

INTRODUCTION TO PARALLEL COMPUTING IN R

prof. dr. Janez Povh
dr. Tomas Martinović (Rcpp)
EuroHPC Competence Center, September 25, 2024

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



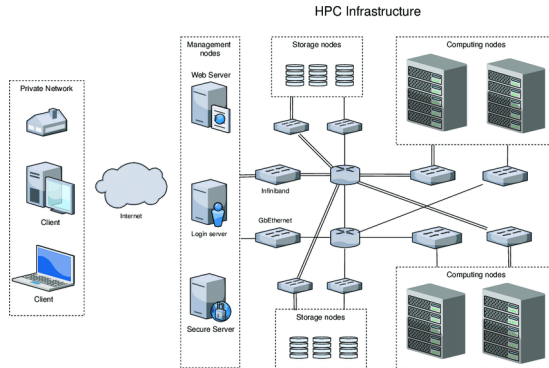
Introduction to HPC

What is HPC



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

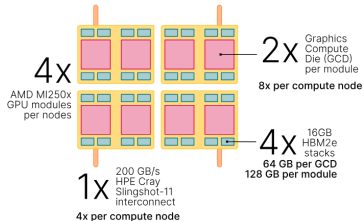
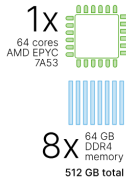
What is HPC



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

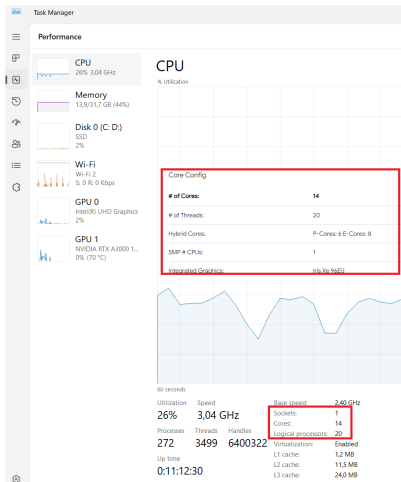
Node structure

2560x compute nodes

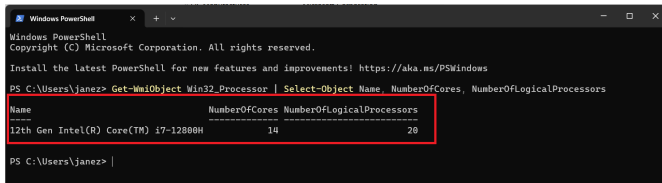


Source: <https://docs.lumi-supercomputer.eu/hardware/lumig/>

How does my computer looks like



How does my computer looks like



A screenshot of a Windows PowerShell terminal window. The window title is "Windows PowerShell". The text inside shows the standard PowerShell startup messages, followed by a command to get system information. The output is a table with three columns: Name, NumberOfCores, and NumberOfLogicalProcessors. The first row of data shows "12th Gen Intel(R) Core(TM) i7-12800H" with 14 cores and 20 logical processors. The command and the output table are highlighted with a red rectangle.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\janez> Get-WmiObject Win32_Processor | Select-Object Name, NumberOfCores, NumberOfLogicalProcessors
```

Name	NumberOfCores	NumberOfLogicalProcessors
12th Gen Intel(R) Core(TM) i7-12800H	14	20

```
PS C:\Users\janez> |
```


How does the cluster look like - add for Barbora



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

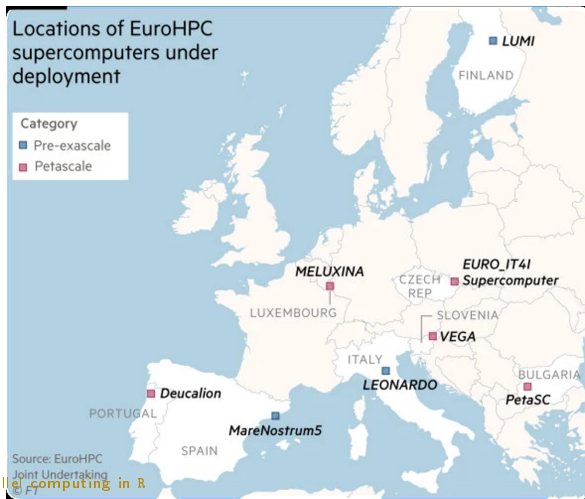
A screenshot of a PuTTY terminal window titled 'login.vega.izum.si - PuTTY'. The window has standard Windows-style window controls (minimize, maximize, close). The terminal output shows the following text:

```
Using username "jpovh".  
Authenticating with public key "rsa-key-20220609"  
Further authentication required  
Keyboard-interactive authentication prompts from server:  
| Verification code: █
```

The cursor is positioned at the end of the 'Verification code:' prompt, indicated by a green block character.

How to access HPC

HPCs are **easily** available.



- Network of EU Competence centers for HPC.

EuroCC2



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

EuroCC2



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

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



Introduction to R

What is R



- Software for Statistical Data Analysis
- Based on S
- Programming Environment
- Interpreted Language
- Data Storage, Analysis, Visualization
- Free and Open Source Software

How to obtain R



- R current version 4.4.0 (released April 2024).
- `http://cran.r-project.org`
- Binary/Windows executable code

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

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.

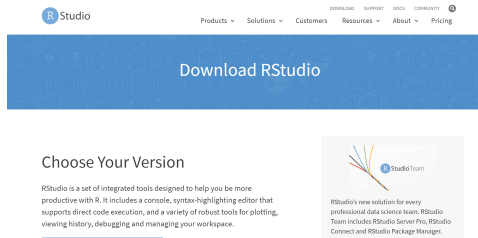


Figure 1: <https://rstudio.com/products/rstudio/download/>

RStudio on IT4I



- Run RStudio on VM.

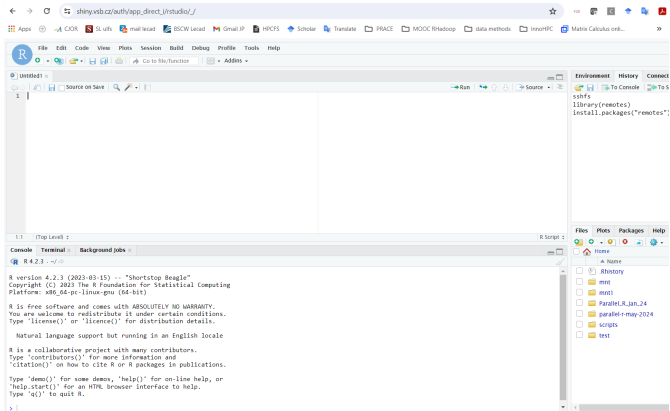
RStudio on IT4I



- Run RStudio on VM.
- Connect to `shiny.vsb.cz/auth`

RStudio on IT4I

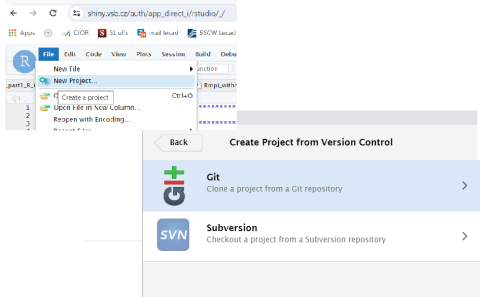
- Run RStudio on VM.
- Connect to `shiny.vsb.cz/auth`



Clone the data

Clone project from GITHUB

<https://github.com/It4innovations/parallel-r-may-2024>



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]
```

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

The background of the slide is a complex, low-poly geometric pattern. It consists of numerous triangles of varying sizes and shades of gray, creating a textured, crystalline effect. The colors range from very light gray to dark charcoal, with the darker tones dominating the right side and bottom of the image.

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	Base speed:	2,40 GHz
5%	1,88 GHz	Sockets:	1
Processes	Threads	Cores:	14
346	5406	Logical processors:	20
Handles		Virtualization:	Enabled
1761616		L1 cache:	1,2 MB
Up time		L2 cache:	11,5 MB
5:17:11:23		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



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 x using the subset function.
- Vectorised functions loop over and repeatedly indexes x, 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:

apply

```
# 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. The R Inferno refers to the apply family of functions as “loop-hiding” - they contain implicit loops. `apply` actually has a loop in its R code. The others do drop down to C for their loops, however, at each step in the C loop, they evaluate the R function passed in. This is what makes them not vectorized, as a true vectorized function performs its loop in C and uses C compiled functions inside that loop.

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 or data frame as input and returns only list as output
- `sapply` function takes list, vector or data frame as 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 or data frame as input and returns only list as output
- sapply function takes list, vector or data frame as 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)
```

Introduction to parallel computing in R

The word "sapply" is written in a stylized, lowercase, yellow font.

