Optimization and Parallel Computing in R Workshop

Osvaldo Espin-Garcia

March 6, 2020

Workshop material

Go to https://github.com/oespinga2/opt-par-R for today's material

Overview

Optimization

1. Introduction
2. A 'simple' example: logistic regression
3. Unconstrained optimization: `optim()` and beyond
4. Constrained optimization
5. Stochastich optimization (genetic algorithms)

Parallel computing

1. Intro, shared vs. distributed memory
2. Out-of-the-box implementations
3. `foreach` package
4. `iterators` package

Packages needed for today

```
# general
install.packages(c('rbenchmark', 'rmarkdown',
                   'knitr', 'gtools'))

# optimization

install.packages(c('optimx', 'pracma', 'ucminf',
                   'adagio', 'nloptr', 'dfoptim',
                   'lbfgs', 'alabama', 'CVXR', 'GA'))

# parallel computing

install.packages(c('parallel', 'foreach', 'iterators',
                   'doParallel', 'snow', 'doSNOW'))
```

Optimization in R

Optimization in R

- Today's material is based on an R User Group Meeting talk by Hans W. Borchers in Sept. 2017 (original slides).
- Current (March, 2020) optimization packages: 132 (R task view).

What is optimization and why do we need it?

- Loosely speaking, optimization is the mathematical/computational procedure to find maxima or minima of a function.
- Some typical uses include:
  - Maximum Likelihood
  - Penalized estimation, e.g. LASSO
  - Nonlinear equations
  - Deep Learning / Support Vector Machines
  - Operations Research, e.g. network flow, resource allocation

Issues commonly found

- optimization can be computationally expensive
- usually no "one size fits all" exists $\rightarrow$ different objective functions and domains
- very high accuracy is usually needed
- global optimum is typically pursued (but local optima are often present)

A 'simple' example: logistic regression

$$l(\beta) = \sum_{i=1}^{n} y_i \log[\phi(x_i^t \beta)] + (1 - y_i) \log[1 - \phi(x_i^t \beta)]$$

where

- $y_i \in \{0, 1\}$ is a binary outcome
- $x_i = (1, \ldots, x_{ip-1})$ is a set of p covariates
- n number of observations
- $\phi(\cdot)$ is the logistic function, i.e. $\phi(z) = \frac{1}{1+e^{-z}}$
- β vector of regression parameters to find

We can find the MLE by simply using, `glm()`, of course.

Logistic regression from scratch

```
llreg1 <- function(beta,y,x){
  ll <- 0
  for(i in 1:length(y)){
    mui <- gtools::inv.logit(sum(x[i,]*beta))
    ll <- ll + y[i]*log(mui) + (1-y[i])*log(1-mui)
  }
  return(-ll)
}

llreg2 <- function(beta,y,x){
    mu <- gtools::inv.logit(x%*%beta)
    ll <- sum( y*log(mu) + (1-y)*log(1-mu))
    return(-ll)
}

llreg3 <- function(beta,y,x){
  mu <- gtools::inv.logit(x%*%beta)
  ll <- sum(dbinom(y,size=1,prob=mu,log=TRUE))
  return(-ll)
}
```

Simulate some data and check functions

```
set.seed(1239)
n <- 100
Beta <- c(1, 0.5, -0.3) # true values
x <- cbind(x0=1,x1=rnorm(n),x2=rbinom(n,size=1,prob=0.4))
mu <- as.numeric(gtools::inv.logit(x%*%Beta))
y <- rbinom(n,1,prob=mu)

# Check functions at beta=c(0,0,0)
beta=c(0,0,0)
a=llreg1(beta,y,x); b=llreg2(beta,y,x); c=llreg3(beta,y,x)

all.equal(a,b); all.equal(b,c); all.equal(a,c)
```

```
## [1] TRUE

## [1] TRUE

## [1] TRUE
```

Objective function performance

- *Always* check your functions performance
- Vectorize them as much as you can
- Is the execution time reasonable for the task at hand?
- Modularize your code

Above, it's clear that `llreg1()` has really poor performance. Why?

