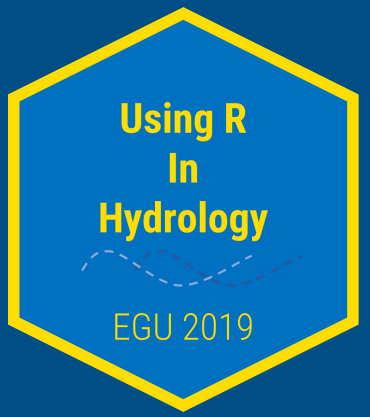
Introduction to parallel and high performance computing for hydrologists

Louise Slater
University of Oxford



louisejslater.com

DrLouiseSlater

Outline:

Key concepts

1. Terminology
2. Why use parallel computing? (and when?)
3. Defining parallelism

Key approaches

1. For loops
2. From for loops to lapply
3. Parallelising the code (parallel / future.apply / foreach)

References and resources

1. The CRAN Task View
2. Cloud computing
3. References

Outline:

Key concepts

1. **Terminology**
2. **Why use parallel computing? (and when?)**
3. **Defining parallelism**

Key approaches

1. For loops
2. From for loops to lapply
3. Parallelising the code (parallel / future.apply / foreach)

References and resources

1. The CRAN Task View
2. Cloud computing
3. References

Terminology

What are we talking about?

High performance computing:

- Using supercomputers or parallel computing techniques to solve computational problems.

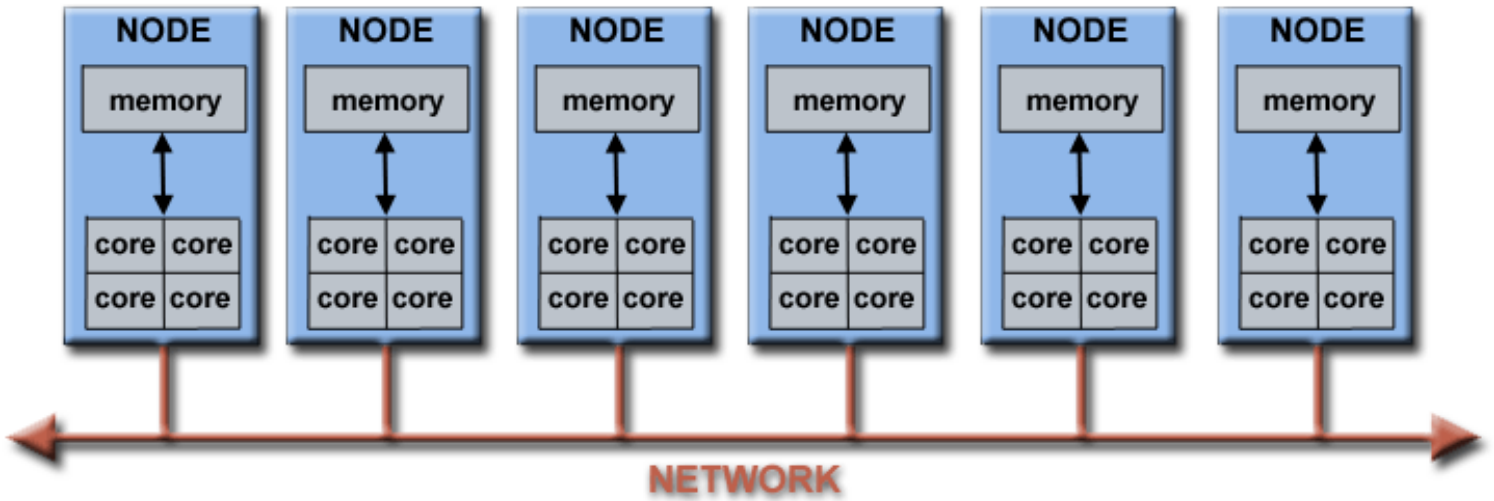
Parallel processing/computing:

- An algorithm is written to divide a large problem into smaller chunks (calculations or processes) that can be solved at the same time

We will adopt a loose definition of HPC and PC as "anything related to pushing R a little further"

Terminology

- A core (processor): basic unit of the Central Processing Unit (CPU): single running version of a program.
- A node: a physical device (e.g. computer) within a network. Can have multiple cores.
- A cluster or supercomputer: hosts multiple nodes



Parallel computing aims to increase performance typically by taking advantage of multiple cores, sometimes across multiple nodes.

