Applied HPC with R

George G. Vega Yon

2023-04-09

Table of contents

Preface

The R programming language (R Core Team 2023) can be amazing 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, then this book is for you. Applied HPC with R is a collection of talks and lectures I have given in the past about speeding up your R code using parallel computing and other resources such as C++. The contents have been mostly developed during my time at USC and UofU[1]

[1]With many to thank, including Paul Marjoram, Zhi Zhang, 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 some powerful features that accelerate your code dramatically[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.

1.1 High-Performance Computing: An overview

Loosely, from R's perspective, we can think of HPC in terms of two, maybe three things:

1. Big data: How to work with data that doesn't fit your computer
2. Parallel computing: How to take advantage of multiple core systems
3. Compiled code: Write your low-level code (if R doesn't have it yet…)

(Checkout CRAN Task View on HPC)

1.2 Big Data

- Buy a bigger computer/RAM (not the best solution!)
- Use out-of-memory storage, i.e., don't load all your data in the RAM. e.g. The bigmemory, data.table, HadoopStreaming R packages
- Efficient algorithms for big data, e.g.: biglm, biglasso
- Store it more efficiently, e.g.: Sparse Matrices (take a look at the `dgCMatrix` objects from the Matrix R package)

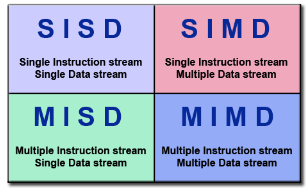


Figure 1.1: Flynn's Classical Taxonomy (Blaise Barney, **Introduction to Parallel Computing**, Lawrence Livermore National Laboratory)

1.3 Parallel computing

We will be focusing on the **S**ingle **I**nstruction stream **M**ultiple **D**ata stream

1.4 Parallel computing

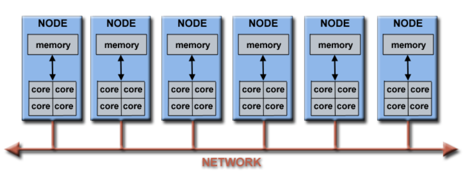
In general terms, a parallel computing program is one in which we use two or more *computational threads* simultaneously. Although computational thread usually means core, there are multiple levels at which a computer program can be parallelized. To understand this, we first need to see what composes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

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

1.4.1 Serial vs Parallel

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

1.5 Parallel computing



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

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

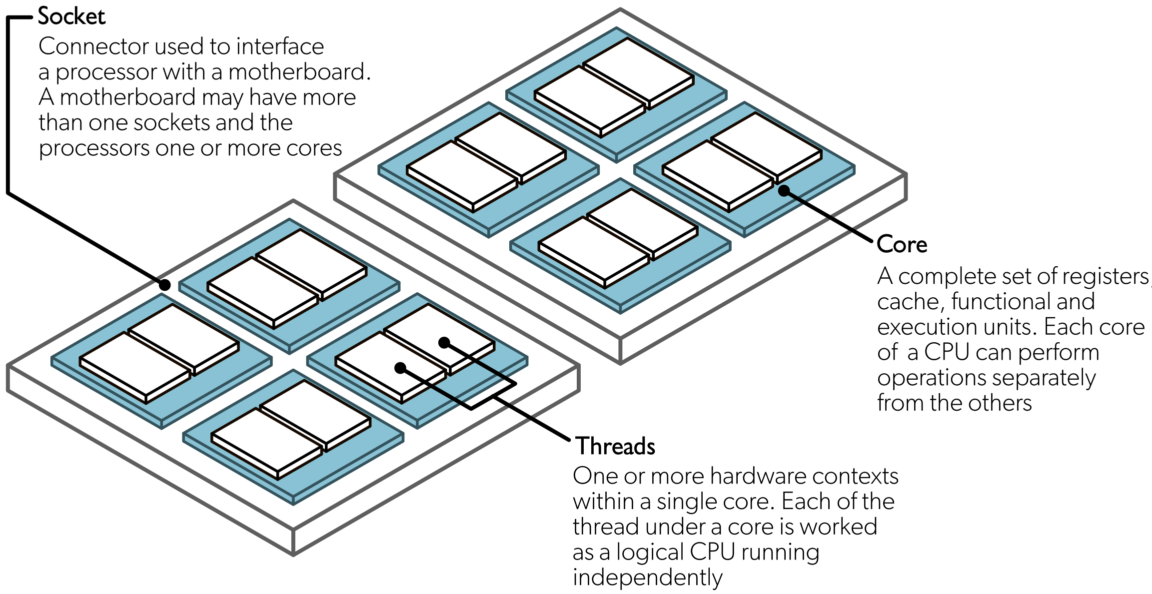


Figure 1.2: Source: Original figure from LUMI consortium documentation (LUMI consortium 2023)

1.6 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)

1.7 GPU vs CPU

- 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).

