PARALLEL R

prof. dr. Janez Povh
EuroHPC Competence Center, January 18 2024

Outline/next

- Introduction to R
- Advanced and Big data management with R
- Parallelization with Rmpi

Introduction to R

What is R

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

How to obtain R

- R current version 4.3.2 (released on 2023-10-31).
- http://cran.r-project.org
- Binary source codes
- Windows executables

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)

Rectangular

- data frame
- matrix

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.

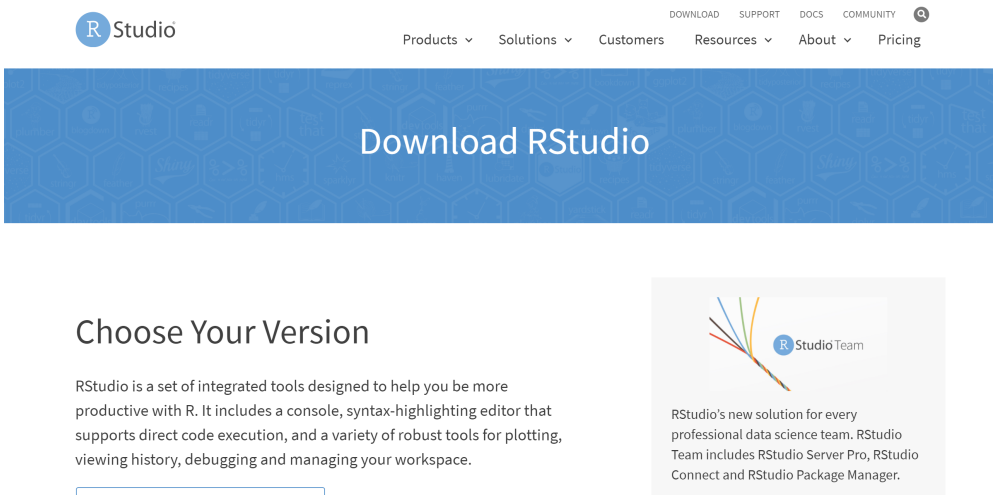


Figure 1: https://rstudio.com/products/rstudio/download/

RStudio on HPCFS

- Connect to HPCFS to gpu02 login node
- Run Rstudio on login node OR
- Connect to compute node and run Rstudio on compute node.

Connect to HPC using NoMachine

Main steps

- Connect to HPCFS to gpu02 login node
- Rund Rstudio on login node OR
- Connect to compute node and run Rstudio on compute node.

Download Nomachine Enterprise Client

NOMACHINE

NoMachine Enterprise Client

Secure remote access to enterprise desktops and applications

Home / Enterprise / NoMachine Enterprise Client

NoMachine Enterprise Client

Enterprise Client enables fast and secure access to your remote PC or desktop computer where you have installed one of the NoMachine server products. It provides a powerful connection interface, named the NoMachine Player, packed with all the features you need to work with your remote desktop. This package is available for those who prefer just a standalone "client-side" application, either because regulatory requirements dictate that connecting clients must stay as thin clients, or simply because a scaled-down client app is your personal preference. It's free to download and can be used to connect remotely from any Windows, Linux, Mac, ARM or Raspberry device to a computer where you have installed a NoMachine Enterprise product or the NoMachine Free Edition.

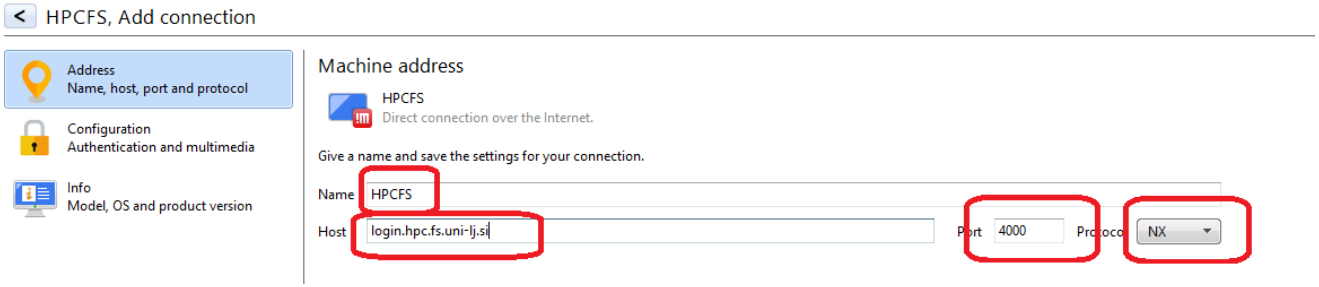
Download

NoMachine Enterprise Client sports the same features regardless of which NoMachine server you are connecting to. To check the features of its parent products, follow the links above.

