# Parallel **R**

Andrew Bzikadze
seryrzu@gmail.com

Saint Petersburg State University, Russia
Faculty of Mathematics and Mechanics
Department of Statistical Modelling

September 12, 2015

# Outline

# Motivation

Why **R**? The **R** language has a lot of advantages:

- Open Source.
- Cross-Platform.
- Free.
- Many basic tools.
- **R** extensions.
- Arguably fast with **R**-way style and well-implemented **R** distribution.

# Motivation

Why **R**? The **R** language has a lot of advantages:

- Open Source.
- Cross-Platform.
- Free.
- Many basic tools.
- **R** extensions.
- Arguably fast with **R**-way style and well-implemented **R** distribution.

Why bother then?

# Motivation

Why **R**? The **R** language has a lot of advantages:

- Open Source.
- Cross-Platform.
- Free.
- Many basic tools.
- **R** extensions.
- Arguably fast with **R**-way style and well-implemented **R** distribution.

Why bother then?

There are 2 major limitations:

- Single-threaded: no out-of-box support of multi-thread calculations.
- Memory-bound: all data should fit in RAM.

# Motivation

Why **R**? The **R** language has a lot of advantages:

- Open Source.
- Cross-Platform.
- Free.
- Many basic tools.
- **R** extensions.
- Arguably fast with **R**-way style and well-implemented **R** distribution.

Why bother then?

There are 2 major limitations:

- Single-threaded: no out-of-box support of multi-thread calculations.
- Memory-bound: all data should fit in RAM.

*Solution:* Parallel Execution. How exactly?

- Single-threaded: multiple CPU's (and cores).
- Memory-bound: spread data from one computer (master) to several computers (slaves).

# Basic Terminology

The two types of parallelism:

- **Implicit** — the OS abstracts parallelism from the user.
- **Explicit** — user controls details of the process.

A **computer cluster** consists of a set of loosely or tightly connected computers that work together so that, in many respects, they can be viewed as a single system (Wiki).

**Master/slave** is a model of communication where one device or process has unidirectional control over one or more other devices (Wiki).

**snow**

*Usage:* Explicit parallelism by using clusters (works on Linux, Windows, Mac OS X).

*Solves:* Single-threaded, memory-bound.

**multicore [deprecated]**

*Usage:* Implicit parallelism by using FORK (doesn't work on Windows).

*Solves:* Single-threaded.

**snow**

*Usage:* Explicit parallelism by using clusters (works on Linux, Windows, Mac OS X).

*Solves:* Single-threaded, memory-bound.

**multicore [deprecated]**

*Usage:* Implicit parallelism by using FORK (doesn't work on Windows).

*Solves:* Single-threaded.

**parallel [mainstream]**

*Usage:* Almost a wrapper of `snow` and `multicore`.

*Solves:* Single-threaded, memory-bound.

# Overview

**snow**

*Usage:* Explicit parallelism by using clusters (works on Linux, Windows, Mac OS X).

*Solves:* Single-threaded, memory-bound.

**multicore [deprecated]**

*Usage:* Implicit parallelism by using FORK (doesn't work on Windows).

*Solves:* Single-threaded.

**parallel [mainstream]**

*Usage:* Almost a wrapper of `snow` and `multicore`.

*Solves:* Single-threaded, memory-bound.

**Hadoop**

**R** + `Hadoop, RHIPE, Segue`.

# Epigraph

*"... R was not built in anticipation of the Big Data revolution. R was born in 1995. Disk space was expensive, RAM even more so, and this thing called The Internet was just getting its legs. Notions of "large-scale data analysis" and "high-performance computing" were reasonably rare. Outside of Wall Street firms and university research labs, there just wasn't that much data to crunch."*

— Q. Ethan McCallum and Stephen Weston "Parallel **R**"

