Applied HPC with R

George G. Vega Yon

2023-04-09

Table of contents

Preface 5

I Parallel computing 6

1 Introduction 7

2 High-Performance Computing: An overview 8
- 2.1 Big Data . . . 8
- 2.2 Parallel computing . . . 8
  - 2.2.1 Serial vs. Parallel . . . 10
- 2.3 High-performance computing in R . . . 11
  - 2.3.1 Some vocabulary for HPC . . . 11
- 2.4 GPU vs. CPU . . . 11
- 2.5 When is it a good idea? . . . 12
- 2.6 Parallel computing in R . . . 12

3 The parallel R package 14
- 3.1 Parallel workflow . . . 14
- 3.2 Types of clusters: PSOCK . . . 14
- 3.3 Types of clusters: Fork . . . 15
- 3.4 Ex 1: Parallel RNG with `makePSOCKCluster` . . . 15
- 3.5 Ex 2: Parallel RNG with `makeForkCluster` . . . 16
- 3.6 Ex 3: Parallel RNG with `mclapply` (Forking on the fly) . . . 17

II Working with a Cluster 19

4 What is Slurm 20
- 4.1 Definitions . . . 20

5 A brief intro to Slurm 22
- 5.1 Step 1: Copy the Slurm script to HPC . . . 22
- 5.2 Step 2: Logging to HPC . . . 23
- 5.3 Step 3: Submitting the job . . . 23

6 Simulating *pi* 25

6.1 Submitting jobs to Slurm 25

6.1.1 Case 1: Single job, single core job 26

6.1.2 Case 2: Single job, multicore job 27

6.2 Jobs with the slurmR package 28

6.2.1 Case 3: Single job, multinode job 29

6.2.2 Case 4: Multi job, single/multi-core 30

6.2.3 Case 5: Skipping the .slurm file 32

III Using C++ 34

7 Rcpp 35

7.1 Before we start 35

7.2 R is great, but… 36

7.3 Enter Rcpp 36

7.4 Why bother? 36

7.5 Example 1: Looping over a vector 37

7.6 How much fast? 37

7.7 Main differences between R and C++ 38

7.8 C++/Rcpp fundamentals: Types 38

7.9 Parts of "an Rcpp program" 39

7.10 Example running .cpp file 40

7.11 Your turn 40

7.11.1 Problem 1: Adding vectors 40

7.11.2 Problem 2: Fibonacci series 41

7.11.3 Problem 2: Fibonacci series (solution) 42

7.12 RcppArmadillo and OpenMP 42

7.12.1 RcppArmadillo and OpenMP workflow 43

7.12.2 Ex 5: RcppArmadillo + OpenMP 43

7.12.3 Ex 6: The future 45

7.12.4 Bonus track 1: Simulating π 46

7.13 See also 48

8 Debugging R w C++/C code 49

8.1 Starting R with a debugger 49

8.2 Using GDB 53

9 Misc 61

9.1 General resources 61

9.2 Data Pointers 61

9.3 The Slurm options they forgot to tell you about… 62

9.4 Good practices (recomendations) 63

9.5 Running R interactively 63
9.6 NoNos when using R 64

References 65

Preface

The R programming language (R Core Team 2023) can be fantastic for most daily tasks. But as soon as you start dealing with more complicated problems, you may face the for-loop bottleneck. 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.[1]

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

[1] 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.

Part I

Parallel computing

1 Introduction

While most people see R as a slow programming language, it has powerful features that dramatically accelerate your code [1]. 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.

[1] Nonetheless, this claim can be said about almost any programming language; there are notable examples like the R package `data.table` (Dowle and Srinivasan 2021) which has been demonstrated to out-perform most data wrangling tools.

2 High-Performance Computing: An overview

From R's perspective, we can think of HPC in terms of two or three things:[1] Big data, parallel computing, and compiled code.

2.1 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:

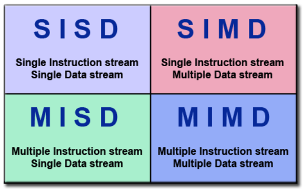
- **Out-of-memory storage**. The idea is simple, instead of using your RAM to load the data, use other methods to load the data. Two notewirthy alternatives are the bigmemory and implyr R packages. The `bigmemory` package provides methods for using "*file-backed*" matrices. On the other hand, `implyr` 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.

2.2 Parallel computing

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

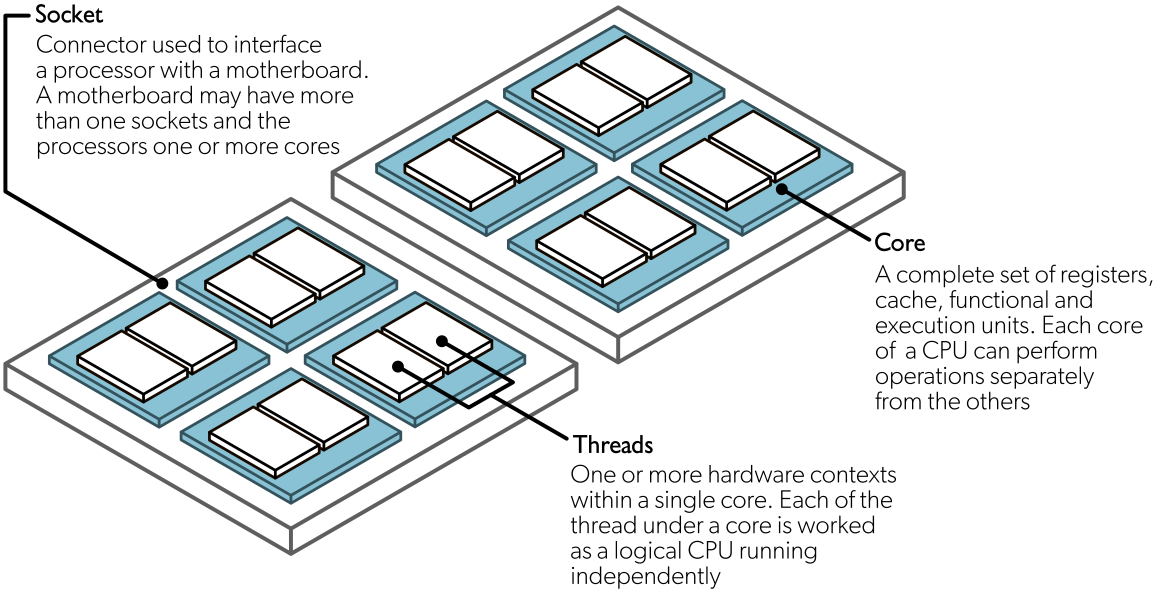
[1] Make sure to check out CRAN Task View on HPC.



