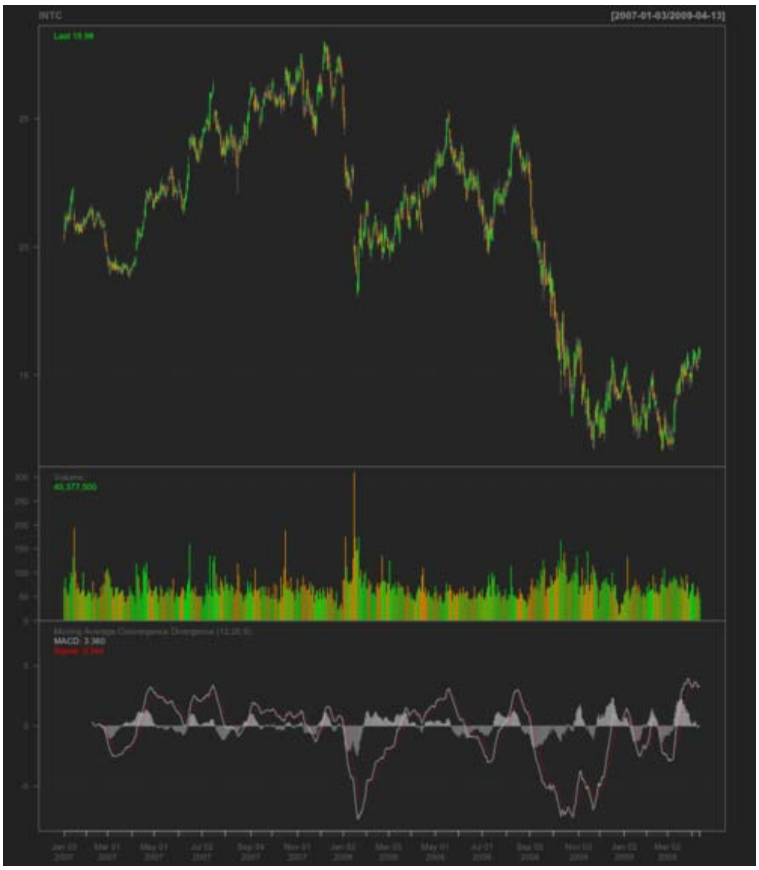
Parallelized Backtesting with `foreach`

Back in March, Bryan Lewis gave a demonstration of parallel backtesting using REvolution R Enterprise and the ParallelR 2.0 suite of packages. In this post, I'll run through the code in detail; the complete script and data file are available for download at the end of this post (after the break).

The goal here is to optimize the parameters for a simple automated trading rule. We will use the MACD oscillator as the basis of the trading rule. MACD compares a short-term (fast) moving average with a long-term (slow) moving average. When the difference between the fast and slow moving averages exceeds a signal line (itself a moving average of the difference) this signals "buy"; otherwise it signals "sell".

The MACD depends on three parameters: the number of historical data points (period) of the fast moving average (nFast), the period of the slow moving average (nSlow) and the period of the signal moving average (nSig). For example, here is the daily close of Intel Corp (INTC) from 2007-2009, along with the MACD with nFast=12, nSlow=26 and nSig=9:

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



(Click to enlarge this and later charts.) The gray line in the lowest panel is the MACD and the red line is the signal. (Incidentally, being able to create a chart like this in just 2 lines of code is a great example of the power of the quantmod package.)

Our trading strategy will be to buy (go long on) INTC when the MACD exceeds the signal, and to sell all INTC and buy IEF (10-year treasuries, a benchmark and safe haven for our cash) when the MACD goes below the signal. We can compare the performance of INTC to our benchmark IEF by looking at the cumulative returns:

```
z <- cbind (INTC$Close, IEF$Close)
colnames(z) <- c("INTC", "IEF")
z <- na.omit (z)
Ra <- Return.calculate(z$INTC)
Rb <- Return.calculate(z$IEF)
chart.CumReturns (cbind (Ra,Rb), main='Returns', legend.loc='topright')
```