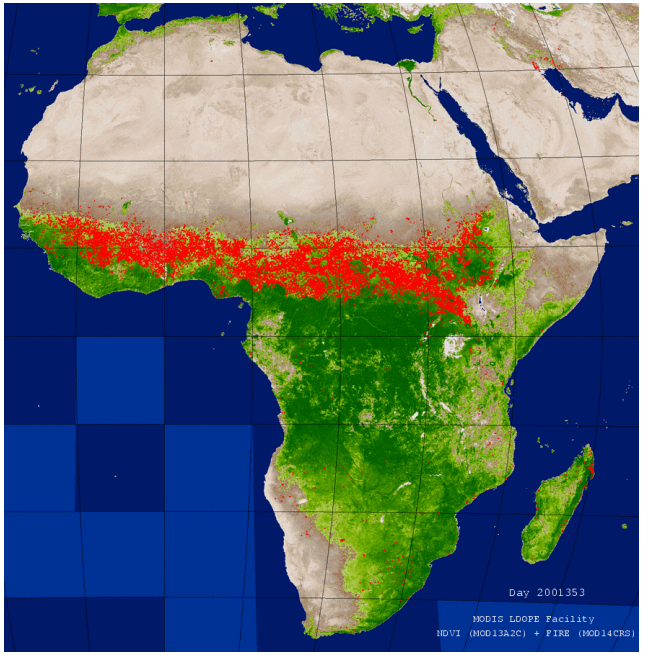
Why would we use parallel computing?

Growing data in hydrology:

- Greater number and size of datasets
- Enhanced resolution and abilities of models

More/better data takes longer to run. So either we spend longer running our codes, or we find ways to speed up their performance

Parallel computing is a practical way to **speed things up**.



When should we use parallel computing?

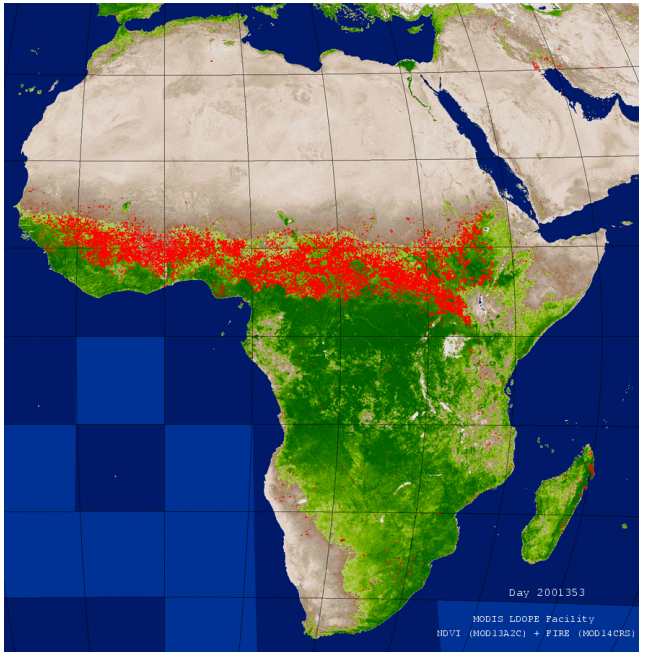
It's important to note that:

- It's not always necessary or desirable to parallelise
- Efficiency does not increase linearly with the number of cores used

That said, in hydrology, parallel computing can save a lot of time.

You **don't need a cluster or HPC** to run parallel code: most laptops have multiple cores...

The examples described here can be implemented ***either*** on your PC ***or*** on a HPC.



Defining parallelism

In serial code:

- chunk 1 runs. chunk 2 starts only once chunk 1 is complete

In parallel code:

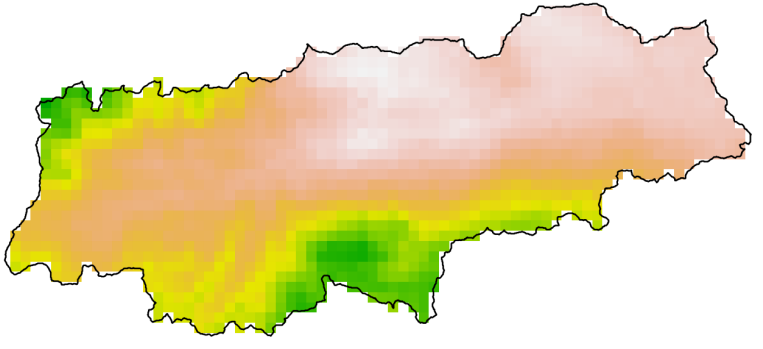
- chunk 1 and chunk 2 run simultaneously