Install Nomachine Client

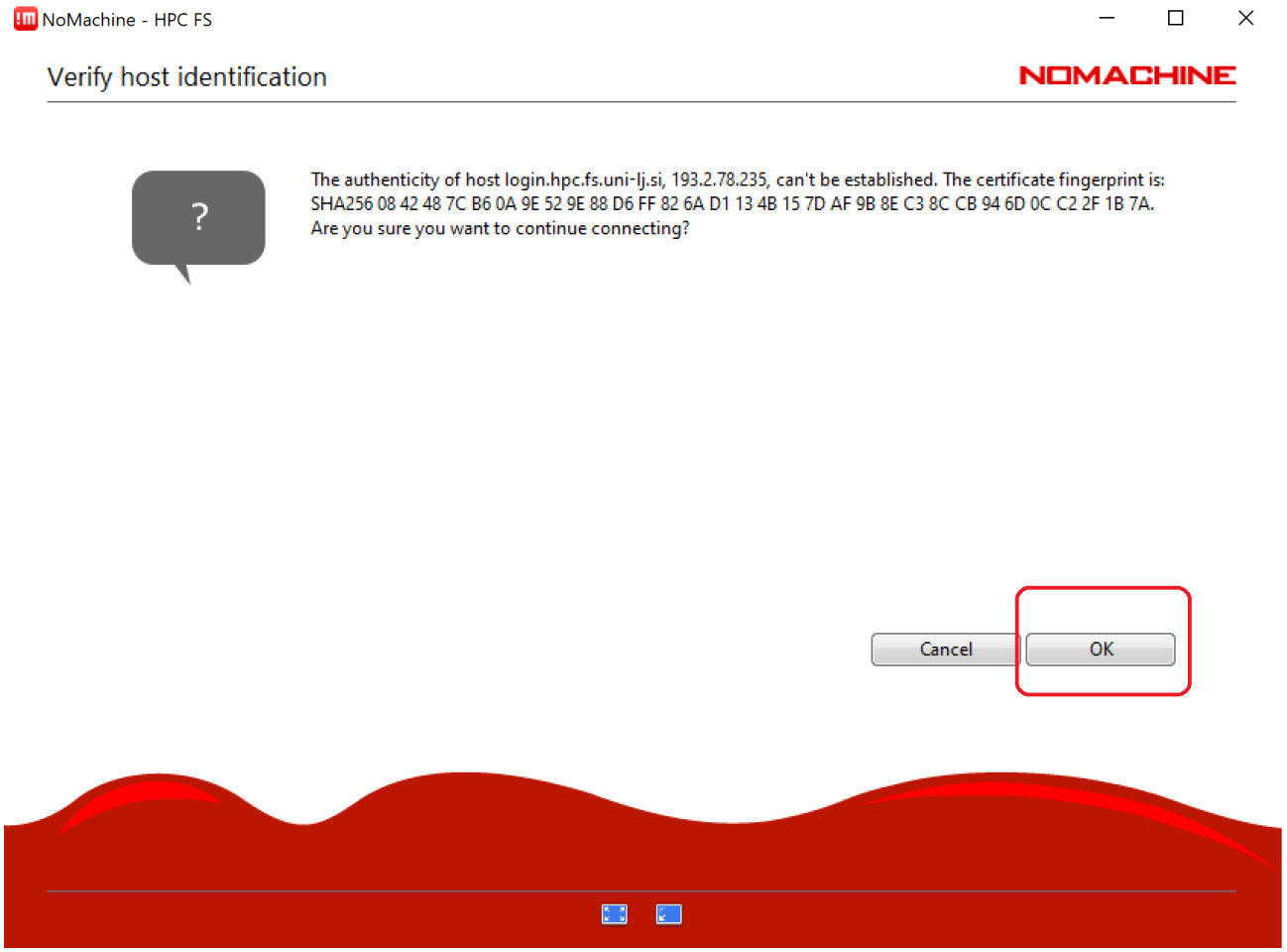
Enter data for HPCFS

HOST: login.hpc.fs.uni-lj.si Name: whatever you want