Iterative Re-Weighted Least Squares (IRWLS)

`glm()` function uses IRWLS to find the maximum likelihood estimates for logistic regression (and glm's in general)

$$\beta^{k+1} = \beta^k - H^{-1}(\beta^k) l'(\beta^k),$$

where - $l'(\beta^k)$ is the gradient of $l(\beta)$ - H is the Hessian matrix of $l(\beta)$

For logistic regression, the first and second derivatives are relatively easy to find, thus, IRWLS can be implemented without much hassle

```
(mle <- glm.fit(x,y,family = binomial())$coef)
```

```
##          x0          x1          x2
##   0.5834096   0.6675419 -0.2332774
```

Unconstrained optimization

A naive call to optim()

```
r1 = optim(par=beta, fn=llreg2, y=y, x=x)
r1$par; r1$value; r1$counts; r1$convergence
```

```
## [1]  0.5832009  0.6677064 -0.2330699
```

```
## [1] 62.2522
```

```
## function gradient
##       80       NA
```

```
## [1] 0
```

```
# difference
all.equal(mle, r1$par, check.attributes = FALSE)
```

```
## [1] "Mean relative difference: 0.0003912279"
```

Improving performance in optim()

Part of the issue with the naive call to optim() is that the default method -Nelder-Mead-, although robust in multiple situations, tends to be relatively slow and all it only uses the fuction values

Let's try the function under the widely used Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm:

```
r2 = optim(par=beta, fn=llreg2, y=y, x=x, method="BFGS")$par

# difference
all.equal(mle, r2, check.attributes = FALSE)
```

```
## [1] "Mean relative difference: 8.280969e-07"
```

Improving performance in optim() (cont'd)

```
# gradient function, same parameters as llreg[1-3]
llgr <- function(beta,y,x){
  mu <- gtools::inv.logit(x%*%beta)
  return(-as.numeric(crossprod(x,(y-mu))))
}

r3 = optim(par=beta, fn=llreg2, gr=llgr, y=y, x=x, method="BFGS")$pa

# difference
all.equal(mle,r3,check.attributes = FALSE)
```

```
## [1] "Mean relative difference: 7.120584e-07"
```

Best practices for optim()

```
r4 = optim(par=beta, fn=llreg2, gr=llgr,
           y=y, x=x,
           method="L-BFGS-B",
           control =
             list(factr = 1e-10,
                  maxit = 50*length(par)))$par
# difference
all.equal(mle,r4,check.attributes = FALSE)
```

```
## [1] TRUE
```

For cases where no analytical gradient is available, compute numerical gradients (`dfoptim`, `numDeriv`, `pracma` or similar), e.g. `gr = function(x) pracma::grad(fn, x)`.

Of course, in this case, we know the "ground truth" provided by `glm.fit()`. In real applications we don't have that luxury.

Beyond optim()

Since the implementation of the routines in optim() were developed in the 1970's better tools have been developed

Function opm in package optimx provides an easy way to compare them all

```
llreg2_ <- function(beta) llreg2(beta,y,x)
llgr_ <- function(beta) llgr(beta,y,x)
llhes_ <- function(beta){
  mu <- gtools::inv.logit(x%*%beta)
  w <- as.numeric(sqrt(mu*(1-mu)))
  return(crossprod(x*w,x*w))
}
ans = opm(par=beta, fn=llreg2_, gr=llgr_,
          hess = llhes_,
          method = "ALL",
          control=list(trace=0))
```

Beyond optim() (cont'd)

	value	fevals	gevals	convergence	xtime	mean.dif
BFGS	62.252	18	7	0	0.007	-6.4e-08
CG	62.252	63	24	0	0.004	5.0e-08
Nelder-Mead	62.252	80	NA	0	0.005	5.4e-05
L-BFGS-B	62.252	9	9	0	0.001	-4.3e-07
nlm	62.252	NA	4	0	0.008	-3.2e-12
nlminb	62.252	5	5	0	0.002	-3.2e-12
lbfgsb3c	62.252	15	15	0	0.008	-1.0e-05
Rcgmin	62.252	30	14	0	0.004	4.6e-10
Rtnmin	62.252	14	14	0	0.007	-2.7e-08
Rvmmin	69.315	2	1	21	0.004	-3.4e-01
snewton	62.252	6	6	92	0.002	7.5e-14
snewtonm	62.252	16	5	0	0.002	-3.2e-12
spg	62.252	21	20	0	0.010	-3.7e-08

Constrained optimization

Box constraints

Consist of restricting the search in a specific parameter domain. See `lower` and `upper` options in optim().