Defining parallelism

- *Implicit* parallelism: when the system (libraries, compiler, etc) is doing the parallelising for you. E.g. a compiler or interpreter automatically exploits the parallelism.
- *Explicit* parallelism: when you are the one explicitly decomposing your work into tasks. This talk focuses on explicit parallelism (e.g. packages snow, foreach, doFuture), which is more commonly used.

Code can be described as "embarrassingly parallel" when there is clear iteration independence, i.e. it is easy to separate the problem into parallel tasks, and there is little to no dependency between those tasks.

Embarrassingly parallel code is quite frequent in large-sample hydrology. Example: computing catchment-averaged precipitation for 100 river basins.



Outline:

Key concepts

1. Terminology
2. Why use parallel computing? (and when?)
3. Defining parallelism

Key approaches

1. **For loops**
2. **From for loops to lapply**
3. **Parallelising the code (parallel / future.apply / foreach)**

References and resources

1. The CRAN Task View
2. Cloud computing
3. References

1. For loops

The for-loop is our basic example of **serial code**.

Let's say we start with a list of three catchments.

Install the **rnrfa** library (Vitolo et al., 2018)

```
if (!require('rnrfa')) install.packages('rnrfa')
library('rnrfa')
```

Import the catchment catalogue and select **three gauging stations**

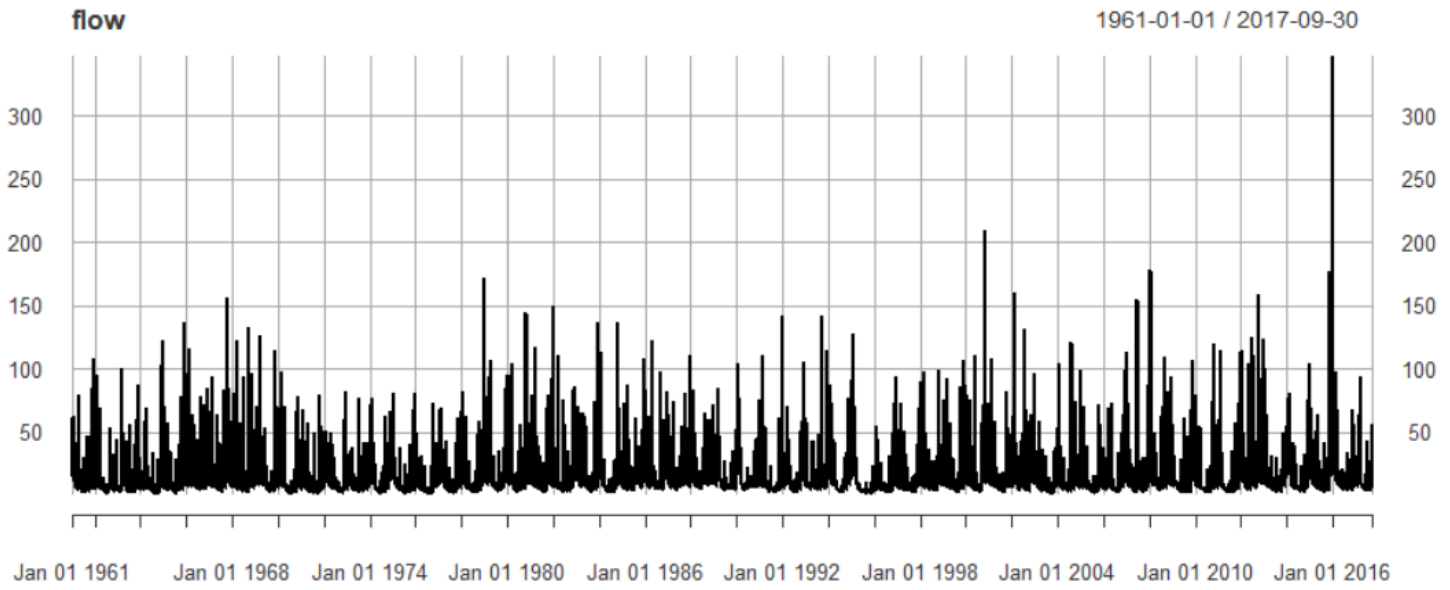
```
Stations_catalogue <- catalogue()
list_of_catchments <- c("21003","22006","27028")
```

1. For loops

In each loop we are going to download gauged daily flow for one catchment.