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

apply



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

apply

```
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()
```

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 `snow` 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`

for

```
N=3000
set.seed(2021)
sum_rand=rep(0,11);
tic()
foreach (i = c(1:12)) %dopar% {
  A=randn(N,N)
  sum_rand[i]=sum(A)
}
time_foreach_dopar=toc()
```

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`

for

```
N=3000
set.seed(2021)
sum_rand=rep(0,11);
tic()
foreach (i = c(1:12)) %dopar% {
  A=randn(N,N)
  sum_rand[i]=sum(A)
}
time_foreach_dopar=toc()
```

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 `parallel::parLapply()`
 - the second creates FORK cluster and uses MULTICORE library (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)
registerDoParallel(clust) # use multicore, set to the number of our cores - needed for
                           foreach dopar
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	4.16	0.50	4.93
time_for_each_sys	4.21	0.37	4.92
time_for_each_dopar_sys	4.11	0.55	4.98
time_for_each_dopar_1_sys	0.13	0.01	1.90
time_for_each_dopar_2_sys	0.11	0.00	1.80

Library parallel



- encapsulates existing libraries multicore, snow
- multicore functionality supports multiple workers only on those operating systems that support the **fork** system call - this excludes Windows.
- two ways of parallelization:
 - The **socket** approach: launches a new version of R on each core via networking (e.g. the same as if you connected to a remote server), but the connection is happening all on your own computer.
 - pros: (i) Works on any system (including Windows); (ii) Each process on each node is unique so it can't cross-contaminate.
 - cons: (i) Each process is unique so it will be slower (ii) Things such as package loading need to be done in each process separately. Variables defined on your main version of R don't exist on each core unless explicitly placed there. (iii) More complicated to implement.
 - use parLapply, parSapply

Library parallel



- The **forking** approach copies the entire current version of R and moves it to a new core.
 - (i) Faster than sockets. (ii) Because it copies the existing version of R, your entire workspace exists in each process. (iii) Easy to implement.
 - Cons (i) Only works on POSIX systems (Mac, Linux, Unix, BSD) and not Windows. (ii) it can cause issues specifically with random number generation or when running in a GUI (such as RStudio). This doesn't come up often.
- use `mclapply`

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
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

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

```
> times_apply[,1:3]
```

	user.self	sys.self	elapsed
time_lapply	1.741	0.011	1.751
time_sapply	1.726	0.007	1.731
time_parLapply	0.007	0.004	1.940
time_parSapply	0.005	0.005	1.842
time_mclapply	0.004	0.238	1.679

Libraries for shared memory parallelization in R



- Parallel for-loop (`foreach...dopar`). Cluster created by `registerDoParallel(N)` and `registerDoSEQ()`. Library `foreach`, `doParallel` needed.
- Parallel apply: `parLapply`, `parSapply`, `mcLapply` need library `parallel`.

Very parallelizable task



Perfectly parallelizable computing task

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

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

tic()
res <- lapply(1:25, f)
t1=toc()
#> 5.025 sec elapsed


tic()
res <- mclapply(1:25, f, mc.cores = 25)
t2=toc()
#> 1.019 sec elapsed
```

Cross-validation of models

- Suppose we want to create N GLM models, where training data sets is random sample of size 70%, and test data sets is the remaining 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

	<code>user.self</code>	<code>sys.self</code>	<code>elapsed</code>
<code>time_ser</code>	139.71	0.11	141.89
<code>time_par</code>	0.76	0.13	30.82

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

The background of the slide is a complex, low-poly geometric pattern. It consists of numerous triangles of varying sizes and shades of gray, creating a textured, crystalline effect. The colors range from very light gray to dark charcoal, with the darker tones dominating the lower and right portions of the image.

Parallelization with Rmpi

What is Rmpi



- Rmpi library: Interface for MPI (Message Passing Interface) in R.
- Enables parallel and distributed computing in the R programming language.
- Facilitates communication and coordination between R processes across multiple nodes.
- Particularly useful for parallelizing computationally intensive tasks like simulations or data processing.
- Users can harness the power of parallel computing for improved performance in certain applications.
- Latest version from Dec 2023, 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 $\{0, 1, 2, \dots, \text{size} - 1\}$)
- `Rmpi::mpi.get.processor.name()` - returns the name of compute node where the process runs.

Hello word example

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

