options]
{
  ...your neat parallel code...
}
```

   You'll need to specify how OMP should handle the data:

   - `shared`: Default, all threads access the same copy.
   - `private`: Each thread has its own copy, uninitialized.
   - `firstprivate` Each thread has its own copy, initialized.
   - `lastprivate` Each thread has its own copy. The last value used is returned.

   Setting `default(none)` is a good practice.

4. Compile!

9.12.2 Ex 5: RcppArmadillo + OpenMP

Our own version of the `dist` function... but in parallel!

```
#include <omp.h>
#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::plugins(openmp)]]
using namespace Rcpp;
// [[Rcpp::export]]
arma::mat dist_par(const arma::mat & X, int cores = 1) {
```

```
  // Some constants
  int N = (int) X.n_rows;
  int K = (int) X.n_cols;

  // Output
  arma::mat D(N,N);
  D.zeros(); // Filling with zeros

  // Setting the cores
  omp_set_num_threads(cores);

#pragma omp parallel for shared(D, N, K, X) default(none)
  for (int i=0; i<N; ++i)
    for (int j=0; j<i; ++j) {
      for (int k=0; k<K; k++)
        D.at(i,j) += pow(X.at(i,k) - X.at(j,k), 2.0);

      // Computing square root
      D.at(i,j) = sqrt(D.at(i,j));
      D.at(j,i) = D.at(i,j);
    }


  // My nice distance matrix
  return D;
}
```

```
# Simulating data
set.seed(1231)
K <- 5000
n <- 500
x <- matrix(rnorm(n*K), ncol=K)
# Are we getting the same?
table(as.matrix(dist(x)) - dist_par(x, 4)) # Only zeros
```

```


     0
250000
```

```
# Benchmarking!
microbenchmark::microbenchmark(
  dist(x),               # stats::dist
  dist_par(x, cores = 1), # 1 core
  dist_par(x, cores = 2), # 2 cores
  dist_par(x, cores = 4), # 4 cores
  times = 1,
  unit = "ms"
)
```

```
Unit: milliseconds
                  expr       min        lq      mean    median        uq
               dist(x) 1179.6251 1179.6251 1179.6251 1179.6251 1179.6251
 dist_par(x, cores = 1)  859.7624  859.7624  859.7624  859.7624  859.7624
 dist_par(x, cores = 2)  630.8682  630.8682  630.8682  630.8682  630.8682
 dist_par(x, cores = 4)  592.1344  592.1344  592.1344  592.1344  592.1344
       max neval
 1179.6251     1
  859.7624     1
```

```
  630.8682     1
  592.1344     1
```

9.12.3 Ex 6: The future

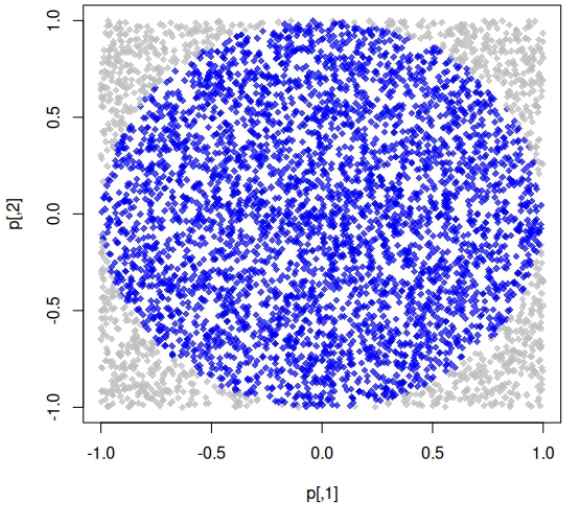
- **future** is an R package that was designed "to provide a very simple and uniform way of evaluating R expressions asynchronously using various resources available to the user."
- `future` class objects are either resolved or unresolved.
- If queried, **Resolved** values are return immediately, and **Unresolved** values will block the process (i.e. wait) until it is resolved.
- Futures can be parallel/serial, in a single (local or remote) computer, or a cluster of them.

Let's see a brief example

```
library(future)
plan(multicore)
# We are creating a global variable
a <- 2
# Creating the futures has only the overhead (setup) time
system.time({
  x1 %<-% {Sys.sleep(3);a^2}
  x2 %<-% {Sys.sleep(3);a^3}
})
##    user  system elapsed
##   0.022   0.021   0.042
# Let's just wait 5 seconds to make sure all the cores have returned
Sys.sleep(3)
system.time({
  print(x1)
  print(x2)
})
## [1] 4
## [1] 8
##    user  system elapsed
##   0.004   0.000   0.005
```

9.12.4 Bonus track 1: Simulating π

- We know that $\pi = \frac{A}{r^2}$. We approximate it by randomly adding points x to a square of size 2 centered at the origin.
- So, we approximate π as $\Pr\{\| x \| \leq 1\} \times 2^2$



The R code to do this

```
pisim <- function(i, nsim) {  # Notice we don't use the -i-
  # Random points
  ans  <- matrix(runif(nsim*2), ncol=2)

  # Distance to the origin
  ans  <- sqrt(rowSums(ans^2))

  # Estimated pi
  (sum(ans <= 1)*4)/nsim
}
```

```
library(parallel)
# Setup
cl <- makePSOCKcluster(4L)
clusterSetRNGStream(cl, 123)
# Number of simulations we want each time to run
nsim <- 1e5
# We need to make -nsim- and -pisim- available to the
# cluster
clusterExport(cl, c("nsim", "pisim"))
# Benchmarking: parSapply and sapply will run this simulation
# a hundred times each, so at the end we have 1e5*100 points
# to approximate pi
microbenchmark::microbenchmark(
  parallel = parSapply(cl, 1:100, pisim, nsim=nsim),
  serial   = sapply(1:100, pisim, nsim=nsim),
  times    = 1
)
```