# Outline

# Outline

# snow: quick look

| | |
|---|---|
| *General use case:* | Main word is cluster, provides explicit parallelism. |
| *Examples:* | Monte Carlo simulations, bootstrapping, cross validation, ensemble machine learning algorithms. |
| *Solves:* | Single-threaded, memory-bound. |
| *Cool features*: | • Different transport mechanisms between Master and Slaves: Sockets, MPI (`rmpi`), NWS (`nws`), PVM (`rpvm`).<br>• Good support of RNG (`rsprng`, `rlecuyer`). |
| *Problems:* | No communication between the workers (slaves). |

# snow: quick look

| | |
|---|---|
| *General use case:* | Main word is cluster, provides explicit parallelism. |
| *Examples:* | Monte Carlo simulations, bootstrapping, cross validation, ensemble machine learning algorithms. |
| *Solves:* | Single-threaded, memory-bound. |
| *Cool features*: | • Different transport mechanisms between Master and Slaves: Sockets, MPI (`rmpi`), NWS (`nws`), PVM (`rpvm`).<br>• Good support of RNG (`rsprng`, `rlecuyer`). |
| *Problems:* | No communication between the workers (slaves). |
| *Warning:* | The input arguments **must fit** into memory when calling `snow` function. Its up to the user to arrange high-performance distributed file systems. |

# Structure of API

| | |
|---|---|
| *Start and stop clusters:* | `makeCluster`, `stopCluster`. |
| *Low-level (cluster-level) functions:* | `cluster*` — `clusterApply`, `clusterApplyLB`, `clusterEvalQ`, `clusterCall`, `clusterSplit`, etc. |
| *High-level functions:* | `par[L,S,A,R,C]apply` — parallel versions of apply and related functions. |
| *(Uniform) RNG:* | L'Ecuyer (package: `rlecuyer`), SPRNG *[deprecated]* (package: `rsprng`). |
| *Timing*: | `snow.time(expr)` — very useful. |

# Start and stop clusters

Basic way:

```
cl <- makeCluster(8, type = "SOCK") # or you may use
                                    # makeSOCKcluster(),
                                    # makePVMcluster(), etc.
stopCluster(cl)
```

First parameter is `spec` — specification. It is `ssh`-configurable unless you type `"localhost"`:

```
cl <- makeCluster(c("localhost","localhost"), type = "SOCK")
stopCluster(cl)
```

*Warning:* Be aware of computational costs of cluster setup.

# Low-level API

All of them are `cluster*` and designed for computing *on* a cluster. Most interesting are as follows.

| | |
|---|---|
| `clusterApply(cl, x, fun, ...)` | Jobs are being "recycled". |
| `clusterApplyLB(cl, x, fun, ...)` | Load Balancing version of `clusterApply()`. |
| `clusterCall(cl, fun, ...)` | Calls a function fun with identical arguments ... on each node in the cluster `cl` and returns a list of the results. |
| `clusterEvalQ(cl, expr)` | Evaluates an expression `expr` on each node in the cluster `cl`; implemented using `clusterCall()`. |
| `clusterMap(cl, fun, ..., MoreArgs = NULL, RECYCLE = TRUE)` | Similar to `mapply`. |

# Example, K-means

Basic one-core way:

```
library(MASS)
result <- kmeans(Boston, 4,
                 nstart=100)
```

# Example, K-means

Basic one-core way:

```
library(MASS)
result <- kmeans(Boston, 4,
                 nstart=100)
```

Before using `snow` it is easier to think `*apply`-way:

```
results <- lapply(rep(25, 4),
  function(nstart)
    kmeans(Boston, 4, nstart=nstart)
  )
i <- sapply(results,
  function(result)
    result$tot.withinss
  )
result <- results[[which.min(i)]]
```

# Example, K-means

Basic one-core way:

```
library(MASS)
result <- kmeans(Boston, 4,
                 nstart=100)
```

Before using snow it is easier to think *apply-way:

```
results <- lapply(rep(25, 4),
  function(nstart)
    kmeans(Boston, 4, nstart=nstart)
  )
i <- sapply(results,
  function(result)
    result$tot.withinss
  )
result <- results[[which.min(i)]]
```

Finally snow version.

