REvolution computing

We do the math

Parallel Computing with R

Bay Area R User Group
May 13, 2009

Agenda

- About Revolution Computing
- Parallel Processing Overview
- Iterators
- Parallel Computing w/foreach
- Q & A

Who is REvolution Computing?

- REvolution Computing is the leading commercial provider of software and support for the open-source statistical computing language R. Our products enable statisticians, scientists and others to derive meaning from large sets of mission-critical data in record time, and to create predictive models that help to answer their most difficult questions.
- Based in New Haven and Seattle with partners worldwide
- Team of expert developers, statisticians and computer scientists.
- 2007, acquired assets of Scientific Computing which included significant High Performance Computing related technology

Customers and partners

- Select Customers:
- Select Partners:

Supporting the R Community

We are an open source company supporting the R community:

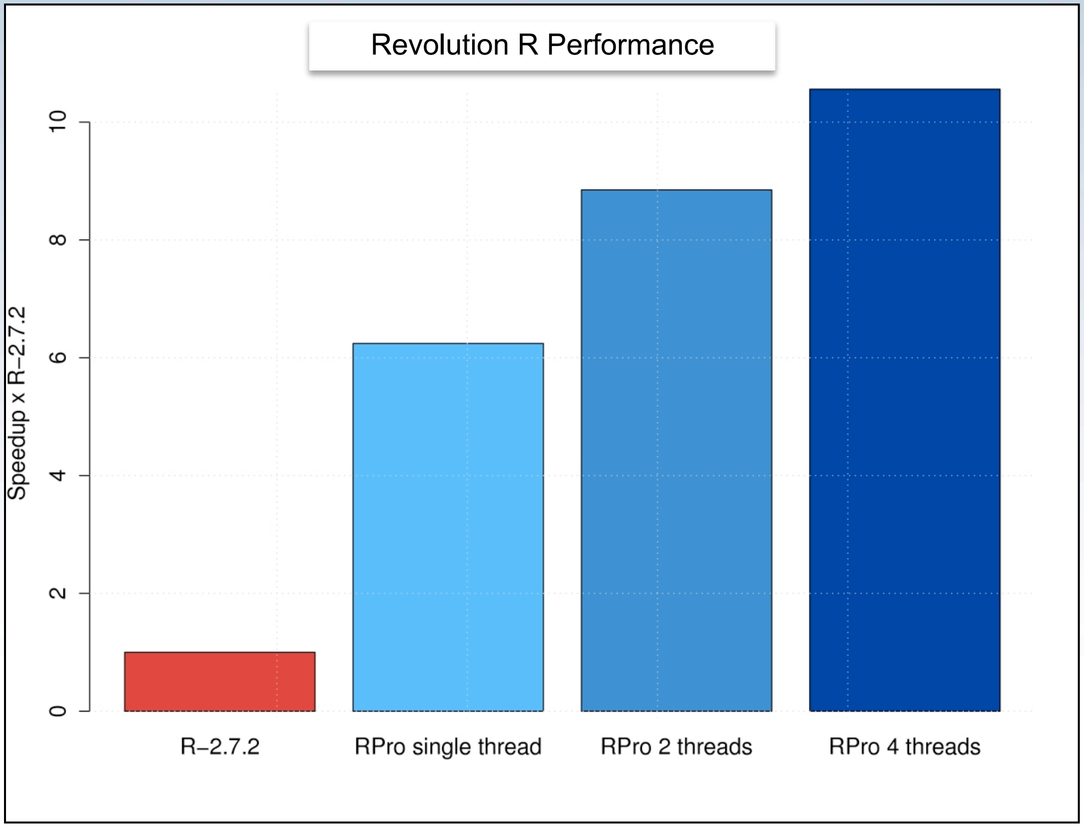
- **Benefactor of R Foundation**
- **Financial supporter of R conferences and user groups**
- **Zero-cost "REvolution R" available to everyone**
- **R Community website:** revolution-computing.com/community
  - **"Revolutions" Blog:** blog.revolution-computing.com
  - **Forum:** revolution-computing.com/forum
- **New functionality developed in core R to contributed under GPL**
  - **64-bit Windows support**
  - **Step-debugging support**
- **Promoting R use in the commercial world**

