Applications - Finance

Revolution Analytics

1. Black-Scholes-Merton Model

2. Serial Function

3. Using apply

4. Using rxExec

Parallel Finance in R

- Monte Carlo methods used to price stocks
- RNG used to explore pricing space
- Embarrassingly parallel – easily parallelized
- Will look at pricing European options
- This is a parallelization example, not a tutorial on option pricing
- This session borrows heavily from
  - Das, S. and Granger, B. “Financial Applications with Parallel R.” Journal of Investment Management, 7(4), 66-77

Outline

1. Black-Scholes-Merton Model
2. Serial Function
3. Using apply
4. Using rxExec

European Option Pricing

- Pricing based on Black-Scholes-Merton (BSM) models:

$$S(t+h) = S(t)e^{(r-\frac{\sigma^2}{2})h+\sigma e(t)\sqrt{h}}$$

- $e(t) \sim N(0,1), i.i.d.$.
- $S(t)$: stock price at time t
- h: small delta t (fraction of year)
- r: risk free interest rate
- σ: stock volatility

More Details

- Will generate multiple paths of stock prices ($S_j(t), t = 1 \ldots T$ for the j-th path)
- Each path has n steps making $h = T/n$ (time step)
- Final price of stock:

$$CallPrice = e^{-rT} \frac{\sum_m max[0, S_j(T) - K]}{m}$$

- where m is the number of paths.
- With this data we are ready to build BSM code

Outline

Naive, Serial Code

```r
mcoption <- function(s0, k, t, v, r, n, m) {
  h <- t/n
  s <- matrix(0, m, n + 1)
  payoff <- matrix(0, m, 1)
  for (j in 1:m) {
    s[j, 1] <- s0
    for (i in 2:(n + 1)) {
      s[j, i] <- s[j, i - 1] * exp((r - 0.5 * v^2) * h + rnorm(1) *
        v * sqrt(h))
    }
    payoff[j] <- max(0, s[j, n + 1] - k)
  }
  callprice <- mean(payoff) * exp(-r * t)
}
```

Your Turn - Brainstorm in group

- What are your observations of the code
- How can you parallelize the BSM function?

Revising `mcoption()`

- The original code is not very R-like
  - nested for loops recall Fortran
  - payoff variable doesn't need to be a matrix
  - last line uses an assignment that is never used
- A slightly more R-like version uses `apply()` to perform the central computation on each row of the s matrix, which we initialize with s0 rather than 0.
  - We have to take the transpose of the object returned by apply to get back our originally-shaped matrix.
  - We then use `ifelse()` to define payoff as a vector.

Outline

Revising `mcoption()`

```
mcoption2 <- function(s0, k, t, v, r, n, m) {
  h <- t/n
  s <- matrix(s0, m, n + 1)
  s <- t(apply(s, 1, function(x) {
    for (i in 2:(n + 1)) {
      x[i] <- x[i - 1] * exp((r - 0.5 * v^2) * h + rnorm(1) *
        v * sqrt(h))
    }
    x
  }))

  final <- s[, n + 1]
  payoff <- ifelse(0 > final - k, 0, final - k)
  mean(payoff) * exp(-r * t)
}
```

Comparing mcoption and mcoption2

```
set.seed(42)
t1 <- system.time(a <- mcoption(100, 100, 1, 0.3, 0.03, 52, 10000))
set.seed(42)
t2 <- system.time(b <- mcoption2(100, 100, 1, 0.3, 0.03, 52, 10000))
all.equal(a, b)
```

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

- Excellent! Two functions give same result.

Near Point of Parallelism

- By recasting the problem as we have done, we can easily see where we can parallelize — the computation of the rows of s.
- Divide m over the number of processors, distribute the computation of the rows over the processors, and combine the results.
- Extract the function used to compute the rows so it can be easily re-used.

