Parallel Binomial Options Pricing on OpenMP and MPI

Sonali Batra
Sarath Chandra Yennamani
Dec 2013

ABSTRACT

In this paper we describe our implementations of parallel European and American binomial options pricing using both OpenMP and MPI architectures. An option provides the holder with the right to buy or sell a specified quantity of an underlying asset at a fixed price at or before the expiration date of the option. The binomial options pricing model is an options valuation method developed by Cox, et al, in 1979. It uses an iterative procedure, allowing for the specification of nodes, or points in time, during the time span between the valuation date and the option's expiration date. The model reduces possibilities of price changes, removes the possibility for arbitrage, assumes a perfectly efficient market, and shortens the duration of the option. Under these simplifications, it is able to provide a mathematical valuation of the option at each point in time specified.

Keywords
Options Pricing, Binomial Options Pricing,European,American,CRR, strike price, call option, put option

1.INTRODUCTION-

OpenMP (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most processor architectures and operating systems, including Solaris, AIX, HP-UX, GNU/Linux, Mac OS X, and Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. It uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer.

Message Passing Interface (MPI) is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in Fortran or the C programming language

In this paper, both European and American Options are priced by the Binomial Options Pricing Method using both OpenMP and MPI architectures. The price tree or lattice is generated by both the CRR (Cox, Ross and Rubinstein) and Equal Probabilities method. In section 1, we elaborate upon Options Pricing and American and European Options. In Section 2, we discuss the Binomial Options Pricing Model. In Section 3, we briefly discuss implementation details and results. Lastly in Section 4, a conclusion and suggestions.

1.OPTIONS PRICING-

An option is a contract that gives the holder a right, but not the obligation to perform a specific action in the future. A Call option gives the holder the right to buy an agreed quantity of a particular commodity at a certain time (Expiration Date) at a certain price (Strike Price). Similarly, a Put Option gives the holder a right to sell a particular commodity, keeping in mind the same conditions imposed above. A European Option can be exercised only at the Expiration Date. In contrast, an American Option can be exercised at any time between its purchase date and expiration date.

2.BINOMIAL OPTIONS PRICING MODEL-

This model uses a "discrete-time" model of the varying price over time of the underlying financial instrument. It traces the evolution of the option's key underlying variable via a binomial lattice (tree), for a given number of time steps between valuation date and option expiration. Each node in the lattice, represents a *possible* price of the underlying, at a *particular* point in time. This price evolution forms the basis for the option valuation.

The valuation process is iterative, starting at each final node, and then working backwards through the tree to the first node (valuation date), where the calculated result is the value of the option.

Option valuation using this method is, as described, a three step process:

1. price tree generation
2. calculation of option value at each final node
3. progressive calculation of option value at each earlier node; the value at the first node is the value of the option.

Price Tree Generation-

The tree of prices is produced by working forward from valuation date to expiration. At each step, it is assumed that the underlying instrument will move up or down by a specific factor (*u* or *d*) per step of the tree. So, if *S* is the current price, then in the next period the price will either be S_{up}=S.u or S_{down} =S.d. The up and down factors are calculated using the underlying volatility, σ and the time duration of a step, *t*, measured in years. From the condition that the variance of the log of the price is $\sigma^2 t$, we have u= $e^{\sigma\sqrt{t}}$ and d= $e^{-\sigma\sqrt{t}}$ =1/u. The above is the original Cox, Ross, & Rubinstein (CRR) method; there are other techniques for generating the lattice, such as "the equal probabilities" tree.

The CRR method ensures that the tree is recombinant, i.e. if the underlying asset moves up and then down (u,d), the price will be the same as if it had moved down and then up (d,u) — here the two paths merge or recombine. This property reduces the number of tree nodes, and thus accelerates the computation of the option price.

Option value at each final node

At each final node of the tree — i.e. at expiration of the option — the option value is simply its intrinsic, or exercise, value.

Max [(S – K), 0], for a call option

Max [(K – S), 0], for a put option

Where: K is the Strike price and S is the spot price of the underlying asset

2.Option value at earlier nodes

Once the above step is complete, the option value is then found for each node, starting at the penultimate time step, and working back to the first node of the tree (the valuation date) where the calculated result is the value of the option. If exercise is permitted at the node, then the model takes the greater of binomial and exercise value at the node.

The steps are as follows:

1) Under the risk neutrality assumption, today's fair price of a derivative is equal to the expected value of its future payoff discounted by the risk free rate. Therefore, expected value is calculated using the option values from the later two nodes

(*Option up* and *Option down*) weighted by their respective probabilities -- "probability" **p** of an up move in the underlying, and "probability" **(1-p)** of a down move. The expected value is then discounted at **r**, the risk free rate corresponding to the life of the option.