A Taxonomy of Parallel Processing

- Multi-threaded processing (lightweight processes)
  - OpenMP / POSIX threads
  - Multiprocessor / Multicore
  - GPU processors (CUDA/NVIDIA ; ct/INTEL)
  - *Usually* shared memory
  - Harder to scale out across networks
  - Examples: multicore (Unix), threaded linear-algebra libraries for R (ATLAS, MKL)

- Multi-process processing (heavyweight processes)
  - *Usually* distributed memory
  - Easier to scale out across networks
  - Examples: SNOW, ParallelR, Rmpi, batch processing

REvolution R SVD Performance



Example data matrix

150,000 x 500

fast.svd

Quad-core Intel Core2 CPU,

Windows Vista 64-bit Workstation

Revolution R Enterprise

An enhanced distribution of R, designed for use in commercial environments.

- **High-performance and scalability**
- **Optimised, multi-threaded math routines**
- **Stable release cycle putting version control in hands of user**
- **Product Support: documentation, installation qualification, and support for every user**
- **Support for regulated environments**
- **Commercial support and training**
- **Support for 64-bit Windows, Linux**
- **Visual Studio IDE integration (soon)**

REvolution Enterprise ParallelR Module

Open-source, script-level parallel/distributed computing with R

- **Easy to use and install on workstations, clusters and grids**
- **Heterogeneous system support**
- **Avoid recoding with MPI for production work**
- **Elegant distributed-shared memory paradigm**
- **Support for enterprise schedulers like Microsoft**
  HPC Server 2008, Platform LSF, SGE

PARALLEL R

ParallelR – scale computationally intensive analysis to improve time to results and better leverage your computing assets

foreach + iterators

foreach

- A for-loop/lapply hybrid
- Similar syntax to list comprehensions

iterators

- Similar to Java iterators
- nextElem ()

Iterators

- Generalized loop variable
- Value need not be atomic
  - Row of a matrix
  - Random data set
  - Chunk of a data file
  - Record from a database
- Create with: `iter`
- Get values with: `nextElem`
- More commonly: argument to `foreach`

Numeric Iterator

```
> i <- iter(1:3)
> nextElem(i)
[1] 1
> nextElem(i)
[1] 2
> nextElem(i)
[1] 3
> nextElem(i)
Error: StopIteration
```

Long sequences

```
> i <- icount(1e9)
> nextElem(i)
[1] 1
> nextElem(i)
[1] 2
> nextElem(i)
[1] 3
> nextElem(i)
[1] 4
> nextElem(i)
[1] 5
```

Matrix dimensions

```
> M <- matrix(1:25,ncol=5)
> r <- iter(M,by="row")
> nextElem(r)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
> nextElem(r)
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    7   12   17   22
> nextElem(r)
     [,1] [,2] [,3] [,4] [,5]
[1,]    3    8   13   18   23
```

Infinite & Irregular sequences

```
iprime <- function() {
 lastPrime <- 1
 nextEl <- function() {
  lastPrime <<- as.numeric(nextprime(lastPrime))
  lastPrime
 }
 it <- list(nextElem=nextEl)
 class(it) <- c('abstractiter','iter')
it}
```

```
> require(gmp)
> p <- iprime()
> nextElem(p)
[1] 2
> nextElem(p)
[1] 3
```

Data File

```
> rec <- iread.table("MSFT.csv",sep=",", header=T, row.names=NULL)
> nextElem(rec)
  MSFT.Open MSFT.High MSFT.Low MSFT.Close MSFT.Volume MSFT.Adjusted
1     29.91     30.25     29.4      29.86    76935100         28.73
> nextElem(rec)
  MSFT.Open MSFT.High MSFT.Low MSFT.Close MSFT.Volume MSFT.Adjusted
1      29.7     29.97    29.44      29.81    45774500         28.68
> nextElem(rec)
  MSFT.Open MSFT.High MSFT.Low MSFT.Close MSFT.Volume MSFT.Adjusted
1     29.63     29.75    29.45      29.64    44607200         28.52
> nextElem(rec)
  MSFT.Open MSFT.High MSFT.Low MSFT.Close MSFT.Volume MSFT.Adjusted
1     29.65      30.1    29.53      29.93    50220200          28.8
```