SISD Single Instruction stream Single Data stream	SIMD Single Instruction stream Multiple Data stream
MISD Multiple Instruction stream Single Data stream	MIMD Multiple Instruction stream Multiple Data stream

Figure 2.1: Flynn's Classical Taxonomy (Blaise Barney, **Introduction to Parallel Computing**, Lawrence Livermore National Laboratory)

multiple levels at which a computer program can be parallelized. To understand this, we first need to see what composes a modern computer:



Socket
Connector used to interface a processor with a motherboard. A motherboard may have more than one sockets and the processors one or more cores

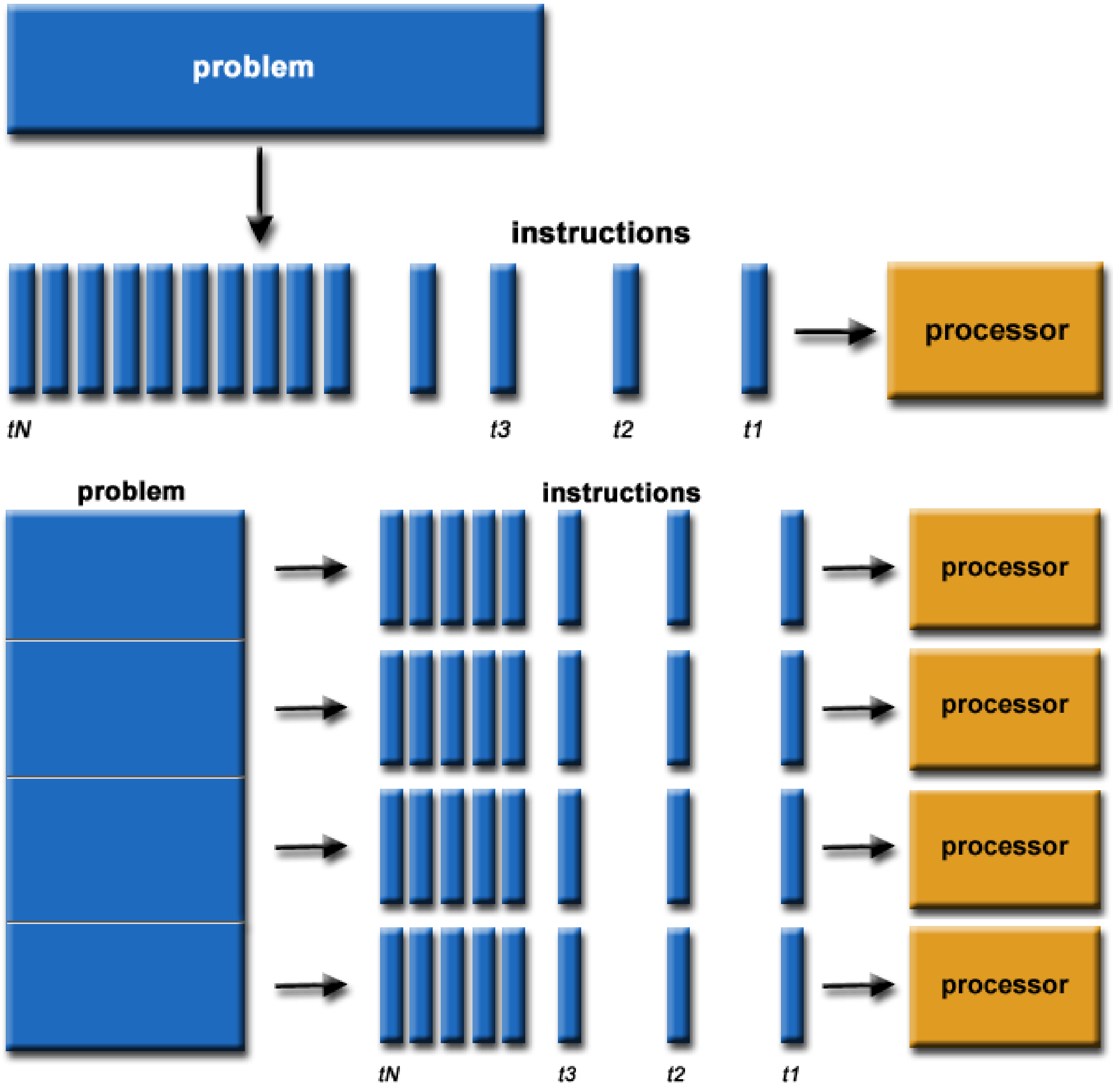
Core
A complete set of registers, cache, functional and execution units. Each core of a CPU can perform operations separately from the others

Threads
One or more hardware contexts within a single core. Each of the thread under a core is worked as a logical CPU running independently

Figure 2.2: Source: Original figure from LUMI consortium documentation (LUMI consortium 2023)

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

2.2.1 Serial vs. Parallel



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

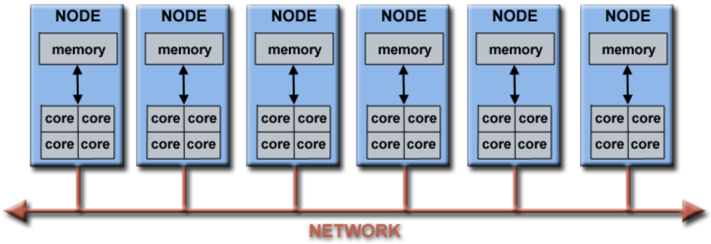


Figure 2.3: source: Blaise Barney, **Introduction to Parallel Computing**, Lawrence Livermore National Laboratory

2.3 High-performance computing in R

2.3.1 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)

2.4 GPU vs. CPU

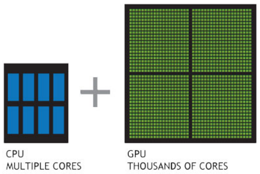


Figure 2.4: NVIDIA Blog

- 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).[2]

[2] 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.