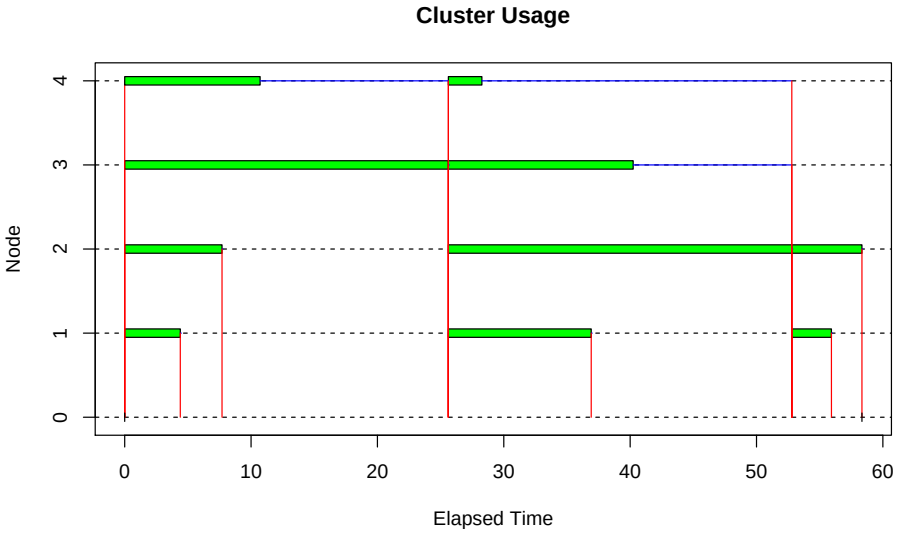
```
ignore <-
  clusterEvalQ(cl, {library(MASS); NULL})
results <-
  clusterApply(cl, rep(25, 4),
    function(nstart)
      kmeans(Boston, 4, nstart=nstart)
  )
i <- sapply(results,
  function(result)
    result$tot.withinss
  )
result <- results[[which.min(i)]]
```

# Load Balancing

`clusterApply()` uses a *robin-round* fashion for scheduling tasks for clusters. It means one time for every cluster. It could be not very wise to do that.

```
set.seed(123)
sleeptime <- abs(rnorm(10, 10, 10))
tm <- snow.time(clusterApply(cl, sleeptime, Sys.sleep))
```

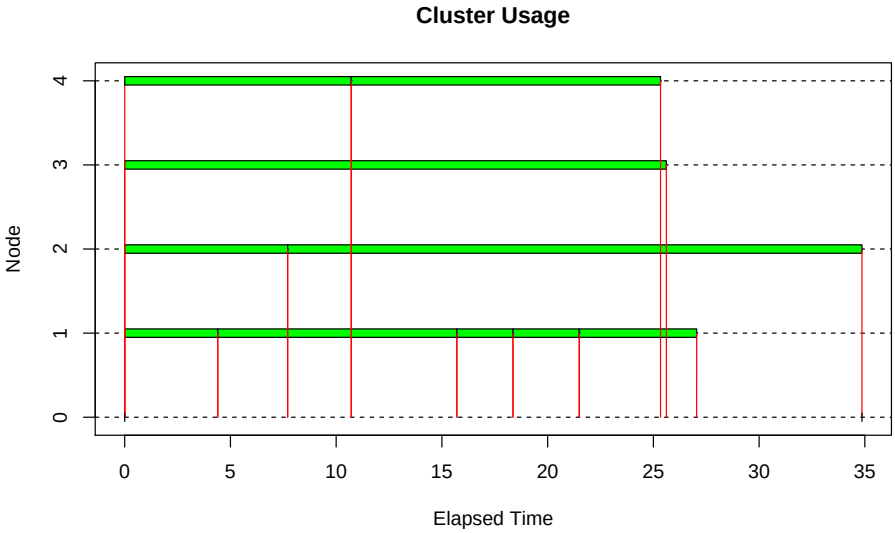
# Load Balancing



# Load Balancing

So we waited for more than 50 seconds. A more efficient way would be pull the tasks to clusters when they are needed. This technique is called "load balancing". Function `clusterApplyLB()` uses that technique.

```
set.seed(123)
sleeptime <- abs(rnorm(10, 10, 10))
tm <- snow.time(clusterApplyLB(cl, sleeptime, Sys.sleep))
```


Cluster Usage
Node
0
1
2
3
4
0
5
10
15
20
25
30
35
Elapsed Time

So, here we waited for about 30 seconds. This is an improvement. The only wasted time was at the end.