Database

```
> library(RSQLite)
> m <- dbDriver('SQLite')
> con <- dbConnect(m, dbname="arrests")
> it <- iquery(con, 'select * from USArrests', n=10)
> nextElem(it)
            Murder Assault UrbanPop Rape
Alabama       13.2     236       58 21.2
Alaska        10.0     263       48 44.5
Arizona        8.1     294       80 31.0
Arkansas       8.8     190       50 19.5
California     9.0     276       91 40.6
Colorado       7.9     204       78 38.7
Connecticut    3.3     110       77 11.1
Delaware       5.9     238       72 15.8
Florida       15.4     335       80 31.9
Georgia       17.4     211       60 25.8
```

Looping with `foreach`

```
foreach (var=iterator) %dopar% { statements }
```

- Evaluate `statements` until `iterator` terminates
- `statements` will reference variable `var`
- Values of `{ … }` block collected into a list

- Runs sequentially (by default) (or force with `%do%`)

```
> foreach (j=1:4) %dopar% sqrt (j)

[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051

[[4]]
[1] 2
```

Combining Results

```
> foreach(j=1:4, .combine=c) %dopar% sqrt(j)
[1] 1.000000 1.414214 1.732051 2.000000

> foreach(j=1:4, .combine='+') %dopar% sqrt(j)
[1] 6.146264
```

- When order of evaluation is unimportant, use `.inorder=FALSE`

Referencing global variables

```
> z <- 2
> f <- function (x) sqrt (x + z)
> foreach (j=1:4, .combine='+') %dopar% f(j)
[1] 8.417609
```

- foreach automatically inspects code and ensures unbound objects are propagated to the evaluation environment

A simple simulation:

```
birthday <- function(n) {
  ntests <- 1000
  pop <- 1:365
  anydup <- function(i)
      any(duplicated(
         sample(pop, n, replace=TRUE)))
  sum(sapply(seq(ntests), anydup)) / ntests
}

x <- foreach (j=1:100) %dopar% birthday (j)
```

Parallel execution, dual-core

```
> s <- sleigh(workerCount=2)
> registerDoNWS(s)
> system.time(
+ x <- foreach (j=1:100) %dopar% birthday (j)
+ )
   user  system elapsed
  0.669   0.137  28.392
```

%dopar%

Modular parallel backends

- doSEQ (default)
- doNWS (NetWorkSpaces)
- doSNOW
- doRMPI
- doMulticore

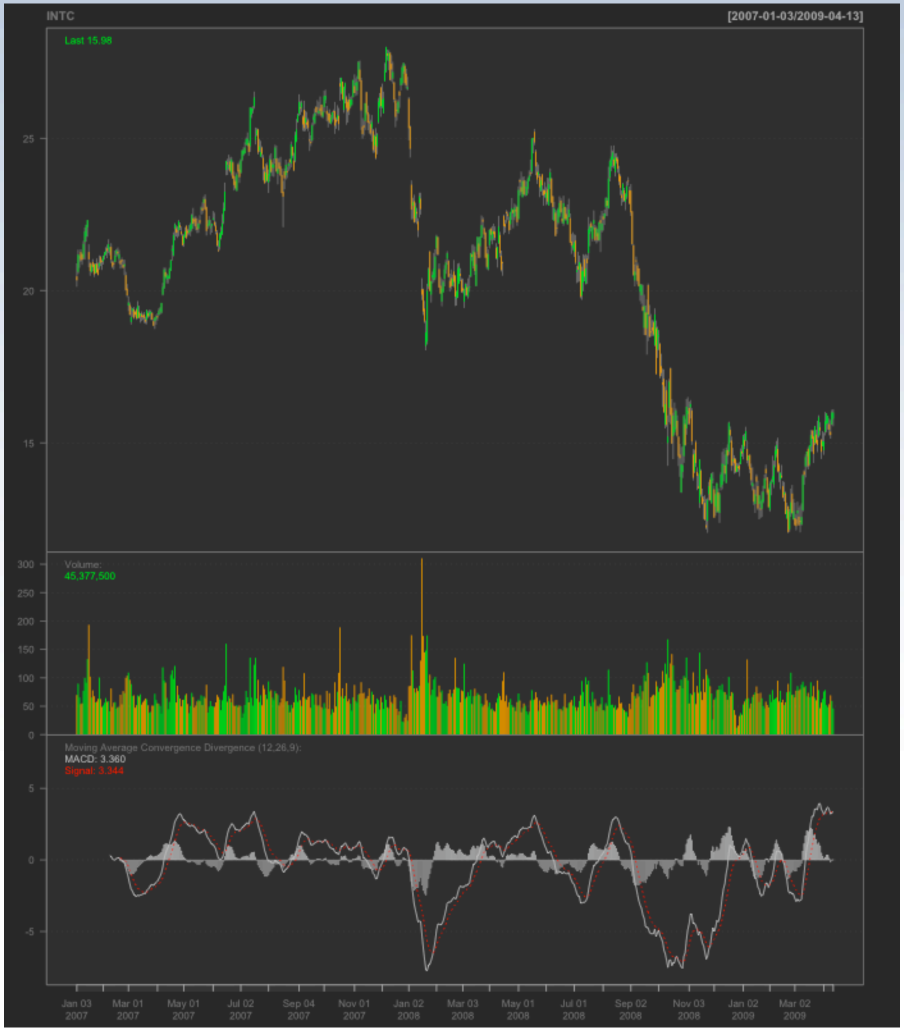
Quick review of distributed computing in R

