Terminology

* A *core* is a general term for either a single processor on your own computer (technically you only have one processor, but a modern processor like the i7 can have multiple cores - hence the term) or a single machine in a cluster network.
* A *cluster* is a collection of objecting capable of hosting cores, either a network or just the collection of cores on your personal computer.
* A *process* is a single running version of R (or more generally any program). Each core runs a single process.

## Methods of Parallelization

Sockets vs forking:

* The socket approach launches a new version of R on each core. Technically this connection is done via networking (e.g. the same as if you connected to a remote server), but the connection is happening all on your own computer[3](http://dept.stat.lsa.umich.edu/~jerrick/courses/stat701/notes/parallel.html#fn3) I mention this because you may get a warning from your computer asking whether to allow R to accept incoming connections, you should allow it.
* The forking approach copies the entire current version of R and moves it to a new core.

Socket:

* Pro: Works on any system (including Windows).
* Pro: Each process on each node is unique so it can’t cross-contaminate.
* Con: Each process is unique so it will be slower
* Con: Things such as package loading need to be done in each process separately. Variables defined on your main version of R don’t exist on each core unless explicitly placed there.
* Con: More complicated to implement.

Forking:

* Con: Only works on POSIX systems (Mac, Linux, Unix, BSD) and not Windows.
* Con: Because processes are duplicates, it can cause issues specifically with random number generation (which should usually be handled by parallel in the background) or when running in a GUI (such as RStudio). This doesn’t come up often, but if you get odd behavior, this may be the case.
* Pro: Faster than sockets.
* Pro: Because it copies the existing version of R, your entire workspace exists in each process.
* Pro: Trivially easy to implement.

**The parallel package**

Merged from multicore and snow and was adopted in the base R installation.

## Forking with mclapply (Windows not available)

result=c()

data=runif(100,0,1)

fun <- function(i) {

set.seed(i)

temp=mean(sample(data,size=100,replace = T))

return(result=c(result,temp))

}

system.time(save1 <- lapply(1:1000, fun))

system.time(save2 <- mclapply(1:1000, fun))

Note:

If you were to run this code on Windows, mclapply would simply call lapply, so the code works but sees no speed gain.

mclapply takes an argument, mc.cores. By default, mclapply will use all cores available to it. If you don’t want to (either becaues you’re on a shared system or you just want to save processing power for other purposes) you can set this to a value lower than the number of cores you have. Setting it to 1 disables parallel processing, and setting it higher than the number of available cores has no effect.

## Using sockets with parLapply

As promised, the sockets approach to parallel processing is more complicated and a bit slower, but works on Windows systems. The general process we’ll follow is

1. Start a cluster with n nodes.
2. Execute any pre-processing code necessary in each node (e.g. loading a package)
3. Use par\*apply as a replacement for \*apply. Note that unlike mcapply, this is not a drop-in replacement.
4. Destroy the cluster (not necessary, but best practices).

result=c()

data=runif(100,0,1)

fun <- function(i) {

set.seed(i)

temp=mean(sample(data,size=100,replace = T))

return(result=c(result,temp))

}

system.time(save1 <- lapply(1:1000, fun))

system.time(save2 <- mclapply(1:1000, fun))

numCores <- **detectCores**()

numCores

cl <- **makeCluster**(numCores-1)

**clusterExport**(cl, "data")

**clusterEvalQ**(cl, {

**library**(ggplot2)

**library**(stringr)

})

There are parallel versions of the three main apply statements: parApply, parLapply and parSapply for apply, lapply and sapply respectively. They take an additional argument for the cluster to operate on.

**parSapply**(cl, 1:1000, fun)

Reference:

http://dept.stat.lsa.umich.edu/~jerrick/courses/stat701/notes/parallel.html