# Parallel Optimization with DEoptim

Presented at St. Louis RUG
February, 2012

FOSS Finance

# What is Differential Evolution (DE)?

- Created by Storn & Price (1997)
- Genetic Algorithm
- Stochastic global optimizer
- DEoptim is a R implementation

# Where DEoptim excels

- Functions with multiple local extrema
- Converges with relatively few function evaluations
- Can find global minimum for very ill-behaved functions (not continuous and/or differentiable)
- Gradient-based methods (e.g. Nelder-Mead) are better for well-behaved functions

# How it works in parallel

- Create entire population
- Split population into chunks
- Evaluate the chunks on the nodes
- Return results to the head node to create population for next iteration
- Wash, Rinse, Repeat

# Why make it parallel?

- Portfolio Optimization Example
  - 100 stocks
  - Optimize for lowest conditional VaR
  - Constrained by maximum 5% CVaR contribution
- Execution takes ~12 min on one 2.8Ghz processor
- More computationally intensive objective functions can take hours or days

# Objective function

```r
# load necessary packages
library(PerformanceAnalytics)
library(PortfolioAnalytics)
library(DEoptim)
library(doSNOW)
# load data (included in DEoptim source)
load("DEoptim/sandbox/10y_returns.rda")
load("DEoptim/sandbox/random_portfolios.rda")
# mean returns and covariance matrix
mu <- colMeans(R)
sigma <- cov(R)
# objective function: note that non-varying parameters mu and
# sigma are not explicitly passed via the function definition
obj <- function(w) {
  if(sum(w)==0) w <- w + 1e-2
  w <- w / sum(w)
  CVaR <- ES(weights=w, method="gaussian",
    portfolio_method="component", mu=mu, sigma=sigma)
  tmp1 <- CVaR$ES
  tmp2 <- max(CVaR$pct_contrib_ES - 0.05, 0)
  out <- tmp1 + 1e3 * tmp2
  return(out)
}
```

FOSS Finance

# Sequential optimization

```
# DEoptim settings
DEctrl <- list(NP=nrow(rp), initialpop=rp, trace=250,
  itermax=5000, reltol=0.000001, steptol=150)

set.seed(1234)
lower <- rep(0,ncol(R))
upper <- rep(1,ncol(R))
# run optimization: note we don't pass mu and sigma via '...'
# because they're not in the function definition. R will find
# them in the global environment
system.time(out <- DEoptim(obj, lower, upper, DEctrl))
Iteration: 250 bestvalit: 0.067308
Iteration: 500 bestvalit: 0.058007
Iteration: 750 bestvalit: 0.056007
Iteration: 1000 bestvalit: 0.055155
Iteration: 1250 bestvalit: 0.054629
Iteration: 1500 bestvalit: 0.054447
Iteration: 1750 bestvalit: 0.054179
Iteration: 2000 bestvalit: 0.053955
   user  system elapsed
738.990   0.020 739.507
```

# Parallel optimization

```
# create socket cluster
cl <- makeSOCKcluster(4)
# load necessary packages on each node
clusterEvalQ(cl, library(PerformanceAnalytics))
# copy necessary objects to each node
clusterExport(cl, list("mu", "sigma"))
# register foreach backend
registerDoSNOW(cl)
set.seed(1234)
# run optimization: note we don't pass mu and sigma again.
# R will find them in each node's global environment
system.time(out <- DEoptim(obj, lower, upper, DEctrl))
Iteration: 250 bestvalit: 0.067308
Iteration: 500 bestvalit: 0.058007
Iteration: 750 bestvalit: 0.056007
Iteration: 1000 bestvalit: 0.055155
Iteration: 1250 bestvalit: 0.054629
Iteration: 1500 bestvalit: 0.054447
Iteration: 1750 bestvalit: 0.054179
Iteration: 2000 bestvalit: 0.053955
   user  system elapsed
 67.980   3.070 356.912
stopCluster(cl) # stop cluster
```

# Parallel optimization notes

- Data communication between nodes can overwhelm gains from processing on multiple CPUs
- Be careful with non-varying objects
  - Exclude them from formal function arguments
  - Copy them to nodes before optimization (clusterExport)
- If mu and sigma were formal function arguments, they would be copied to each node for all 2037 function evaluations!

# Comparison

- 2.8Ghz quad core AMD
- Sequential optimization takes ~12 min
- Parallel optimization takes ~6 min
  - Only ~60% CPU utilization
  - Improvement will be closer to linear if CPU utilization is closer to 100%