Table of Contents

Parallelization in R 1

Parallelization in R

Here comes a brief introduction on how to run an R code in parallel on multiple cores. Depends, of course, whether your computer's processor (CPU) has more than one core (here you can find how to figure out the number of cores on Windows).

Parallel calculation in R works in this way (hugely simplified description):

1. Initiate the `library (parallel)` (or other libraries supporting parallel backend).
2. Created several "workers" (usually one worker = one core).
3. The workers are a bit like a newly launched instance of R - they don't know anything, their global environment has no variables and no uploaded libraries, so whatever you want them to do, you need to supply them (objects, libraries etc.)
4. Use some of the functions which are able to run the loop in parallel. E.g. most of the functions like `apply`, `lapply`, `sapply` have their parallel alternatives: `parApply`, `parLapply`, `parSapply`. These functions will distribute the job evenly among workers, collect results from them and output them out.
5. When you finish using parallel backend and you don't need workers any more, stop them, otherwise, they will keep hanging in the memory.

If we stick to using `library (parallel)`, then the simplified description of the script looks like this:

1.
```
library (parallel)
```

2.
```
cl <- makeCluster (4)
```

   - this creates four workers (clusters), and assigns their description to variable `cl`, which will be important further - all functions able to run parallel need this description so as they know where the workers actually are and how many of them;

3.
```
clusterExport (cl, varlist = c("object1", "object2"))
```

   - this function exports the variables (here named `object1` and `object2`, but it could be whatever) from current R instance into the global environment of newly created workers - so as they could use them. Note that the first argument is the description of workers `cl`)

4.
```
result <- parSapply (cl, some.vector, FUN = function (i)
{some.function1; some.function2})
```

   - this is one of the functions which is running in parallel, analogy of `sapply`. Note that the first argument is again the description of workers. It takes elements in `some.vector` and evenly ditributes them among workers; then it process the `some.function1` and `some.function2` on these workers, and collects results - here into variable `results`.

5.
```
stopCluster (cl)
```

- this stops the workers and closes the parallel backend.

Let's demonstrate the parallelization on the example of a function which generates a large number (10^6) of random values from the normal distribution and then calculates their mean; this is replicated many times (100 times here).

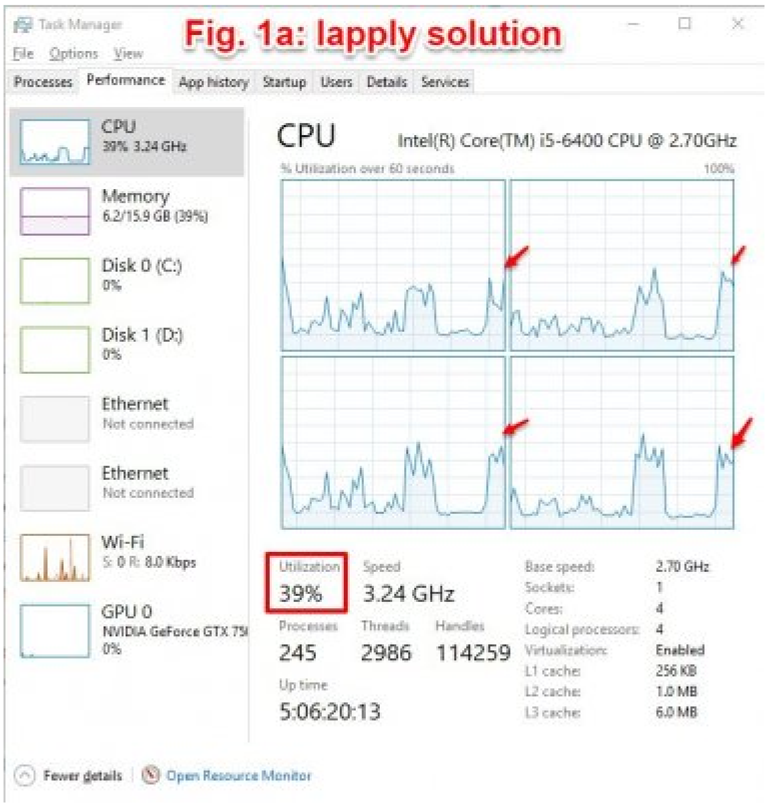
```
lapply (1:100, FUN = function (x) mean (rnorm (1000000)))
```

Parallel version (using library `parallel`) looks like this:

```
library (parallel)
cl <- makeCluster (4)
res <- parLapply (cl, X = 1:100, fun = function (x) mean (rnorm (1000000)))
stopCluster (cl)
```

Both functions return the list where each item is the mean value (all are close to zero since this is mean of the normal distribution with zero mean and unit standard deviation).

