



Parallel Computing in R

from `parallel` to `foreach` and future

Chao Cheng

School of Statistics and Management
Shanghai University of Finance and Economics, China

November 25, 2022





Outline

- 1 Parallel Computation in R
 - Introduction
 - The Parallel package
 - Advanced topics
 - Easy parallel computation
 - Other stuff
- 2 Acknowledgement





Introduction

A toy example

- $1 + 2 + \dots + 100$
- Compute the sum of a vector

Abstraction

- The whole job can be break into small parts and they can be done independently of each other.
- Map + Reduce

Useful cases

- Simulation
- Bootstrap, MCMC and cross validation, etc.
- Elementwisely update an vector in ADMM algorithm (Parallel in C++)



Basic parallel computation for simulation

- Start multiple R sessions
- Preparation: load necessary packages, etc.
- Run simulation scripts, possibly according to session ID.
- Collect and summary the results by hand.

Abstraction

- Create workers
- Prepare workers
- Run script in parallel and collect the results.





The parallel package

- It's derived from `snow` and `multicore` packages.
- Useful reference:
 - Parallel R. (This book is a bit old.)
 - `parallel`'s documentations.
 - `parallel`'s vignettes.





A simple template

```
library(parallel)
# use all the cores of this machine
cls <- makeCluster(detectCores())

# initializing workers
clusterEvalQ(cls, fun)
# pass VARLIST from master to all the workers
clusterExport(cls, VARLIST)

# split full index of all tasks to workers
idx_split <- clusterSplit(cls, idx_full)
# carry out the task parFUNCTION parallely
res <- parLapply(cls, idx_split, parFUNCTION)

# stop workers
stopCluster(cls)
```



A simple example

```
library(parallel)
a <- rnorm(12)
slow_function <- function(invec){
  ...      # a slow function
}

cls <- makeCluster(4)
ind_seq <- clusterSplit(cls, a)
clusterExport(cls, varlist = "slow_function")
res_par <- parSapply(cls, ind_seq, slow_function)
res <- sum(res_par)
```





Rabbit hole 1: PSOCK vs FORK

```
a <- rnorm(100)
```

PSOCK

```
cls <- makeCluster(4)      # default type is PSOCK
```

FORK

```
cls <- makeCluster(4, type = "FORK")    # NOT available on Windows
```

```
parSapply(cls, 1 : 10, function(id){  
  return(a[id])  
})
```





