Parallel Computing :: **CHEAT SHEET**

Splitting :

Splitting a code by :

1. **Task** (different tasks on same data)
2. **Data** (one task on different data)

Hardware needs :

CPU (+2 cores)

RAM (shared memory vs distributed memory)

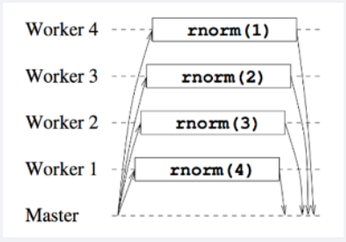
2 ideas in parallel computing :

1. **Map-Reduced Models :**
(distributed data; physically on different devices)
- **Hadoop**
- **Spark**

R Packages:
- sparklyr, iotools
- pbdr (programming with big data in R)

2. **Master - Worker Models :**
(M tasks on C cores; usually 1 < C << M)

R Packages:
- snow, snowFT, snowfal
- foreach
- future, future.apply



Not always parallel computing:

stop/start cluster takes time
overhead (communication time b/w master and workers ; not good for repeatedly sending big data!

Sequential vs Parallel:

```
library(microbenchmark)
microbenchmark( FUN1(…), FUN2(…),
times = 10)
```

parallel.R : core package

```
library(parallel)
ncores <- detectCores(logical=F) # physical cores
cl <- makeCluster(ncores)
clusterApply(cl, x = c(…), fun = FUN) # FUN(x,…)
stopCluster(cl)
```

Initialization of workers :

```
clusterCall(cl,FUN)  # calls FUN on workers
clusterEvalQ(cl, exp) # eval an exp. on workers
## clusterEvalQ(cl, library(foo))
clusterExport(cl, varlist) # varlist on workers
## clusterExport(cl, c("mean")) where mean = 10
```

Data Chunk on workers :

1. generated on workers
```
# clusterApply(cl,x, FUN) e.g FUN(){ rnorm()}
```
2. generated on master and pass to workers
```
# ind <- splitIndices(200, 5)
# clusterApply(cl, ind, FUN)
# (-) : not efficient in Big Data : heavy
```
3. chunk on workers # copy of original Data on all workers
```
# clusterExport(cl, M)       e.g. M is a matrix
# clusterApply(cl, x, FUN)   FUN contains subset M
```

foreach.R : Sequential

```
library(foreach)          # by default return a list
foreach(n = rep(5,3), m = 10^(0:2)) %do% FUN(n,m)
foreach(n, .packages = "X") %do% FUN(n)
# FUN needs package X to be run
foreach(n, .export = c("Y") ) %do% FUN(n,b=Y)
# FUN needs outside object/function "Y"
foreach(n,.combine = rbind) %do% FUN(n) #row bind
foreach(n,.combine = '+') %do% FUN(n) #rbind + colSum
foreach(n,.combine = c) %do% FUN(n) # vector
foreach(n,.combine = c) %:% when(n > 2) %do% FUN(n)
```

future.R : asynchronously

```
library(future) (variables run as soon as created)
plan(multicore)
# plans : sequential, cluster, multicore, multiprocess
x %<-% mean(rnorm(100))
y %<-% mean(rnorm(100))
```

future.apply.R : parallel _apply

```
library(future.apply) (parallel _apply functions)
plan(multicore)     # can be other plans
future_apply(n,FUN),future_lapply(…),future_sapply(…)
```

foreach.R : Parallel

```
needs backend packages support parallel computing
• doParallel(parallel.R),doFuture (future.R),doSEQ
```

doParallel.R : backend of foreach

```
library(doParallel)
cl <- makeCluster(ncores)     # ncores = 2,3,…
registerDoParallel(cl)     # register the backend
foreach(…) %dopar% FUN(…)
```

doFuture.R : backend of foreach

```
library(doFuture)
registerDoFuture()
plan(cluster , workers = 3) # can be other plans
foreach(…) %dopar% FUN(…)
```

Load Balancing: for uneven task times

```
clusterApplyLB(cl,x,FUN) # not for small task time
clusterApply(cl, x = splitIndices(10,2), FUN)
library(itertools)
foreach(s=isplitVector(1:10,chunks =2))%dopar% FUN
# e.g. FUN = sapply(s,"*",100)
future_sapply(…, future.scheduling = 1)
```