```
library(Rmpi)
n=30
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 {
  where=getwd()
  A=matrix(rnorm(n^2), nrow=n)
  A=A+t(A)
  a = max(eigen(A)$values)
  cat(size, rank, host, a, "\n")
}
```

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

Compute smallest eigenvalue of n symmetric matrices of size $N \times N$

```
#!/bin/bash
#SBATCH --export=ALL,LD_PRELOAD=
#SBATCH --job-name Rmpi
#SBATCH --partition=rome --mem=24GB --time=02:00
#SBATCH --nodes=8
#SBATCH --ntasks-per-node 48 ## maximum is 48
#SBATCH --output=logs/%x_%j.out

module load OpenMPI/4.1.4-GCC-11.3.0
module load R/4.2.1-foss-2022a
srun Rscript Rmpi_master_slave.R
```

Go to Barbora



- Create directory

```
mkdir /home/rstudio/mnt/
```

- copy to it files

```
Job_Rmpi_master_slave.sbatch, Rmpi_master_slave.R
```

- mount this directory

```
sshfs -o IdentityFile=/home/rstudio/.ssh/id_ed25519 it4i-jpovh@barbora.it4i.cz:. /home/rstudio/mnt/
```

- connect to barbora with ssh

```
ssh -i /home/rstudio/.ssh/id_ed25519 it4i-jpovh@barbora.it4i.cz
```


Connect to Barbora



```
rstudio@9cfcdf2562f:~$ sshfs -o IdentityFile=/home/rstudio/.ssh/id_ed25519 it4i-jpov@barbora.it4i.cz: /home/rstudio/mt/
rstudio@9cfcdf2562f:~$ ssh -i /home/rstudio/.ssh/id_ed25519 it4i-jpov@barbora.it4i.cz
client_global_hostkeys.private.confirm: server gave bad signature for ED25519 key 1: incorrect signature
Last login: Tue May 28 00:13:59 2024 from 195.113.175.60
```



...running on Red Hat Enterprise Linux 8.4

Public Service Announcement: Aptainer on the Karolina cluster
Posted: (2024-05-18 10:23:47)

Aptainer is now a part of the operating system, you do not need to load the module.

```
$ aptainer --version
aptainer version 1.3.1-1.el8
```

```
[it4i-jpov@login2.barbora ~]$ █
```

Run

```
sbatch Job_Rmpi_master_slave.sbatch
```

Results in log file



```
[1] "size rank host max_eigen_value"  
[1] "384 0 cn48 NaN"  
[1] "384 110 cn50 8.25199803297607"  
[1] "384 173 cn52 8.01187455128492"  
[1] "384 68 cn49 8.05800653948316"  
[1] "384 200 cn53 8.81600769867893"  
[1] "384 258 cn54 8.12244071842822"  
[1] "384 332 cn55 7.61927646789373"  
[1] "384 338 cn56 4.9472190383247"
```

How parallelise without slurm?

Compute smallest eigenvalue of n symmetric matrices of size $N \times N$

```
rm(list=ls()) # R code: parallel version
library(snow)
library(Rmpi)
nclus=6
cl <- snow::makeMPIcluster(nclus) #alter either n or mc to affect run time
n=30
N_per_proc=100
#x=matrix(runif(n),n,1)
#x=cbind(1,x)
min_eig_values=function(n,N){
  a=c()
  for (ind in 1:N){
    A=matrix(rnorm(n^2),nrow=n)
    A=A+t(A)
    a[ind] = max(eigen(A)$values)
  }
  return(a)
}
ptim=proc.time()[3]
b=clusterCall(cl,min_eig_values,n=n,N=N_per_proc)
b=unlist(b)
hist(b)
tim=proc.time()[3]-ptim
#Rmpi::mpi.quit()
snow::stopCluster(cl)
```

- 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

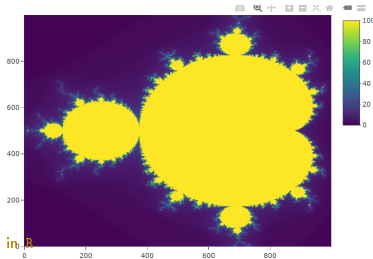
- Two-dimensional set: simple definition, great complexity,
- Definition:** The Mandelbrot set is set of all points c in complex plane for which the sequence

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

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 $c_{\min} = -1.5 - i$ and $c_{\max} = 0.5 + i$ 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++

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