# High-level API

| | |
|---|---|
| `parLapply(cl, x, fun, ...)` | Parallel version of lapply(). |
| `parSapply(cl, X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)` | Parallel version of sapply(). |
| `parApply(cl, X, MARGIN, FUN, ...)` | Parallel version of apply(). |
| `parRapply(cl, x, fun, ...)` | Row apply() for matrix. |
| `parCapply(cl, x, fun, ...)` | Column apply() for matrix. |

The most useful is `parLapply()` function. It is different from `clusterApply()` because it splits the task into "equal" tasks.

```
parLapply
```

```
## function (cl, x, fun, ...)
## docall(c, clusterApply(cl, splitList(x, length(cl)), lapply,
##     fun, ...))
## <environment: namespace:snow>
```

where `splitList()` is an internal `snow` function.

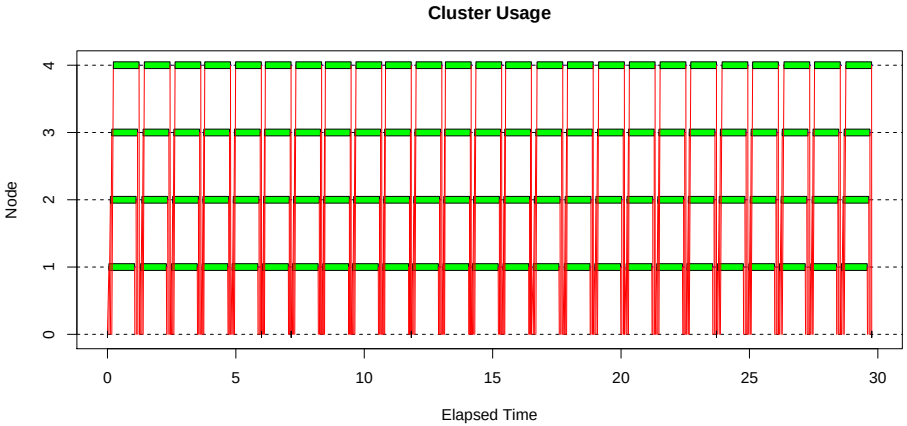
# Example of comparison `clusterApply()` and `parLapply()`

`parLapply()` could be more efficient if you have more tasks than workers. Another situation — you send large arguments to `parLapply()`. Let's take a look at the example.

```
bigsleep <- function(sleeptime, mat) Sys.sleep(sleeptime)
bigmatrix <- matrix(0, 2000, 2000)
sleeptime <- rep(1, 100)
```

Firstly, let's try `clusterApply()`.

# Example of comparison `clusterApply()` and `parLapply()`



# Example of comparison `clusterApply()` and `parLapply()`

Definitely not highly efficient. Those gaps are due to I/O time. Ideally we should have 25 seconds... Let's give `parLapply()` a try.

# Example of comparison `clusterApply()` and `parLapply()`

# Load Balancing parLapply?

# Load Balancing parLapply?

Short answer: no, there is no such a function in snow package.
Good news: it is possible to write your own.

# Random Number Generation

There are 2 basic steps.

1. Configure your cluster workers to use a generator.

```
library(rlecuyer)
clusterSetupRNG(cl, type = 'RNGstream')
```

```
## [1] "RNGstream"
```

2. Be happy to generate your numbers.

```
unlist(clusterCall(cl, runif, 1))
```

```
## [1] 0.12701112 0.75958186 0.72850979 0.09570262
```

# Outline

# `Multicore` *[deprecated]* : quick look

If it is deprecated, why even think about it?