What if the solver does not support bound contraints? $\rightarrow$ transfinite trick

Suppose we want to solve an optimization problem for $\theta = (\theta_1, \ldots, \theta_p)$ s.t. $l_i \leq \theta_i \leq u_i$, $i = 1, \ldots, p$.

Transfinite trick: Define $h : R^p \rightarrow [l_i, u_i]$, e.g.

$$h : \theta_i \rightarrow l_i + \frac{u_i - l_i}{2}[1 + \tanh(\theta_i)],$$

then optimize the composite function $g(\theta) = f(h(\theta))$, i.e. $g : R^p \rightarrow [l_i, u_i] \rightarrow R$

$$\theta^* = \min g(\theta) = f(h(\theta))$$

then $\theta^\dagger = h(\theta^*)$ will be a minimum of f in $[l_i, u_i]$.

Transfinite trick example

Let's minimize the logistic function with parameter domain in $[0, 0.5]$

```
Tf <- adagio::transfinite(0, 0.5, 3)
h <- Tf$h
hinv <- Tf$hinv

f <- function(beta) llreg2_(hinv(beta)) # f: R^p -> R
g <- function(beta) pracma::grad(f, beta)

soltf <- lbfgs::lbfgs(f, g, beta, epsilon=1e-10, invisible=1)

round(hinv(soltf$par),3)
```

```
## [1] 0.466 0.500 0.000
```

More general constraints

In general, we are interested in solving problems of the form:

$$\min f(\beta) \quad \text{s.t. } g(\beta) \geq 0, h(\beta) = 0.$$

Multitude of packages can tackle this problem e.g. `dfoptim`, `alabama`, `nloptr`, `Rsolp`. Some are derivative-free, some use augmented Lagrangian approaches.

A couple useful tricks

- For linear equality constraints, i.e. $A\beta = c$, we can find a solution by minimizing a new function $g(\xi) = f(\beta^* + B_0^t \xi)$ without constraints, where β^* is a special solution of $A\beta = c$ and B_0 is a basis of the *null space* of A.
- Equality constraints can be implemented if unavailable by specifing two sets of inequality constraints, i.e. $g(\beta) \geq 0$ and $g(\beta) \leq 0$.

A slighltly more advanced example

Let's try to maximize logistic regression with solution subject to a sum constraint of positive parameters, that is:

$$\max l(\beta) \quad \text{s.t.} \sum_{j=0}^{p-1} \beta_j = 1, \beta_j >= 0.$$

Null space trick (note that the box constraint is not enforced)

```
A <- matrix(1, 1, length(beta)) # \sum \beta_j = 1
N <- pracma::nullspace(A)
beta0 <- qr.solve(A, 1) # A beta = 1

fun <- function(s) llreg2_(beta0 + N %*% s)

sol0 <- ucminf::ucminf(c(0,0), fun)
xmin <- c(beta0 + N %*% sol0$par)

round(xmin,3); sum(xmin)
```

```
## [1]  0.583  0.662 -0.244
```

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

Augmented Lagrangian

```
fheq <- function(beta) sum(beta) - 1
fhin <- function(beta) c(beta)

sol1 <- alabama::auglag(beta, fn=llreg2_, gr=llgr_,
       heq = fheq, hin = fhin,
       control.outer = list(trace = FALSE,
                            method = "nlminb"))

round(sol1$par, 3); sum(sol1$par)
```

```
## [1] 0.412 0.588 0.000
```

```
## [1] 0.9999999
```