2.5 When is it a good idea?

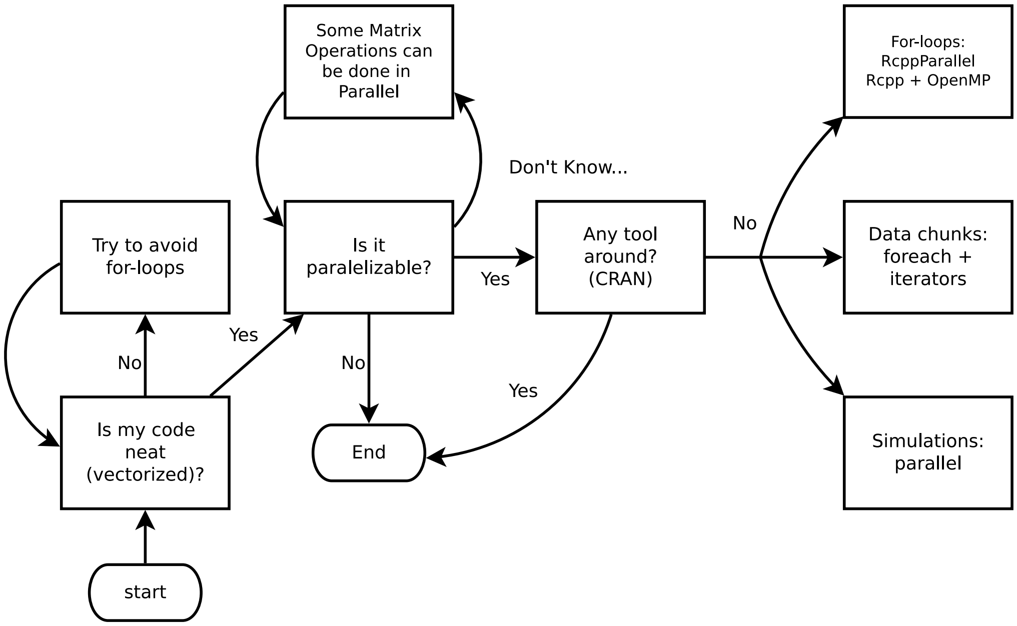

Some Matrix Operations can be done in Parallel
For-loops: RcppParallel Rcpp + OpenMP
Don't Know...
Try to avoid for-loops
Is it paralelizable?
Any tool around? (CRAN)
No
Data chunks: foreach + iterators
Yes
Yes
No
No
Yes
Is my code neat (vectorized)?
End
Simulations: parallel
start

Figure 2.5: Ask yourself these questions before jumping into HPC!

2.6 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.

3 The parallel R package

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.

3.1 Parallel workflow

(Usually) We do the following:

1. Create a `PSOCK/FORK` (or other) cluster using `makePSOCKCluster/makeForkCluster` (or `makeCluster`)
2. Copy/prepare each R session (if you are using a `PSOCK` cluster):

    a. Copy objects with `clusterExport`

    b. Pass expressions with `clusterEvalQ`

    c. Set a seed

3. Do your call: `parApply`, `parLapply`, etc.
4. Stop the cluster with `clusterStop`

3.2 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!

3.3 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 `makeCluster`: passed to **snow** (Simple Network of Workstations)

3.4 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.)

```r
# 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
```

3.5 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...

3.6 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,]  0.085384184 0.002390790  0.006576204 -0.003998278
[2,]  0.002390790 0.081142190  0.001846963  0.001476244
[3,]  0.006576204 0.001846963  0.085175347 -0.002807348
[4,] -0.003998278 0.001476244 -0.002807348  0.082425477
```

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

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

Part II

Working with a Cluster

4 What is Slurm

ℹ 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.

4.1 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.

5 A brief intro to Slurm

For a quick-n-dirty intro to Slurm (Yoo, Jette, and Grondona 2003), 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`.

5.1 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.

5.2 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
   ```

5.3 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.

6 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:

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

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

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

6.1 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:

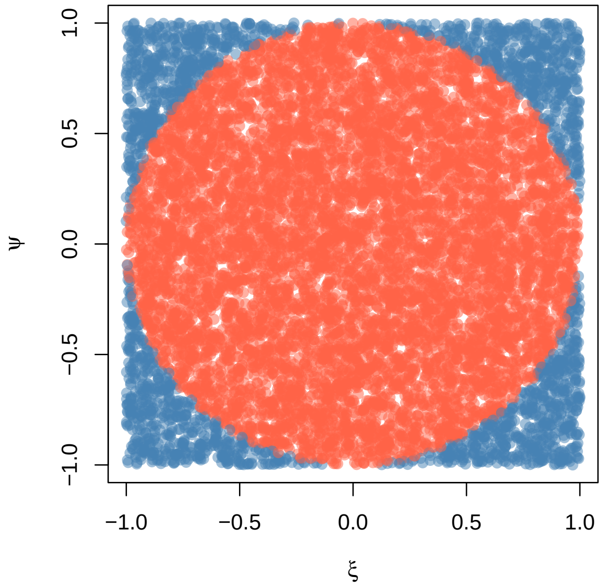


Figure 6.1: 10,000 random points drawn within the unit circle.

- **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.

