PARALLEL DATA ANALYSIS WITH R

prof. dr. Janez Povh
dr. Tomas Martinović (Rcpp)
EuroHPC Competence Center, November 18, 2026

Outline/next

- Introduction to HPC
- Introduction to R
- Parallel R within one node
- Parallelization with Rmpi

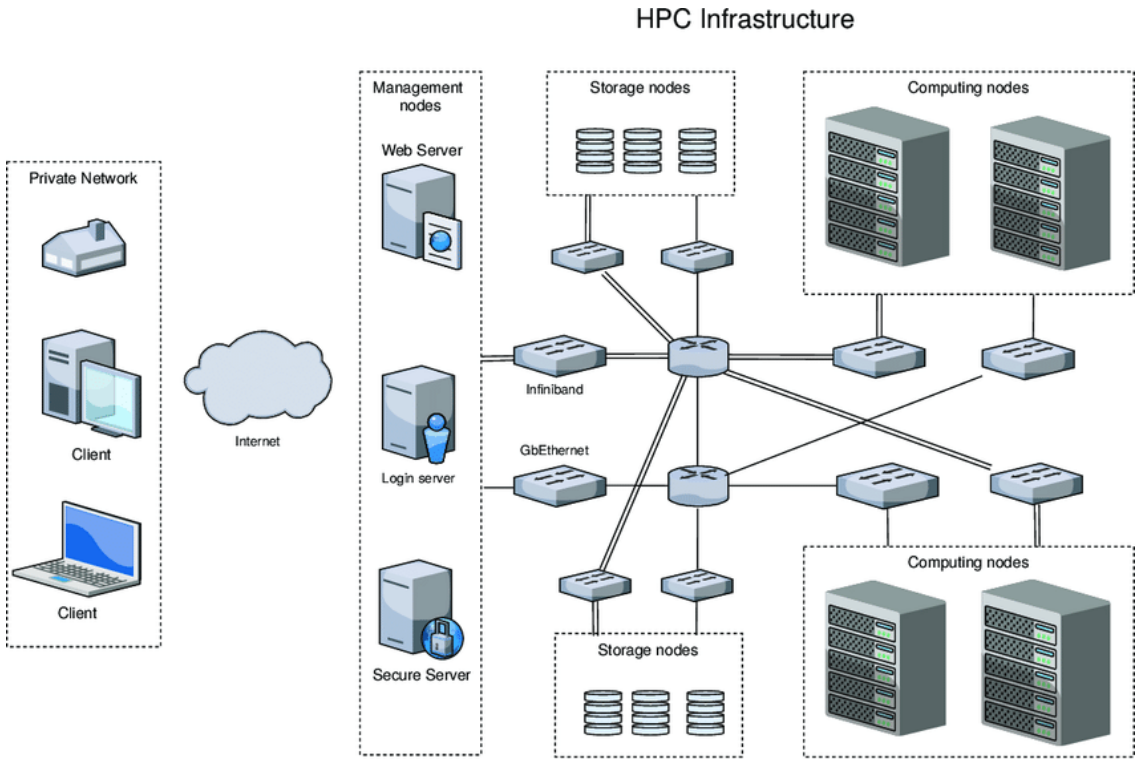
Introduction to HPC

What is HPC



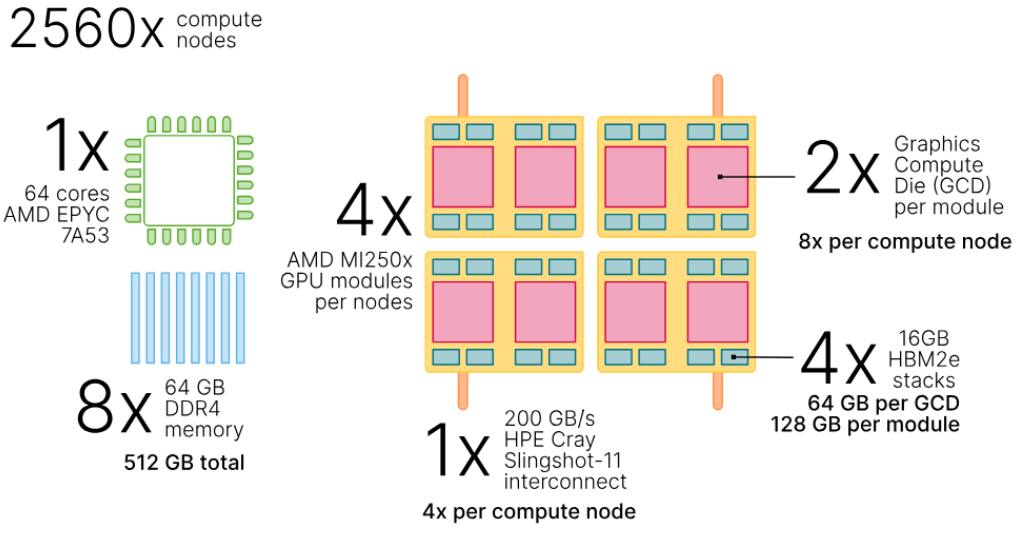
Source: https://www.vyzkumne-infrastruktury.cz/en/2022/06/lumi-supercomputer-has-been-inaugurated/

What is HPC



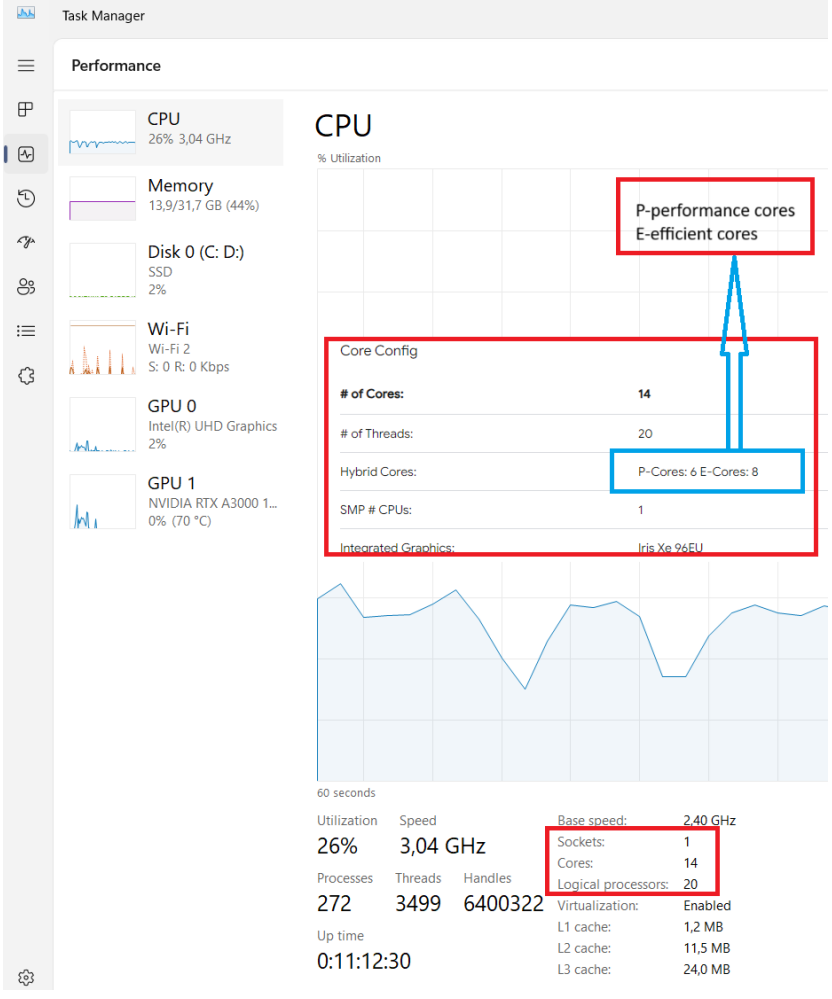
Source: Reghenzani, F. et al, IEEE Access, 8, 208566-208582.