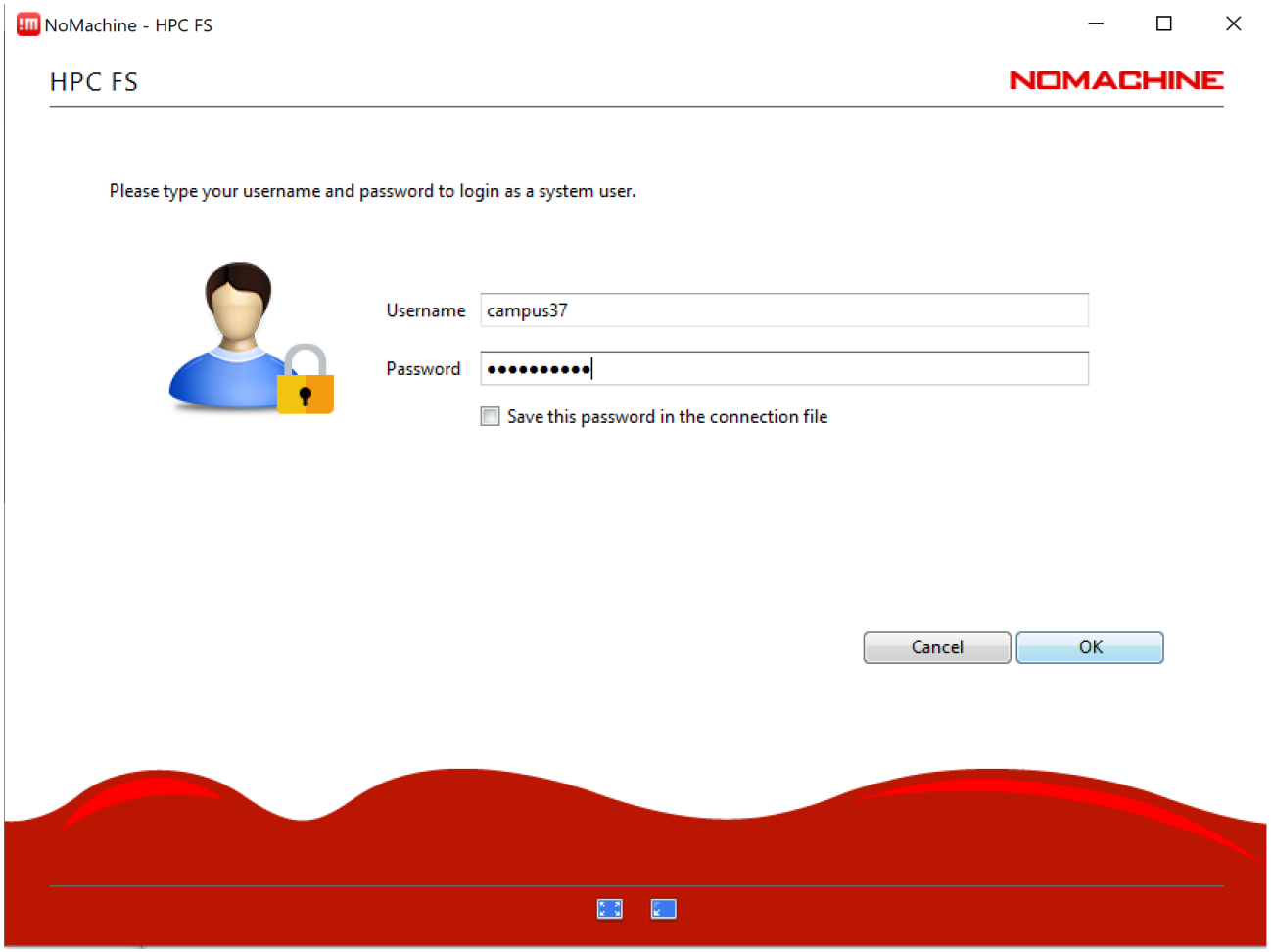



NoMachine
Machines
NOMACHINE
Add
Edit
Connect
Search
View
Order
Settings
HPC FS
NX