6.1.1 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
```

6.1.2 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.[1] 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):

[1] 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
```

6.2 Jobs with the slurmR package

The `slurmR` R package (Vega Yon and Marjoram 2019, 2022) 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.

6.2.1 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.[2] 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,[3] 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

# 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)
```

[2] See installation instructions **here**

[3] 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.

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

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

Bashfile (03-parsapply-slurmr.slurm):

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

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

6.2.2 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

```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):

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

```r
# 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):

```sh
#!/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])`

6.2.3 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

Part III

Using C++

7 Rcpp

When parallel computing is not enough, you can boost your R code using a lower-level programming language[1] 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 (Eddelbuettel and François 2011; Eddelbuettel 2013; Eddelbuettel and Balamuta 2018) 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.

7.1 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!

[1] 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.

7.2 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

7.3 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++

7.4 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));
```

7.5 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
```

7.6 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 cld
   add1R(1:1000) 33.371 35.0785 50.97206 35.759 36.495 1494.941   100  a 
 add1Cpp(1:1000)  2.238  2.6305 10.57369  3.029  5.284  663.015   100   b
```

7.7 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).

7.8 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`

7.9 Parts of "an Rcpp program"

```
1 #include<Rcpp.h>
2 using namespace Rcpp
3 // [[Rcpp::export]]
4 NumericVector add1(NumericVector x) {
5   NumericVector ans(x.size());
6   for (int i = 0; i < x.size(); ++i)
7     ans[i] = x[i] + 1;
8   return ans;
9 }
```

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.

7.10 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.

7.11 Your turn

7.11.1 Problem 1: Adding vectors

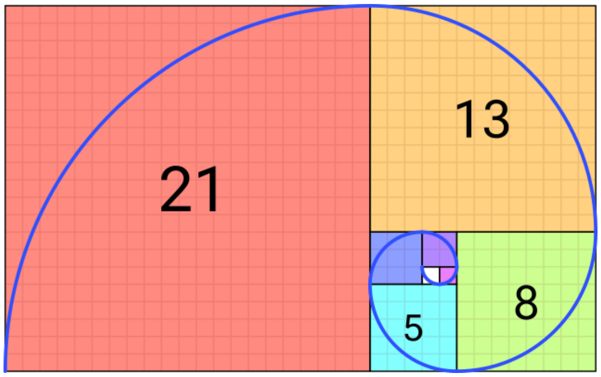
1. Using what you have just learned about Rcpp, write a function to add two vectors of the same length. Use the following template

```
#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 :)");
```

7.11.2 Problem 2: Fibonacci series



Each element of the sequence is determined by the following:

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

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!

7.11.3 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) 5351.777 5802.3785 6513.57586 5943.7375 6295.514 13654.174   100
 fibCpp(20)   11.842   12.7245   28.71172   19.2465   27.513   806.393   100
 cld
  a
   b
```

7.12 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
  ```

7.12.1 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!

7.12.2 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      max
                dist(x) 2223.023 2223.023 2223.023 2223.023 2223.023 2223.023
 dist_par(x, cores = 1) 2414.402 2414.402 2414.402 2414.402 2414.402 2414.402
 dist_par(x, cores = 2) 1865.621 1865.621 1865.621 1865.621 1865.621 1865.621
 dist_par(x, cores = 4) 1223.261 1223.261 1223.261 1223.261 1223.261 1223.261
 neval
     1
     1
     1
     1
```

7.12.3 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

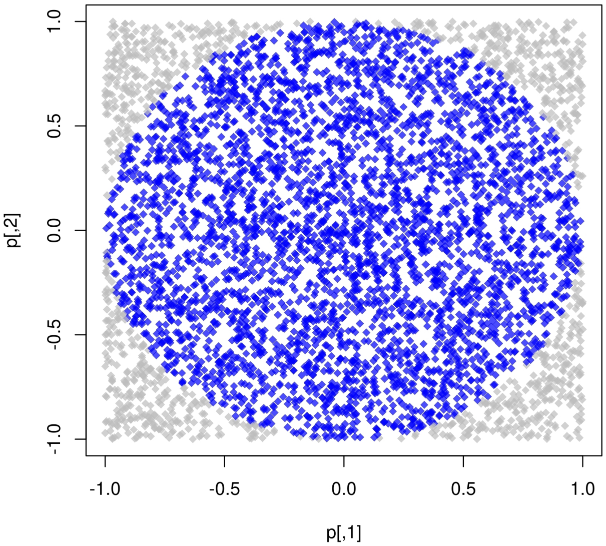
```
library(future)
plan(multicore)
```

```
# 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
##   0.023   0.008   0.030
# 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
##   0.003   0.000   0.003
```

7.12.4 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 268.7455 268.7455 268.7455 268.7455 268.7455 268.7455     1
   serial 329.4686 329.4686 329.4686 329.4686 329.4686 329.4686     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 
```

7.13 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

8 Debugging R w C++/C code

Although debugging R code is easy, the same doesn't apply to compiled code in R[1]. This chapter shows a few ways to debug your R + C++ code. You will need the GNU Debugger (GDB) and Valgrind.

8.1 Starting R with a debugger

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.
```

[1] 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.

```
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.

>
```