Node structure

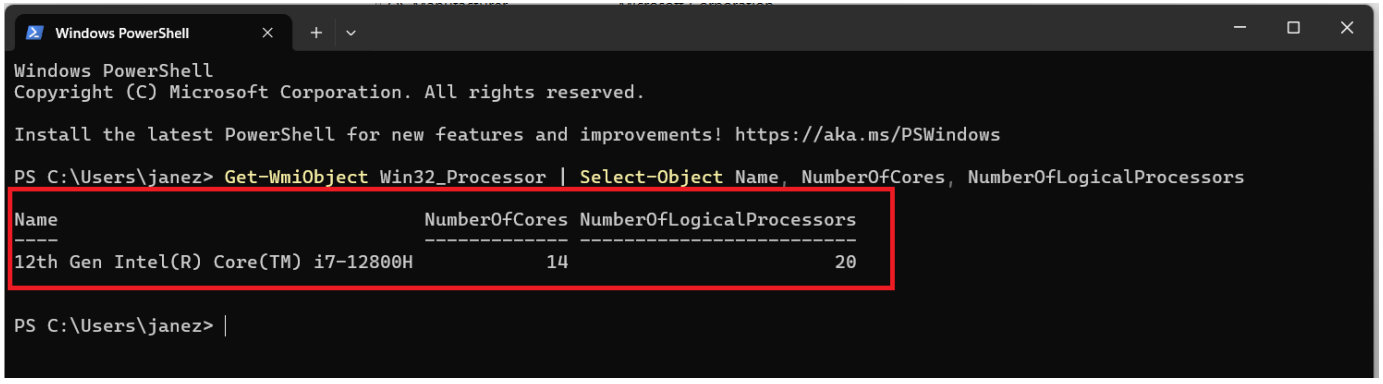


Source: https://docs.lumi-supercomputer.eu/hardware/lumig/

How does my computer look like



How does my computer looks like



Why HPC

Some computations are (very) extensive:

- cpu-extensive: take too much cpu time

Why HPC

Some computations are (very) extensive:

- cpu-extensive: take too much cpu time
- memory-extensive: Take too much memory

Why HPC

Some computations are (very) extensive:

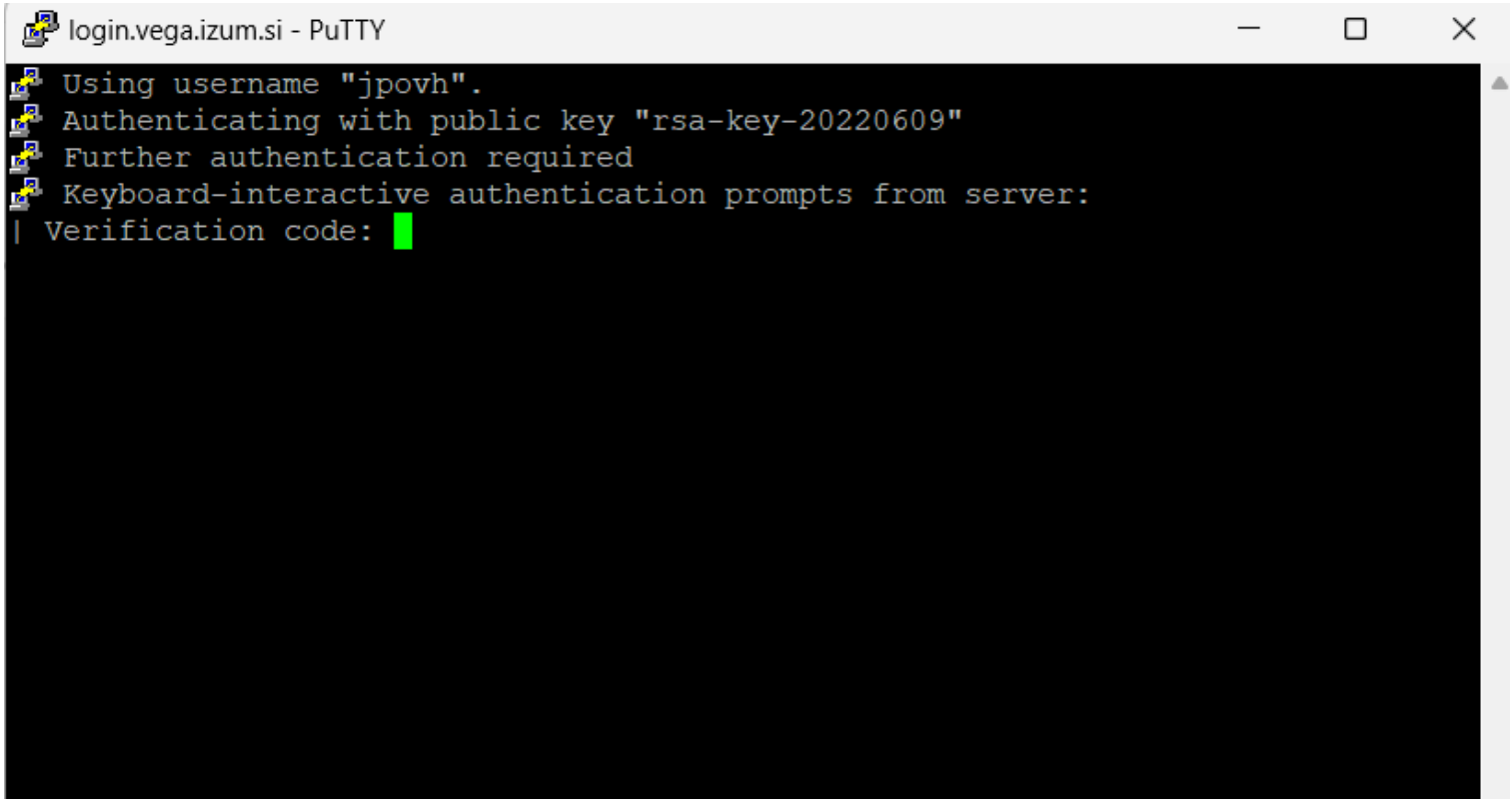
- cpu-extensive: take too much cpu time
- memory-extensive: Take too much memory
- I/O-extensive: Take too much time to read/write from disk

Why HPC

Some computations are (very) extensive:

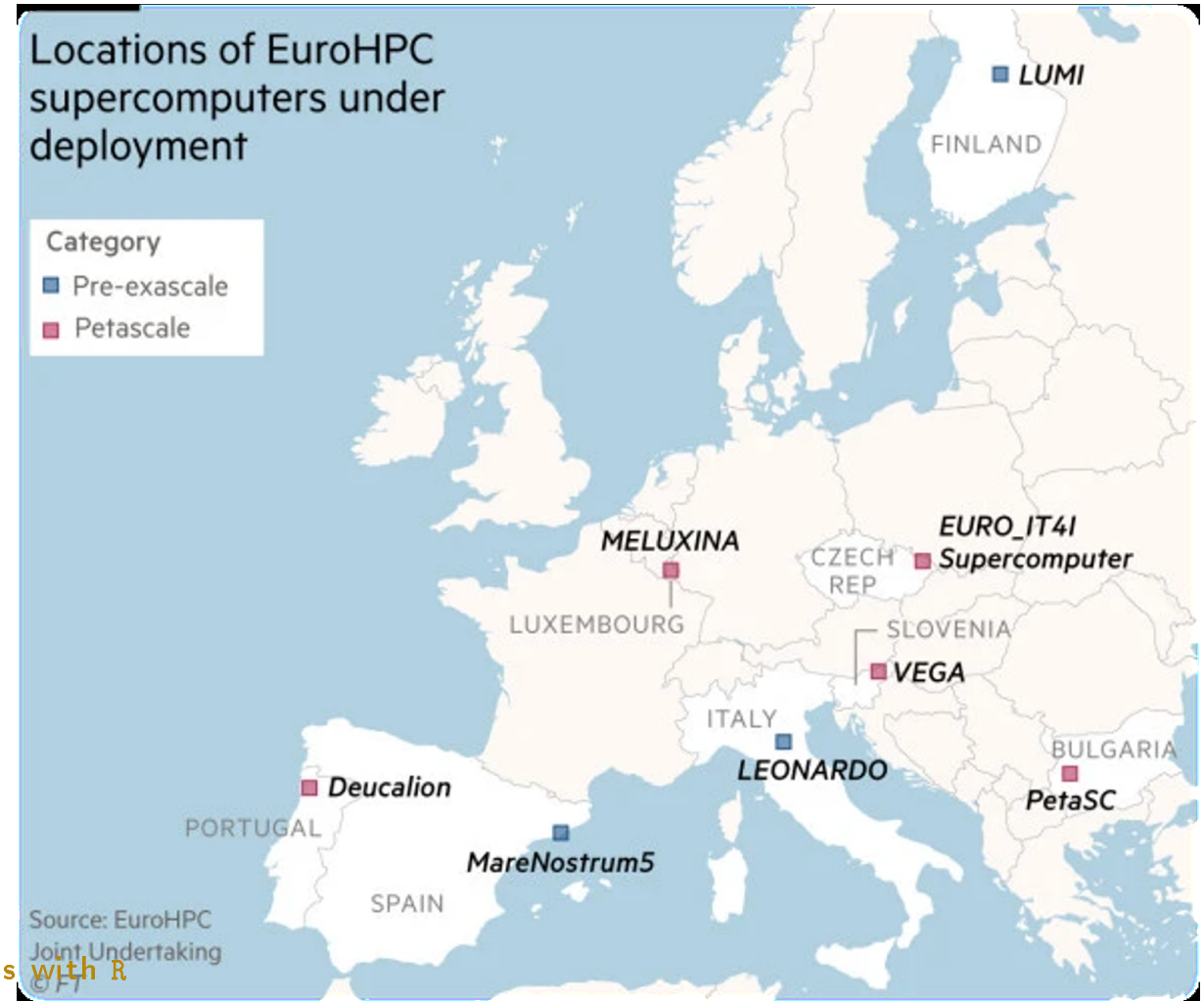
- cpu-extensive: take too much cpu time
- memory-extensive: Take too much memory
- I/O-extensive: Take too much time to read/write from disk
- network-extensive: Take too much time to transfer over the network.

User experience



How to access HPC

HPCs are easily available.



EuroCC3

- Network of EU Competence centers for HPC.

EuroCC3

- Network of EU Competence centers for HPC.
- Almost all members of EuroHPC JU involved.

EuroCC3

- Network of EU Competence centers for HPC.
- Almost all members of EuroHPC JU involved.
- Training, support for industry, talent attraction,...

Outline/next

- Introduction to HPC
- Introduction to R
- Parallel R within one node
- Parallelization with Rmpi

Introduction to R

What is R?



- A language and environment for statistical computing
- Data manipulation, analysis, modelling, and visualization
- Based on the S programming language
- Interpreted, interactive, and highly extensible
- Core components are written in R, C, and Fortran
- Free, open source, and cross-platform

Installing R



- Current release: R 4.6.2, 29 October 2026
- Download from CRAN: `https://cran.r-project.org`
- Available for Linux, macOS, and Windows
- Windows and macOS users can install precompiled binaries
- Linux users may also install R using their package manager

Pros and Cons

Pros:

- Free and Open Source
- Strong User Community
- Highly extensible, flexible
- Implementation of high-end statistical methods
- Flexible graphics and intelligent defaults

Pros and Cons

Pros:

- Free and Open Source
- Strong User Community
- Highly extensible, flexible
- Implementation of high-end statistical methods
- Flexible graphics and intelligent defaults

Cons

- Steep learning curve
- Slow for large datasets

Data types



- R Supports virtually any type of data
- Numbers, characters, logicals (TRUE/ FALSE)
- Arrays of virtually unlimited sizes
- Simplest: Vectors and Matrices
- Lists: Can Contain mixed type variables
- Data Frame: Rectangular Data Set
- Datasets in `tidy` format.

Data structures in R

Linear

- vectors (all same type)
- lists (mixed types)

Data structures in R

Linear

- vectors (all same type)
- lists (mixed types)

Rectangular

- data frame
- matrix

Running R



- I recommend RStudio, an IDE for R.

Running R

- I recommend RStudio, an IDE for R.
- It is available as RStudio Desktop and **RStudio Server**, which runs on a remote server and allows accessing RStudio using a web browser.

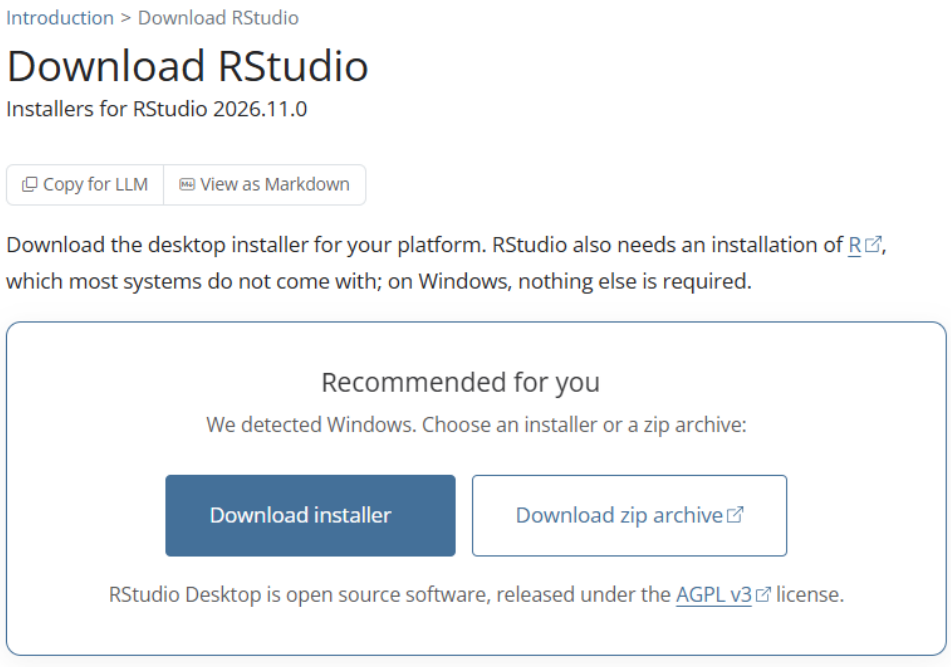
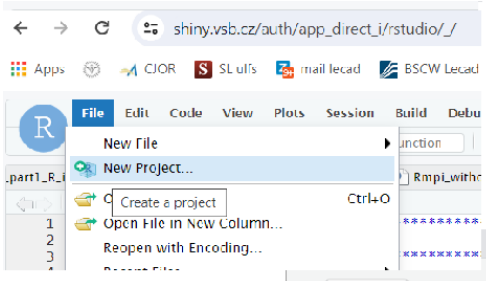