```
Unit: milliseconds
     expr      min       lq     mean   median       uq      max neval
 parallel 279.9740 279.9740 279.9740 279.9740 279.9740 279.9740     1
   serial 372.6806 372.6806 372.6806 372.6806 372.6806 372.6806     1
```

```
ans_par <- parSapply(cl, 1:100, pisim, nsim=nsim)
ans_ser <- sapply(1:100, pisim, nsim=nsim)
stopCluster(cl)
```

```
     par      ser        R
3.141762 3.141266 3.141593
```

9.13 See also

- Package parallel
- Using the iterators package
- Using the foreach package
- 32 OpenMP traps for C++ developers
- The OpenMP API specification for parallel programming
- 'openmp' tag in Rcpp gallery
- OpenMP tutorials and articles

For more, check out the CRAN Task View on HPC

10. Debugging R w C++/C code

Although debugging R code is easy, the same doesn't apply to compiled code in R[12]. This chapter shows a few ways to debug your R + C++ code. You will need the GNU Debugger (GDB) and Valgrind.

Before we start, remember that we will not deal with the good old-fashioned `Rprint("Your code is working up to here")` approach. Printing messages while your program runs can be very informative, but using Valgrind and GDB is, in my humble opinion, faster as, most of the time, those will scream at you, indicating the location of your problem.

> **Note**
>
> The manual Writing R Extensions [1] has a fair amount of information about debugging compiled code in R here. Dirk Eddelbuettel (lead author of Rcpp) has an excellent post on Stackoverflow and recommends a tutorial hosted on BioConductor [10].

10.1 Debugging with Valgrind

As a starting point, we will use Valgrind. Valgrind provides a mature framework for memory debugging and profiling. We must launch the program through the command line to use a debugger within R. To lunch R with Valgrind, we use the following:

```
$ R --debugger=valgrind
```

Which will result in something like the following:

```
==31245== Memcheck, a memory error detector
==31245== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==31245== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==31245== Command: /usr/lib/R/bin/exec/R
==31245==

R version 4.2.3 (2023-03-15) -- "Shortstop Beagle"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

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

  Natural language support but running in an English locale

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

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

>
```

[12] Debugging **only** C++/C code is easy, though. If you already work with compiled code, you must be aware of VS Code and the many other tools out there for debugging C++/C code.

Once R executes with Valgrind, the debugger will catch any memory leaks generated by your C++/C code. The following is a faulty Rcpp program that creates a pointer using `new` and "forgets" to delete it.

```
#include <Rcpp.h>

using namespace Rcpp;

// [[Rcpp::export]]
NumericVector faulty_program(int n) {

    // Here is the faulty line
    NumericVector * x_ptr = new NumericVector(n);

    return *x_ptr;

}

/***R
# Calling the faulty program
faulty_program(10)
*/
```

We can use the `-e` flag in the `R` command to compile the `Rcpp` script using `sourceCpp`:

```
R --debugger=valgrind -e 'Rcpp::sourceCpp("rcpp-debugging-faulty.cpp")'
```

```
==1804== Memcheck, a memory error detector
==1804== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==1804== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==1804==  Command:  /usr/local/lib/R/bin/exec/R  -e  Rcpp::sourceCpp("rcpp-debugging-
faulty.cpp")
==1804==

R version 4.5.3 (2026-03-11) -- "Reassured Reassurer"
Copyright (C) 2026 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu

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

  Natural language support but running in an English locale

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

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

> Rcpp::sourceCpp("rcpp-debugging-faulty.cpp")

> faulty_program(10)
 [1] 0 0 0 0 0 0 0 0 0 0
>
==1804==
==1804== HEAP SUMMARY:
==1804==     in use at exit: 60,883,195 bytes in 11,665 blocks
==1804==   total heap usage: 31,690 allocs, 20,025 frees, 93,845,132 bytes allocated
==1804==
```

```
==1804== LEAK SUMMARY:
==1804==    definitely lost: 32 bytes in 1 blocks
==1804==    indirectly lost: 0 bytes in 0 blocks
==1804==      possibly lost: 0 bytes in 0 blocks
==1804==    still reachable: 60,883,163 bytes in 11,664 blocks
==1804==                       of which reachable via heuristic:
==1804==                         newarray           : 4,264 bytes in 1 blocks
==1804==         suppressed: 0 bytes in 0 blocks
==1804== Rerun with --leak-check=full to see details of leaked memory
==1804==
==1804== For lists of detected and suppressed errors, rerun with: -s
==1804== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

By the end of the output, in the `LEAK SUMMARY` section, we see `definitely lost: 24 bytes in 1 block`, *i.e.*, a memory leak. If we change the program by deleting the pointer before returning, the leak will be solved:
New program:

```
    NumericVector res = *x_ptr;
    delete x_ptr;

    return res;
```