Derivative-free

```
fhin2 <- function(beta){
  ui <- rbind(-rep(1,length(beta)),rep(1,length(beta)))
  ci <- c(-1,1)
  return(as.numeric(ui%*%beta-ci))
}
sol2 <- cobyla(beta, fn=llreg2_, lower=rep(0,length(beta)), hi

round(sol2$par, 3); sum(sol2$par)
```

```
## [1] 0.412 0.588 0.000
```

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

CVXR: An R Package for Disciplined Convex Optimization

CVXR is a package that provides a very flexible modeling language for convex optimization problems

We can recreate logistic regression under the CVXR framework:

```
p <- length(beta)
betaHat <- Variable(p)
obj <- -sum(x[y <= 0, ] %*% betaHat) - sum(logistic(-x %*% betaHat))
problem <- Problem(Maximize(obj))
result <- solve(problem)

beta_res <- as.numeric(result$getValue(betaHat))

all.equal(beta_res, mle, check.attributes = FALSE)
```

```
## [1] TRUE
```

CVXR (cont'd)

The power of this package lies in how easily we can add constraints

```
problem1 <- Problem(Maximize(obj),
                    constraints = list(betaHat >= 0))
result1 <- solve(problem1)

beta_res1 <- as.numeric(result1$getValue(betaHat))

round(beta_res1,3)
```

```
## [1] 0.497 0.682 0.000
```

CVXR (cont'd)

An example from before:

```
constraint1 <- betaHat <= 0.5
constraint2 <- betaHat >= 0
problem2 <- Problem(Maximize(obj),
                    constraints = list(constraint1, constraint2))

result2 <- solve(problem2)

beta_res2 <- as.numeric(result2$getValue(betaHat))

round(beta_res2,3)
```

```
## [1] 0.466 0.500 0.000
```

```
all.equal(hinv(soltf$par),beta_res2)
```

```
## [1] TRUE
```

```
##
```

CVXR (cont'd)

Another example from before

```
constraint3 <- list(sum(betaHat) == 1, betaHat>=0)
problem3 <- Problem(Maximize(obj), constraint3)
result3 <- solve(problem3)

beta_res3 <- as.numeric(result3$getValue(betaHat))

round(beta_res3,3); sum(beta_res3)
```

```
## [1] 0.412 0.588 0.000
```