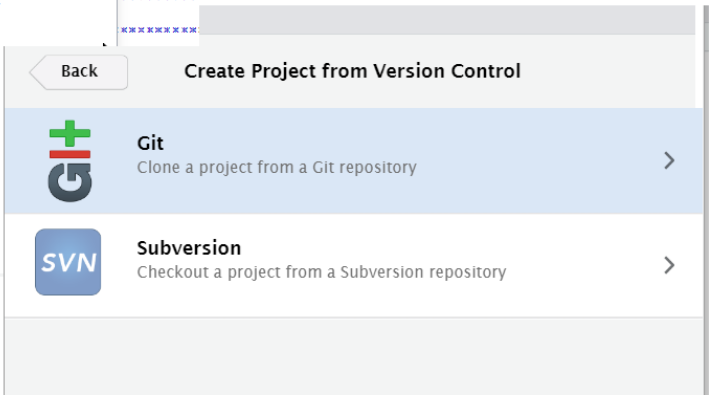
Figure 1: `https://rstudio.com/products/rstudio/download/`

Clone the data

Clone project from GITHUB

https://github.com/janezpovh/parallel-R-nov-26





Creating the first script file

Create and save simple data file

```
N=1000;
Data=data.frame(group=character(N),ints=numeric(N),reals=numeric(N))
Data$group=sample(c("a","b","c"), 1000, replace=TRUE);
Data$ints=rbinom(N,10,0.5);
Data$reals=rnorm(N);

head(Data)
Data

write.table(Data, file='data/Data_Ex_1.txt', append = FALSE, dec = ".",col.names =
 TRUE)

ls()
rm(list = ls())
```

Load and analyse the data

Load data

```
Data_read<-read.table(file='data/Data_Ex_1.txt',header = TRUE)
# first few rows
head(Data_read)
#10 th row
Data_read[10,]
# column group
Data_read$group
Data_read[,1]
```

Analyse the data

Simple analysis

```
# compute means and counts by groups
group count_ints mean_ints
a | 337 | 5.014837
b | 338 | 5.032544
c | 325 | 4.990769

# primitive solution
Group_lev=levels(Data_read$group)

Tab_summary=data.frame(group=character(3),count_ints=integer(3),mean_ints=numeric(3))
Tab_summary$group<-Group_lev
for (i in c(1:3)){
  sub_data = subset(Data_read,group==Group_lev[i])
  Tab_summary$count_ints[i]<-nrow(sub_data)
  Tab_summary$mean_ints[i]<-mean(sub_data$ints)
}
```

Analyze the data by split, aggregate



split, aggregate

```
s <- split(Data_read, Data_read$group)
Tab_summary1<-cbind(aggregate(ints~group,data = Data_read,FUN=length),aggregate(ints~
  group,data = Data_read,FUN=mean))
Tab_summary1<-Tab_summary1[,-3]
```

Outline/next

- Introduction to HPC
- Introduction to R
- Parallel R within one node
- Parallelization with Rmpi

Parallel R within one node

What system do I have

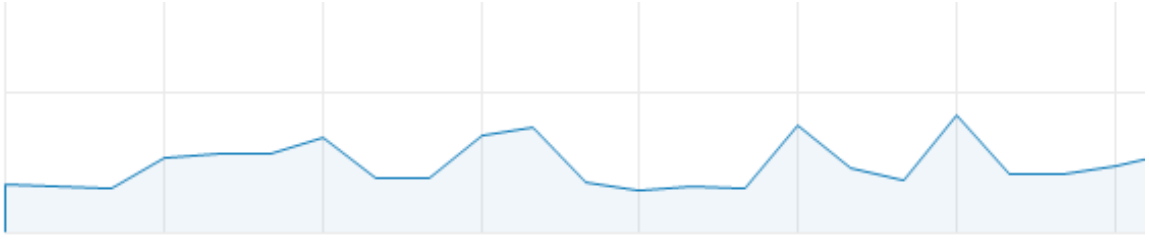
How many cores

```
library(parallel)
detectCores()
> detectCores()
[1] 20
```

What system do I have

How many cores

```
library(parallel)
detectCores()
> detectCores()
[1] 20
```



60 seconds

Utilization	Speed	Processes	Threads	Handles	Up time
5%	1,88 GHz	346	5406	1761616	5:17:11:23

Base speed:	2,40 GHz
Sockets:	1
Cores:	14
Logical processors:	20
Virtualization:	Enabled
L1 cache:	1,2 MB
L2 cache:	11,5 MB
L3 cache:	24,0 MB

Speed up without paralelization - vectorization

Vectorization:

- a programming technique used to avoid explicit loops in order to improve the performance and readability of code.
- a function can be called on a vector and operate on each element, rather than requiring a loop across elements
- Example Use `sum()` or `mean()` instead of loop variant.

Speed up without paralelization - vectorization

EURO

Vectorization:

- vectorization is faster: R is an interpreted language and not a compiled one:
  - R needs to repeatedly interpret what your code means for each iteration of the loop;
  - each iteration of the for loop requires indexing into the main object using the subset function.
- Vectorised functions loop over and repeatedly indexes the main object, but this is done in the compiled language C and has been optimized to take advantage of the fact that the elements of a vector are contiguous in memory.

Benchmarking

Let us benchmark a loop and vectorised function:

Benchmarking

```
# Library for timing comparison
library(microbenchmark)

x <- 1:100000
microbenchmark(a = sum(x),
b = {
  s0 <- 0
  for (i in seq_along(x)) {
    s0 <- s0 + x[i]
  }
})
```

apply, lapply, sapply



These are not typical vectorised functions - they are "loop-hiding", since they contain implicit loops. apply actually has a loop in its R code.

apply, lapply, sapply

```
apply(X, MARGIN, FUN)
Here:
-x: an array or matrix
-MARGIN=1: the manipulation is performed on rows
-MARGIN=2: the manipulation is performed on columns
-MARGIN=c(1,2): the manipulation is performed on rows and columns
-FUN: tells which function to apply. Built functions like mean, median, sum, min, max and
      even
user-defined functions can be applied
```

apply

For data constructed above (`Data_read`) compute row and columns means using `apply`

apply

For data constructed above (Data_read) compute row and columns means using apply

apply

```
Data_read<-read.table(file='data/Data_Ex_1.txt',header = TRUE)

Data_col_means_1 <- colMeans(Data_read[,-1])
Data_col_means_2 <- apply(Data_read[,-1],2,FUN =mean)

Data_row_means_1 <- rowMeans(Data_read[,-1])
Data_row_means_2 <- apply(Data_read[,-1],1,FUN =mean)

Data_both_squares <- apply(Data_read[,-1],c(1,2),FUN = function(x) return(x^2))
```

lapply

- lapply function takes list, vector, matrix or data frame as linear input and returns only list as output
- sapply function takes list, vector, matrix or data frame as linear input. It is similar to lapply function but returns only vector as output.