Rabbit hole 1: PSOCK vs FORK

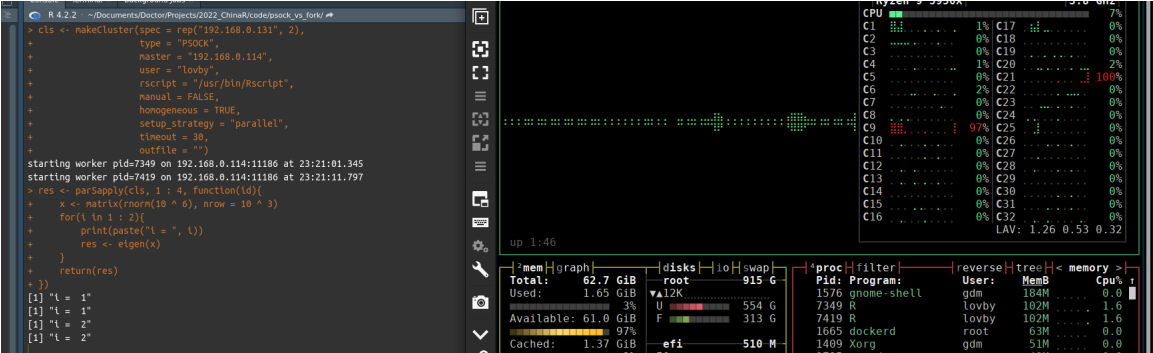


Figure 1: PSOCK connects to remote server



Rabbit hole 1: PSOCK vs FORK

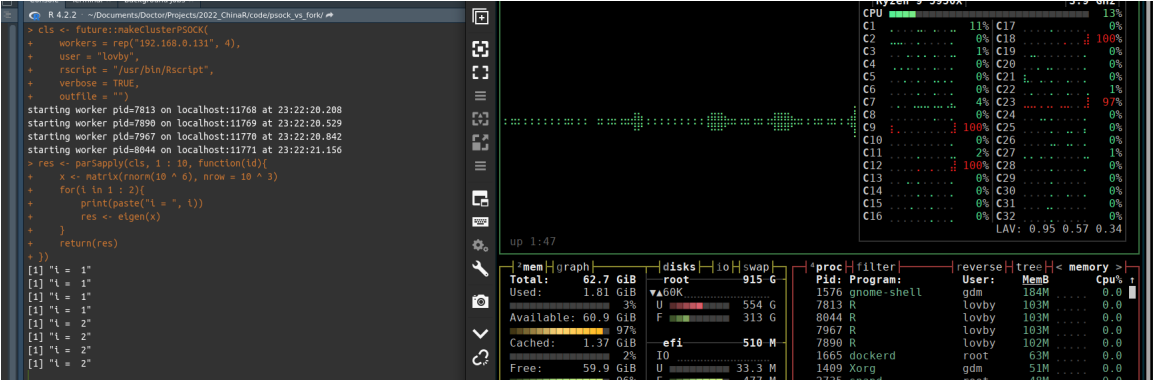


Figure 2: PSOCK (via future/parallelly) connects to remote server



Rabbit hole 1: PSOCK vs FORK



Figure 3: Memory consumption of FORK, part 1



Rabbit hole 1: PSOCK vs FORK

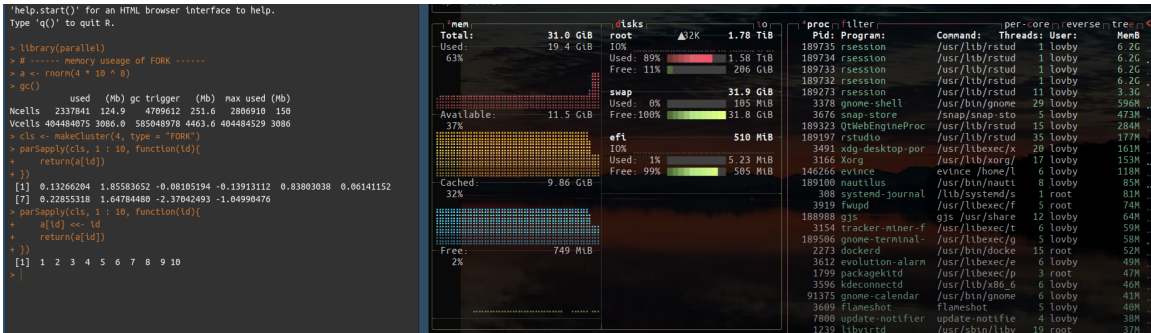


Figure 4: Memory consumption of FORK, part 2



Rabbit hole 1: PSOCK vs FORK

PSOCK

- Pros:
 - Use socket connection, a general approach.
 - All system, locally or remotely with suitable setup such as MPI
- Cons:
 - Might be hard to configure.
 - Manually transport the data.

FORK

- Pros: use FORK mechanism, no worry about variable transportation.
- Cons: Only for one machine, not available on Windows.



Rabbit hole 2: Parallel random number generation

```
> # --- PSOCK will have their own seed ---
> cls <- makeCluster(4)
> parSapply(cls, 1 : 4, function(id){
+   rnorm(10)
+ })
```

	[,1]	[,2]	[,3]	[,4]
[1,]	-0.1284523	0.90577746	-0.2007189	1.5838963
[2,]	-0.2812603	-0.12137209	-0.5616048	2.1205495
[3,]	-1.4523742	-0.82587203	-1.0321580	-0.7770089
[4,]	0.2909059	0.60436821	0.1179803	-1.4674607
[5,]	0.5116825	2.42765426	-1.3673113	1.5117072
[6,]	-1.7666148	0.50982495	0.5123056	0.3701816
[7,]	0.5683548	0.55647039	1.1734904	0.7364532
[8,]	0.6602347	-0.09715067	1.3660750	1.1635798
[9,]	0.1688014	-0.47356787	-1.5571017	0.4500263
[10,]	1.0467846	0.07022752	0.6334269	-0.5432841

(a) No control on PSOCK

```
> # --- fork will copy the seed setting of main session ---
> set.seed(10)
> cls <- makeCluster(4, type = "FORK")
> parSapply(cls, 1 : 4, function(id){
+   rnorm(10)
+ })
```