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.

```cpp
#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")'
```

```
==51926== Memcheck, a memory error detector
==51926== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==51926== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==51926== Command: /usr/lib/R/bin/exec/R -e Rcpp::sourceCpp("rcpp-debugging-faulty.cpp")
==51926==
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.

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

> faulty_program(10)
 [1] 0 0 0 0 0 0 0 0 0 0
>
>
==51926==
==51926== HEAP SUMMARY:
==51926==     in use at exit: 55,450,057 bytes in 10,732 blocks
==51926==   total heap usage: 31,980 allocs, 21,248 frees, 95,390,528 bytes allocated
==51926==
==51926== LEAK SUMMARY:
==51926==    definitely lost: 24 bytes in 1 blocks
==51926==    indirectly lost: 0 bytes in 0 blocks
==51926==      possibly lost: 0 bytes in 0 blocks
==51926==    still reachable: 55,450,033 bytes in 10,731 blocks
==51926==                       of which reachable via heuristic:
==51926==                         newarray           : 4,264 bytes in 1 blocks
==51926==         suppressed: 0 bytes in 0 blocks
==51926== Rerun with --leak-check=full to see details of leaked memory
==51926==
==51926== For lists of detected and suppressed errors, rerun with: -s
==51926== 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")'
```

```
==50468== HEAP SUMMARY:
==50468==     in use at exit: 55,450,707 bytes in 10,731 blocks
==50468==   total heap usage: 32,017 allocs, 21,286 frees, 95,411,007 bytes allocated
==50468==
==50468== LEAK SUMMARY:
==50468==    definitely lost: 0 bytes in 0 blocks
==50468==    indirectly lost: 0 bytes in 0 blocks
==50468==      possibly lost: 0 bytes in 0 blocks
==50468==    still reachable: 55,450,707 bytes in 10,731 blocks
==50468==                       of which reachable via heuristic:
==50468==                         newarray           : 4,264 bytes in 1 blocks
==50468==         suppressed: 0 bytes in 0 blocks
==50468== Rerun with --leak-check=full to see details of leaked memory
==50468==
==50468== For lists of detected and suppressed errors, rerun with: -s
==50468== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

No more memory leaks.

8.2 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:

```cpp
#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)
*/
```

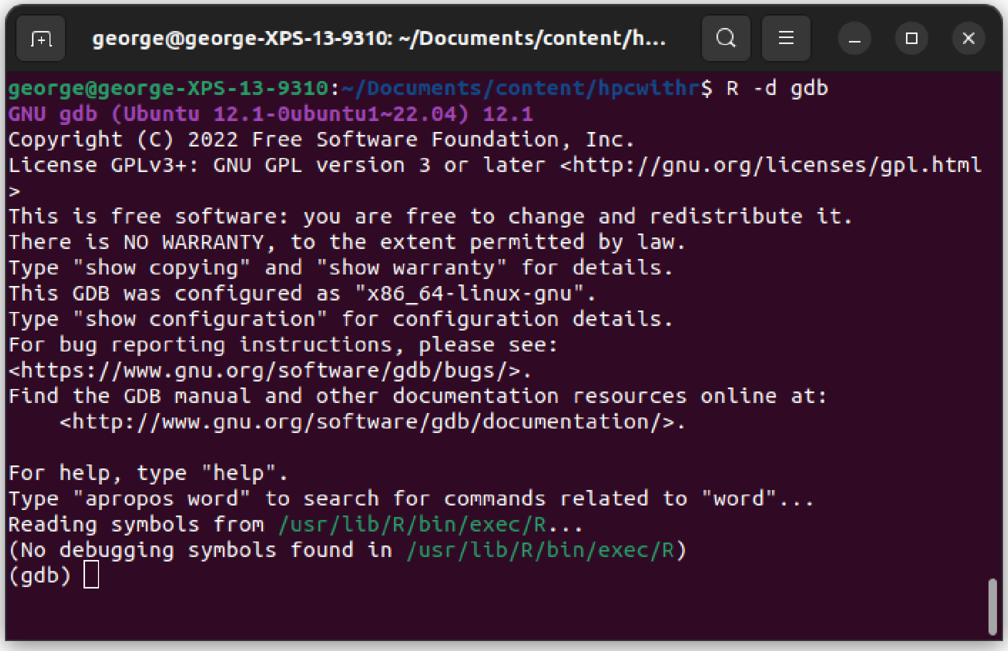
In it, we try to access a location in the memory that hasn't been allocated yet, namely, a `NumericVector` declared as a pointer but never assigned. Using `R --debugger=valgrind` generates the following code:

```
==54716== Memcheck, a memory error detector
==54716== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==54716== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==54716== Command: /usr/lib/R/bin/exec/R -e Rcpp::sourceCpp("rcpp-debugging-not-mapped.cpp
==54716==

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.
```

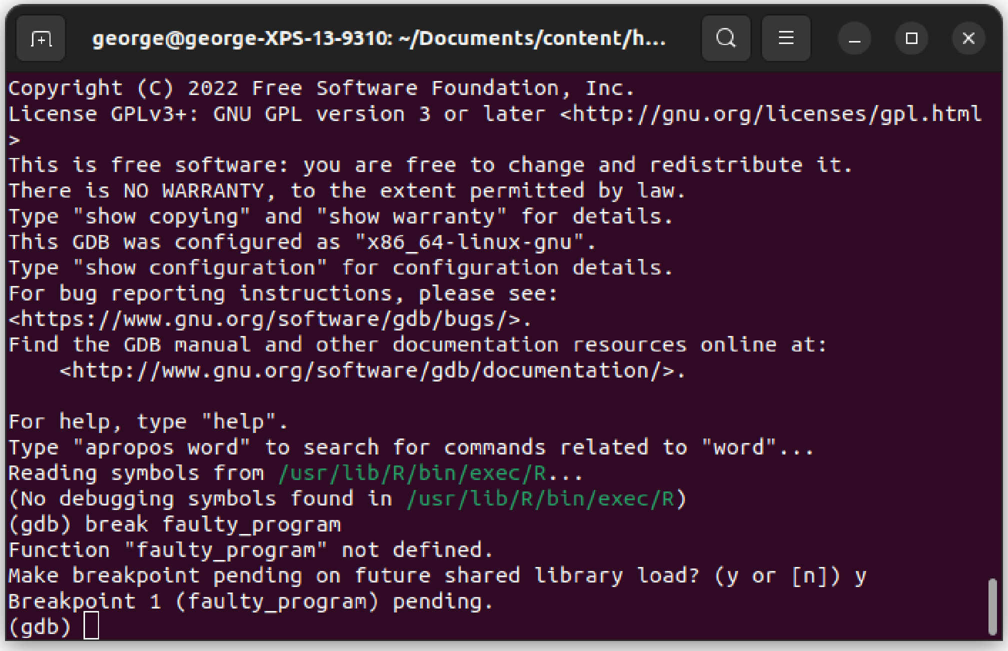
```
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  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.