For data constructed above (Data_read) compute row and columns sums using lapply

lapply

- lapply function takes list, vector, matrix or data frame as linear input and returns only list as output
- sapply function takes list, vector, matrix or data frame as linear input. It is similar to lapply function but returns only vector as output.

For data constructed above (Data_read) compute row and columns sums using lapply

lapply

```
Data_col_sums_1 <- apply(Data_read[,-1],2,FUN =sum)
Data_col_sums_2 <- lapply(Data_read[,-1],FUN =sum)

typeof(Data_col_sums_1)
typeof(Data_col_sums_2)


Data_abs <- lapply(Data_read[,-1],FUN =abs)
Data_sq <- lapply(Data_read[,-1],FUN = function(x){x^2})

typeof(Data_abs)
length(Data_abs)
```

sapply

For data constructed above (`Data_read`) compute row and columns sums using `sapply`

sapply

For data constructed above (Data_read) compute row and columns sums using sapply

sapply

```
Data_col_sums_1 <- apply(Data_read[,-1],2,FUN =sum)
Data_col_sums_2 <- lapply(Data_read[,-1],FUN =sum)
Data_col_sums_3 <- sapply(Data_read[,-1],FUN =sum)

typeof(Data_col_sums_1)
typeof(Data_col_sums_2)
typeof(Data_col_sums_3)


Data_col_sums_4 <- lapply(list(Data_read$ints,Data_read$reals),FUN =sum)
Data_col_sums_5 <- sapply(list(Data_read$ints,Data_read$reals),FUN =sum)
Data_col_len_1 <- lapply(list(Data_read$ints,Data_read$reals),FUN =length)
Data_col_len_2 <- sapply(list(Data_read$ints,Data_read$reals),FUN =length)
```

for loop

Let us compute sums of all elements of K random matrices of order $N \times N$

for

```
N=1000
K=60
set.seed(2021)
sum_rand=rep(0,K-1);
tic()
time_for_sys=system.time({
  for (i in c(1:K)){
    A=rand(N,N)
    sum_rand[i]=sum(A)
  }
})
time_for=toc()
```

foreach do loop

Let us compute sums of all elements of K random matrices of order $N \times N$

for

```
set.seed(2021)
sum_rand=rep(0,K-1);
tic()
time_foreach_sys=system.time({
  foreach (i = c(1:K)) %do% {
    A=rand(N,N)
    sum_rand[i]=sum(A)
  }
})
time_foreach=toc()

> time_for_foreach
elapsed
time_for        1.412
time_foreach    1.294
```

Multisession (PSOCK) vs. Multicore (FORK)



MULTISESSION / PSOCK

```
Main R
|
+-- start new R --> Worker 1
+-- start new R --> Worker 2
+-- start new R --> Worker 3
+-- start new R --> Worker 4
```

Independent R sessions

Objects transferred by serialization

MULTICORE / FORK

```
Main R
|
+-- fork --> Worker 1
+-- fork --> Worker 2
+-- fork --> Worker 3
+-- fork --> Worker 4
```

Copies of current R process

Memory initially inherited via copy-on-write

When multisession (PSOCK)?



Reasons for multisession (PSOCK):

- Portability: code works on Windows, Linux, macOS.
- More predictable behavior with RStudio, connections, external libraries, etc.
- Safety and reproducibility are more important than minimizing overhead.

When multisession (PSOCK)?

Reasons for multisession (PSOCK):

- Portability: code works on Windows, Linux, macOS.
- More predictable behavior with RStudio, connections, external libraries, etc.
- Safety and reproducibility are more important than minimizing overhead.
- **Cost:** objects have to be serialized and transferred to workers, so large data can create communication and memory overhead.

When multicore (FORK)?

Reasons for multicore (FORK):

- You are on Linux/macOS, not Windows.
- Computations repeatedly use large, mostly read-only objects (copy-on-write principle).
- You want fast worker creation and lower data-transfer overhead.
- You know your code/libraries are fork-safe.
- **Cost:** if you start changing big data objects, then data transfer is needed to workers.

Libraries `parallel`, `doParallel`

- `parallel` package comes in the base R installation
- `parallel` works great for any task that you pass to the apply family (e.g., lapply becomes parLapply).
- `doParallel` package works great when you want to use parallel variant of for-loops (foreach–do), and might be a little easier to use.

doParallel

- This library is meant for use with `foreach`, which lets you use a particular type of for-loop, that looks like:

```
foreach(i=list_of_elements) %do% {thing with i}.
```

doParallel

- This library is meant for use with `foreach`, which lets you use a particular type of for-loop, that looks like:

```
foreach(i=list_of_elements) %do% {thing with i}.
```

- Foreach allows this to be parallelized, using dopar:

```
foreach(i=listOfThings) %dopar% {thing with i}.
```

doParallel

- This library is meant for use with `foreach`, which lets you use a particular type of for-loop, that looks like:

```
foreach(i=list_of_elements) %do% {thing with i}.
```

- Foreach allows this to be parallelized, using `dopar`:

```
foreach(i=listOfThings) %dopar% {thing with i}.
```

- Note that: parallelization with `dopar` depends on which backend you use.
  - `doParallel` is one such backend - it tells `foreach` to use `parallel`.
  - There are others: `doFuture`, `doMPI` (another parallel backend, using message passing interface), `doSnow` (another backend, using the `snow` package for creating parallel processes),...
  - By default, `doParallel` uses multicore functionality on Unix-like systems and multisession functionality on Windows.

Parallel foreach-dopar loop

Let us compute sums of all elements of K random matrices of order $N \times N$ using foreach ...dopar from foreach and doParallel

foreach do vs. foreach dopar

```
N=3000
bench_1<-microbenchmark::microbenchmark(foreach (i = c(1:K)) %dopar% {
  library(pracma)
  A= rand(N,N)
  sum_rand[i]=sum(A)
},
foreach (i = c(1:K)) %do% {
  A=rand(N,N)
  sum_rand[i]=sum(A)
},times = 10
)
```

```
      min        lq      mean    median        uq       max neval
821.7110  844.1936  922.3553  870.8398  898.5949  1404.979    10
842.3294  846.7332  908.1153  857.0934  871.2040  1305.258    10
```

Parallel foreach-dopar loop

Let us compute sums of all elements of K random matrices of order $N \times N$ using foreach ...dopar from foreach and doParallel

foreach do vs. foreach dopar

```
N=3000
bench_1<-microbenchmark::microbenchmark(foreach (i = c(1:K)) %dopar% {
  library(pracma)
  A= rand(N,N)
  sum_rand[i]=sum(A)
},
foreach (i = c(1:K)) %do% {
  A=rand(N,N)
  sum_rand[i]=sum(A)
},times = 10
)
```

```
     min        lq      mean    median        uq       max neval
821.7110 844.1936 922.3553 870.8398 898.5949 1404.979    10
842.3294 846.7332 908.1153 857.0934 871.2040 1305.258    10
```

Do you observe any difference?