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

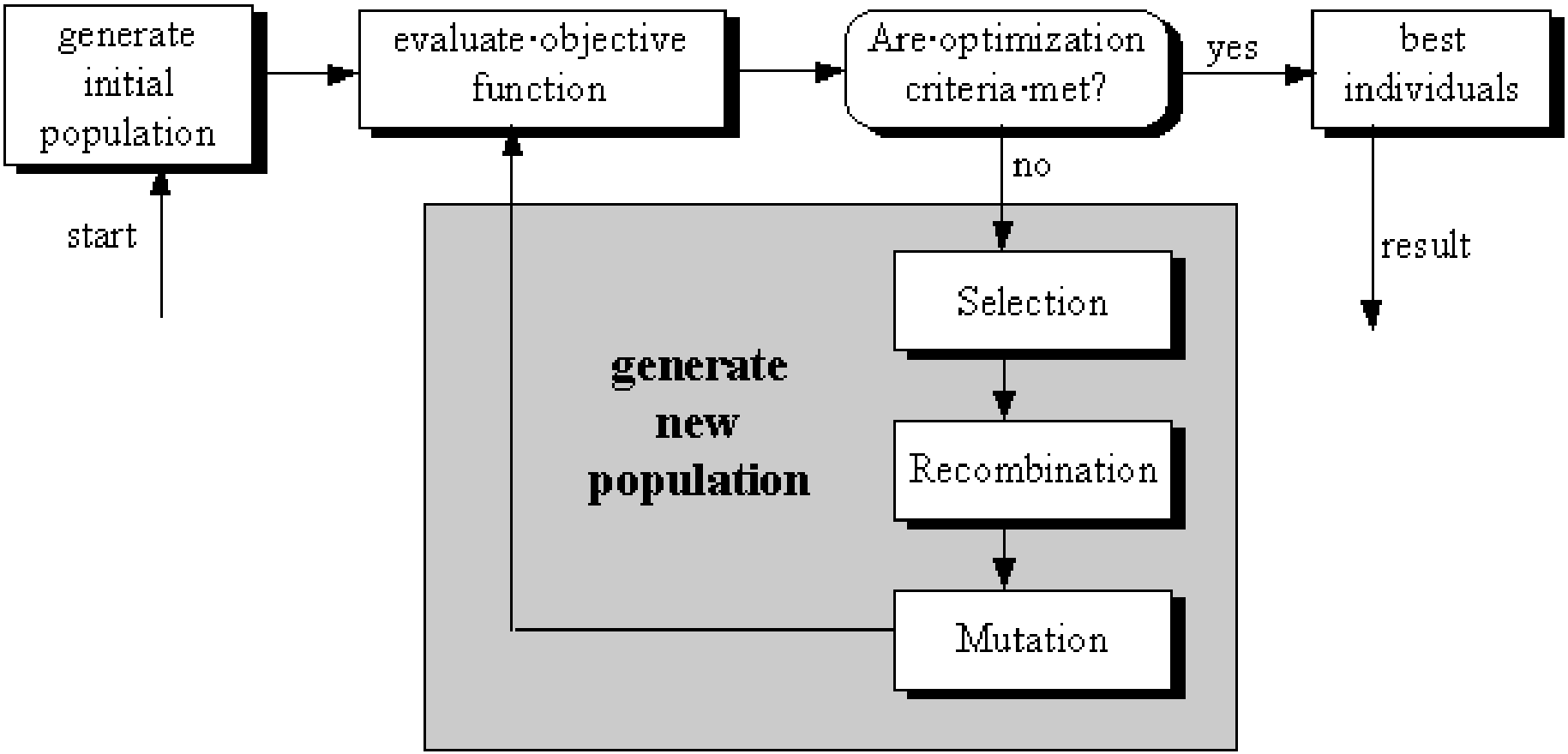
```
all.equal(sol2$par,beta_res3)
```

```
## [1] "Mean relative difference: 1.053877e-06"
```

Stochastic optimization

Genetic algorithms (GAs)

- mimic nature's evolutionary processes
- are typically designed to solve discrete optimization problems
- tend to work well in large search space cases



GA example

Issues with GAs

- No clear convergence criteria
- No guarantee of reaching a global optimum $\rightarrow$ approx. solutions
- Application-specific, algorithm parameters need to be tuned appropriately

Parallel computing in R

Parallel computing in R

- Current (March, 2020) high-performance and parallel computing packages: 95 (R task view).
- Most recent computers come equipped with a fair ammount of processing power, e. g. recent Intel Core i9 chips come with 8 cores

Some remarks

- Computation has become increasingly inexpensive in the recent times
- Scientific computing has benefited greatly from these advances and many routines and algorithms have incorporated parallelism
- It is important to know when/where/if any of these routines are being used within a given R package
- This part of the workshop will focus on embarrasingly parallel problems