	[,1]	[,2]	[,3]	[,4]
[1,]	0.01874617	0.01874617	0.01874617	0.01874617
[2,]	-0.18425254	-0.18425254	-0.18425254	-0.18425254
[3,]	-1.37133055	-1.37133055	-1.37133055	-1.37133055
[4,]	-0.59916772	-0.59916772	-0.59916772	-0.59916772
[5,]	0.29454513	0.29454513	0.29454513	0.29454513
[6,]	0.38979430	0.38979430	0.38979430	0.38979430
[7,]	-1.20807618	-1.20807618	-1.20807618	-1.20807618
[8,]	-0.36367602	-0.36367602	-0.36367602	-0.36367602
[9,]	-1.62667268	-1.62667268	-1.62667268	-1.62667268

(b) Same seed on FORK



Rabbit hole 2: Parallel random number generation

- Manually use `set.seed()` on every worker. **not recommended**
- Use 'L'Ecuyer-CMRG' multiple RNG stream.
 - `RNGkind("L'Ecuyer-CMRG")` on your main session.
 - `set.seed()` on your main session.
 - `clusterSetupRNGstream()` to set your workers' seed.





Rabbit hole 3: Variable transportation

- Explicit functions and variables will always be transported.
- FORK will copy the main session at creation.
- Others should be taken care of by hand.
- Additional configuration of `clusterExport` when nested in a function call.





How many frameworks can one developer maintains?

There are so many different parallel backends:

- snow
- multicore
- parallel
- MPI
- Redis
- Hadoop
- Spark
- Slurm
- ...

How to support them? How to maintain code?





Foreach

`foreach` defines a simple but powerful framework for map/reduce parallel computation.

Package author/code writer

Decide **which part of code** can run in parallel.

End user

Decide **how** to run in parallel based on their available resources.

`foreach` is syntactically structured in the form of a for loop. But actually it works like **-apply** functions.





Foreach

```
library(foreach)
# library(doParallel)
# registerDoParallel()
a <- 10
foreach(i = 1 : 12, j = 12 : 1, .combine = rbind) %dopar%{
  Sys.sleep(0.5)
  # print will be dropped when run in parallel
  print(paste("i = ", i, ", j = ", j, sep = ""))
  # 'a' is passed automatically
  data.frame(i, j, a)
}
```





future and future.apply

- `future` provides a simple and uniform way of evaluating R expressions asynchronously using various resources available to the user.
- `future.apply` provides worry-free parallel alternatives to base-R "apply" functions.

```
library(future.apply)      # default plan is sequential
# plan(cluster)
x <- rnorm(16)
future_lapply(1 : 5, function(id){
  print(paste("id = ", id, sep = " "))      # normal print kept
  Sys.sleep(0.5)
  sum(x[1 : id])
})
```





- Asynchronous computation. Not constrained by a for-loop or apply syntax.
- Available extensions:
 - `future.apply`
 - `doFuture`: **backends for** `foreach`, `BiocParallel` **and** `plyr`.
 - `furrr`





future

```
library(future)
plan(cluster)

x <- future({
  x <- matrix(rnorm(10 ^ 6), nrow = 10 ^ 3)
  for(i in 1 : 5){
    print(paste("i = ", i))
    res <- eigen(x)
  }
  return(res)
}, seed = T)      # not block the main session

resolved(x)      # check whether the future is resolved
a <- rnorm(10)    # we can do other stuff at the main session
```





Personal suggestions

- Nested parallel is **NOT** recommended. At least it should be done with careful configuration.
- `future.apply` **VS** `foreach`
 - Familiar with `foreach`: just use the `doFuture` backends.
 - New to parallel: `future.apply` is a good start point for your code.
 - `future` backend will relay the printed messages.
 - Performance in parallel are close, **so-called**.
 - Performance for sequential are slower than for-loop.