Creating cluster with doParallel

- Option 1: (use doParallel)

Create cluster

```
clust <- makeCluster(n_cores-1)
registerDoParallel(clust)
getDoParName()
.
.
stopCluster(clust)
#registerDoSEQ()  # alternative - register sequential mode
```

Creating cluster with doParallel



- Option 2: use `parallel`

Create cluster

```
registerDoParallel(cores=n_cores-1)
getDoParName()
.
.
registerDoSEQ()  # alternative - register sequential mode
```

- In linux:
  - the first option uses SNOW library and utilizes multisession approach `parallel::parLapply()`
  - the second creates FORK cluster and uses multicore approach via (it effectively utilizes `parallel::mclapply()`).
- In windows: both use SNOW. They create PSOCK clusters and basically utilize `parallel::parLapply()`

Parallel foreach dopar loop - option 1



Let us compute sums of all elements of K random matrices of order $N \times N$ using `foreach ...dopar` from `foreach, doParallel`. Create cluster!

Option 1

```
set.seed(2021)
clust <- makeCluster(n_cores-1)  # use multisession, set to the number of parallel R
     sessions
registerDoParallel(clust)
getDoParName()
sum_rand=rep(0,K-1);
tic()
time_foreachdopar_1_sys=system.time({
  print("for each-dopar (cluster allocated)")
  foreach (i = c(1:K)) %dopar% {
    library(pracma)
    A=rand(N)
    sum_rand[i]=sum(A)
}}
)
time_foreach_dopar_1=toc()
stopCluster(clust)
```

Parallel foreach dopar loop - option 2



Let us compute sums of all elements of K random matrices of order $N \times N$ using `foreach ...dopar` from `foreach, doParallel`. Create cluster!

Option 2

```
set.seed(2021)
registerDoParallel(n_cores-1)  # use multicore, set to the number of our cores - needed
     for foerach dopar
getDoParName()
sum_rand=rep(0,K-1);
tic()
time_foreachdopar_2_sys=system.time({
  print("for each-dopar (cluster allocated)")
  foreach (i = c(1:K)) %dopar% {
    library(pracma)
    A=rand(N)
    sum_rand[i]=sum(A)
}}
)
time_foreach_dopar_1=toc()
registerDoSEQ()   #this registers sequential mode - equivalent
```

Timings

Timings

	user.self	sys.self	elapsed
time_for_sys	0.994	0.060	1.059
time_foreach_sys	1.148	0.046	1.201
time_foreachdopar_sys	0.045	0.011	0.692
time_foreachdopar_1_sys	0.036	0.010	0.647
time_foreachdopar_2_sys	0.015	0.012	0.617

Library parallel: Socket/Multisession Clusters

- The `parallel` package incorporates functionality from `multicore` and `snow`.

Library `parallel`: Socket/Multisession Clusters

- The `parallel` package incorporates functionality from `multicore` and `snow`.
- The socket-based/multisession approach:
  - starts separate R worker processes;
  - communicates with workers through sockets;
  - works on Linux, macOS, and Windows;
  - can use workers on the local or remote computers.

Library `parallel`: Socket/Multisession Clusters

- The `parallel` package incorporates functionality from `multicore` and `snow`.
- The socket-based/multisession approach:
  - starts separate R worker processes;
  - communicates with workers through sockets;
  - works on Linux, macOS, and Windows;
  - can use workers on the local or remote computers.
- Advantages:
  - Independent and isolated R sessions.
  - More portable and generally safer than forking.

Library `parallel`: Socket/Multisession Clusters

- The `parallel` package incorporates functionality from `multicore` and `snow`.
- The socket-based/multisession approach:
  - starts separate R worker processes;
  - communicates with workers through sockets;
  - works on Linux, macOS, and Windows;
  - can use workers on the local or remote computers.
- Advantages:
  - Independent and isolated R sessions.
  - More portable and generally safer than forking.
- Limitations:
  - Data must be serialized and transferred to workers.
  - Packages and objects may need to be loaded or exported.
  - Worker creation and communication introduce overhead.

Library `parallel`: Socket/Multisession Clusters



- The `parallel` package incorporates functionality from `multicore` and `snow`.
- The socket-based/multisession approach:
  - starts separate R worker processes;
  - communicates with workers through sockets;
  - works on Linux, macOS, and Windows;
  - can use workers on the local or remote computers.
- Advantages:
  - Independent and isolated R sessions.
  - More portable and generally safer than forking.
- Limitations:
  - Data must be serialized and transferred to workers.
  - Packages and objects may need to be loaded or exported.
  - Worker creation and communication introduce overhead.
- Typical functions: `makeCluster()`, `parLapply()`, and `parSapply()`.

Library `parallel`: Fork-Based Parallelism



- The forking approach creates worker processes as copies of the current R process.
- Forked workers are used by `mclapply()`.

Library `parallel`: Fork-Based Parallelism



- The forking approach creates worker processes as copies of the current R process.
- Forked workers are used by `mclapply()`.
- Advantages:
  - Workers are fast to create.
  - Workers initially inherit the current R session, including loaded packages, functions, and objects.
  - Memory pages are initially shared using *copy-on-write*.
  - It is particularly efficient for large, mostly read-only objects.

Library parallel: Fork-Based Parallelism



- The forking approach creates worker processes as copies of the current R process.
- Forked workers are used by `mclapply()`.
- Advantages:
  - Workers are fast to create.
  - Workers initially inherit the current R session, including loaded packages, functions, and objects.
  - Memory pages are initially shared using *copy-on-write*.
  - It is particularly efficient for large, mostly read-only objects.
- Limitations:
  - Forking is available on Unix-like systems, such as Linux and macOS, but not on Windows.
  - Modified memory pages must be copied separately for each worker.
  - Some libraries, connections, threads, and graphical interfaces may not be fork-safe.
  - Parallel random-number generation must be handled carefully to ensure independent and reproducible streams.

Parallel versions of lapply

By using library `parallel` and `parSapply`, `mclapply` compute sums of all elements of K random matrices of order $N \times N$. Create cluster!

parallel versions of apply

```
mat_sum<-function(x){
  library(pracma)
  A=rand(x)
  return(sum(A))
}

time_lapply<-system.time({
  set.seed(2021)
  sum_rand_lapply=lapply(rep(N,K),FUN=mat_sum)
})

time_sapply<-system.time({
  set.seed(2021)
  sum_rand_sapply=sapply(rep(N,K),FUN=mat_sum)
})
```

Parallel versions of lapply

parallel versions of apply

```
#forking
time_mcLapply<-system.time({
  set.seed(2021)
  sum_rand_mcLapply=mclapply(X=rep(N,K),FUN=mat_sum,mc.cores = n_cores)
})

# socketing/multisession
clust <- makeCluster(n_cores, type="PSOCK")
time_parLapply<-system.time({
  set.seed(2021)
  sum_rand_parLapply=parLapply(clust,rep(N,K),fun=mat_sum)
})
stopCluster(clust)

clust <- makeCluster(n_cores, type="PSOCK")
time_parSapply<-system.time({
  set.seed(2021)
  sum_rand_parSapply=parSapply(clust,rep(N,K),FUN=mat_sum)
})
stopCluster(clust)
```

Parallel versions of lapply

parallel versions of apply - 2 cores

```
times_apply<-rbind(time_lapply,time_sapply,time_parLapply,time_parSapply,time_
  mcLapply)

> times_apply[,1:3]
                user.self   sys.self elapsed
             user.self sys.self elapsed
time_lapply         1.050    0.168   1.227
time_sapply         1.021    0.171   1.201
time_parLapply      0.002    0.002   0.568
time_parSapply      0.002    0.001   0.570
time_mcLapply       0.001    0.008   0.643
```