First, download and plot gauged daily flow for one catchment, just to check everything is working:

```
flow <- gdf(id = "27028")
plot(flow)
```



1. For loops

The for-loop is our basic example of serial (sequential) code.

Here is a simple example of a for loop where we download the *gauged daily flow* (gdf function) and compute the mean value for each catchment.

```
my_list <- list()
for (i in list_of_catchments) {
  flow <- gdf(id = i)
  mean_flow <- mean(flow)
  my_list[[i]] <- mean_flow
}
```

The result of each iteration is a single value (variable mean_flow):

```
str(my_list)
```

```
## List of 3
##  $ 21003: num 16.1
##  $ 22006: num 2.13
##  $ 27028: num 15.3
```

1. For loops

We can wrap the “iteration” code inside **local()** to make sure it is evaluated in a local environment.

This simple adjustment prevents the loop from assigning values to the global environment and can prevent errors.

```
my_list <- list()
for (i in list_of_catchments) {
  my_list[[i]] <- local({
    flow <- gdf(id = i)
    mean_flow <- mean(flow)
    mean_flow
  })
}
```

The output remains the same.

2. From for loops to Lapply

Instead of executing each task in a sequence (where just one of the processors is used), we can use multiple processors on our computer to speed things up.

But before we get there, we need to write our code as a function. We do this by turning our for loop into a lapply call. The **lapply** function in base-R is a substitute to the loop, and is often faster.

Lapply works by applying a **function** over a **list** or vector (list-apply).

```
my_function <- function(x) {
  flow <- gdf(id = x)
  mean_flow <- mean(flow)
  mean_flow
}
my_list <- lapply(list_of_catchments, my_function)
```

Typically we would combine the two, like this:

```
my_list <- lapply(list_of_catchments, function(x) {
    flow <- gdf(id = x)
    mean_flow <- mean(flow)
    mean_flow
```

3. Parallelising the code

There are many different methods to do this.

Three of the most common are:

1. The parallel package (in base-R), which uses lapply functions and builds on the work done for CRAN packages **multicore** (Urbanek, 2009–2014) and **snow** (Tierney et al., 2003–present)

2. The future.apply package (Bengtsson, 2019), which makes things easier via the **future_lapply** function

3. The doParallel package, which uses foreach (Calaway et al., 2018) and the doFuture package (Bengtsson, 2019)

3.1. The parallel package

The parallel package is part of base R. It builds on earlier CRAN parallel computation packages:

- **multicore** (shared memory platforms - unix; now retired from CRAN)
- snow (distributed memory platforms)

and allows us to run large chunks of computations in parallel.

It works with both ***shared memory*** (when multiple processing elements share the same location in memory) and ***distributed memory*** (which requires explicit commands to transfer data from one processing element to another)

Check the online vignette for the parallel package, by simply typing in the R console:

```
vignette("parallel")
```

3.1. The parallel package

Load the library

```
library(parallel)
```

Check the number of CPU cores on your computer

```
ncores <- detectCores()
ncores
```

Start the cluster

```
my_cluster <- makeCluster(ncores)
```

3.1. The parallel package

When parallelising you have to explicitly export/declare everything you need to perform the parallel operation to each thread

clusterEvalQ loads an expression on each cluster node:

```
clusterEvalQ(my_cluster, {library(rnrfa)})
```

ClusterExport exports objects from the base workspace to the global environments of each node (in this case we don't really need it because we won't be using the Station catalogue, so this is just an example)

```
clusterExport(my_cluster, list( 'Stations_catalogue'))
```

Now we can run the **parallel version of lapply** from the **snow** package:

```
parLapply(my_cluster, list_of_catchments,
          function(x) {
            flow <- gdf(id = x)
            mean_flow <- mean(flow)
            mean_flow
          })
```

3.1. The parallel package

Close the cluster

After running your parallel code it is useful to stop the cluster.