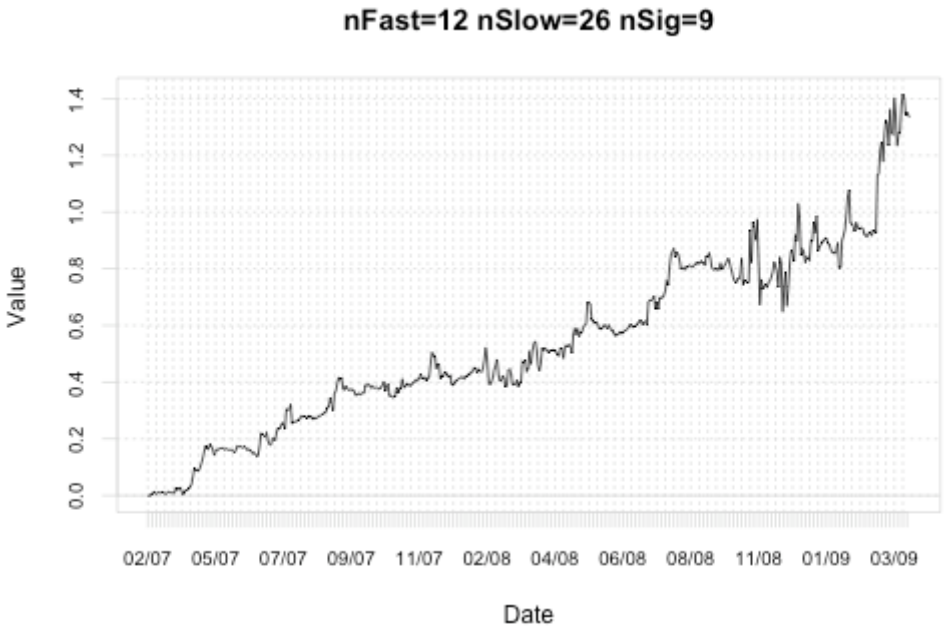
INTC was handily outperforming our benchmark IEF until early 2008. Since then there have been periodic small rallies which our trading strategy may be able to capitalize on.

The key here is optimizing our trading strategy to detect these short rallies. The MACD depends on three parameters nFast, nSlow and nSig, and varying any or all of them might result in better (or worse!) overall performance. We can write a simple function to calculate the returns from our trading strategy given these parameters. Given a series of indicators z (in our example, the INTC close) and the MACD parameters, it takes a long position on the return series `long` on a buy signal and reverts to the `benchmark` series otherwise:

```
# Define a very simple MACD-based example trading rule.  This rule
# takes a buy or sell position on the 'long' instrument,
# converting into the 'benchmark' instrument on the sell signal.
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))
}
```

We can apply this rule using the parameter values used in the chart above:

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



If we'd used this trading rule with the default parameters since 2007 we'd have done fairly well with a steadily growing investment. A standard way of evaluating our performance compared to the risk we're taking on is to calculate the Sharpe Ratio:

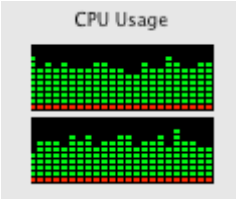
```
Dt <- na.omit(R - Rb)
sharpe <- mean(Dt)/sd(Dt)
print (paste("Ratio = ",sharpe))
SharpeRatio (na.omit(Ra))  # for buy-and-hold strategy
```

For the default parameters we get a Sharpe Ratio of 0.073 (compared to a Sharpe Ratio of -0.015 for simply holding INTC), but we might be able to do better than that by choosing other parameters. For the sake of simplicity, let's hold the nSig parameter at 9 days. A brute-force method would be to simply try thousands of combinations of nFast and nSlow and see which ones result in the highest Sharpe Ratio. Here's one way of doing so, testing 4466 combinations subject to some common-sense constraints (namely, that nFast is at least 5 days and nSlow at least 2 days longer):

```
M <- 100
S <- matrix(0,M,M)
for (j in 5:(M-1)) {
  for (k in min((j+2),M):M) {
    R <- simpleRule(z$INTC,j,k,9, Ra, Rb)
    Dt <- na.omit(R - Rb)
    S[j,k] <- mean(Dt)/sd(Dt)
  }
}
```