What is happening at the computer if you run `lapply` vs `parLapply` solution? For this, I had a looked into the *Windows Task Manager*, the folder *Processes* and *Performance* (you get there by right-clicking on the bottom toolbar of the Windows and choosing the option Task manager, see e.g. here). The `lapply` solution uses only part of the computer capacity - in my example, it is using ~ 39% of CPU, which (given that if I do any calculation the CPU runs on 10-15%) represents approximately one-fourth of the CPU capacity (my CPU has 4 cores, so it is like if it runs only on a single core, Figs 1a and 1b below). In case of `parLapply` the situation is different - first, four instances of R program are launched (which is possible to see in the Process window of the Task manager, Fig. 2a), and CPU is running on 100% of capacity, with each R instance running on roughly 25% (Fig. 2b).


Task Manager
Fig. 1a: lapply solution
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
Open Resource Monitor
% Utilization over 60 seconds
100%

CPU 39% 3.24 GHz
Memory 6.2/15.9 GB (39%)
Disk 0 (C:) 0%
Disk 1 (D:) 0%
Ethernet Not connected
Ethernet Not connected
Wi-Fi S: 0 R: 8.0 Kbps
GPU 0 NVIDIA GeForce GTX 75 0%

CPU Intel(R) Core(TM) i5-6400 CPU @ 2.70GHz

Utilization 39%
Speed 3.24 GHz
Processes 245
Threads 2986
Handles 114259
Up time 5:06:20:13

Base speed: 2.70 GHz
Sockets: 1
Cores: 4
Logical processors: 4
Virtualization: Enabled
L1 cache: 256 KB
L2 cache: 1.0 MB
L3 cache: 6.0 MB

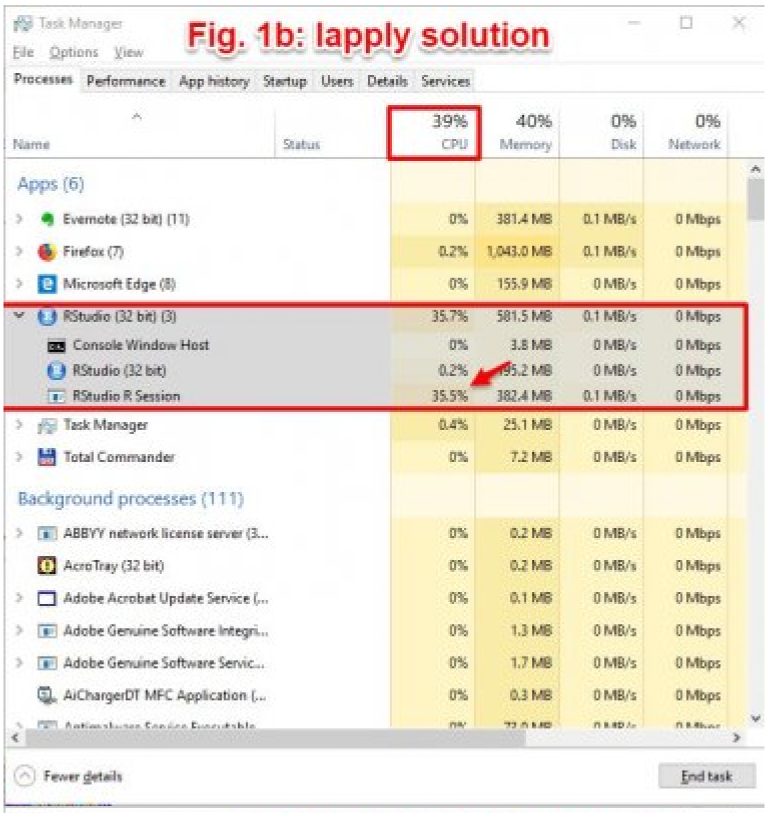


Fig. 1b: lapply solution

Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
End task

Name	Status	39% CPU	40% Memory	0% Disk	0% Network
Apps (6)					
Evernote (32 bit) (11)		0%	381.4 MB	0.1 MB/s	0 Mbps
Firefox (7)		0.2%	1,043.0 MB	0.1 MB/s	0 Mbps
Microsoft Edge (8)		0%	155.9 MB	0 MB/s	0 Mbps
RStudio (32 bit) (3)		35.7%	581.5 MB	0.1 MB/s	0 Mbps
Console Window Host		0%	3.8 MB	0 MB/s	0 Mbps
RStudio (32 bit)		0.2%	[illegible]5.2 MB	0 MB/s	0 Mbps
RStudio R Session		35.5%	382.4 MB	0.1 MB/s	0 Mbps
Task Manager		0.4%	25.1 MB	0 MB/s	0 Mbps
Total Commander		0%	7.2 MB	0 MB/s	0 Mbps
Background processes (111)					
ABBYY network license server (3...		0%	0.2 MB	0 MB/s	0 Mbps
AcroTray (32 bit)		0%	0.2 MB	0 MB/s	0 Mbps
Adobe Acrobat Update Service (...		0%	0.1 MB	0 MB/s	0 Mbps
Adobe Genuine Software Integri...		0%	1.3 MB	0 MB/s	0 Mbps
Adobe Genuine Software Servic...		0%	1.7 MB	0 MB/s	0 Mbps
AiChargerDT MFC Application (...		0%	0.3 MB	0 MB/s	0 Mbps