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

> faulty_program(10)
==54716== Invalid read of size 8
==54716==    at 0x1222229A: get__ (PreserveStorage.h:52)
==54716==    by 0x1222229A: copy__<Rcpp::Vector<14, Rcpp::PreserveStorage> > (PreserveStor
==54716==    by 0x1222229A: Vector (Vector.h:64)
==54716==    by 0x1222229A: faulty_program(int) (rcpp-debugging-not-mapped.cpp:11)
==54716==    by 0x12222468: sourceCpp_1_faulty_program (rcpp-debugging-not-mapped.cpp:33)
==54716==    by 0x495391D: ??? (in /usr/lib/R/lib/libR.so)
==54716==    by 0x4953E9C: ??? (in /usr/lib/R/lib/libR.so)
==54716==    by 0x49ABD77: Rf_eval (in /usr/lib/R/lib/libR.so)
==54716==    by 0x49AD2AE: ??? (in /usr/lib/R/lib/libR.so)
==54716==    by 0x49AE0F4: Rf_applyClosure (in /usr/lib/R/lib/libR.so)
==54716==    by 0x49AB84B: Rf_eval (in /usr/lib/R/lib/libR.so)
==54716==    by 0x49B120B: ??? (in /usr/lib/R/lib/libR.so)
==54716==    by 0x498E694: ??? (in /usr/lib/R/lib/libR.so)
==54716==    by 0x49AB71F: Rf_eval (in /usr/lib/R/lib/libR.so)
==54716==    by 0x49AD2AE: ??? (in /usr/lib/R/lib/libR.so)
==54716==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==54716==

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

Traceback:
 1: .Call(<pointer: 0x122223f0>, 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 ...
==54716==
==54716== Process terminating with default action of signal 11 (SIGSEGV)
==54716==    at 0x4DB4A7C: __pthread_kill_implementation (pthread_kill.c:44)
==54716==    by 0x4DB4A7C: __pthread_kill_internal (pthread_kill.c:78)
==54716==    by 0x4DB4A7C: pthread_kill@@GLIBC_2.34 (pthread_kill.c:89)
==54716==    by 0x4D60475: raise (raise.c:26)
==54716==    by 0x4D6051F: ??? (in /usr/lib/x86_64-linux-gnu/libc.so.6)
==54716==    by 0x12222299: copy__<Rcpp::Vector<14, Rcpp::PreserveStorage> > (PreserveStor
==54716==    by 0x12222299: Vector (Vector.h:64)
==54716==    by 0x12222299: faulty_program(int) (rcpp-debugging-not-mapped.cpp:11)
==54716==
==54716== HEAP SUMMARY:
==54716==     in use at exit: 55,590,287 bytes in 10,969 blocks
==54716==   total heap usage: 31,943 allocs, 20,974 frees, 95,324,688 bytes allocated
==54716==
==54716== LEAK SUMMARY:
==54716==    definitely lost: 0 bytes in 0 blocks
==54716==    indirectly lost: 0 bytes in 0 blocks
==54716==      possibly lost: 5,007 bytes in 15 blocks
==54716==    still reachable: 55,585,280 bytes in 10,954 blocks
==54716==                       of which reachable via heuristic:
==54716==                         newarray           : 4,264 bytes in 1 blocks
==54716==         suppressed: 0 bytes in 0 blocks
==54716== Rerun with --leak-check=full to see details of leaked memory
==54716==
==54716== For lists of detected and suppressed errors, rerun with: -s
==54716== 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.

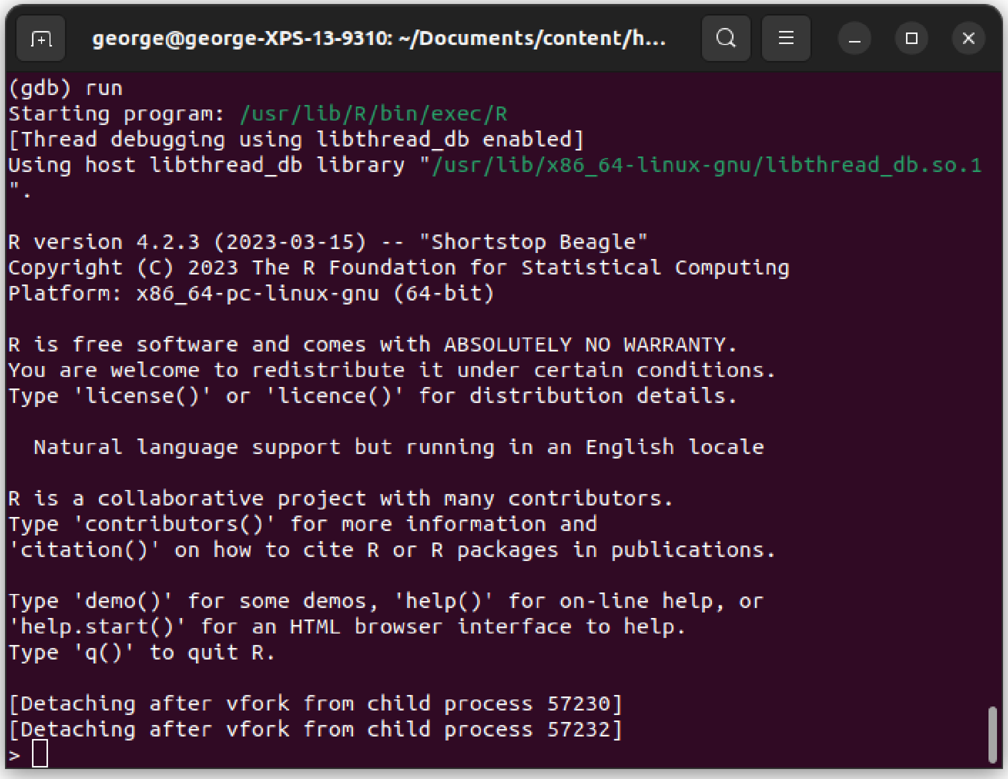


```
george@george-XPS-13-9310:~/Documents/content/hpcwithr$ R -d gdb
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html
>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /usr/lib/R/bin/exec/R...
(No debugging symbols found in /usr/lib/R/bin/exec/R)
(gdb) 
```



```
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html
>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /usr/lib/R/bin/exec/R...
(No debugging symbols found in /usr/lib/R/bin/exec/R)
(gdb) break faulty_program
Function "faulty_program" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (faulty_program) pending.
(gdb) 
```