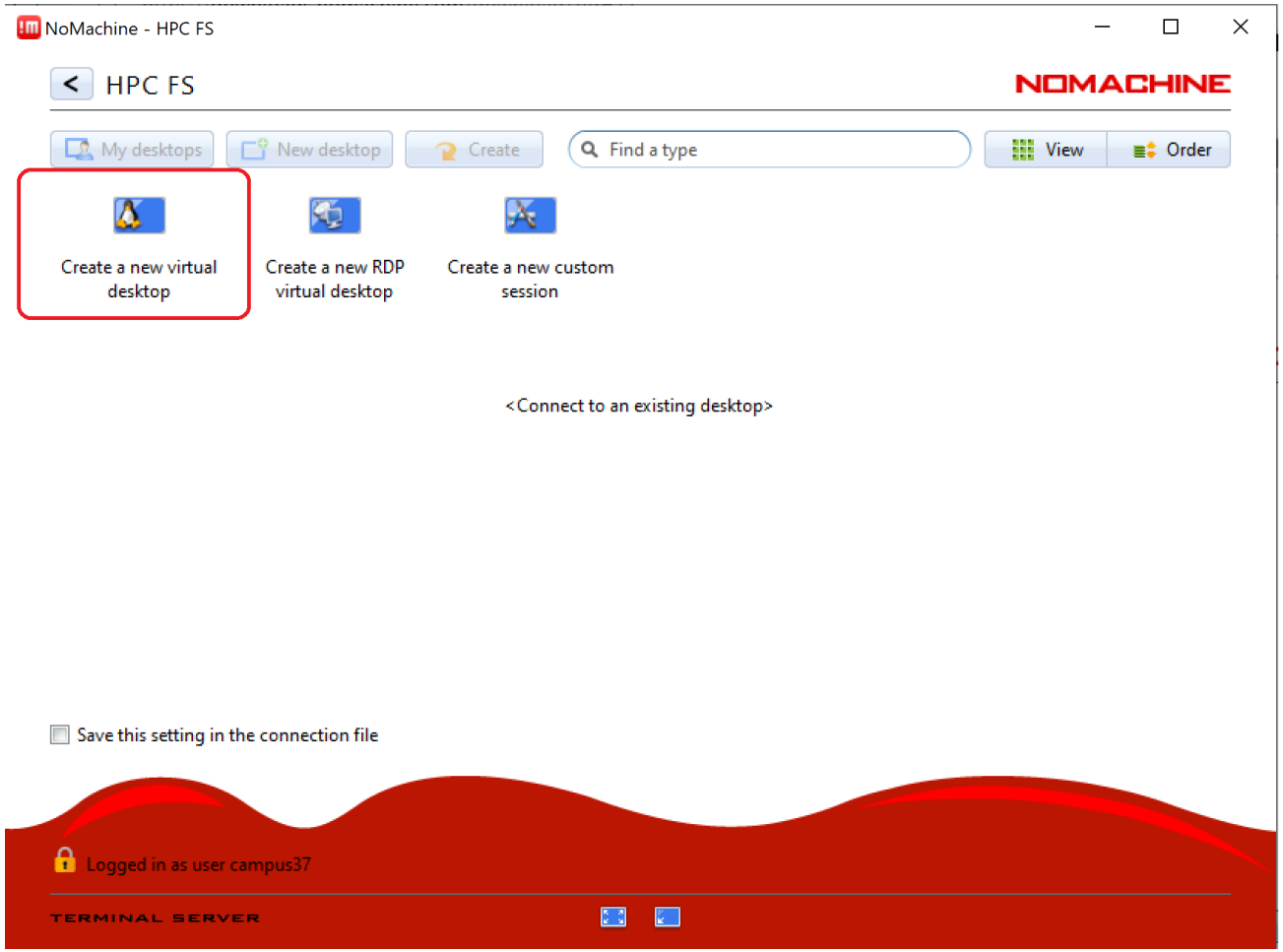
Skip autentification warning



Enter credentials

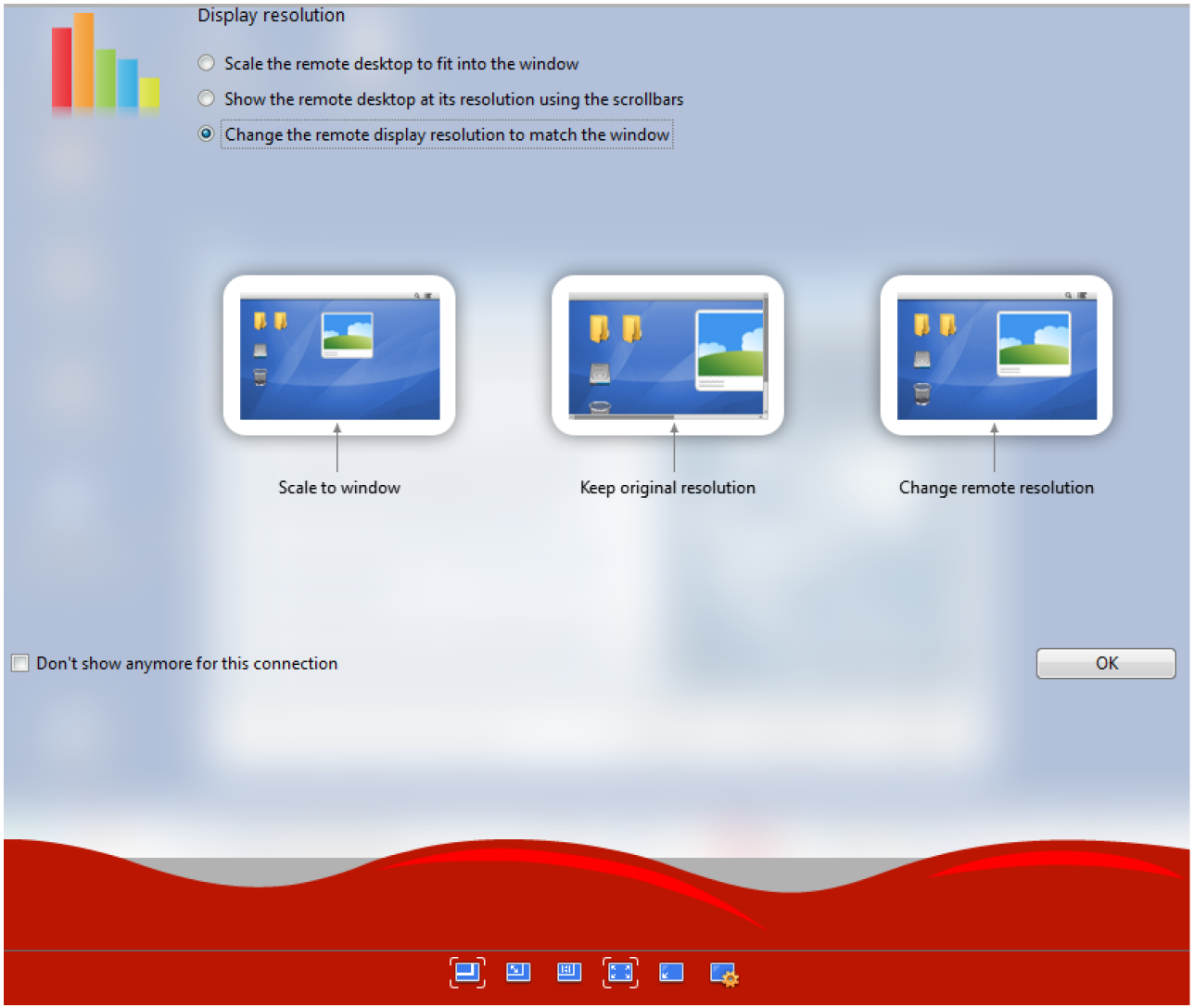


Create new virtual desktop



Define remote desktop setting

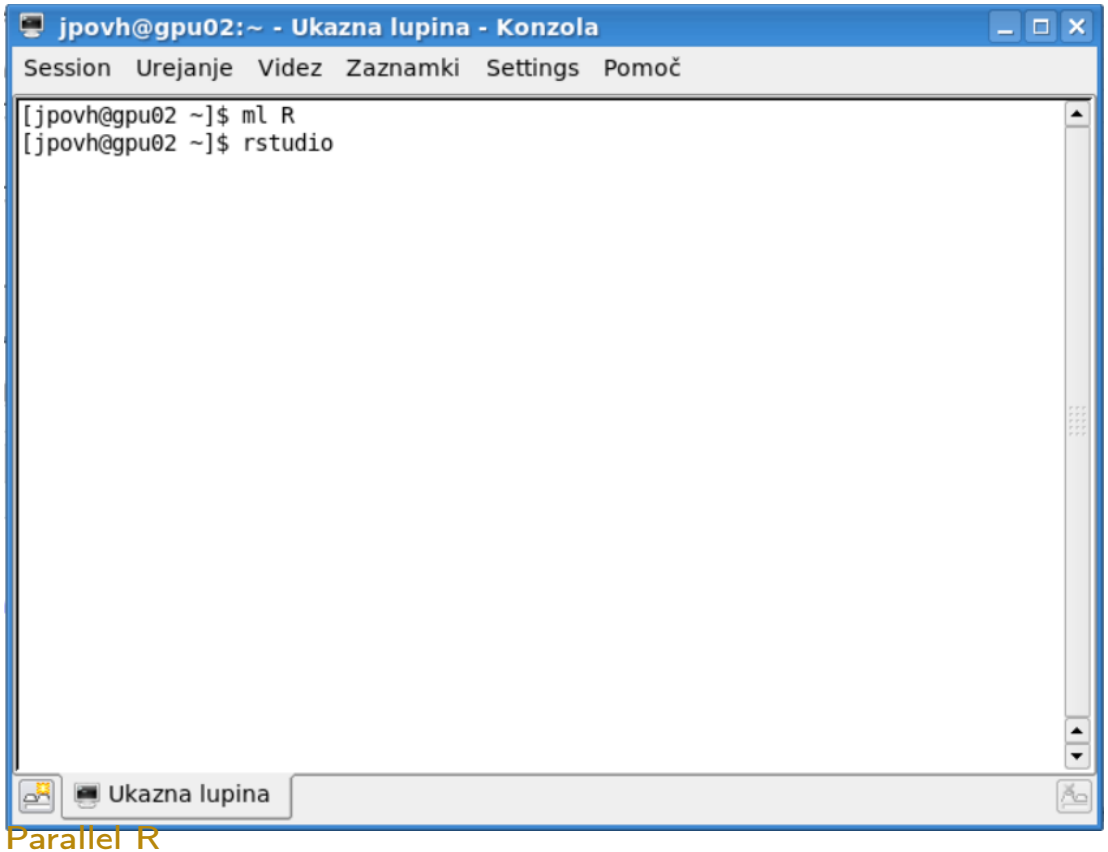
Define remote desktop setting



Konzola/console/terminal window