I want progress bars

- `RcppProgress` allows to display a progress bar in the R console for long running computations taking place in c++ code, supports OpenMP.
- `pbapply` is a lightweight package that adds progress bar to vectorized R functions ('*apply'). It supports several parallel backends.
- `progress` shows ASCII progress bars.
- `progressr` provides a minimal API for reporting progress updates in R.
 - Developer is responsible for providing progress updates.
 - End user decides if, when, and how progress should be presented.





progressr

```
library(progressr)

slow_sum <- function(x) {
  p <- progressr::progressor(along = x)
  sum <- 0
  for (kk in seq_along(x)) {
    Sys.sleep(0.5)
    sum <- sum + x[kk]
    p(message = sprintf("Added %g", x[kk]))
  }
  sum
}

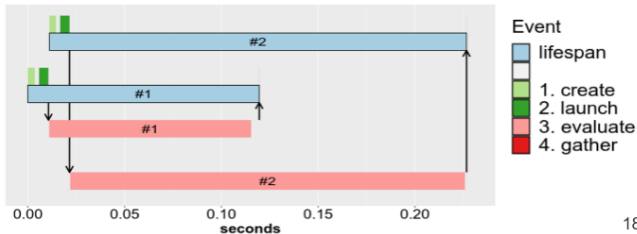
# handlers("default")      # default handler is "txtprogressbar"
with_progress(y <- slow_sum(1:10))
handlers("progress")
with_progress(y <- slow_sum(1:10))
```



Profile parallel code (WIP)

Profiling 2 futures with 2 workers

```
plan(cluster, workers = 2)
fs <- lapply(1:2, function(x) future(slow(x))
vs <- value(fs)
```



18

Figure 6: Profile parallel code

Figure source at <https://www.jottr.org/2022/06/23/future-user2022-slides/>



Profile parallel code (WIP)

```
remotes::install_github(  
  "git@github.com:HenrikBengtsson/future.git",  
  ref = "9875992", force = TRUE)
```

```
> plan(cluster, workers = 2)  
> slow_fcn <- function(x) {  
+   Sys.sleep(x / 10)  
+   sqrt(x)  
+ }  
>  
> js <- capture_journals({  
+   fs <- lapply(3:1, FUN = function(x) future(slow_fcn(x)))  
+   value(fs)  
+ })  
> lapply(js, function(indata){lndata[1 : 6, c("event", "type", "parent", "duration")]})  
[[1]]  
      event      type parent      duration  
1      create overhead  <NA> 2.614021e-03 secs  
6      launch overhead  <NA> 8.135939e-02 secs  
2      getWorker overhead launch 1.218319e-04 secs  
3      eraseWorker overhead launch 7.087874e-02 secs  
4 attachPackages overhead launch 4.148483e-05 secs  
5 exportGlobals overhead launch 8.690357e-04 secs  
  
[[2]]  
      event      type parent      duration  
1      create overhead  <NA> 2.676725e-03 secs  
6      launch overhead  <NA> 4.583597e-03 secs  
2      getWorker overhead launch 1.337528e-04 secs  
3      eraseWorker overhead launch 4.656315e-04 secs  
4 attachPackages overhead launch 3.361702e-05 secs  
5 exportGlobals overhead launch 6.408691e-04 secs
```





Profile parallel code (WIP)

```
library(future)
n_worker <- 3
n <- 6

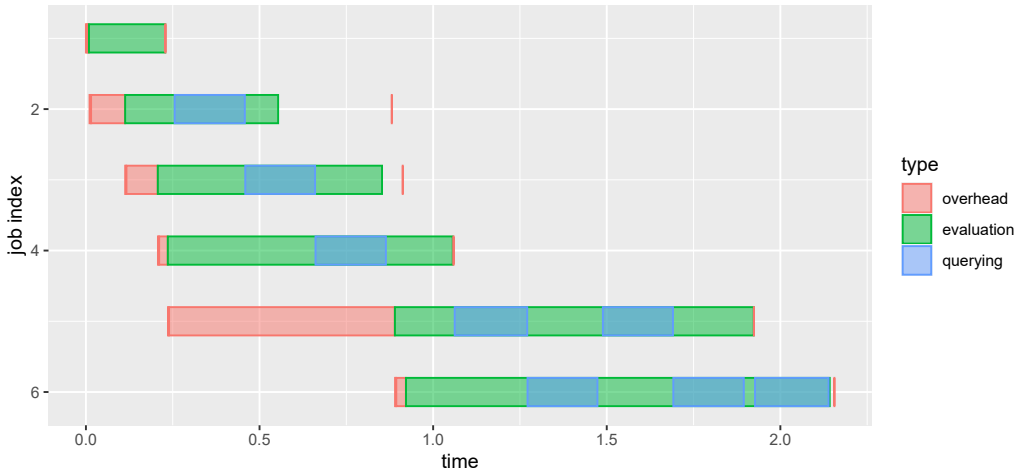
plan(cluster, workers = n_worker)
slow <- function(x) {
  Sys.sleep(x / 5)
  return(x)
}

tmp <- capture_journals({
  fs <- lapply(1 : n, function(x) future(slow(x)))
  vs <- value(fs)
})
```