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



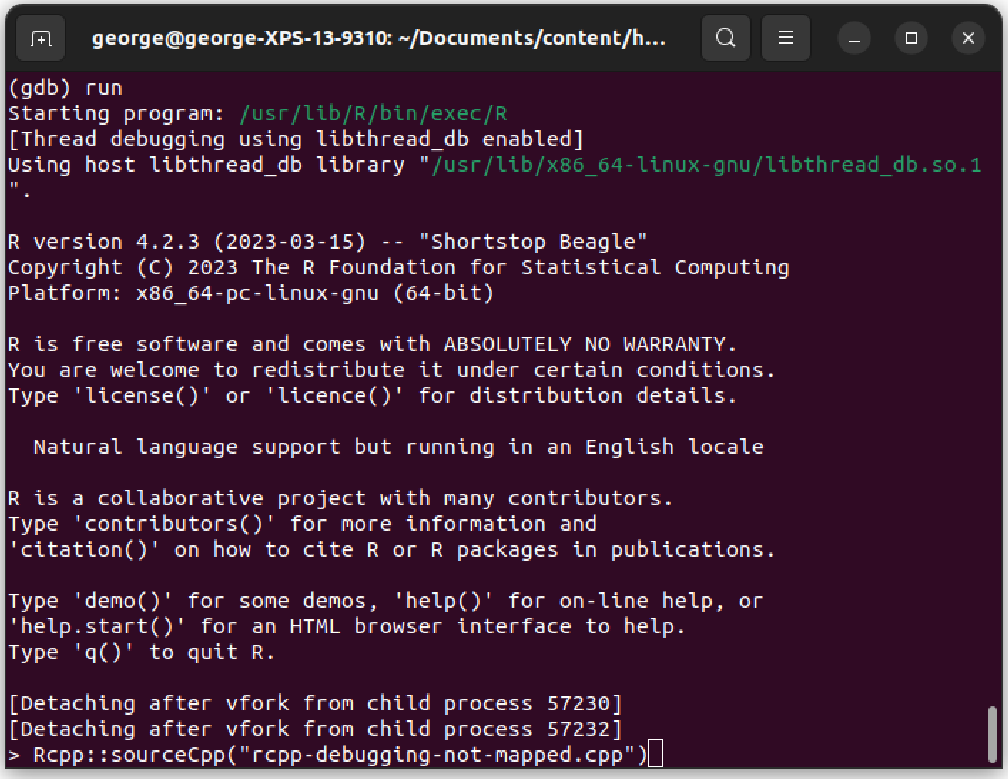
```
(gdb) run
Starting program: /usr/lib/R/bin/exec/R
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/x86_64-linux-gnu/libthread_db.so.1
".

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.

[Detaching after vfork from child process 57230]
[Detaching after vfork from child process 57232]
> 
```

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

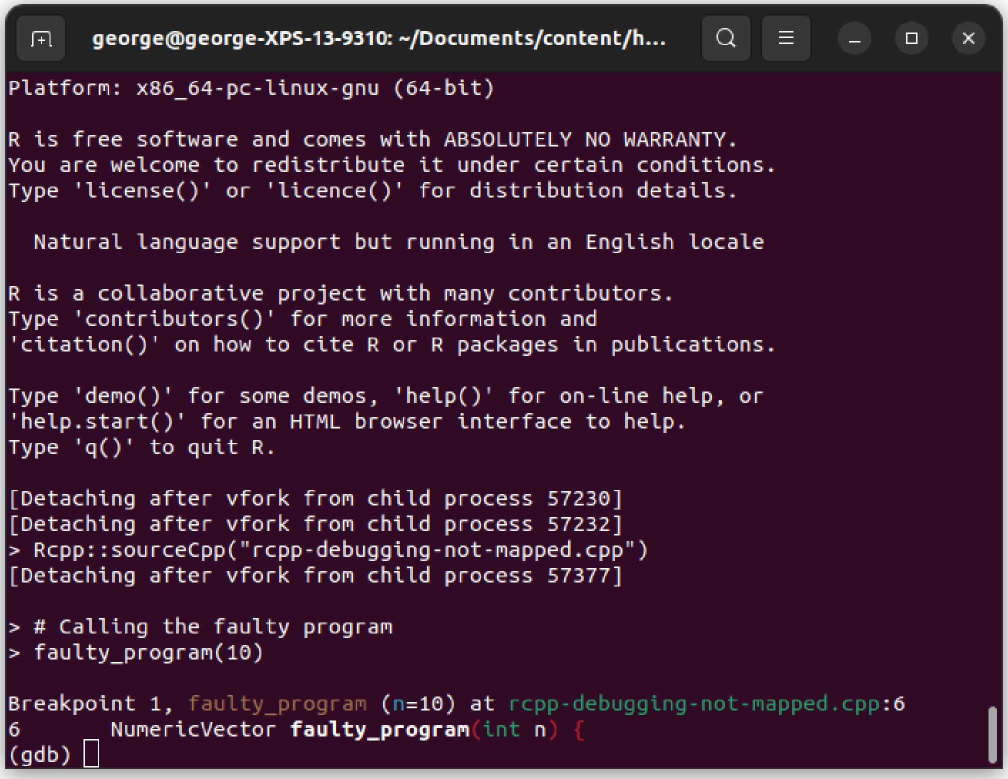
5. Once the program has paused, we can inspect the context.