Shared vs. distributed memory

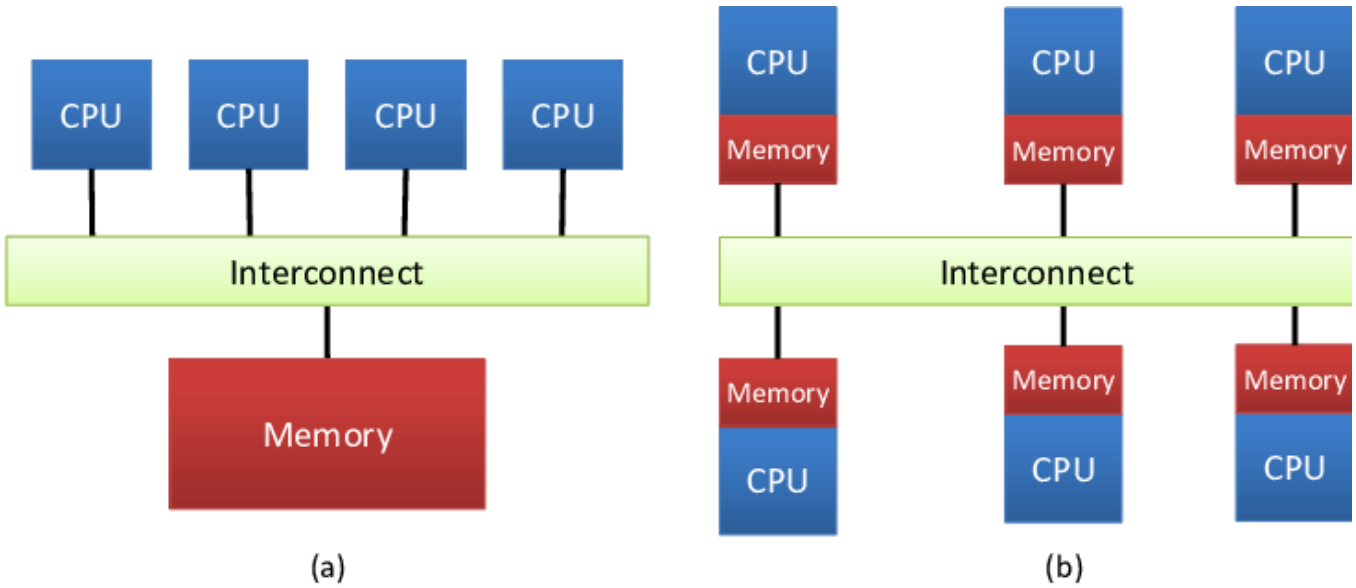


Figure 2

Out-of-the-box implementations

Since R v2.14.0, the package `parallel` is part of the base distribution

`parallel` comes with parallel versions of the family of *apply functions, e.g. apply, lapply, sapply.

Example code:

```
n <- 10; sd <- 2
# calculate the number of cores
no_cores <- parallel::detectCores() - 1
# initiate cluster
cl <- makeCluster(no_cores)
# run
parLapply(cl, 2:4, function(mean)rnorm(n,mean,sd))
```

```
## Error in checkForRemoteErrors(val): 3 nodes produced errors; firs
```

```
# stop cluster
stopCluster(cl)
```

Initializing the cores

In parallel computing, required information (n and sd) needs to be passed to all the cores prior to execution

This is achieved as follows:

```
n <- 10; sd <- 2
# initiate cluster
cl <- parallel::makeCluster(no_cores)
parallel::clusterExport(cl, c("n","sd"))
# run
parLapply(cl, 2:4, function(mean)rnorm(n,mean,sd))
# stop cluster
stopCluster(cl)
```

In addition of `clusterExport`, `parallel` has additional function to initialize variables, functions or packages in remote clusters, see the help page of `?clusterExport` for more details

Overhead

Despite what one may think, parallel computation is not always faster, why?

The reason is **overhead**

This is because by using multiple cores one needs to initialize and pass information among them. This preparation/communication adds some computational burden. Consequently, the performance increase is highly dependent on the type of application. Typically, fast computations with efficient use of processing power won't benefit as much as more time-consuming applications.

Random number generation

In many instances, we are interested in making our results reproducible, which is usually achieved in the sequential setting by setting up a *seed*.

The specific way of setting a seed in parallel implementations is:

```
cl <- parallel::makeCluster(no_cores)

clusterSetRNGStream(cl, rep(403,6) )
res1 <- parLapplyLB(cl,rep(100,3),function(n){
  rnorm(n,mean=1,sd=2)})

clusterSetRNGStream(cl, rep(403,6) )
res2 <- parLapplyLB(cl,rep(100,3),function(n){
  rnorm(n,mean=1,sd=2)})

stopCluster(cl)
all.equal(res1,res2)
```

```
## [1] TRUE
```

A limitation

- `parallel` was designed for usage in shared memory architectures, For distributed memory architectures, package `snow` provides a robust alternative
- interestingly, `snow` works well for either architechture, thus, it is a good practice to stick with it

foreach

foreach

Because of all the housekeeping that needs to be done using `parallel`, it tends to be burdensome to keep track of all variables/packages/functions that need to be passed to remote cores