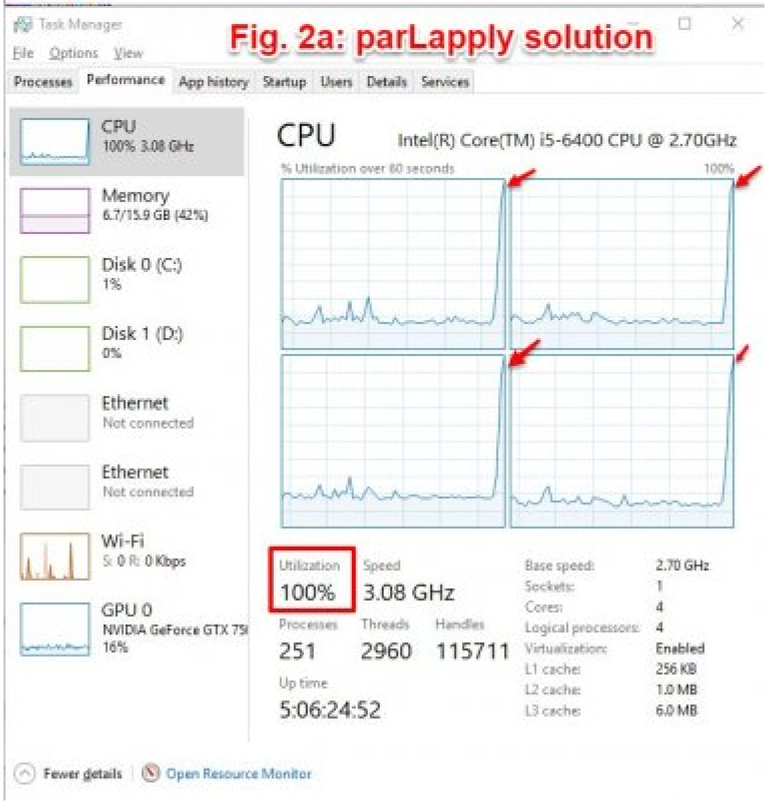


Fig. 2a: parLapply solution

Task Manager
File Options View
Processes Performance App history Startup Users Details Services
% Utilization over 60 seconds
100%
Fewer details
Open Resource Monitor

CPU
100% 3.08 GHz

Memory
6.7/15.9 GB (42%)

Disk 0 (C:)
1%

Disk 1 (D:)
0%

Ethernet
Not connected

Ethernet
Not connected

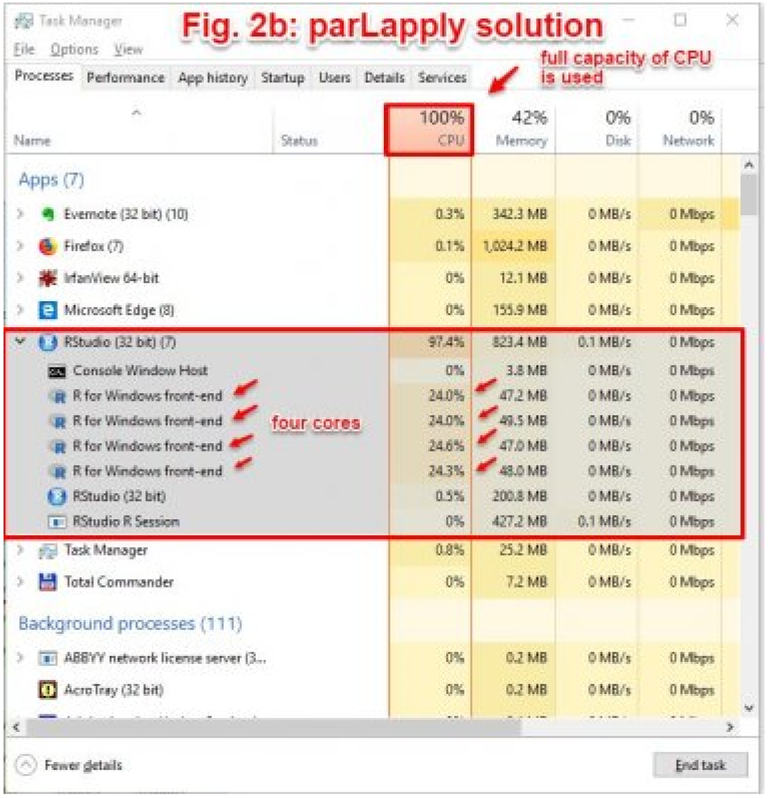
Wi-Fi
S: 0 R: 0 Kbps

GPU 0
NVIDIA GeForce GTX 75
16%

CPU Intel(R) Core(TM) i5-6400 CPU @ 2.70GHz

Utilization	Speed	
100%	3.08 GHz	
Processes	Threads	Handles
251	2960	115711
Up time		
5:06:24:52		