Old program

```
    return *x_ptr;
```

Re-running R with Valgrind returns the following (only the last few lines):

```
R --debugger=valgrind -e 'Rcpp::sourceCpp("rcpp-debugging-faulty-fixed.cpp")'
```

```
==1861== HEAP SUMMARY:
==1861==     in use at exit: 60,883,837 bytes in 11,664 blocks
==1861==   total heap usage: 31,727 allocs, 20,063 frees, 93,865,542 bytes allocated
==1861==
==1861== LEAK SUMMARY:
==1861==    definitely lost: 0 bytes in 0 blocks
==1861==    indirectly lost: 0 bytes in 0 blocks
==1861==      possibly lost: 0 bytes in 0 blocks
==1861==    still reachable: 60,883,837 bytes in 11,664 blocks
==1861==                       of which reachable via heuristic:
==1861==                         newarray           : 4,264 bytes in 1 blocks
==1861==         suppressed: 0 bytes in 0 blocks
==1861== Rerun with --leak-check=full to see details of leaked memory
==1861==
==1861== For lists of detected and suppressed errors, rerun with: -s
==1861== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

No more memory leaks.

10.2 Using GDB

Sometimes, we need to go further and inspect what's going on **inside** the program. GBD is excellent for that. With GBD, we can set breakpoints that allow us to review the program while it is executed.

The following Rcpp code generates a `memory not mapped` type error:

```
#include <Rcpp.h>
```

```
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector faulty_program(int n) {

    // Here is the faulty line
    NumericVector * x_ptr;

    return *x_ptr;

}

/***R
# Calling the faulty program
faulty_program(10)
*/
```

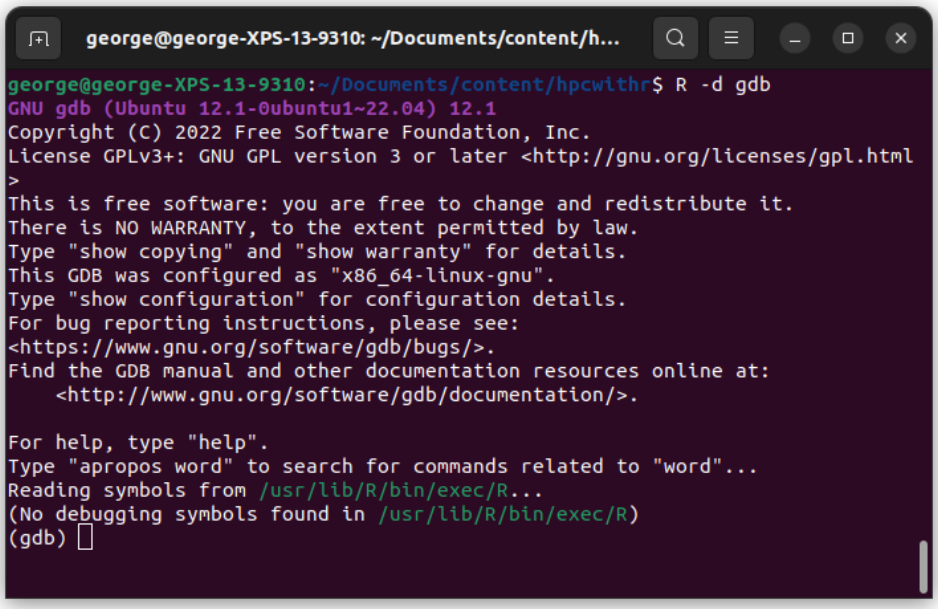
In it, we try to access a location in the memory that hasn't been allocated yet, namely, a `NumericVector` declared as a pointer but never assigned. Using `R --debugger=valgrind` generates the following code:

```
==1918== Memcheck, a memory error detector
==1918== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==1918== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==1918== Command: /usr/local/lib/R/bin/exec/R -e Rcpp::sourceCpp("rcpp-debugging-not-
mapped.cpp")
==1918==

R version 4.5.3 (2026-03-11) -- "Reassured Reassurer"
Copyright (C) 2026 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu

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

  Natural language support but running in an English locale

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

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

> Rcpp::sourceCpp("rcpp-debugging-not-mapped.cpp")

