R with future

Fabian Freund, Brigitte Wellenkamp
(KIM Hohenheim, bwHPC)

July 26th, 2022
ZOOM

UNIVERSITY OF HOHENHEIM bw|HPC

Parallel computation in general

Computing in parallel: “Separate” computations are performed on different threads/cores

- Parallel without communication
- Parallel with communication
- in R: opens new instances of R (forked, socket cluster, MPI, via SLURM jobs)

Other specifics:

- Parallelisation over CPUs vs. over GPUs (latter not covered today)
- Multithreading: Parallelisation with shared memory, typically working on the same (big) task

Parallelisation without communication

- No communication needed (only collection of results)
  - Running script on different data sets
  - Simulations (distribute replications and/or parameters sets)

Main issues when computing in parallel

- Load balancing
- Copying of objects between R instances
- (Nested parallelisms)
- Setting seeds
- (Overhead of parallelisation approach)

Working on bwUniCluster

To run the code on bwUniCluster, open an interactive session

```
salloc -t 120 -p single -n 2
module load math/R/4.1.2
```

Part 1 and part 2 of this course should also run on most local computers (if > 1 CPU).

Prerequisites

Install the following R packages on bwUniCluster (we use R 4.1.2)

```
install.packages(c("future","parallelly",
                   "future.apply","doFuture",
                   "future.batchtools",
                   "tictoc")) #tictoc for very easy timing
```

futureverse project leader: Henrik Bengtsson (Department of Epidemiology & Biostatistics at UCSF)

Homepage, latest releases on CRAN.

Main package: future

R libraries needed today

```r
library("future")
library("parallelly")
library("future.apply")
library("doFuture")
```

```
## Loading required package: foreach
```

```r
library("future.batchtools")
library("tictoc")
```

The idea behind R future

- R usually allocates and computes in one step
- future wraps the assignment up in a new object - which can then be computed/resolved on any R process
  - (and NOT NECESSARILY IN THE ONE IT WAS DEFINED IN).

```
library(future)
x <- 1
mode(x)
```

```
## [1] "numeric"
```

```
future_x <- future(1,lazy = TRUE)
mode(future_x)
```

```
## [1] "environment"
```

```
mode(value(future_x))
```

```
## [1] "numeric"
```

General scheme of future

- Any `future(...)` you run can be run in serial or parallel (forked, socket cluster,. . . , via SLURM)
- You simply specify how any futures are run by `plan(...)`. This is called the parallel resp. serial backend
- Seeds can be set in such a way that results stay reproducible regardless of backend (via l'Ecuyer RNG)

⇒ Code stays the same when you switch the backend, essentially you only change `plan(...)`

Some interesting plans on a single node

- `plan(strategy=sequential)`: Serial execution of futures within the main R session
- `plan(strategy=multicore,workers=n)`: Forks n workers from main R (not on Windows, not in GUIs as Rstudio)
- `plan(multisession,workers=n)`: n separate (background) R processes as workers
- The main R waits for all futures to be distributed across workers - main R only blocked after this if $\#$ workers $<$ $\#$ futures
- Change the `plan` by simply invoking a different `plan`

Details on future

```
str(future)
```

- lazy: Should future(...) be evaluated asap or only when queried for value?
- seed (default=FALSE): Should seeds be set (via L'Ecruyer RNG, reproducible regardless of backend used)
- globals: control over R objects that future needs from the global environment (auto-identified by default)
- packages: specific packages needed to evaluate the future

The difference between futures and assign (aka <-)

```r
#availableCores(); availableWorkers()
plan(strategy=multisession,workers=4)
testf1 <- function(){Sys.sleep(6);return(Sys.getpid())}
s1 <- replicate(3,future(testf1()))
sapply(s1,resolved)
```

```
## [1] FALSE FALSE FALSE
```

```r
sapply(s1,value)
```

```
## [1] 24236 24237 24235
```

```r
replicate(3,value(future(testf1())))
```

```
## [1] 24236 24236 24236
```

Exercise: Does this behave differently if workers=2?

The difference between futures and assign (aka <-)

```
#availableCores(); availableWorkers()
plan(strategy=multisession,workers=4)
testf1 <- function(){Sys.sleep(6);return(Sys.getpid())}
s1 <- replicate(3,future(testf1()))
sapply(s1,resolved)
```

```
## [1] FALSE FALSE FALSE
```

```
sapply(s1,value)
```

```
## [1] 24236 24237 24235
```

```
replicate(3,value(future(testf1())))
```

```
## [1] 24236 24236 24236
```

Exercise: Does this behave differently if workers=2? Check package ‘promises‘ for non-blocking futures

Futures make any loop or vectorization parallel

```
set.seed(44)
testf1 <- function(){Sys.sleep(sample(6))
                     return(Sys.getpid())}
res1 <- vector("list",10)
for (i in 1:10){
res1[[i]] <- future(testf1(),seed=TRUE)
}
str(res1[[1]])
```

```
## Classes 'MultisessionFuture', 'ClusterFuture', 'Multipro
```

```
res2 <- sapply(res1,value)
table(res2)
```

```
## res2
## 24234 24235 24236 24237
##     3     3     2     2
```

"Exercise": Use vectorization (*apply) instead of the loop

Timing via R package tictoc

```
tic()
Sys.sleep(3) #Any code
toc()
```

```
## 3.004 sec elapsed
```

```
tic()
Sys.sleep(3)
toc()
```

```
## 3.006 sec elapsed
```

A test function and some R objects

```r
test_node <- function(i,exportsth=NULL){
  str(exportsth) #Do sth. cheap w. object
  p1 <- date() #get date
  sleep_t <- sample(10,1) #random sleep time
  Sys.sleep(sleep_t) #sleep in R
  x <- Sys.info() #System info, including host name
  return(c(run=i,time_start=p1,time_end=date(),
           pid=Sys.getpid(),host=x["nodename"],
           sleep_time=sleep_t))}

small_thing <- "cookies"
big_thing <- matrix(rnorm(1e6),nrow=1000)
```

Exercise: Compare future's `plans`

Test the behaviour of plans `multisession` and `multicore` with 3 workers

- Use the test function `test_node` and "force" it to import a big or small object
- Use some method to run several (4) instances of this test (which then get distributed via `plan`)
- Time the executions
- Are there differences? How is the load balancing?

Hints: general structure

```
library(....), ...

#For each plan to test

tic()
set.seed(...)
plan(...)

LOOP i 1:4 start
... <- future(...(i),seed=TRUE)
LOOP end

LOOP start
value(future i)
LOOP end

toc()
```