- NetWorkSpaces
  - GPL, also commercially supported by REvolution Computing
  - Very cross-platform, distributed shared-memory paradigm
  - Fault-tolerant
- Rmpi
  - Fine-grained control allows very high-performance calculations
  - Can be tricky to configure
  - Limited Windows and heterogeneous cluster support
- SNOW
  - Limited Windows support (single machine only)
  - Meta-package: supports MPI, sockets, NWS, PVM

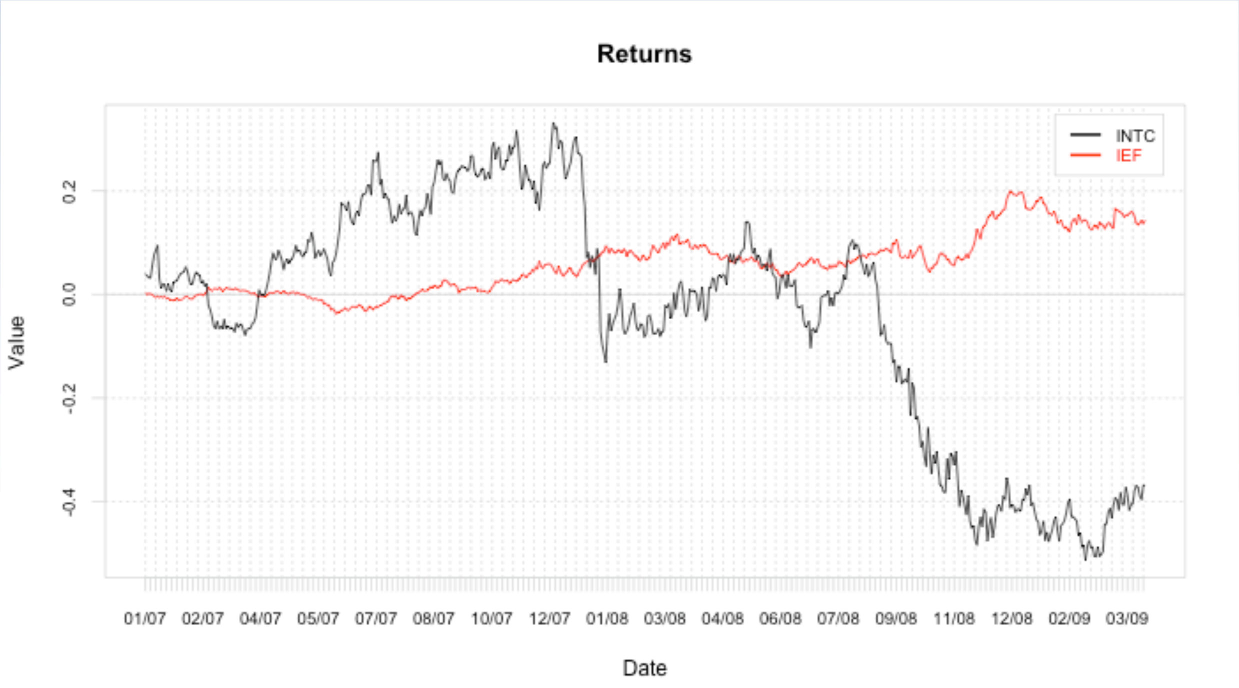
Financial Backtesting Example

- Automated trading of Intel (INTC) stock
- Put everything in INTC on "buy" signal