Base speed:	2.70 GHz
Sockets:	1
Cores:	4
Logical processors:	4
Virtualization:	Enabled
L1 cache:	256 KB
L2 cache:	1.0 MB
L3 cache:	6.0 MB


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fig. 2b: parLapply solution
full capacity of CPU is used
four cores
Fewer details
End task

Name	Status	100% CPU	42% Memory	0% Disk	0% Network
Apps (7)					
Evernote (32 bit) (10)		0.3%	342.3 MB	0 MB/s	0 Mbps
Firefox (7)		0.1%	1,024.2 MB	0 MB/s	0 Mbps
IrfanView 64-bit		0%	12.1 MB	0 MB/s	0 Mbps
Microsoft Edge (8)		0%	155.9 MB	0 MB/s	0 Mbps
RStudio (32 bit) (7)		97.4%	823.4 MB	0.1 MB/s	0 Mbps
Console Window Host		0%	3.8 MB	0 MB/s	0 Mbps
R for Windows front-end		24.0%	47.2 MB	0 MB/s	0 Mbps
R for Windows front-end		24.0%	49.5 MB	0 MB/s	0 Mbps
R for Windows front-end		24.6%	47.0 MB	0 MB/s	0 Mbps
R for Windows front-end		24.3%	48.0 MB	0 MB/s	0 Mbps
RStudio (32 bit)		0.5%	200.8 MB	0 MB/s	0 Mbps
RStudio R Session		0%	427.2 MB	0.1 MB/s	0 Mbps
Task Manager		0.8%	25.2 MB	0 MB/s	0 Mbps
Total Commander		0%	7.2 MB	0 MB/s	0 Mbps
Background processes (111)					
ABBYY network license server (3...		0%	0.2 MB	0 MB/s	0 Mbps
AcroTray (32 bit)		0%	0.2 MB	0 MB/s	0 Mbps

This makes the calculation considerably faster; result of benchmarking using function `microbenchmark` from library `microbenchmark`, show that the non-parallel version is in average 2.5 times slower than parallel (column mean, 7.56 sec vs 2.95 sec; note that I modified argument `times` in `microbenchmark` function to only 10 (default is 100), which means that the function repeats each version 10-times and calculates the time statistics):

```
mb <- microbenchmark::microbenchmark (
  {
    lapply (1:100, FUN = function (x) mean (rnorm (1000000)))
  },
  {
    library (parallel)
    cl <- makeCluster (4)
    res <- parLapply (cl, X = 1:100, fun = function (x) mean (rnorm
(1000000)))
    stopCluster (cl)
  },
  times = 10)
mb
```

```
Unit: seconds
...
      min       lq     mean   median       uq      max neval cld
 7.389548 7.522466 7.566548 7.585431 7.605311 7.703006    10   b
 2.853429 2.890022 2.954747 2.943975 2.968527 3.114184    10  a
```

The reason why the parallel version is not truly four times faster than non-parallel version (although it is running on four cores in the case of my computer) is that parallel processes spend some time overhead by managing the parallelization - splitting the data, sending them to individual R workers, collecting them and merging them together. The best use of parallelization is when a single operation takes considerable time, so the time spent by calculation is much higher than time spent by communication of R with individual cores. Indeed, if I increase the number of random values from which the mean is calculated from 10^6 to 10^7, the relative speed of parallel version increases to almost four (83.8 sec non-parallel vs 21.5 sec parallel)(do not try to run the code below unless you have a fairly powerful computer, since the calculation already takes some time; and note that I decreased the `times = 5`, otherwise it takes ages):

```
mb <- microbenchmark::microbenchmark (
  {
    lapply (1:100, FUN = function (x) mean (rnorm (10000000)))
  },
  {
    library (parallel)
    cl <- makeCluster (4)
    res <- parLapply (cl, X = 1:100, fun = function (x) mean (rnorm
(10000000)))
    stopCluster (cl)
  },
  times = 5)
mb
```

```
Unit: seconds
...
      min       lq     mean   median       uq      max neval cld
 83.08273 83.82933 83.95855 83.97395 84.39401 84.51273     5   b
 21.42050 21.43552 21.58001 21.49912 21.58116 21.96373     5  a
```