GCCW19-A4 Andrew Vaillancoutrt 7729503

Please see README.md for how to run programs.

Q1.

Results discussion (see *q1_results.ods* for tables and charts)

1.1. (varying steps, 1 year)
When varying the number of steps, the payoff appears to converge at a value. More steps, more accuracy.

1.2 (varying strike price, 1 year)
As may be expected, with a higher strike price, there is a lower payoff, as the threshold for when the option may be sold is higher.

1.3 (varying sigma, 12% interest rate, 1 year)
The option holder benefits from a higher volatility, as a volatile stock can appreciate in value more rapdily, with the option to not buy in the event that a stock value plummets.

1.4 (varying sigma , 5% interest rate, 1 year)
There is an interest rate advantage in a call option. An increase in interest rates will lead to either saving in outgoing interest on the loaned amount or an increase in the receipt of interest income on the saving account. Both will be positive for a call position.

1.5 (varying sigma, 5% interest rate, 1/2 year)
Unsurprisingly, the payoff has the same degree as 1.4 but at a lesser scale as the stock has had less time to appreciate.

Map-Reduce Design

As far as I could tell, this algorithm was not as easy as question 2 to run in a “map-reduce” style on AWS. My idea for binomial lattice consisted of creating 4 EC2 instances, and having each generate its own binomial lattice. To find the starting values for the 4, 4^{th} level nodes I built a 3 step lattice from m This is rather inefficient as many of the leaf ndoes are calcualted multiple times by different instances. The 4 inner leaf nodes are actually calculated by each instance.
It saves its output as su.txt on its own volume, and sends its results as sd.txt to the node with it’s ID+1.
Not super elegant or easy to manage. In hind sight It propbably would have been easier to have each node make its calcluate its tree and then have one node collect all those values and do the last few steps on its own, which could have been a reducer in EMR Hadoop. I did try this but I think I was having trouble with the calculation, using the formulas from the notes..

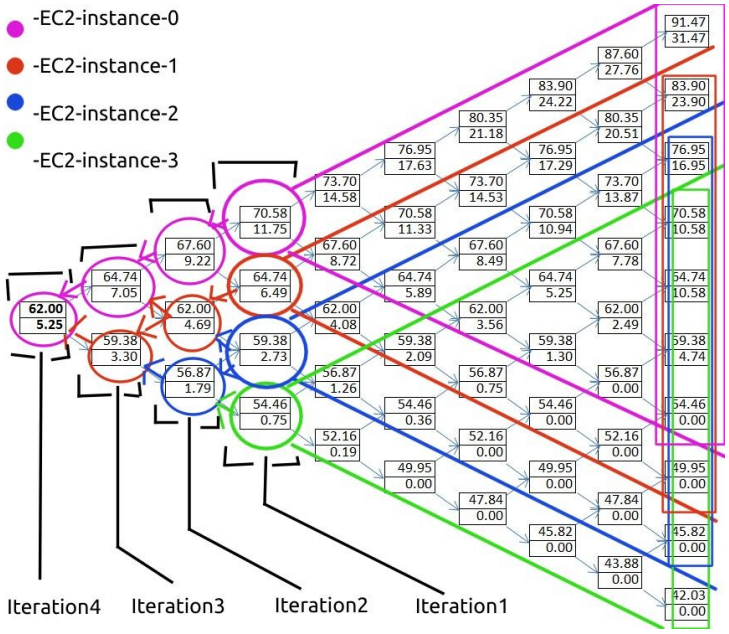
Iteration 1: each node builds its own tree. Starting values were calculated manually.

Iteration 2: has instances 0-2 gathering the iteration1 value from its “neghbouring” instance and using that with it’s own value from iteration one to calculate the value for the next level in the lattice.

Iteration3: repeats this operation but only on instance-0 and 1.

Instance-0 does the last reduction and we have our final option price.

A visual representation of my design follows.



Q2.

Plots and Analysis

Because 1.1 and 1.2 used the same random seed and only vary in strike price, their walks are identical. The increased value of 1.2 is due to the higher strike price.

1.3 has both a lower volatility and a higher strike price and therfore sees a lower payoff than both 1.1 and 1.2.

1.4 was ran with an extrememely high volatility and thus its walks vary widely and see larger jumps between steps.

1.5 was run with a high interest rate, but otherwise the same values as 1.1 and 1.2. The plot shows a clear upward trend when compared to 1.1 and 1.2. Thsi is because, like mentioned above in Q1, high interest rates are an advantage in a call option.

Map-Reduce

The Monte-Carlo Option algorithm was better suited to map-reduce. My desktop program generated a simple .csv file of the results of each walk and displays a plot of each walk. For AWS, I simplified the output as only the final value (and with no plotting of course) of each walk. I created a 3 node Elastic Map Reduce cluster and used this simplified version as the mapper program. The reducer program simply takes an average of all the outputs. I experimented with 2 different mapper outputs. The first just outputted the payoff for its simualtions. The other printed out all the simulation paths (like the table paths) and then the reducer just calculated the payoff for all using the formula, exactly as in the provided program:

```
opt_value = exp(-r * years) * sum([max(path[-1] - K, 0) for path in paths]) / num_sims
```

Using EMR on AWS is quite easy for programs well suited to map reduce. Create an S3 bucket to store your mapper and reduce programs, and the output values, and fire up a Hadoop cluster with your desired number of instances.