- Put everything in IEF (10-year Treasury bonds) on "sell" signal
- Signal: MACD (**moving-average** convergence/divergence) Oscillator
  - Buy when "short-term" MA exceeds "long-term" MA for "a while"
  - Short-term: fast moving average of **nFast** days
  - Long-term: slow-moving average of **nSlow** days (nSlow > nFast)
  - A while: MA of (short+long)/2: **nSig** days

```
> chartSeries(INTC)
> addMACD(fast=12,
  slow=26, signal=9)
```



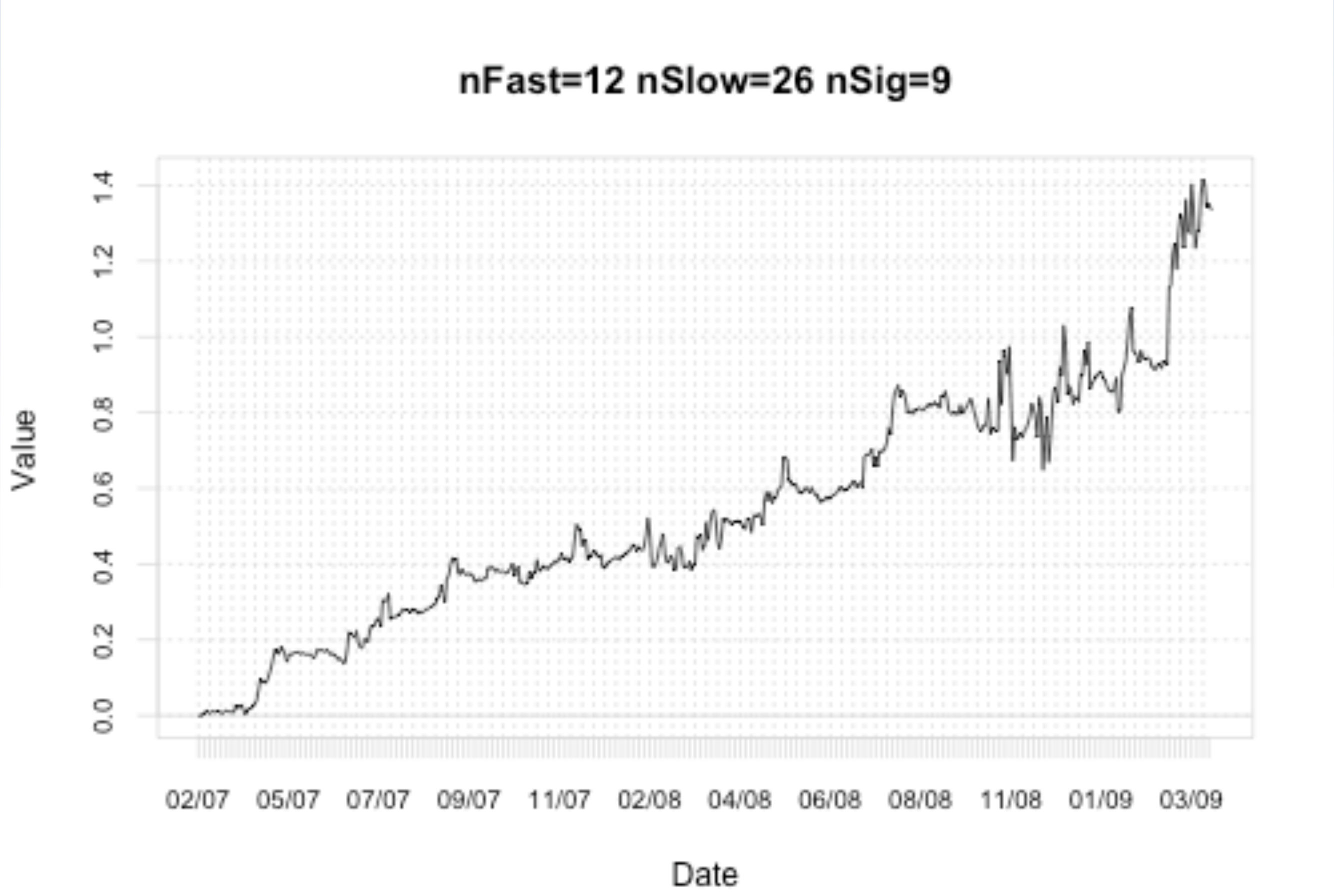
```
> Ra <- Return.calculate(Cl(INTC))
> Rb <- Return.calculate(Cl(IEF))
> chart.CumReturns (cbind (Ra,Rb))
```