# `Multicore` *[deprecated]* : quick look

If it is deprecated, why even think about it? The reason is the package `parallel`.
*Wait a little bit...*

# `Multicore` *[deprecated]* : quick look

If it is deprecated, why even think about it? The reason is the package `parallel`.
*Wait a little bit...*

| | |
|---|---|
| *General use case:* | Main word is fork (thus no Windows support), provides implicit parallelism. |
| *Examples:* | `lapply()` runs for ages on your Intel Core i999. |
| *Solves:* | Single-threaded. |
| *Problems*: | • No Windows support.<br>• No internal RNG support.<br>• Runs only on one computer.<br>• Cannot be used with **R** `GUI`.<br>• No internal Load Balancing, however, it can be imitated. |

# `Multicore` *[deprecated]* : quick look

If it is deprecated, why even think about it? The reason is the package `parallel`.
*Wait a little bit...*

| | |
|---|---|
| *General use case:* | Main word is fork (thus no Windows support), provides implicit parallelism. |
| *Examples:* | `lapply()` runs for ages on your Intel Core i999. |
| *Solves:* | Single-threaded. |
| *Problems*: | • No Windows support.<br>• No internal RNG support.<br>• Runs only on one computer.<br>• Cannot be used with **R** `GUI`.<br>• No internal Load Balancing, however, it can be imitated. |
| *Warning:* | Jobs started by `multicore` share the same state (because of `fork`). |

# Multicore *[deprecated]* : quick look

We will only consider *high-level* API and let *low-level* to be out-of-scope.

| | |
|---|---|
| `mclapply()` | Parallel version of `lapply()`. |
| `mcmapply()` | Parallel version of `mapply()`. |
| `pvec()` | Somewhat an high-level analog of low-level `clusterSplit()` function. |
| `parallel()` and `collect()` | `parallel()` creates a new process with `fork()`, evaluate expression in parallel and after that the result is retrieved by the `collect()`. |

# Multicore: mclapply

`mclapply()` is a parallel `lapply()`.

Syntax is as follows:

```
mclapply(X, FUN, ..., mc.preschedule = TRUE, mc.set.seed = TRUE,
  mc.silent = FALSE, mc.cores = getOption("mc.cores"))
```

where

1. `mc.preschedule = TRUE` — how jobs are created for `X`.
2. `mc.set.seed = TRUE` — do you need to randomly seed slaves, or fork it?
3. `mc.silent = FALSE` — hide info from 'stdout' for all parallel forked processes. 'stderr' is not affected.
4. `mc.cores == getOption("mc.cores")` — number of **workers** (not cores, actually) to start.

Meaning:

- TRUE: divide data in `mc.cores`-jobs beforehand and fork it to `mc.cores`-processes.
- FALSE: for each piece of data construct a new job (up to `mc.cores`).

Meaning:

- TRUE: divide data in `mc.cores`-jobs beforehand and fork it to `mc.cores`-processes.
- FALSE: for each piece of data construct a new job (up to `mc.cores`).

Rule of thumb: use

- TRUE: you don't need load balance (for instance, if there are lot's of values in the data).
- FALSE: the variance of job completion is very high (so, you need load balance).

# Outline

# parallel

| | |
|---|---|
| *General use case:* | Main word is mainstream,<br>almost a wrapper of `snow` and `multicore` packages. |
| *Examples:* | Anything above. |
| *Solves:* | Single-threaded and (partially) memory-bound. |
| *Cool features*: | • Preinstalled into **R** since 2.14.0.<br>• Full RNG support with no dependency on `rlecuyer` package.<br>• Almost nothing to learn (if you are still awake).<br>• Can be easily used on any platform including Windows.<br>• Highly compatible with `snow` and `multicore`. |

| | |
|---|---|
| *General use case:* | Main word is mainstream,<br>almost a wrapper of `snow` and `multicore` packages. |
| *Examples:* | Anything above. |
| *Solves:* | Single-threaded and (partially) memory-bound. |
| *Cool features*: | • Preinstalled into **R** since 2.14.0.<br>• Full RNG support with no dependency on `rlecuyer` package.<br>• Almost nothing to learn (if you are still awake).<br>• Can be easily used on any platform including Windows.<br>• Highly compatible with `snow` and `multicore`. |
| *Warning:* | On Windows you can't use more than one machine.<br>It also can be difficult to configure multiple Linux machines. |

How many cores?

```
library(parallel)
mc.cores <- detectCores()
mc.cores
```

```
## [1] 8
```

*Warning:* It is important to take into account that you maybe have hyper-threading.