Open console and type

- `module load R`
- `R`



Connect to compute node

Type to console

```
LD_PRELOAD= srun -p haswell --pty --x11 bash -i
module load RStudio-Server/2022.07.2-576-foss-2022a-Java-11-R-4.2.1
module help RStudio-Server/2022.07.2-576-foss-2022a-Java-11-R-4.2.1
```

Connect to compute node

Type to console

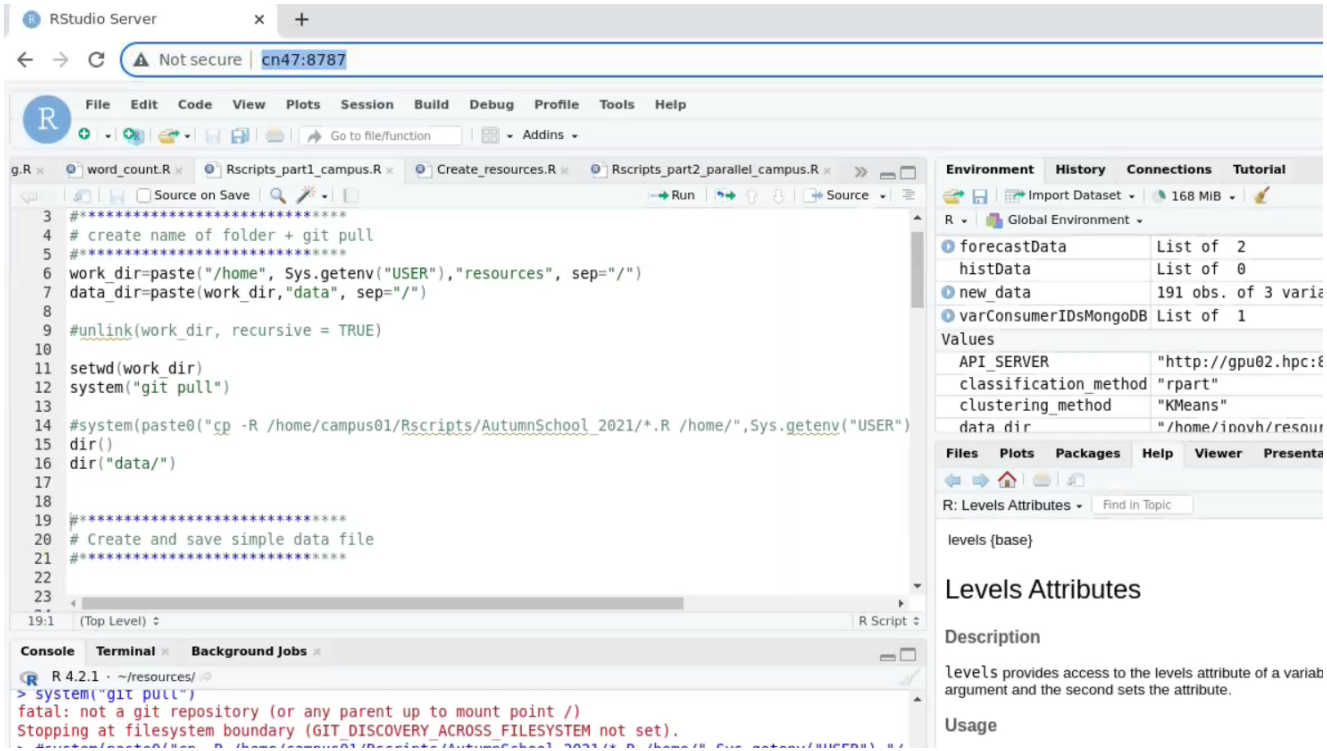
```
LD_PRELOAD= srun -p haswell --pty --x11 bash -i
module load RStudio-Server/2022.07.2-576-foss-2022a-Java-11-R-4.2.1
module help RStudio-Server/2022.07.2-576-foss-2022a-Java-11-R-4.2.1
```