> faulty_program(10)
==1918== Invalid read of size 8
==1918==    at 0x2F34F5D6: get__ (PreserveStorage.h:52)
==1918==         by  0x2F34F5D6:  copy__<Rcpp::Vector<14,  Rcpp::PreserveStorage>  >
(PreserveStorage.h:66)
==1918==    by 0x2F34F5D6: Vector (Vector.h:64)
==1918==    by 0x2F34F5D6: faulty_program(int) (rcpp-debugging-not-mapped.cpp:11)
==1918==          by  0x2F34F758:  sourceCpp_1_faulty_program  (rcpp-debugging-not-
mapped.cpp:33)
==1918==    by 0x495DB7D: R_doDotCall (dotcode.c:754)
==1918==    by 0x495E0F6: do_dotcall (dotcode.c:1437)
==1918==    by 0x49ACA5C: Rf_eval (eval.c:1260)
==1918==    by 0x49AE67D: R_execClosure (eval.c:2393)
==1918==    by 0x49AF3D6: applyClosure_core (eval.c:2306)
```

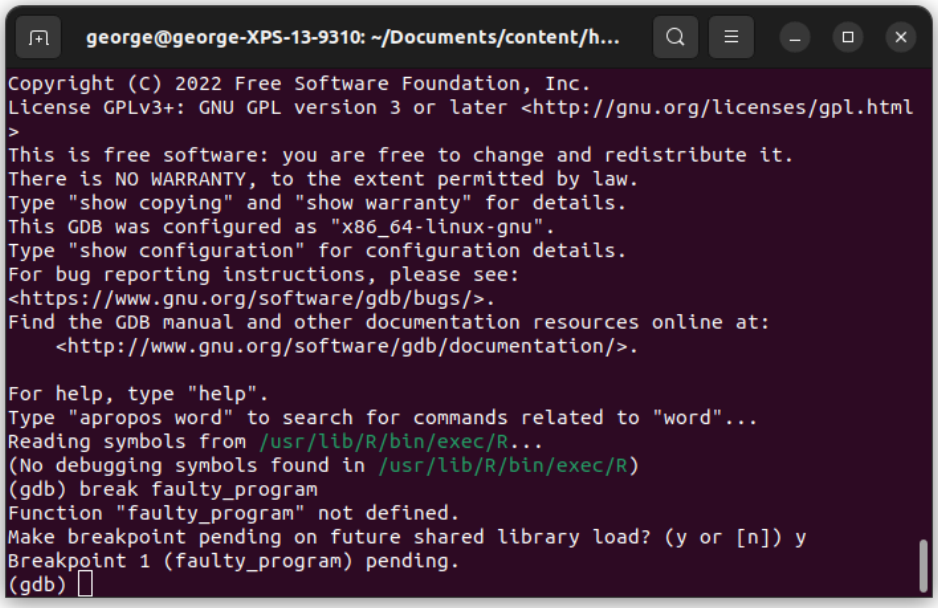
```
==1918==    by 0x49AC575: Rf_applyClosure (eval.c:2328)
==1918==    by 0x49AC575: Rf_eval (eval.c:1280)
==1918==    by 0x49B30E3: do_eval (eval.c:3959)
==1918==    by 0x499F3E4: bcEval_loop (eval.c:8118)
==1918==    by 0x49AC069: bcEval (eval.c:7501)
==1918==    by 0x49AC069: bcEval (eval.c:7486)
==1918==    by 0x49AC43A: Rf_eval (eval.c:1167)
==1918==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==1918==