Unlike `snow` package no additional packages (like `rlecuyer`) are needed.

Fork (no Windows) way:

```
RNGkind("L'Ecuyer-CMRG")
unlist(mclapply(rep(1,4), runif))

## [1] 0.3768615 0.3824588 0.3845725 0.9092709
```

Cluster way:

```
detach("package:snow", character.only=TRUE)
library(parallel)
RNGkind("L'Ecuyer-CMRG")
cl <- makeCluster(2, type="PSOCK")
unlist(clusterCall(cl, function(x) runif(2)))

## [1] 0.3114024 0.9506436 0.6032429 0.8057068

stopCluster(cl)
```

# parallel RNG: reproducible results

Basic way to get reproducible results would be `mc.reset.stream()` — the parallel random number generator is reinitialized using the current seed on the master.

```
detach("package:snow", character.only=TRUE)
library(parallel)
RNGkind("L'Ecuyer-CMRG")
cl <- makeCluster(2, type="PSOCK")
clusterSetRNGStream(cl, 123)
unlist(clusterCall(cl, function(x) runif(2)))
```

```
## [1] 0.1663742 0.3898457 0.3411064 0.9712727
```

```
clusterSetRNGStream(cl, 123)
unlist(clusterCall(cl, function(x) runif(2)))
```

```
## [1] 0.1663742 0.3898457 0.3411064 0.9712727
```

```
stopCluster(cl)
```

# Differences from `multicore` and `snow`

Let's sum up the differences between modern `parallel` package and his predecessors.

`parallel` > `multicore`

- Prefix `mc` in `mcfork()`, `mcexit()`, `mckill()`, `mcparallel()`, `mccollect()`, `mc.cores`.
- Different default value of `mc.cores` argument.
- New mc.reset.stream() function.

Also useful `detectCores()` is added.

`parallel` $\neq$ `snow`

- New function `clusterSetRNGStream()` initializes parallel RNG.
- `snow.time()` function not included.
- `makeCluster()` supports additional types FORK.

# Outline

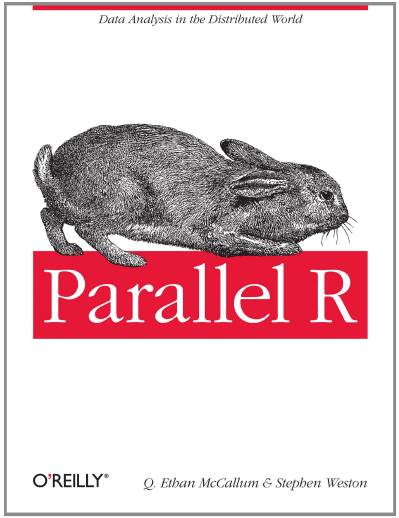
# Out of scope

We covered 3 (2.5 really) packages: `snow`, `multicore`, `parallel`. What else?

- *Revolution Analytics* `foreach` package for iteration over a set of values.
- MapReduce via Java Hadoop: `RHIPE` (negotiator between you with your MapReduce functions and Hadoop).
- `Segue` for Amazon Elastic MapReduce lovers. Be aware of terminating clusters.
- `doRedis`.
- `http://cloudNumbers.com`
- **R** and GPUs: `gputools` etc.

# Literature

Main reference:




**Useful links:**

- Advanced **R** by Hadley Wickham.
- The **R** Inferno by Patrick Burns.
- **R** Packages by Hadley Wickham.
- Writing **R** Extensions.
- Los Angeles **R** Users Group: Parallelization in **R**, Revisited by Ryan R. Rosario.
- Package parallel manual.