Copy the following text to command line

```
[jpovh@cn61 ~]$  module help RStudio-Server/2022.07.2-576-foss-2022a-Java-11-R-4.2.1

----------------------------------------------------- Module Specific Help for "RStudio-Server/2022.07.2-576-foss-2022a-Java-11-R-4.2.1" -----

Description
===========
This is the RStudio Server version.
RStudio is a set of integrated tools designed to help you be more productive with R.

The server can be started with:
 MYTMP=`mktemp -d` && trap "rm -rf ${MYTMP}" INT QUIT ABRT KILL TERM CHLD && \
  echo -e "provider=sqlite\ndirectory=${MYTMP}/db" > ${MYTMP}/db.conf &&     \
  rserver --server-daemonize=0 --www-port=8787 --rsession-which-r=$(which R) \
   --server-user=${USER} --secure-cookie-key-file=${MYTMP}/secure-cookie-key \
   --server-data-dir=${MYTMP}/sdd --database-config-file=${MYTMP}/db.conf


More information
================
 - Homepage: https://www.rstudio.com/
```

Run Rstudio on compute node

- open web browser (chrome or firefow)
- type in `http://cnXX:8787/` - XX is the name of compute node where you are



GITHUB

Clone data for today from GITHUB

`https://github.com/janezpovh/Parallel_R_Jan_24.git}`

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]
```

Load and analyse the data

Load data

```
  # 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)
}
```

Analyse the data with dplyr, magrittr

- Library dplyr: "select", "filter", "group_by","arrange", "mutate" and "summarize".
- Library magrittr: "%>%"

dplyr

```
llibrary(dplyr)
library(magrittr)
Tab_summary1<-group_by(Data_read,group) %>% dplyr::summarise(count_ints=n(),mean_ints=
     mean(ints))

# other operations on rows and columns
Data_read_group_ints<-Data_read %>% select(group,ints)
# add new variable reals/ints
Data_read<-mutate(Data_read,ratio=reals/ints)
Data_read<-Data_read %>% mutate(ratio1=ints/reals)
#arrange
#sort accordind to increasing group
Data_read<-Data_read %>% arrange(desc(group))
Data_read<-Data_read %>% arrange(group)
```

Analyze the data by split, aggregate, sapply

split, aggregate, sapply

```
s <- split(Data_read, Data_read$group)
Tab_summary1<-t(sapply(s, function(x) return(c(mean(x$ints),length(x$group)) )))