Very parallelizable task

Perfectly parallelizable computing task

```
# simple very parallel
library(parallel)
library(tictoc)

f <- function(...) {
  Sys.sleep(1)
  "DONE"
}


for (p in 1:8) {
  t <- system.time(
  mclapply(1:25, f, mc.cores = p)
  )
  cat("workers =", p,
  " elapsed =", t["elapsed"], "\n")
}
```

```
workers = 1  elapsed = 25.005
workers = 2  elapsed = 13.008
workers = 3  elapsed = 9.005
workers = 4  elapsed = 7.006
workers = 5  elapsed = 5.013
workers = 6  elapsed = 5.006
workers = 7  elapsed = 4.01
workers = 8  elapsed = 4.005
```

Not so parallelizable task

Weakly parallelizable computing task

```
# simple very parallel
library(parallel)
library(tictoc)

g <- function(...) {
  sum(rnorm(1e7))
}

for (p in 1:8) {
  t <- system.time(
  mclapply(1:25, g, mc.cores = p)
  )
  cat("workers =", p,
  " elapsed =", t["elapsed"], "\n")
}
```

```
workers = 1  elapsed = 10.431
workers = 2  elapsed = 5.524
workers = 3  elapsed = 5.634
workers = 4  elapsed = 5.667
workers = 5  elapsed = 5.624
workers = 6  elapsed = 5.886
workers = 7  elapsed = 5.594
workers = 8  elapsed = 5.575
```

Summary: libraries for shared memory parallelization in R



- Parallel for-loop (`foreach...dopar`). Cluster created by `registerDoParallel(N)` and `registerDoSEQ()`. Library `foreach`, `doParalel` needed. Type of parallelization: multisession (socket, with call makeCluster, stopCluster), multicore (fork) if use registerDoparalel, registerDoSEQ
- Parallel `apply`: needs library `parallel`. `parLapply`, `parSapply` create multisession (socket), while mclapply creates multicore

The future package?

One interface – different execution strategies

- `future` separates what to compute from how and where to compute it.
- The same R code can run sequentially, with independent R sessions (`multisession`), or using forked processes (`multicore`).
- No need to explicitly create, manage, export data to, and stop clusters.
- Required globals and packages are automatically identified and made available to workers.
- Changing the parallelization strategy often requires changing only the `plan()`.

Main idea

Write the computation once – choose the execution strategy separately.

The future package?



One interface – different execution strategies

- `future` separates what to compute from how and where to compute it.
- The same R code can run sequentially, with independent R sessions (`multisession`), or using forked processes (`multicore`).
- No need to explicitly create, manage, export data to, and stop clusters.
- Required globals and packages are automatically identified and made available to workers.
- Changing the parallelization strategy often requires changing only the `plan()`.

Main idea

Write the computation once – choose the execution strategy separately.

```
plan(sequential) # no parallelization
plan(multisession, workers = 4) # PSOCK workers
plan(multicore, workers = 4) # forked workers (Linux/macOS)
```

Example with future

Run this function in serial, multisession and multicore mode.

```
# One computationally expensive task
f <- function(i) {
  x <- rnorm(5e6)
  c(task = i, statistic = mean(exp(x)))
}
```

Run R scrip in three modes



▸ Open / Copy code

Run function in three modes

```
library(future)
library(future.apply)
n_tasks <- 8
n_workers <- 2

# One computationally expensive task
f <- function(i) {
  x <- rnorm(5e6)
  c(task = i, statistic = mean(exp(x)))
}

# 1. Sequential
plan(sequential)
set.seed(2026)
t_seq <- system.time({
  res_seq <- future_lapply(
  1:n_tasks, f,
  future.seed = TRUE
  )
})
```

Run R scrip in three modes - results



Results

```
> print(times)
sequential.elapsed   multisession.elapsed     multicore.elapsed
2.878                2.007                    1.743
```

Why library future?

- With `parallel` and `doParallel`, the code is more tightly connected to the way parallel workers are created:
  - `makeCluster()`, `parLapply()`, `stopCluster()`
  - `registerDoParallel()`, `%dopar%`
- `future` separates two questions:
  - What computation should be performed?
  - How and where should it be performed?

Cross-validation of models

- Suppose we want to cretae N GLM models, where training data sets is random sample of size 70%, and test data sets is the remaing set.
- If $N = 10$ and we use 10% training set, we have the usual 10-fold cross-validation.
- We do it on dataset `K_data_clean.txt`.
- Use `R` script `parallel_cross-validation.R`
- Timings with 10 cores

```
          user.self  sys.self  elapsed
time_ser    139.71       0.11   141.89
time_par      0.76       0.13    30.82
```

Outline/next

- Introduction to HPC
- Introduction to R
- Parallel R within one node
- Parallelization with Rmpi

Parallelization with Rmpi

What is Rmpi

- `Rmpi`: an R interface to MPI (Message Passing Interface) for distributed-memory parallel computing.
- Allows multiple R processes to communicate using standard MPI operations such as point-to-point communication, broadcasting, gathering, and reduction.
- Particularly suitable for HPC clusters, where computations can be distributed across multiple CPU cores and multiple compute nodes.
- Supports both direct MPI execution and a manager–worker model, enabling parallel simulations, numerical computations, and distributed data processing.
- Requires an MPI implementation (e.g., Open MPI) and therefore must usually be compiled against the MPI environment available on the HPC system.
- Current CRAN version: **0.7-4.1** (November 2026); see `https://cran.r-project.org/web/packages/Rmpi/Rmpi.pdf`.

Few basic command

- `Rmpi::mpi.comm.size(0)`: returns the number of active processes in current computing task/job
- `Rmpi::mpi.comm.rank(0)`: returns the ID of current process (number from $\{0, 1, 2, \ldots, \text{size} - 1\}$
- `Rmpi::mpi.get.processor.name()` - returns the name of compute node where the process runs.

Hello world example

EURO

Open / Copy code

Compute largest eigenvalue of $n \times n$ random symmetric matrices

```
# Load Rmpi
library(Rmpi)
n <- 30
# MPI information
size <- Rmpi::mpi.comm.size(0)
rank <- Rmpi::mpi.comm.rank(0)
host <- Rmpi::mpi.get.processor.name()

if (rank == 0) {
  cat("size", "rank", "host", "max_eigen_value", "\n")
  cat(size, rank, host, "NaN", "\n")
} else {
  A <- matrix(rnorm(n^2), nrow = n)
  A <- A + t(A)
  a <- max(eigen(A, symmetric = TRUE, only.values = TRUE)$values)
  cat(size, rank, host, a, "\n")
}
# Wait until all MPI processes have finished their work
Rmpi::mpi.barrier(0)

# Finish MPI cleanly
Rmpi::mpi.finalize()
```

How to distribute this task across cluster



- Save the scripts from previous slide into separate file, called e.g. `Rmpi_master_slave.R`
- Create separate `.batch` file, where the parallelization is defined, e.g., `Job_Rmpi_master_slave.sbatch`