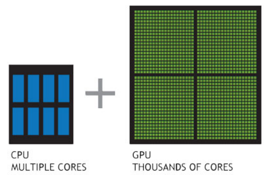


Figure 1.3: NVIDIA Blog

1.8 When is it a good idea?

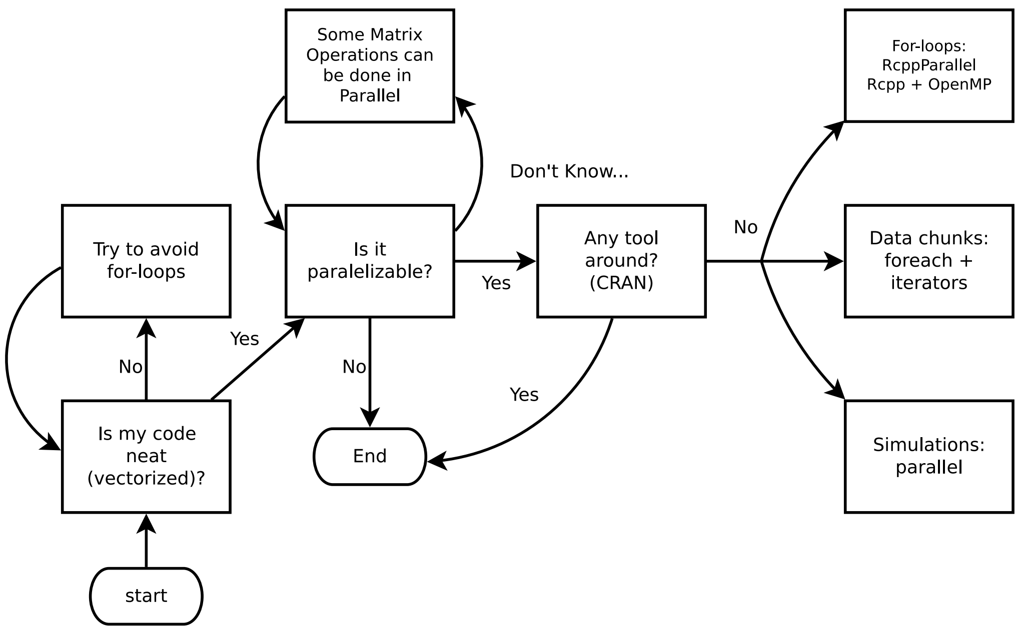


Figure 1.4: Ask yourself these questions before jumping into HPC!

1.9 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.

2 The parallel R package

2.1 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`

2.2 Types of clusters: PSOCK

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

```r
# 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 of the current computer, i.e. across multiple computers!

2.3 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 it is altered (need to double check when this happens!)
- Not available on Windows.

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

2.4 Ex 1: Parallel RNG with `makePSOCKCluster`

```
# 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 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
```

```r
stopCluster(cl)
all.equal(ans0, ans1) # All equal!
```

```
[1] TRUE
```

2.5 Ex 2: Parallel RNG with `makeForkCluster`

In the case of `makeForkCluster`

```r
# 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

```r
# 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 simpler way of doing this...

2.6 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

```r
# 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
```

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

Part II

Working with a Cluster

3 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`.

3.1 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.

3.2 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
   ```

3.3 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!

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.

4 Simulating pi (once more)

This is the same old example that lots of people (including me) have been using to illustrate parallel computing with R. The example is very simple, 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 say $\pi = A/r^2$, 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

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

4.1 Submitting jobs to Slurm

The main way that we will be working is by submitting jobs using the `sbatch` function. This function takes as a 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:

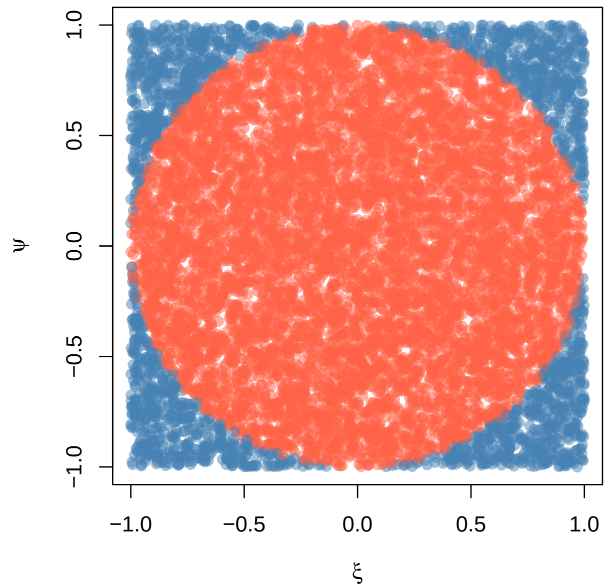


Figure 4.1: 10,000 random points drawn within the unit circle.

- The Slurm flags `#SBATCH`.
- Loading R `module load R/4.2.2`.
- Executing the R script.

Submission is then done as follows:

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

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

4.1.1 Case 1: Single job, single core job

The most basic way is submitting a job using the `sbatch` command. In this case, you need to have 2 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
```

```r
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:

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

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

4.1.2 Case 2: Single job, multicore job

Now, imagine that we would like to use more than one processor for this job, using something like 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):

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

[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.

```r
# 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):

```sh
#!/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
```

4.2 Jobs with the slurmR package

The `slurmR` R package (Vega Yon and Marjoram 2019, 2022) is a lightweight wrapper of Slurm. The package's main functions are the `*apply` family–mostly 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.

4.2.1 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 (see installation instructions **here**)

Once you have the `slurmR` package in your system, you can proceed as follow

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)

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
```

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

Another way to submit jobs is using **job arrays**. A job array is essentially a job that is 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

}
```

```r
# 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):

```bash
#!/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])`

4.2.3 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 and used to submit the R script to the queue.

Part III

Using C++

5 Rcpp

6 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 straight forward… 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
```

6.1 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);
```

6.2 RcppArmadillo and OpenMP workflow

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!

6.3 Ex 5: RcppArmadillo + OpenMP

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

```cpp
#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
```

```r
  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
```

6.4 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

6.5 Ex 6: The future (cont'd)

```r
library(future)
plan(multicore)
# We are creating a global variable
a <- 2
```

```r
# 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
```

6.6 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, checkout the CRAN Task View on HPC

6.7 Session info

```
R version 4.2.3 (2023-03-15)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 22.04.2 LTS

Matrix products: default
BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.10.0
LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.10.0
```

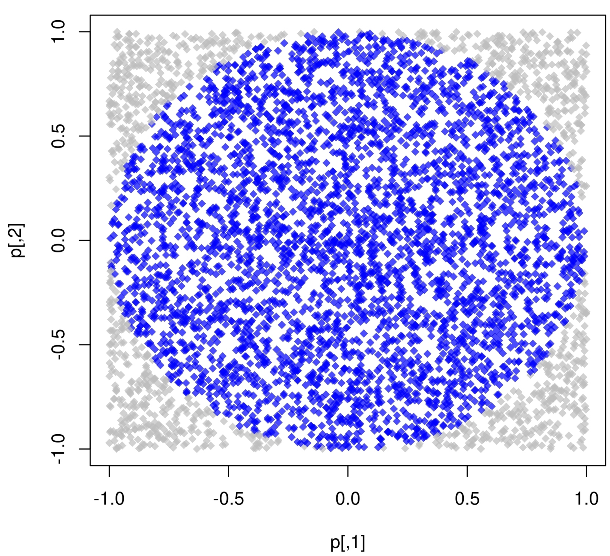
```
locale:
 [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
 [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
 [9] LC_ADDRESS=C               LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
 [1] compiler_4.2.3  magrittr_2.0.2  fastmap_1.1.0   cli_3.6.1
 [5] tools_4.2.3     htmltools_0.5.4 yaml_2.3.5      Rcpp_1.0.10
 [9] stringi_1.7.6   rmarkdown_2.20  knitr_1.37      stringr_1.4.0
[13] jsonlite_1.7.3  xfun_0.37       digest_0.6.29   rlang_1.1.0
[17] evaluate_0.15
```

6.8 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

```r
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
}
```

6.9 Bonus track 1: Simulating π (cont'd)

```r
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
```

```r
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 Misc

7.1 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

7.2 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).

7.3 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

7.4 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")`.

7.5 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.

7.6 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.

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.