A very simple trading rule:

```
simpleRule <- function (z, fast=12, slow=26,
                        signal=9, long, benchmark)
{
  x <- MACD (z, nFast=fast, nSlow=slow,
                        nSig=signal, maType="EMA")
  position <- sign(x[,1]-x[,2])
  s <- xts(position,order.by=index(z))
  return (long*(s>0) + benchmark*(s<=0))
}
```

```
> R.def <- simpleRule (z$INTC, fast=12, slow=26, signal=9,
  long=Ra, benchmark=Rb)
> chart.CumReturns(R.def, main="nFast=12 nSlow=26 nSig=9")
```

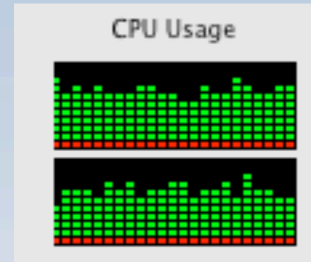


Optimizing the parameters

- Goal is to find the “best” nFast, nSlow, nSig
- “Best” is the trading rule that maximizes Sharpe Ratio
  - Measure of return given the risk

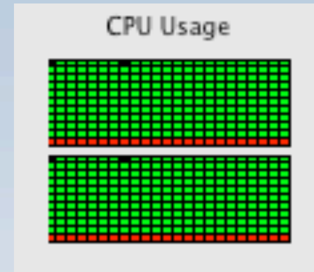
```
> Dt <- na.omit(R - Rb)
> sharpe <- mean(Dt)/sd(Dt)
> print (paste("Ratio = ",sharpe))
[1] "Ratio =  0.0727204602764740"
```

Brute-force parameter optimization (fix nSig=9):



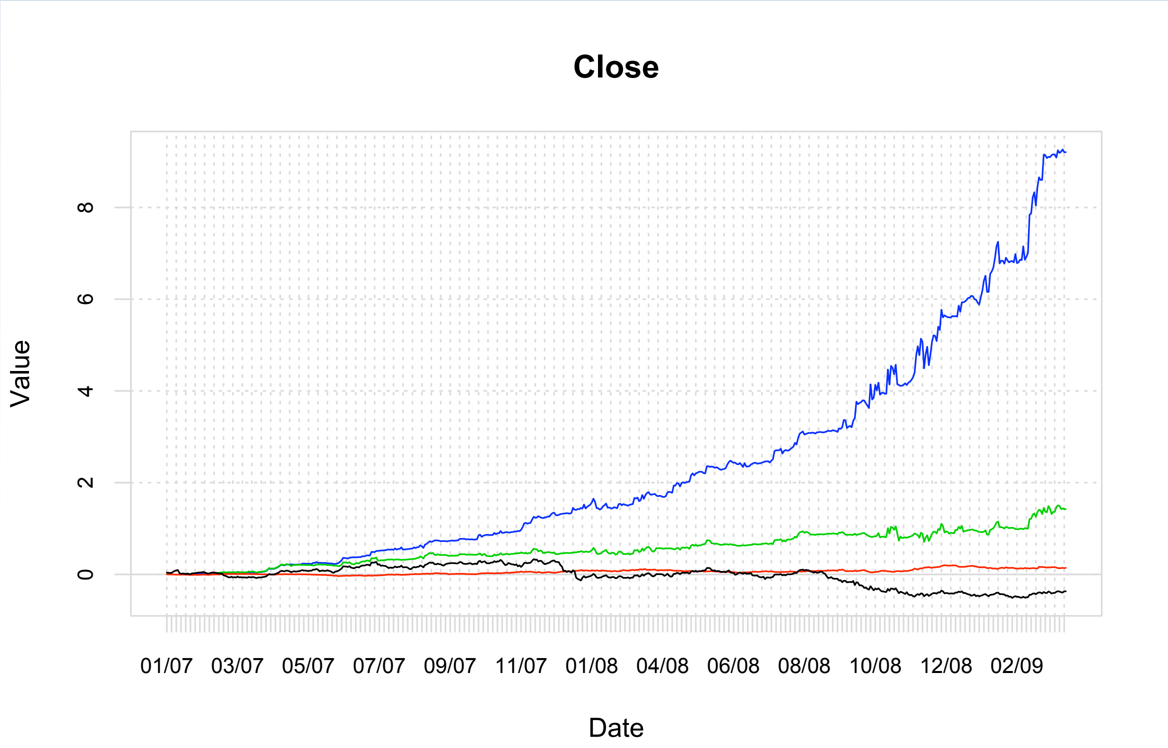

```
M <- 100
S <- matrix(0,M,M)

for (j in 5:(M-1)) {
  for (k in min ((j+2),M):M) {
    R <- simpleRule (INTC$Close,j,k,9, Ra, Rb)
    Dt <- na.omit (R - Rb)
    S[j,k] <- mean (Dt)/sd(Dt)
  }
}
```