   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
```

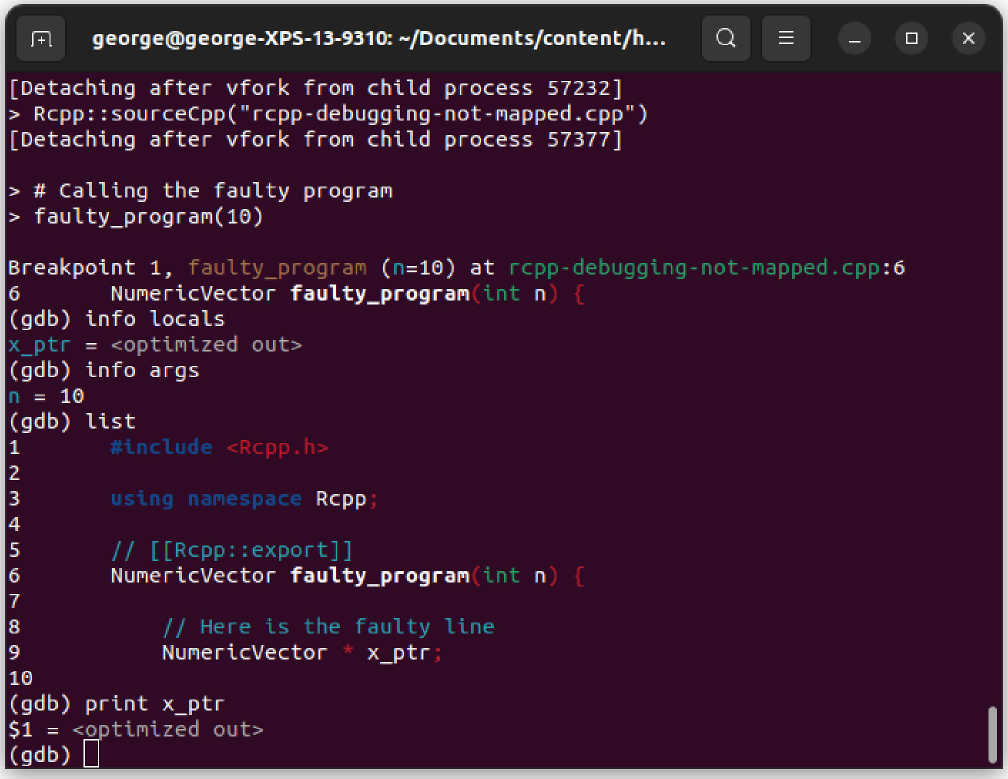


```
(gdb) run
Starting program: /usr/lib/R/bin/exec/R
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/x86_64-linux-gnu/libthread_db.so.1
".

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.

[Detaching after vfork from child process 57230]
[Detaching after vfork from child process 57232]
> Rcpp::sourceCpp("rcpp-debugging-not-mapped.cpp")
```



```
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.

[Detaching after vfork from child process 57230]
[Detaching after vfork from child process 57232]
> Rcpp::sourceCpp("rcpp-debugging-not-mapped.cpp")
[Detaching after vfork from child process 57377]

> # Calling the faulty program
> faulty_program(10)

Breakpoint 1, faulty_program (n=10) at rcpp-debugging-not-mapped.cpp:6
6         NumericVector faulty_program(int n) {
(gdb) 
```

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



```
[Detaching after vfork from child process 57232]
> Rcpp::sourceCpp("rcpp-debugging-not-mapped.cpp")
[Detaching after vfork from child process 57377]

> # Calling the faulty program
> faulty_program(10)

Breakpoint 1, faulty_program (n=10) at rcpp-debugging-not-mapped.cpp:6
6         NumericVector faulty_program(int n) {
(gdb) info locals
x_ptr = <optimized out>
(gdb) info args
n = 10
(gdb) list
1         #include <Rcpp.h>
2
3         using namespace Rcpp;
4
5         // [[Rcpp::export]]
6         NumericVector faulty_program(int n) {
7
8             // Here is the faulty line
9             NumericVector * x_ptr;
10
(gdb) print x_ptr
$1 = <optimized out>
(gdb) 
```

9 Misc

9.1 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

9.2 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).

9.3 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

9.4 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")`.

9.5 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.

9.6 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

Dowle, Matt, and Arun Srinivasan. 2021. *Data.table: Extension of 'Data.frame'*. https://CRAN.R-project.org/package=data.table.

Eddelbuettel, Dirk. 2013. *Seamless R and C++ Integration with Rcpp*. New York: Springer. https://doi.org/10.1007/978-1-4614-6868-4.

Eddelbuettel, Dirk, and James Joseph Balamuta. 2018. "Extending extitR with extitC++: A Brief Introduction to extitRcpp." *The American Statistician* 72 (1): 28–36. https://doi.org/10.1080/00031305.2017.1375990.

Eddelbuettel, Dirk, and Romain François. 2011. "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software* 40 (8): 1–18. https://doi.org/10.18637/jss.v040.i08.

LUMI consortium. 2023. "Documentation - Distribution and Binding." https://docs.lumi-supercomputer.eu/runjobs/scheduled-jobs/distribution-binding/.

R Core Team. 2023. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

Vega Yon, George, and Paul Marjoram. 2019. "slurmR: A Lightweight Wrapper for HPC with Slurm." *The Journal of Open Source Software* 4 (39). https://doi.org/10.21105/joss.01493.

———. 2022. *slurmR: A Lightweight Wrapper for 'Slurm'*. https://github.com/USCbiostats/slurmR.

Yoo, Andy B., Morris A. Jette, and Mark Grondona. 2003. "SLURM: Simple Linux Utility for Resource Management." In *Job Scheduling Strategies for Parallel Processing*, edited by Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, 2862:44–60. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/10968987_3.