 *** caught segfault ***
address (nil), cause 'memory not mapped'

Traceback:
 1: .Call(<pointer: 0x2f34f6e0>, n)
 2: faulty_program(10)
 3: eval(ei, envir)
 4: eval(ei, envir)
 5: withVisible(eval(ei, envir))
 6: source(file = srcConn, local = env, echo = echo)
 7: Rcpp::sourceCpp("rcpp-debugging-not-mapped.cpp")
An irrecoverable exception occurred. R is aborting now ...
==1918==
==1918== Process terminating with default action of signal 11 (SIGSEGV): dumping core
==1918==    at 0x4D6BB2C: __pthread_kill_implementation (pthread_kill.c:44)
==1918==    by 0x4D6BB2C: __pthread_kill_internal (pthread_kill.c:78)
==1918==    by 0x4D6BB2C: pthread_kill@@GLIBC_2.34 (pthread_kill.c:89)
==1918==    by 0x4D1227D: raise (raise.c:26)
==1918==    by 0x4D1232F: ??? (in /usr/lib/x86_64-linux-gnu/libc.so.6)
==1918==         by  0x2F34F5D5:  copy__<Rcpp::Vector<14,  Rcpp::PreserveStorage>  >
(PreserveStorage.h:65)
==1918==    by 0x2F34F5D5: Vector (Vector.h:64)
==1918==    by 0x2F34F5D5: faulty_program(int) (rcpp-debugging-not-mapped.cpp:11)
==1918==
==1918== HEAP SUMMARY:
==1918==     in use at exit: 60,975,513 bytes in 11,879 blocks
==1918==   total heap usage: 31,659 allocs, 19,780 frees, 93,822,239 bytes allocated
==1918==
==1918== LEAK SUMMARY:
==1918==    definitely lost: 0 bytes in 0 blocks
==1918==    indirectly lost: 0 bytes in 0 blocks
==1918==      possibly lost: 5,987 bytes in 19 blocks
==1918==    still reachable: 60,969,526 bytes in 11,860 blocks
==1918==                       of which reachable via heuristic:
==1918==                         newarray           : 4,264 bytes in 1 blocks
==1918==         suppressed: 0 bytes in 0 blocks
==1918== Rerun with --leak-check=full to see details of leaked memory
==1918==
==1918== For lists of detected and suppressed errors, rerun with: -s
==1918== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
```

To inspect an error with GDB, we have to follow these steps:

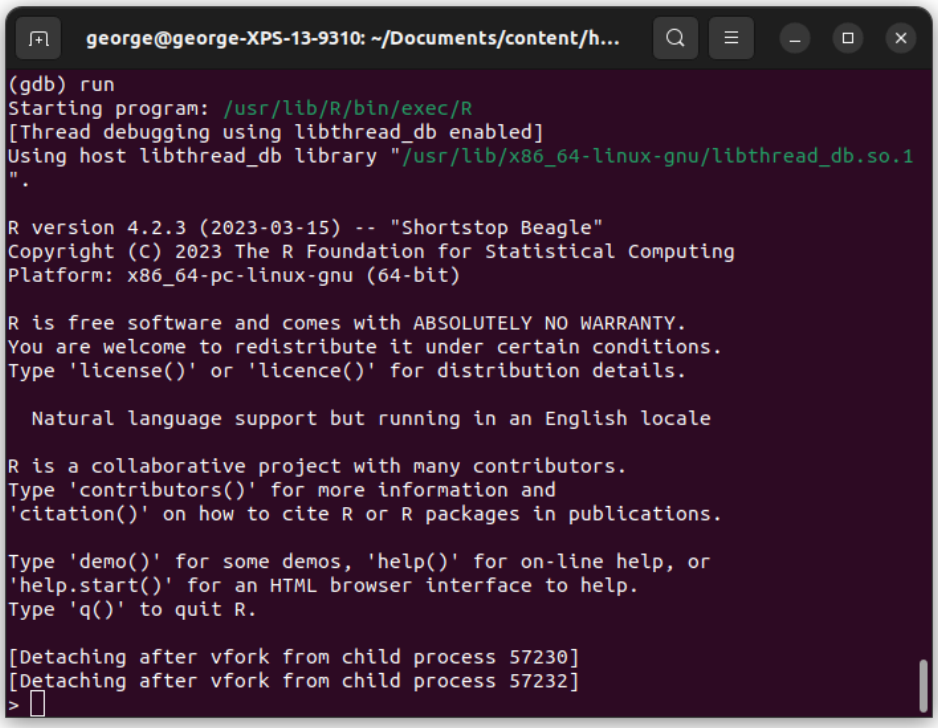
1. Run R with gdb as debugger: `R --debugger=gdb`. R won't start immediately, so we have time to add breakpoints.



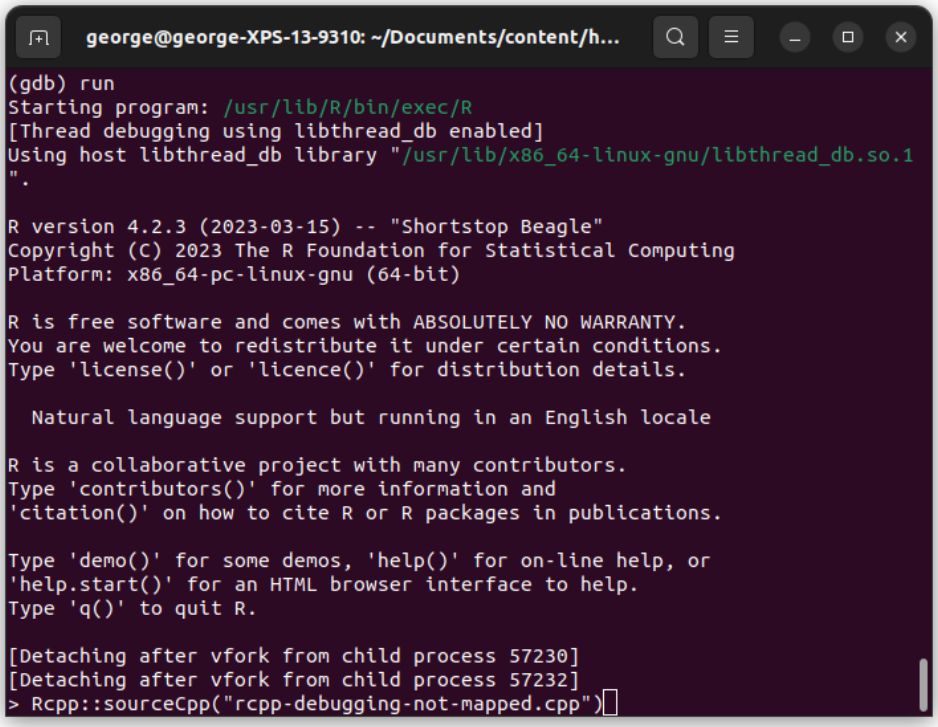
2. We can set a breakpoint on the given function with `break faulty_program`. GDB will grab it on the fly, so choose `yes`. It most likely will warn you that there's no symbol for that function.



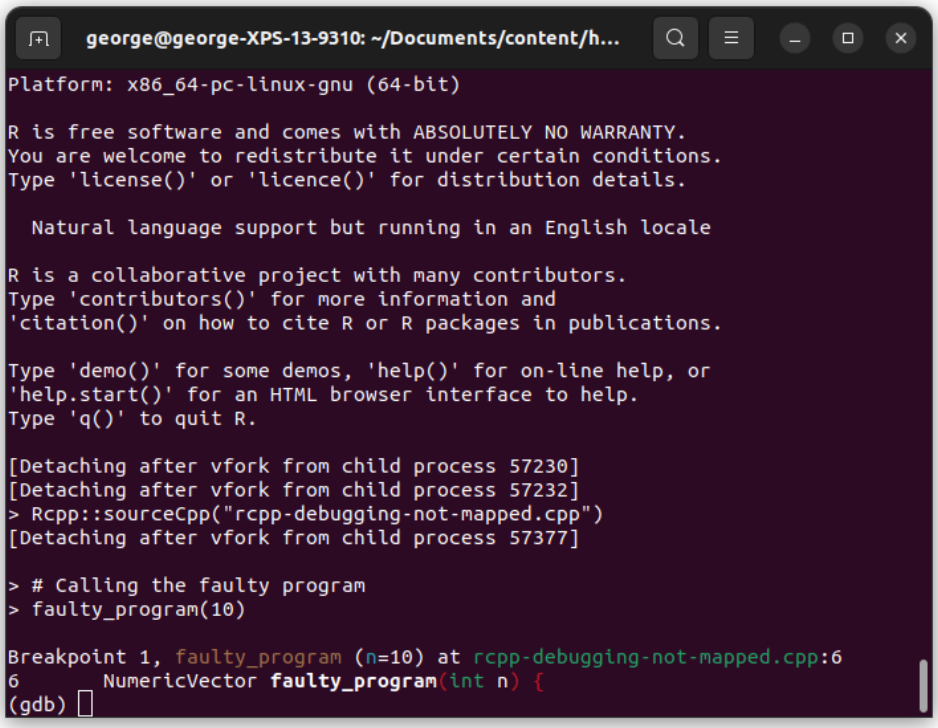
3. Run R using the `run` command in gdb:



4. Source the program using `Rcpp::sourceCpp`, and wait for `gdb` to pause the program once it reaches the breakpoint.



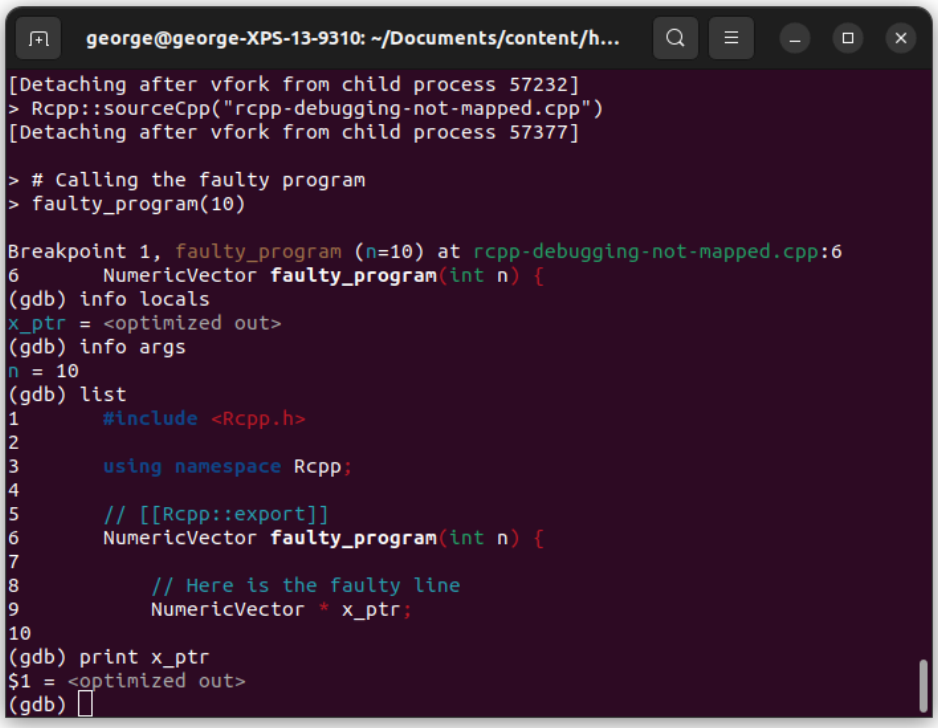
5. Once the program has paused, we can inspect the context.



Because of the number of options it has, using GBD can be overwhelming. Here is the list of commands I use the most:

```
help        # Get help
info locals # List the local variables (scope)
info args   # List the arguments passed to the function
list        # See the last few lines of the source code
continue    # Continue running the program
next        # Execute the next step
bt          # Show the entire call stack (backtrace)
up          # Go up one level in the call stack
down        # Go down one level in the call stack
print       # Print/display an expression
```

And here is an example using `info locals`, `info args`, `list`, and `print`.



6. Finally, to exit the program, type `exit` (similar to `q()` in R.)

Appendices

A. Notes about scoping in C++

This chapter will present a more advanced but important topic in C++: RAII and scoping. RAII stands for Resource Acquisition Is Initialization, and it is a programming idiom that ensures that resources are properly released when they are no longer needed. From the perspective of an R user, this provides a nice opportunity for scoping in C++. Let's take a look with an example:

```
class Agent {
public:
    int fun();
};

// Declaration of the Model class
class Model {
private:
    Agent agent_;
    int x_;

public:
    inline static Model * instance_ = nullptr;
    virtual int run();
    Model(int x) : agent_(Agent()), x_(x) {};

    // Key function that allows access to `Model`
    // by calls within the call stack.
    static Model & get();

    int get_x();
};

// Implementation of the Model methods
inline int Model::run() {
    Model::instance_ = this;
    return agent_.fun();
};

inline Model & Model::get() {return *instance_;};
inline int Model::get_x() {return x_;};

// Implementation of the Agent method
inline int Agent::fun() {return Model::get().get_x();};

// [[Rcpp::export]]
int test_scoping_cpp() {

    // Create an instance of Model
    Model Model(42);

    // Run the Model
    return Model.run();
}

// [[Rcpp::export]]
bool is_it_on() {
```

```
    return Model::instance_ != nullptr;
}
```

Checking whether it works or not:

```
test_scoping_cpp()
```

```
[1] 42
```

It does! Let's split this into pieces. First of all, the `Agent` class does not have explicit access to the `Model` class; in no place of its definition has it as a member. The key lies in the `static` function `Model::get()`, which can be call by any function within the call stack without having to access to `Model`:

```
Model::run() -> Agent::fun() -> Model::get() -> Model::get_x()
```

And the `Agent` function access the model through the static member. Furthermore, we can see that after we exited `Model::run()`, the variable `Model::instance_` is still not null:

```
is_it_on()
```

```
[1] TRUE
```

Now, this is not entirely safe, as the variable `instance_` would still be available even after calling the `Model::run()` function. To address this, we can use an auxiliary class that resets the `instance_` to `nullptr`:

```
#include <Rcpp.h>
class Agent {
public:
    int fun();
};

class Scope;