For a manually created cluster, it is best to stop it as follows:

```
stopCluster(my_cluster)
```

3.2. Using the future.apply package

The **future.apply** package allows us to easily parallelise lapply.

```
if (!require('future.apply')) install.packages('future.apply')
library('future.apply')
```

The function **plan(multiprocess)** parallelises the code on your local computer for you.

It queries the machine (using availableCores()) to infer how many cores can be used simultaneously, so there is no need to explicitly specify how many cores are available:

```
plan(multiprocess)
#plan(sequential) # returns to sequential processing
```

3.2. Using the future.apply package

Now all we have to do is to replace lapply() with **future_lapply()**:

```
my_list <- future_lapply(list_of_catchments, function(x) {
  flow <- gdf(id = x)
  mean_flow <- mean(flow)
  mean_flow
})
```

3.3. With the foreach framework

The doParallel foreach framework is another way of parallelising code. It is somewhat similar to the lapply approach. You have to explicitly create the cluster:

```
library(doParallel)
detectCores() # As before, check cores
my_cluster <- makeCluster(ncores) # Create cluster
registerDoParallel(my_cluster) # Register cluster
getDoParWorkers() # Check how many cores are being used
```

Implement foreach as:

```
result <- foreach(x=list_of_catchments) %dopar% {
  library(rnrfa)
  flow <- gdf(id = x)
  mean_flow <- mean(flow)
  mean_flow
}
stopCluster(my_cluster)
```

3.3. With the foreach framework

The ***simplified*** version of foreach can be implemented with the **doFuture** package:

```
if (!require('doFuture')) install.packages('doFuture')
library('doFuture')
```

doFuture provides access to the **future framework** and the full range of parallel backends that comes with it

```
registerDoFuture()
plan(multiprocess)

my_list <- foreach(x = list_of_catchments) %dopar% {
  flow <- gdf(id = x)
  mean_flow <- mean(flow)
  mean_flow
}
```

Summary of these approaches

According to the author of the future package (see here), one of the simplest approaches to parallel computing is to parallelize your for-loop with the future package:

1. Rewrite your for-loop such that each iteration is done *inside a local() call*
2. Rewrite this new for-loop as a *lapply call* (straightforward)
3. Replace the lapply call with a *parallel implementation* of your choice

Outline:

Key concepts

1. Terminology
2. Why use parallel computing? (and when?)
3. Defining parallelism

Key approaches

1. For loops
2. From for loops to lapply
3. Parallelising the code (parallel / future.apply / foreach)

References and resources

1. **The CRAN Task View**
2. **Cloud computing**
3. **References**

The CRAN Task View for HPC

The Task View can be accessed online at https://cran.r-project.org/web/views/HighPerformanceComputing.html:

CRAN Task View: High-Performance and Parallel Computing with R

Maintainer: Dirk Eddelbuettel
Contact: Dirk.Eddelbuettel at R-project.org
Version: 2018-10-30
URL: https://CRAN.R-project.org/view=HighPerformanceComputing

This CRAN task view contains a list of packages, grouped by topic, that are useful for high-performance computing (HPC) with R. In this context, we are defining 'high-performance computing' rather loosely as just about anything related to pushing R a little further: using compiled code, parallel computing (in both explicit and implicit modes), working with large objects as well as profiling.

Unless otherwise mentioned, all packages presented with hyperlinks are available from CRAN, the Comprehensive R Archive Network.

Several of the areas discussed in this Task View are undergoing rapid change. Please send suggestions for additions and extensions for this task view to the task view maintainer .

Suggestions and corrections by Achim Zeileis, Markus Schmidberger, Martin Morgan, Max Kuhn, Tomas Radivoyevitch, Jochen Knaus, Tobias Verbeke, Hao Yu, David Rosenberg, Marco Enea, Ivo Welch, Jay Emerson, Wei-Chen Chen, Bill Cleveland, Ross Boylan, Ramon Diaz-Uriarte, Mark Zeligman, Kevin Ushey, Graham Jeffries, Will Landau, Tim Flutre, Reza Mohammadi, Ralf Stubner, and Bob Jansen (as well as others I may have forgotten to add here) are gratefully acknowledged.

Contributions are always welcome, and encouraged. Since the start of this CRAN task view in October 2008, most contributions have arrived as email suggestions. The source file for this particular task view file now also reside in a GitHub repository (see below) so that pull requests are also possible.