Luckily, package `foreach` greatly helps with this

Basic call for foreach

```
cl <- parallel::makeCluster(no_cores)
doParallel::registerDoParallel(cl)

res <- foreach(..., # controls the "loop"
         .combine, # how the results are put together
         # (usually equals c, rbind, cbind)
         .inorder=TRUE,
        .errorhandling=c('stop', 'remove', 'pass'),
        .packages=NULL,
        .export=NULL,
        .noexport=NULL,
        .verbose=FALSE) %dopar%{

         # do something for a given iteration of the "loop"` #

          }
stopCluster(cl)
```

Appeal of foreach

- loop-like interfase
- seamless passing of needed variables, dataframes, functions (need to explictly define packages, however)
- flexibility in the way results are combined

An example (logistic regression, anyone?)

Cross-validation, a parallelized version

```
# number of folds
cvfolds <- 5

# data from the GA example
data.cv <- data.frame(y=Y,X[,gsub("[.]",":",bestvars)])

# regression formula
reg_formula <- as.formula(paste0("y~",paste(bestvars, collapse="+"))

# divide data in equally-sized folds (at random)
set.seed(28197)
data.cv$fold <- cut(sample(nrow(data.cv)),
                    breaks=cvfolds,labels=FALSE)
```

Cross-validation in parallel

```
cl <- makeCluster(no_cores)
registerDoSNOW(cl) # could also use registerDoParallel()
res.fe <- foreach(foldi=1:cvfolds,
                  .combine = c,
                  .inorder=TRUE,
                  .verbose=TRUE) %dopar%
  {
    fit <- glm(reg_formula,data.cv[data.cv$fold!=foldi,],
        family=binomial())
    pred <- predict(fit,data.cv[data.cv$fold==foldi,])
    resi <- mean((data.cv$y[data.cv$fold==foldi]-pred)^2)
    return(resi)
  }
stopCluster(cl)
```

One practical recommendation

Suppose you have a dataframe (or a vector) called "`data`" that can be somehow indexed (or split) by variable `indx`, e.g. a replicate, a fold, a centre, etc.

A not-so-great idea

Can you say why?

```
cl <- parallel::makeCluster(no_cores)
doParallel::registerDoParallel(cl)

res <- foreach(indxi = 1:nindx, .combine = rbind,
         .inorder=FALSE,
        .errorhandling='remove',
        .verbose=TRUE) %dopar%{

         datai = data[data$indx==indxi,]

         # ... do something with datai... #

       }
stopCluster(cl)
```

A better idea

Why?

```
cl <- parallel::makeCluster(no_cores)
doParallel::registerDoParallel(cl)

res <- foreach(datai = isplit(data, list(indxi=data$indx)),
         .combine = rbind,
         .inorder=FALSE,
         .errorhandling='remove',
         .verbose=TRUE) %dopar%{

         # ... do something with datai... #

       }
stopCluster(cl)
```

iterators

iterators

In most cases, it's better to pass only the portion of the data we are dealing with for a given iteration/core.

icount

Performs a sequential count

```
cl <- makeCluster(no_cores)
registerDoSNOW(cl)
clusterSetRNGStream(cl, rep(4039,6) )
res <- foreach(iter = icount(10),
          .combine='rbind',
          .verbose=FALSE) %dopar% {
    return(summary(rnorm(1000,mean=iter)))
}
stopCluster(cl)
```

icount (cont'd)

```
round(res,3)
```

```
##              Min. 1st Qu. Median  Mean 3rd Qu.   Max.
## result.1   -1.676   0.304  1.023 1.038   1.744  4.101
## result.2   -0.865   1.361  2.017 2.038   2.702  5.212
## result.3   -0.126   2.258  2.987 2.980   3.669  5.967
## result.4    0.851   3.285  3.949 3.969   4.646  7.461
## result.5    1.602   4.302  4.983 4.991   5.681  8.124
## result.6    2.215   5.391  6.014 6.023   6.660  9.518
## result.7    3.595   6.355  7.009 7.014   7.650 10.476
## result.8    4.907   7.327  8.068 8.013   8.665 10.894
## result.9    5.522   8.387  9.087 9.046   9.726 12.031
## result.10   7.122   9.290 10.003 9.982  10.641 13.244
```