// Declaration of the Model class
class Model_s {
    friend Scope;
private:
    Agent agent_;
    int x_;

public:
    inline static Model_s * instance_ = nullptr;
    virtual int run();
    Model_s(int x) : agent_(Agent()), x_(x) {};
    static Model_s & get();
    int get_x();
};

// Declaration & Implementation of the Scope method
class Scope {
private:
    Model_s * prev_;
public:
    explicit Scope(Model_s * model) : prev_(Model_s::instance_) {
        Model_s::instance_ = model;
    };
    ~Scope() {
        Model_s::instance_ = prev_;
    };
};
```

```
// Implementation of the Model_s methods
inline int Model_s::run() {

    // Here is the important point. The constructor
    // of `Scope` sets the value of `instance_` to
    // be the current instance of `Model`, but then
    // once we exit the call, the destructor re-assigns
    // `instance_` to `nullptr`.
    Scope scope(this);

    return agent_.fun();
};

inline Model_s & Model_s::get() {return *instance_;};
inline int Model_s::get_x() {return x_;};

// Implementation of the Agent method
inline int Agent::fun() {return Model_s::get().get_x();};

// [[Rcpp::export]]
int test_scoping_cpp2() {

    // Create an instance of Model
    Model_s Model(42);

    // Run the Model
    return Model.run();
}

// [[Rcpp::export]]
bool is_it_on2() {
    return Model_s::instance_ != nullptr;
}
```

The simple class `Scope` provides a nice way of controlling the duration of this static variable:

```
class Scope {
private:
    Model_s * prev_;
public:
    explicit Scope(Model_s * model) : prev_(Model_s::instance_) {
        Model_s::instance_ = model;
    };
    ~Scope() {
        Model_s::instance_ = prev_;
    };
};
```

Once it is invoked, it sets the value for the static member of `Model_s` for the duration of `Scope`. Once `Model_s::run()` exists, the destructor of `Scope` resets the value of `Model_s::instance_` to its previous value, `nullptr`, in this case. Nonetheless, this provides a way to allow nested calls setting the proper value of `instance_` as the callstack grows. Here is the result of the new implementation

```
test_scoping_cpp2() # Checking it works
```

```
[1] 42
```

```
is_it_on2()         # And that it is turned off
```

```
[1] FALSE
```

> ⚠ Warning
>
> While implementing this, I was using `Model` for both code chunks in C++. Although it still allows to me compile, the symbol `Model` is shared by both implementations, which broke my code (the `is_it_on2()` function was returning `TRUE` when it was supposed to return `FALSE`).

A.1 Multi-thread versions

If you are using parallel computing via OpenMP and similar, it is important to play it safe. By default, copies of `Model` will share the value of `_instance`, which may be undesirable. Instead, it is recommended to use the `thread_local` keyword:

```
class Model {
    static thread_local instance_;
}

// Needs to be initialized outside of the class
thread_local Model::instance_ = nullptr;
```

This will ensure that, if you are creating copies of the `Model`, each thread has individual access to their own copy.

A.2 Working with multiple compiled versions

Another caveate that I have experienced is when having two different C++ library objects (what Rcpp creates as `.so`, `.o`, or `.dll`, depending on the OS), problems can happen. Particularly, I observed this during the development of the `epiworldR` and `measles` R packages. Both of them had compiled libraries depending on the `epiworld` C++ template library, so when I load them together, using the static member access was not working as expected; in those cases, it is better to be more explicit and simply pass `Model` as another argument; in other words, don't be "smart" about it and stick to something more reliable.

B. Misc

B.1 General resources

The Center for Advanced Research Computing (formerly HPCC) has tons of resources online. Here are a couple of useful links:

- **Center for Advanced Research Computing Website** https://carc.usc.edu
- **User forum (very useful!)** https://hpc-discourse.usc.edu/categories
- **Monitor your account** https://hpcaccount.usc.edu/
- **Slurm Jobs Templates** https://carc.usc.edu/user-information/user-guides/high-performance-computing/slurm-templates
- **Using R** https://carc.usc.edu/user-information/user-guides/software-and-programming/r

B.2 Data Pointers

IMHO, these are the most important things to know about data management at USC's HPC:

1. Do your data transfer using the transfer nodes (it is faster).
2. Never use your home directory as a storage space (use your project's allotted space instead).
3. Use the scratch filesystem for temp data only, i.e., never save important files in scratch.
4. Finally, besides of **Secure copy protocol (scp)**, if you are like me, try setting up a GUI client for moving your data (see this).

B.3 The Slurm options they forgot to tell you about...

First of all, you have to be aware that the only thing Slurm does is allocate resources. If your application uses parallel computing or not, that's another story.

Here some options that you need to be aware of:

- `ntasks` (default 1) This tells Slurm how many processes you will have running. Notice that processes need not to be in the same node (so Slurm may reserve space in multiple nodes)
- `cpus-per-task` (defatult 1) This is how many CPUs each task will be using. This is what you need to use if you are using OpenMP (or a package that uses that), or anything you need to keep within the same node.
- `nodes` the number of nodes you want to use in your job. This is useful mostly if you care about the maximum (I would say) number of nodes you want your job to work. So, for example, if you want to use 8 cores for a single task and force it to be in the same node, you would add the option `--nodes=1/1`.
- `mem-per-cpu` (default 1GB) This is the MINIMUM amount of memory you want Slurm to allocate for the task. Not a hard barrier, so your process can go above that.
- `time` (default 30min) This is a hard limit as well, so if you job takes more than the specified time, Slurm will kill it.
- `partition` (default "") and `account` (default"") these two options go along together, this tells Slurm what resources to use. Besides of the private resources we have the following:
  - **quick partition**: Any job that is small enough (in terms of time and memory) will go this way. This is usually the default if you don't specify any memory or time options.
  - **main partition**: Jobs that require more resources will go in this line.

- **scavenge partition**: If you need a massive number resources, and have a job that shouldn't, in principle, take too long to finalize (less than a couple of hours), and **you are OK with someone killing it**, then this queue is for you. The Scavenge partition uses all the idle resources of the private partitions, so if any of the owners requests the resources, Slurm will cancel your job, i.e. you have no priority (see more).
- **largemem partition**: If you need lots of memory, we have 4 1TB nodes for that.

More information about the partitions here

B.4 Good practices (recomendations)

This is what you should use as a minimum:

```
#SBATCH --output=simulation.out
#SBATCH --job-name=simulation
#SBATCH --time=04:00:00
#SBATCH --mail-user=[you]@usc.edu
#SBATCH --mail-type=END,FAIL
```

- `output` is the name of the logfile to which Slurm will write.
- `job-name` is that, the name of the job. You can use this to either kill or at least be able to identify what is what you are running when you use `myqueue`
- `time` Try always to set a time estimate (plus a little more) for your job.
- `mail-user`, `mail-type` so Slurm notifies you when things happen

Also, in your R code

- Any I/O should be done to either Scratch (`/scratch/[your usc net id]`) or Tmp `Sys.getenv("TMPDIR")`.

B.5 Running R interactively

1. The HPC has several pre-installed pieces of software. R is one of those.
2. To access the pre-installed software, we use the **Lmod module system** (more information **here**)
3. It has multiple versions of R installed. Use your favorite one by running

   ```
   module load R/4.2.2/[version number]
   ```

   Where `[version number]` can be 3.5.6 and up to 4.0.3 (the latest update). The `usc` module automatically loads gcc/8.3.0, openblas/0.3.8, openmpi/4.0.2, and pmix/3.1.3.
4. It is never a good idea to use your home directory to install R packages, that's why you should try using a **symbolic link instead**, like this

   ```
   cd ~
   mkdir -p /path/to/a/project/with/lots/of/space/R
   ln -s /path/to/a/project/with/lots/of/space/R R
   ```

   This way, whenever you install your R packages, R will default to that location
5. You can run interactive sessions on HPC, but this recommended to be done using the `salloc` function in Slurm, in other words, NEVER EVER USE R (OR ANY SOFTWARE) TO DO DATA ANALYSIS IN THE HEAD NODES! The options passed to salloc are the same options that can be passed to `sbatch` (see the next section.) For example, if need to do some analyses in the `thomas` partition (which is private and I have access to), I would type something like

   ```
   salloc --account=lc_pdt --partition=thomas --time=02:00:00 --mem-per-cpu=2G
   ```

   This would put me in a single node allocating 2 gigs of memory for a maximum of 2 hours.

B.6 NoNos when using R

- Do computation on the head node (compile stuff is OK)
- Request a number of nodes (unless you know what you are doing)
- Use your home directory for I/O
- Save important information in Staging/Scratch

References

C. News

C.1 Version 2026.03.18

- Added a new example in the `parallel` package chapter showing how to replace a for-loop with `parLapply()`, including a discussion of per-simulation seeding as an alternative to `clusterSetRNGStream`.

C.2 Version 2025.09.03

- Started the news section of the book. I will try to keep track of the changes made to the book here. Previous versions of the book will now be archived using releases on GitHub.

Bibliography

[1] R Core Team, "R: A Language and Environment for Statistical Computing." Vienna, Austria, 2023. [Online]. Available: https://www.r-project.org/

[2] M. Dowle and A. Srinivasan, "data.table: Extension of `data.frame`." 2021. [Online]. Available: https://cran.r-project.org/package=data.table

[3] LUMI consortium, "Documentation - Distribution and binding." [Online]. Available: https://docs.lumi-supercomputer.eu/runjobs/scheduled-jobs/distribution-binding/

[4] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management," in *Job Scheduling Strategies for Parallel Processing*, vol. 2862, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds., in Lecture Notes in Computer Science, vol. 2862., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60. doi: 10.1007/10968987_3.

[5] G. Vega Yon and P. Marjoram, "slurmR: A lightweight wrapper for HPC with Slurm," *The Journal of Open Source Software*, vol. 4, no. 39, July 2019, doi: 10.21105/joss.01493.

[6] G. Vega Yon and P. Marjoram, "slurmR: A Lightweight Wrapper for 'Slurm'." 2022. [Online]. Available: https://github.com/USCbiostats/slurmR

[7] D. Eddelbuettel and R. François, "Rcpp: Seamless R and C++ Integration," *Journal of Statistical Software*, vol. 40, no. 8, pp. 1–18, 2011, doi: 10.18637/jss.v040.i08.

[8] D. Eddelbuettel, *Seamless R and C++ Integration with Rcpp*. New York: Springer, 2013. doi: 10.1007/978-1-4614-6868-4.

[9] D. Eddelbuettel and J. J. Balamuta, "Extending extitR with extitC++: A Brief Introduction to extitRcpp," *The American Statistician*, vol. 72, no. 1, pp. 28–36, 2018, doi: 10.1080/00031305.2017.1375990.

[10] K. Rue-Albrecht, D. Cassol, J. Rainer, and L. Shepherd, *Welcome | Bioconductor Packages: Development, Maintenance, and Peer Review*.