The best Sharpe Ratio from all the tests is 0.194, when nFast is 5 and nSlow is 7. On my dual-core MacBook, that code takes about 77 seconds (in stopwatch-time) to run. REvolution R does

a good job of making use of the dual cores for mathematical operations like these, but even then the CPUs are not fully utilized. This is typical of what I see in the Activity Monitor while that code runs:



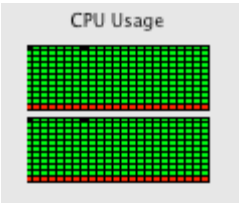
Of course, we could refine the optimization further by using a longer date history and cross-validating against data held back from the optimization, but we'll leave that as an exercise for the reader. My goal here is rather to show how we can improve the performance by parallelizing that brute-force optimization.

This is an example of an "embarassingly parallel" problem: each pair of parameter values can be tested independently of all the other combinations, as the results (the Sharpe Ratio) are all independent of each other. On a multiprocessor system, we can reduce the time required for the calculation by running some of the tests in parallel. The ParallelR suite of packages, available with REvolution R Enterprise, makes this very easy for us. All I need to do is replace a "`for`" loop with a "`foreach`" loop.

`foreach` from ParallelR is similar to the standard for loop in R, except that the iterations of the loop may run in parallel, using additional instances of R launched automatically on the current workstation. `foreach` also differs from for in that it returns a value: the result of each iteration collected (by default) into a list. Here's how I'd rewrite the code above to run the outer loop with 2 parallel instances of R:

```
require ('doNWS') # load the ParallelR packages
s <- sleigh(workerCount=2) # max 2 parallel R sessions
setSleigh(s)
registerDoNWS(s) # use NetWorkSpaces as the back-end engine
SS <- 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(z$INTC,j,k,9,Ra,Rb)
      Dt <- na.omit(R - Rb)
      x[k] <- mean(Dt)/sd(Dt)
    }
    return(x)
  }
stopSleigh(s) # Shutdown parallel workers
```

This version of the code gives the same results, but runs in just 54 seconds according to my stopwatch. It also makes much better use of my dual CPUs:



Getting a 30% reduction in the processing time was pretty simple: all I needed to do was change one for loop to a `foreach` / `%dopar%` loop, and initialize the system with a "`sleigh`" object to indicate that two R sessions should be used on the current machine in parallel. I could get even better gains on a machine with more processors (on a quad-core I could run 4 sessions in parallel), or by distributing the computations across a cluster of networked machines. ParallelR makes all of this very easy, with a simple change to the `sleigh` call.

Also of note is that I did not require any special code to populate the "worker" R sessions with the various objects required for the computation. Some other parallel computing systems for R require a lot of housekeeping to make sure that each R session has all the data it needs, but with ParallelR this isn't necessary. ParallelR automatically analyzed my code, and knew that it needed to transmit objects like `Rb` to the worker sessions. The only housekeeping I needed to do was to indicate that the packages `xts` and TTR should be loaded for each worker session, and that the results should be combined into a matrix rather than a list.

So to sum up, ParallelR and the `foreach` function provide a simple mechanism to speed up "embarrassingly parallel" problems, even on modest hardware like a dual-core laptop. In many cases, with just a simple conversion from the `for` syntax to the foreach syntax you can get significant speedups without having to worry about many of the housekeeping details of setting up worker R sessions. And for the really big problems, you just need to change one line of code to move your job onto a distributed cluster or grid.

If you want to try this out yourself, download the two files below. The first file is the data for INTC and IEF, downloaded from Vhayu. You'll need to run the script `parallelbacktesting.R` in REvolution R Enterprise 2.0. You'll also need version 0.6 or later of the xts package.

Download Vhayu.Rdata (47.6K)
Download parallelbacktesting.R (7.3K)

The previous information was published on David Smith's Revolutions blog on May 14, 2009.
http://blog.revolution-computing.com/2009/05/parallelized-backtesting-with-foreach.html