Tab_summary2<-cbind(aggregate(ints~group,data = Data_read,FUN=length),aggregate(ints~
     group,data = Data_read,FUN=mean))
Tab_summary2<-Tab_summary2[,-3]
```

Outline/next

- Introduction to R
- Advanced and Big data management with R
- Parallelization with Rmpi

Advanced and Big data management with R

What system do I have

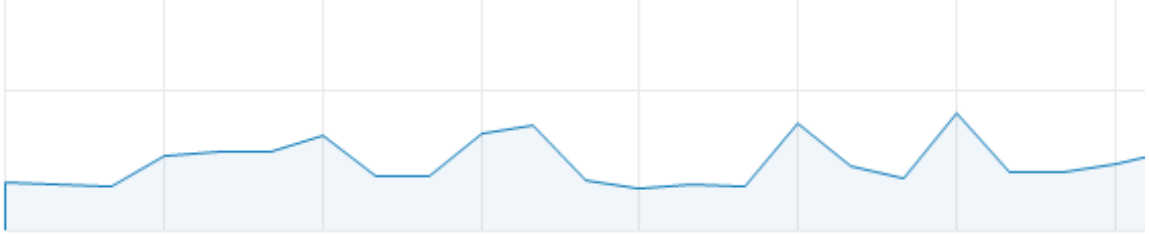
How many cores

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

What system do I have

How many cores

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



60 seconds

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

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)
```

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 12 random matrices of order 3000×3000

for

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

foreach do loop

Let us compute sums of all elements of 12 random matrices of order 3000×3000

for

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

Parallel foreach dopar loop

Let us compute sums of all elements of 12 random matrices of order 3000×3000 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 12 random matrices of order 3000×3000 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?

Parallel foreach dopar loop

Let us compute sums of all elements of 12 random matrices of order 3000×3000 using `foreach ...dopar` from `foreach, doParallel`. Create cluster!

for

```
N=3000
set.seed(2021)
registerDoParallel(12)  # use multicore, set to the number of our cores - needed for
     foerach dopar

sum_rand=rep(0,11);
tic()
foreach (i = c(1:12)) %dopar% {
   A=randn(N,N)
   sum_rand[i]=sum(A)
}
time_foreach_dopar_1=toc()
registerDoSEQ()
```

Parallel foreach dopar loop

Let us compute sums of all elements of 12 random matrices of order 3000×3000 using `foreach ...dopar` from `foreach, doParallel`. Create cluster!

for

```
N=3000
set.seed(2021)
registerDoParallel(12)  # use multicore, set to the number of our cores - needed for
     foerach dopar

sum_rand=rep(0,11);
tic()
foreach (i = c(1:12)) %dopar% {
   A=randn(N,N)
   sum_rand[i]=sum(A)
}
time_foreach_dopar_1=toc()
registerDoSEQ()
```

Do you observe any difference?

Library parallel

- encapsulates existing libraries `multicore, snow`
- 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 12 random matrices of order 3000×3000. Create cluster!

parallel versions of apply

```
mat_sum<-function(x){
  A=rand(x)
  return(sum(A))
}
tic()
time_lapply<-system.time({
  set.seed(2021)
  sum_rand_lapply=lapply(rep(3000,12),FUN=mat_sum)
  time_lapply=toc()
})

time_sapply<-system.time({
  set.seed(2021)
  sum_rand_sapply=sapply(rep(3000,12),FUN=mat_sum)
})
```

Parallel versions of lapply

parallel versions of apply

```
time_mcLapply<-system.time({
  set.seed(2021)
  sum_rand_mcLapply=mclapply(X=rep(3000,12),FUN=mat_sum,mc.cores = 12)
})

time_parLapply<-system.time({
  clust <- makeCluster(12, type="PSOCK")
  set.seed(2021)
  sum_rand_parLapply=parLapply(cl,rep(3000,1000),fun=mat_sum)
  stopCluster(clust)
})

time_parSapply<-system.time({
  clust <- makeCluster(12, type="PSOCK")
  set.seed(2021)
  sum_rand_parSapply=parSapply(cl,rep(3000,20),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        5.120    0.954   6.072
time_sapply        5.049    0.885   5.932
time_parLapply     0.076    0.209  47.999
time_parSapply     0.021    0.105   4.286
time_mcLapply      0.003    0.040   0.531
```

Libraries for shared memory parallelization in R

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

Outline/next

- Introduction to R
- Advanced and Big data management with R
- Parallelization with Rmpi

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 $\{0, 1, 2, \ldots, \text{size} - 1\}$
- `Rmpi::mpi.get.processor.name()` - returns the name of compute node where the process

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
```

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)
```