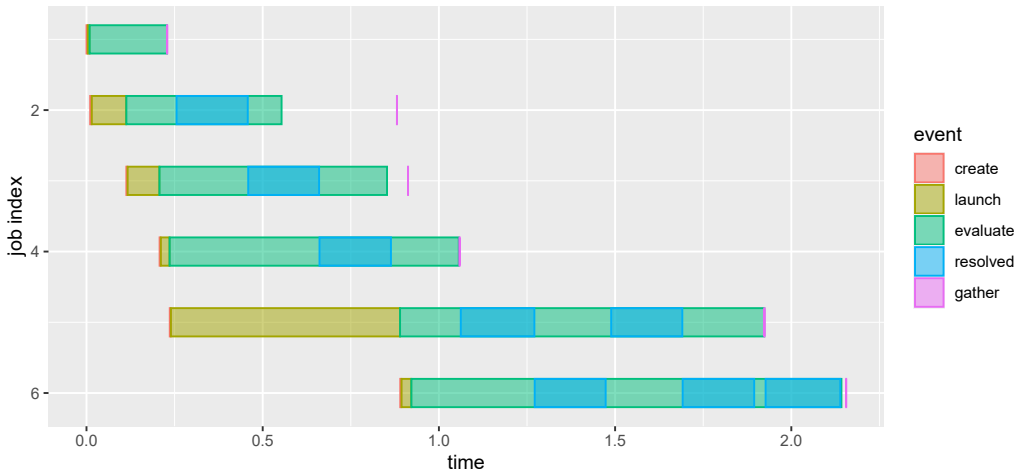
Profile parallel code (WIP)

Job num: 6. Worker num: 3



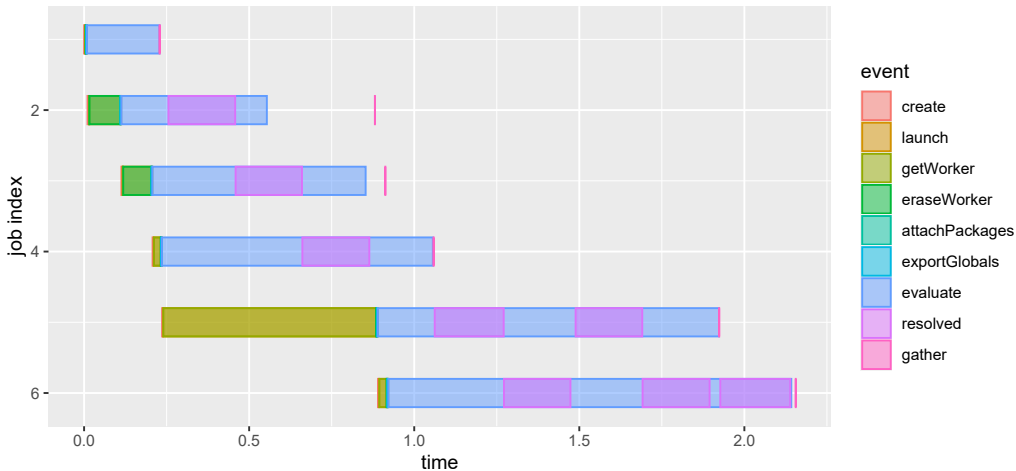
Profile parallel code (WIP)

Job num: 6. Worker num: 3



Profile parallel code (WIP)

Job num: 6. Worker num: 3





Parallel is NOT all you need

How about we use $\frac{(1+n) \times n}{2}$?





Acknowledgement



Parallel computing with R
using foreach, future, and
other packages

BRYAN LEWIS



Future: Simple Async, Parallel
& Distributed Processing in R -
What's Next?

HENRIK BENGTSSON