The `ctv` package supports these Task Views. Its functions `install.views` and `update.views` allow, respectively, installation or update of packages from a given Task View; the option `coreOnly` can restrict operations to packages labeled as *core* below.

The CRAN Task View for HPC

- Explicit parallel computing
- Implicit parallel computing
- Grid computing
- Hadoop, Random numbers, Resource managers and batch schedulers
- Applications, GPUs
- Large memory and out-of-memory data
- Easier interfaces for Compiled code
- Profiling tools

Cloud computing

It's worth mentioning that cloud computing is another way to increase code efficiency **for those who do not have access** to a powerful computer to run parallel code.

There is no software to install or configure. You can run R directly in the browser, and you can execute your parallel code in the cloud.

Come to see our PICO: 'Using R in hydrology'

Friday 12th, PICO spot 5b (8:30-10:15)

In this fantastic session HS1.2.7:

Innovative methods to facilitate open science and data analysis in hydrology.

Hydrol. Earth Syst. Sci. Discuss., https://doi.org/10.5194/hess-2019-50
Manuscript under review for journal Hydrol. Earth Syst. Sci.
Discussion started: 18 February 2019

Hydrology and Earth System Sciences Discussions

Using R in hydrology: a review of recent developments and future directions

Louise J. Slater[a], Guillaume Thirel[b], Shaun Harrigan[c], Olivier Delaigue[b], Alexander Hurley[d], Abdou Khouakhi[e], Ilaria Prodoscimi[f], Claudia Vitolo[c], and Katie Smith[g]

[a]School of Geography and the Environment, University of Oxford, OX1 3QY, UK
[b]IRSTEA, HYCAR Research Unit, 1 rue Pierre-Gilles de Gennes, 92160 Antony, France
[c]Forecast Department, European Centre for Medium-Range Weather Forecasts (ECMWF), Shinfield Park, Reading, RG2 9AX, UK
[d]School of Geography, Earth and Environmental Sciences, University of Birmingham, B15 2TT, UK
[e]School of Architecture, Civil and Building Engineering, Loughborough University, Loughborough, UK
[f]Department of Environmental Sciences, Informatics and Statistics, Ca' Foscari University of Venice, Venice, Italy
[g]Centre for Ecology & Hydrology, Maclean Building, Crowmarsh Gifford, Wallingford, OX10 8BB, UK

Correspondence: Louise J. Slater (louise.slater@ouce.ox.ac.uk)

Abstract. The open-source programming language R has gained a central place in the hydrological sciences over the last decade, driven by the availability of diverse hydro-meteorological data archives and the development of open-source computational tools. The growth of R's usage in hydrology is reflected in the number of newly published hydrological packages, the strengthening of online user communities, and the popularity of training courses and events. In this paper, we explore the benefits and advantages of R's usage in hydrology, such as: the democratization of data science and numerical literacy, the enhancement of reproducible research and open science, the access to statistical tools, the ease of connecting R to and from other languages, and the support provided by a growing community. This paper provides an overview of important packages at every step of the hydrological workflow, from the retrieval of hydro-meteorological data, to spatial analysis and cartography, hydrological modelling, statistics, and the design of static and dynamic visualizations, presentations and documents. We discuss some of the challenges that arise when using R in hydrology and useful tools to overcome them, including the use of hydrological libraries, documentation and vignettes (long-form guides that illustrate how to use packages); the role of Integrated Development Environments (IDEs); and the challenges of Big Data and parallel computing in hydrology. Last, this paper provides a roadmap for R's future within hydrology, with R packages as a driver of progress in the hydrological sciences, Application Programming Interfaces (APIs) providing new avenues for data acquisition and provision, enhanced teaching of hydrology in R, and the continued growth of the community via short courses and events.

References

Helpful tutorials and references to go further:

- M.Jones (2017), Quick Intro to Parallel Computing in R
- H.Bengtsson (2019), author of the future package, Parallelize a For-Loop by Rewriting it as an Lapply Call
- F.Schwendinger et al. (2017), High Performance Computing with Applications in R
- G.Lockwood (2015), Using R on HPC Clusters or Parallel Options for R
- A.Mirshani (2019), Parallel computing cheat sheet (note: the link downloads the PDF)