With foreach:




```
s <- sleigh(workerCount=2)
registerDoNWS(s)

S <- foreach (j=5:(M-1), .combine=rbind,
          .packages=c('xts','TTR')) %dopar% {
    x <- rep(0,M)
    for (k in min ((j+2),M):M) {
      R <- simpleRule (INTC$Close,j,k,9,Ra,Rb)
      Dt <- na.omit (R - Rb)
      x[k] <- mean (Dt)/sd(Dt)
    }
    return(x)
  }
```

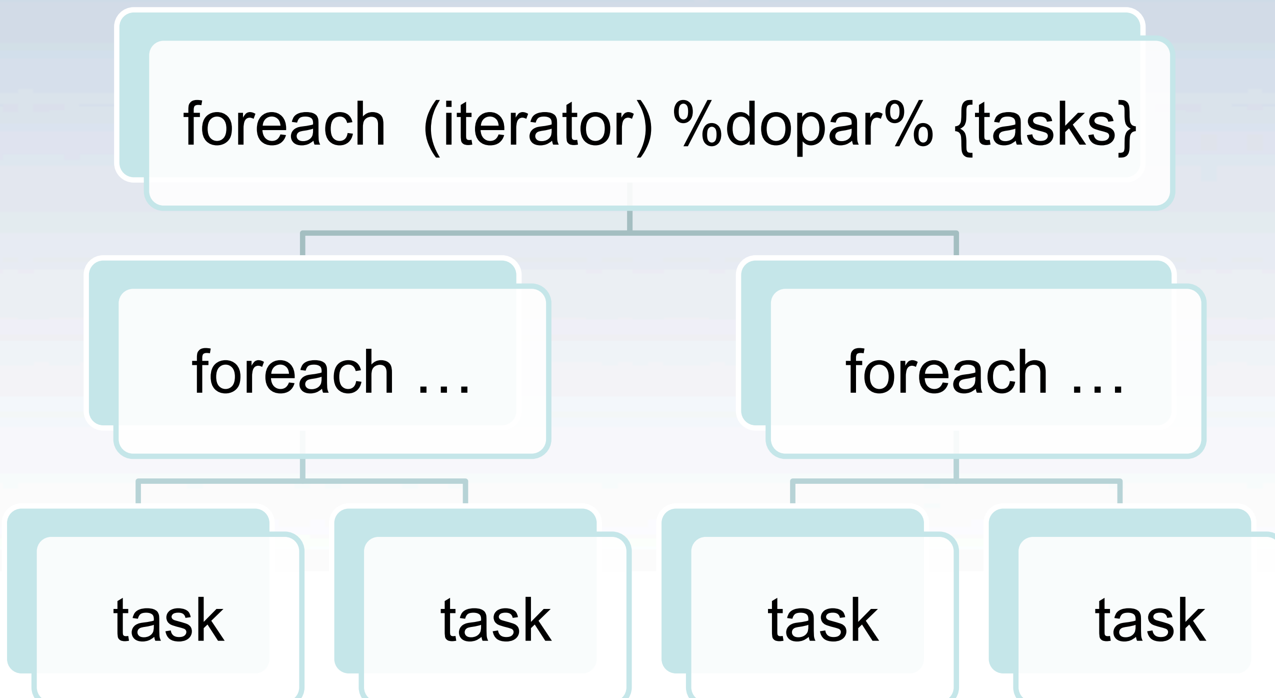
```
j <- which (S==max(S), arr.ind=TRUE)
Ropt <- simpleRule (Cl(INTC),j[1],j[2],9,Ra,Rb)
chart.CumReturns (cbind (Ra,Rb,R.def,Ropt))
```