How to distribute this task across cluster



Open / Copy code

SLURM job description

```
#!/bin/bash
#SBATCH --export=ALL,LD_PRELOAD=
#SBATCH --job-name Rmpi
#SBATCH --partition=qcpu
#SBATCH --time=02:00
#SBATCH --nodes=2
#SBATCH --ntasks-per-node 128 ## maximum is 128
#SBATCH --output=logs/%x_%j.out
#SBATCH --account atr-26-4
module --force purge
module load R/4.4.2-gfbf-2024a
module load OpenMPI/5.0.3-GCC-13.3.0
export R_LIBS_USER="$HOME/R/library/4.4"
mkdir -p "$R_LIBS_USER"
Rscript -e '
.libPaths(c(Sys.getenv("R_LIBS_USER"), .libPaths()))
if (!requireNamespace("Rmpi", quietly = TRUE)) {
  cat("Rmpi not installed - installing...\n")
  install.packages(
  "Rmpi",
  lib = Sys.getenv("R_LIBS_USER"),
  repos = "https://cloud.r-project.org"
  )
}
library(Rmpi)
cat("Rmpi OK\n")
'
mpirun Rscript Rmpi_master_slave.R
```

Go to Karolina

```
# Copy folder code  to Karolina
scp -i ~/.ssh/id_ed25519 -r ~/parallel-R-nov-26/code it4i-jpovh@karolina.it4i.cz:~/

# Go to Karolina
ssh -i ~/.ssh/id_ed25519 it4i-jpovh@karolina.it4i.cz

#Go to folder code and run

sbatch Job_Rmpi_direct.sbatch

# check results in file code/logs/*.out
```

Rcpp

- It is a package originally develop by Dirk Eddelbuettel and Romain François
- It aims to ease the extension of `R` with C++ code.
- It allows to load C++ code in an interactive session.
- It has framework to help when creating package with `Rcpp`
- `Credit:` this content was prepared based on materials from dr. Tomas Martinovic from Technical university Ostrava.

Create C++ function within R

- Create C++ function within R by using cppFunction()
- Rcpp does all the nasty work (compiling, linking)

Mandelbrot set

- Two-dimensional set: simple definition, great complexity,
- Definition: The Mandelbrot set is set of all points c in complex plane for which the sequence

$$z_{n+1} = z_n^2 + c$$

does not diverge to infinity when iterated starting at $z_1 = c$.

- Theorem: Complex point c is in the Mandelbrot set if and only if $|z_n| \leq 2$, for all $n \in \mathbb{N}$.

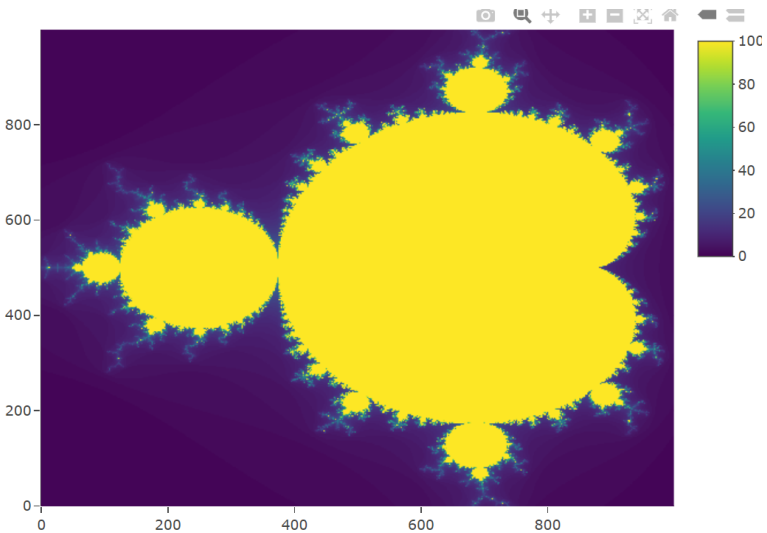
Mandelbrot set

- Two-dimensional set: simple definition, great complexity,
- Definition: The Mandelbrot set is set of all points c in complex plane for which the sequence

$$z_{n+1} = z_n^2 + c$$

  does not diverge to infinity when iterated starting at $z_1 = c$.
- Theorem: Complex point c is in the Mandelbrot set if and only if $|z_n| \leq 2$, for all $n \in \mathbb{N}$.

Visualization: (maximum number of iterations is 100)



Inner algorithm in R

For any point $c = a + ib \in \mathbb{C}$ we check, if it is in the Mandelbrot set by the following procedure:

```
mandelbrot <- function(c, max_iter = 100) {
 z <- c
 for (i in 1:max_iter-1) {
  z <- z ^ 2 + c
  if (abs(z) > 2) {
   return(i)
  }
 }
 return(max_iter)
}
```

Inner algorithm in C++

C++ code for the inner algorithm using Rcpp:

```
Rcpp::cppFunction(
"
int Mandel(double real, double im, int max_iter = 100)
{
 std::complex<double> c(real, im);
 std::complex<double> z = c;
 for (int i=0; i< max_iter; i++){
  z = z * z + c;
  if (std::abs(z) > 2) {
   return i;
  }
 }
 return max_iter;
}
"
)
```

Outer (meta) algorithm

1. Input: A - discrete rectangular grid of complex points between $c_{\text{min}} = -1.5 - i$ and $c_{\text{max}} = 0.5 + i$ with resolution 1000 in each dimension.
2. For every point $c \in A$ check, if it is in the Mandelbrot set by the inner algorithm.
3. Visualise the results.

Three implementations

1. The outer and the inner algorithm written in R
2. The outer in R and the inner algorithm in C++
3. The outer and the inner algorithm written in C++

Three implementations

1. The outer and the inner algorithm written in R
2. The outer in R and the inner algorithm in C++
3. The outer and the inner algorithm written in C++

```
Unit: milliseconds
     expr        min         lq       mean     median         uq        max neval
 for_loop 19064.2023 19064.2023 19640.1962 19640.1962 20216.1902 20216.1902     2
    for_c  3773.9743  3773.9743  3890.8985  3890.8985  4007.8227  4007.8227     2
        c   442.7557   442.7557   463.9417   463.9417   485.1277   485.1277     2
```

What is RcppParallel



- Parallel algorithms for `Rcpp` code, built on Intel TBB.
- Threads, not processes: shared memory ⇒ no data copying, no cluster set-up, low overhead.
- You write a Worker: a `struct` whose `operator()(begin, end)` processes a range of indices.
- `parallelFor(begin, end, worker)` splits the range into chunks and runs them on a thread pool with work stealing (automatic load balancing).
- `RVector`/`RMatrix`: thread-safe views of R memory allocated beforehand on the main thread.
- Rule: no R API inside the worker (no `Rcpp` objects, allocation, `Rprintf`) – R is single-threaded.
- Threads set from R: `setThreadOptions(numThreads = 4)`

Mandelbrot: key changes



1. Load the package and its header:

```
cppFunction(depends = "RcppParallel", includes = "#include <RcppParallel.h>", ...)
```

2. Loop body moves into a Worker; each thread gets only rows `[begin, end)`:

```
struct MandelWorker : public RcppParallel::Worker {
 const RcppParallel::RVector<double> xs, ys;  // thread-safe views of inputs
 RcppParallel::RMatrix<int> out;              // thread-safe view of output
 void operator()(std::size_t begin, std::size_t end) {
  for (std::size_t i = begin; i < end; i++)  // was: i = 0 .. resolution-1
  ...                                       // inner loops unchanged
 }
};
```

3. Allocate output on the main thread, then replace the loop by `parallelFor`:

```
IntegerMatrix out(x.size(), y.size());
MandelWorker worker(x, y, out, max_iter);
RcppParallel::parallelFor(0, x.size(), worker);
```

4. Set the number of threads from R:

```
RcppParallel::setThreadOptions(numThreads = 4)
```