The `computeRow()` function

This function encapsulates the parallelizable part of the algorithm:

```
computeRow <- function(s0, n, r, v, h) {
  x <- rep(s0, n + 1)
  for (i in 2:(n + 1)) {
    x[i] <- x[i - 1] * exp((r - 0.5 * v^2) * h + rnorm(1) * v *
      sqrt(h))
  }
  x
}
```

Outline

A First Cut at `mcparallel()`

- We basically rewrite the middle of `mcoption2()`, to call `computeRow()` repeatedly via `rxExec()`:

```
mcparallel <- function(s0, k, t, v, r, n, m, numw) {
  h <- t/n
  s <- unlist(rxExec(computeRow, s0, n, r, v, h, timesToRun = m,
    taskChunkSize = m/numw))
  s <- matrix(s, nrow = m, ncol = n + 1, byrow = TRUE)
  final <- s[, n + 1]
  payoff <- ifelse(0 > final - k, 0, final - k)
  mean(payoff) * exp(-r * t)
}
```

Remember

- Remember that `rxExec()` always returns a list
- To get a vector of values, you must use `unlist()`.
- To return us to the format `mcoption2()` was using, we call `matrix()` to put the returned vector into a matrix.

Running `mcparallel()`

- `mcparallel()` returns the solution
- Need performance data (for this exercise)

```
rxSetComputeContext(RxLocalParallel())
rxOptions(numCoresToUse = 8)
numw <- 8
set.seed(42)
t3 <- system.time({
  mcparallel(s0 = 100, k = 100, t = 1, v = 0.3, r = 0.03, n = 52,
    m = 10000, numw)
})
```

Time Three Ways

- `mcoption()` by itself (sequential version)
- `mcoption2()` by itself (sequential version)
- `mcparallel()` with `numw = 4`

Lab for Session 3

- Can we improve our parallel result?
- Our parallel result in many ways returns us to the nested for loop
- Would be nice to take advantage of the efficiency of `apply()` for each chunk - return a list of matrices, rather than a list of row vectors.

Second try at parallelizing

```r
computeRow2 <- function(s0, n, r, v, h, chunksize) {
  s <- matrix(s0, nrow = chunksize, ncol = n + 1)
  s <- t(apply(s, 1, function(x, n, r, v, h) {
    for (i in 2:(n + 1)) {
      x[i] <- x[i - 1] * exp((r - 0.5 * v^2) * h + rnorm(1) *
        v * sqrt(h))
    }
    x
  }, n = n, r = r, v = v, h = h))
  s
}
```

Second try at parallelizing

```
mcparallel2 <- function(s0, k, t, v, r, n, m, numw) {
  h <- t/n
  chunksize <- m/(2 * numw)
  s <- rxExec(computeRow2, s0 = s0, n = n, r = r, v = v, h = h,
    chunksize = chunksize, timesToRun = m/chunksize)
  s <- do.call("rbind", s)
  final <- s[, n + 1]
  payoff <- ifelse(0 > final - k, 0, final - k)
  mean(payoff) * exp(-r * t)
}
```

Running `mcparallel2()`

```
rxSetComputeContext(RxLocalParallel())
rxOptions(numCoresToUse = 8)
numw <- 8
set.seed(42)
t4 <- system.time({
  out2 <- mcparallel2(s0 = 100, k = 100, t = 1, v = 0.3, r = 0.03,
    n = 52, m = 10000, numw)
})
```

Compare Run-Times

```
t1
```

```
##    user  system elapsed
##   4.675   0.014   4.703
```

```
t2
```

```
##    user  system elapsed
##   4.846   0.016   4.878
```

```
t3
```

```
##    user  system elapsed
##   0.246   0.221  11.964
```

```
t4
```

```
##    user  system elapsed
##   0.102   0.181  11.625
```

Exercise

Can you make it more efficient?

Parallel Options Pricing Summary

- Implemented Black-Scholes-Merton option pricing
- While this scales reasonably well, this is not necessarily optimal even in R.
- More optimal `mcoption()` code might scale less well
- Decent scaling even with naïve implementation

Questions?

Thank you

Revolution Analytics is the leading commercial provider of software and support for the popular open source R statistics language.

www.revolutionanalytics.com
1.855.GET.REVO
Twitter: @RevolutionR