An example of explicit multi-paradigm ||ism

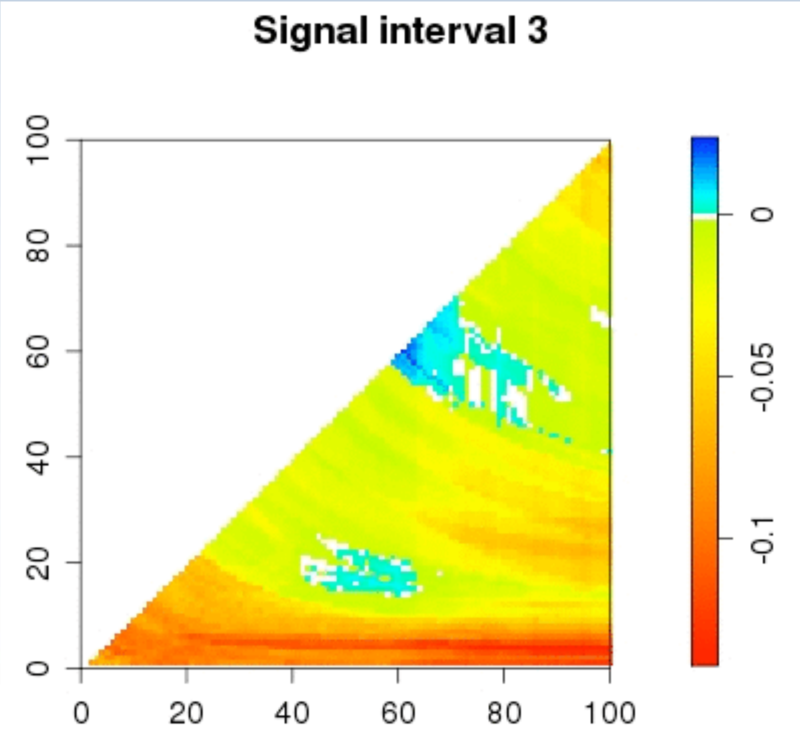
CLUSTER

SMP



```
require ('snow')
require ('foreach')
require ('doSNOW')

cl <- makeCluster (c ('n1', 'n2'))
registerDoSNOW ()

foreach (iterator,
     .packages=c ('foreach', 'doMETHOD')
%dopar%
     {
       registerMETHOD ()
       foreach (iterator) %dopar% {
         tasks…
       }
     }
```

```
require ("spatstat")

function showIm (S) {
   (wrapper for image) … }

for (j in 3:20) {
 S <- S3[,,j]
 showIm(S)
}
```



Pitfalls to avoid

- Sequential vs Parallel Programming
- Random Number Generation
  - `library(sprngNWS)`
  - `sleigh(workerCount=8, rngType='sprngLFG')`
- Node failure
- Deadlocks
- Cosmic Rays

Conclusions

- Parallel computing is easy!
- Write loops with foreach / %dopar%
  - Works fine in a single-processor environment
  - Third-party users can register backends for multiprocessor or cluster processing
  - Speed benefits without modifying code
- Easy performance gains on modern laptops / desktops
- Expand to clusters for meaty jobs
  - Appropriate unused PCs overnight!

Thank You!

dsmith@revolution-computing.com

blog.revolution-compuing.com