Note that if this iterator is run without an argument, i.e. `icount()`, it will keep counting indefinitely.

iter

This function iterates over a variety of objects, more commonly matrices or dataframes. In particular, it allows to iterate over columns, rows or individual cells

```
iters.df <- expand.grid(mean=0:2,sd=3:5)

cl <- makeCluster(no_cores)
registerDoSNOW(cl)
clusterSetRNGStream(cl, rep(4039,6) )
res <- foreach(iter = iter(iters.df, by='row'),
               .combine='rbind',
               .verbose=FALSE) %dopar%
  {
    mean.iter = iter$mean
    sd.iter = iter$sd
    x = rnorm(1000, mean=mean.iter, sd=sd.iter)
    return(c(summary(x),SD=sd(x)))
  }
stopCluster(cl)
```

iter (cont'd)

```
round(res,3)
```

```
##                Min. 1st Qu. Median   Mean 3rd Qu.   Max.    SD
## result.1   -8.027  -2.088  0.068  0.114   2.231  9.302 3.038
## result.2   -7.595  -0.917  1.052  1.114   3.107 10.636 2.934
## result.3   -7.379  -0.225  1.960  1.941   4.008 10.900 3.061
## result.4  -13.590  -2.792 -0.068 -0.035   2.725 12.497 4.029
## result.5  -11.597  -1.861  0.798  0.875   3.583 14.844 4.026
## result.6  -10.372  -0.691  2.272  2.052   4.660 13.577 3.969
## result.7  -18.924  -3.044  0.068  0.117   3.301 17.592 4.823
## result.8  -15.156  -2.213  1.328  1.302   4.805 18.830 5.149
## result.9  -15.025  -1.223  2.045  2.068   5.249 19.381 5.098
```

isplit

This iterator allows to divide a given vector or dataframe into groups according to a factor or list of factors

```
x <- rnorm(200)
f <- factor(sample(1:10, length(x), replace=TRUE))

cl <- makeCluster(no_cores)
registerDoSNOW(cl)
res <- foreach(iter = isplit(x, list(f=f)),
               .combine='rbind',
               .verbose=FALSE) %dopar% {

  factoriter <- iter$key$f
  xiter <- iter$value

  return(c(f=as.numeric(factoriter),
           summary(xiter),SD=sd(xiter)))
}
stopCluster(cl)
```

isplit

```
round(res,3)
```

```
##              f   Min. 1st Qu. Median   Mean 3rd Qu.  Max.    SD
## result.1     1 -1.271  -0.667 -0.427 -0.193   0.065 1.207 0.740
## result.2     2 -1.414  -0.696  0.006 -0.104   0.150 1.859 0.819
## result.3     3 -1.220  -0.329 -0.108 -0.071   0.394 0.843 0.553
## result.4     4 -1.401  -0.532 -0.034  0.073   0.482 1.809 0.935
## result.5     5 -2.008  -0.837  0.086 -0.062   0.557 1.910 0.892
## result.6     6 -2.722  -0.769 -0.191 -0.288   0.220 2.921 1.138
## result.7     7 -2.579  -0.670 -0.097 -0.145   0.498 1.206 0.946
## result.8     8 -2.058  -0.257  0.410  0.188   0.976 1.930 1.083
## result.9     9 -1.089  -0.164  0.669  0.489   0.867 1.758 0.906
## result.10 10 -1.899  -0.271  0.127  0.151   0.701 2.402 0.938
```

Take-home messages

- *always* benchmark your code
- squeeze as much performance as you can in your objective function/gradient (if possible/available)
- know your solver! do research on best practices and useful tricks
- take advantage of available computing power
- be mindful of what you are passing to the cores, this can greatly impact performance

Resources

- Numerical Optimization in R: Beyond optim
- On Best Practice Optimization Methods in R
- `CVXR` vignette
- `foreach` vignette
- Intro to parallel computing in R
- A guide to parallelism in R
- Compute Canada
- SciNet