The following formula is applied at each node:

Binomial Value = [p × Option up + (1-p) × Option down] × exp (- r × t).

Where p= $(e^{(r-q)t}$-d)/ (u-d)

where q is the dividend yield of the underlying corresponding to the life of the option.

2) This result is the "Binomial Value". It represents the fair price of the derivative at a particular point in time (i.e. at each node), given the evolution in the price of the underlying to that point. It is the value of the option if it were to be held — as opposed to exercised at that point.

3) If exercise is permitted at the node, then the model takes the greater of binomial and exercise value at the node.

3. IMPLEMENTATION DETAILS AND RESULTS-

OpenMP Implementation:

We used simple #pragma omp parallel for directive to speed up my calculations in for loops.

Just parallelizing for loops produced race conditions at the output as thread execution is not synchronous. Thread 2 might run 2^{nd} iteration of the for loop before Thread 1 ran 1^{st} iteration modifying the data which 1^{st} iteration depends on.

We avoided this problem by moving the data obtained in every iteration to a temporary array. Thus for loop operates on the temporary array of previous iteration without modifying actual data.

MPI Implementation:

We divided the Tree with Spot, Option values into chunks and distributed along the nodes in MPI_COMM_SIZE.

Chunk Size = Steps/No. of Nodes

Each processor is guided to work on its chunk/share of the binomial tree.

The sequential code shown below fills the nodes with Spot Prices.

1. for(StepPtr = ONE; StepPtr < steps; StepPtr++) {
2. tree[StepPtr].spot = tree[StepPtr - ONE].spot * up;
3. for(SpotPtr = 0; SpotPtr < StepPtr; SpotPtr++)
4. tree[SpotPtr].spot *= down; }

Below mentioned steps portray how this sequential code is implemented in MPI:

1. Look at Step2 in above code,Current tree-node spot value uses previous node's spot value. Now, If the required previous node is available in the processor-node's chunk then there is no issue. The first node in processor chunk needs last node from its predecessor processor.

2. Therefore each processor with rank 'r' sends its final node's spot value in the chunk to processor with rank 'r+1'.

3. Rest of the steps in code is easily parallelizable by directing the processor to its chunk.

Another snippet of sequential code given below is when walking backwards through Binomial Tree.

1. /*calculateoption*/
2. tree[SpotPtr].option= down_prob * tree[SpotPtr].option + up_prob * tree[SpotPtr + ONE].option;
3. /*calculatespotprice*/

4. tree[SpotPtr].spot = tree[SpotPtr+ONE].spot / up;

This sequential code is implemented in MPI following these steps

1. Look at Step2 and 4 in above code, Current tree-node option value and spot value uses next node's option value. Now, If the required next node is available in the processor-node's chunk then there is no issue. The last node in processor chunk needs first node from its successor processor.

2. Therefore each processor with rank 'r' sends its first node's option and spot values in the chunk to processor with rank 'r-1'.

3. Rest of the steps in code is easily parallelizable.

Results–

Table 1
Execution time and speedup using MPI Implementation

Steps	Max Time				Avg Time			
	Time	Speedup			Time	Speedup		
	p = 1	p = 5	p = 10	p = 20	p = 1	p = 5	p = 10	p = 20
5000	2.48	2.98	3.93	6.52	2.09	4.35	5.80	9.50
10000	10.79	2.84	3.79	6.50	8.96	4.12	5.49	9.43
20000	46.54	2.82	3.92	6.68	37.65	4.06	5.75	9.70
40000	94.78	1.36	1.91	3.50	73.90	1.89	2.68	4.64
60000	220.76	1.56	2.19	3.79	168.59	2.15	3.12	5.30
80000	394.03	1.53	2.18	3.77	302.15	2.13	3.10	5.33
100000	795.86	3.88	2.81	4.85	655.78	5.63	4.30	7.95

Table 2
Execution time and speedup using OpenMP Implementation

Steps	Max Time				Avg Time			
	Time	Speedup			Time	Speedup		
	p = 1	p = 5	p = 10	p = 15	p = 1	p = 5	p = 10	p = 15
5000	2.48	1.43	2.66	3.30	2.09	1.99	2.67	3.31
10000	10.79	1.53	3.03	4.15	8.96	1.92	3.24	4.22
20000	46.54	1.86	3.44	4.79	37.65	2.05	3.58	4.92
40000	94.78	1.18	2.07	5.52	73.90	1.20	2.08	5.44
60000	220.76	1.21	2.13	5.71	168.59	1.21	2.15	5.61
80000	394.03	1.20	2.13	5.73	302.15	1.20	2.18	5.33
100000	795.86	1.19	2.09	5.69	655.78	1.21	2.11	5.45

Fig.1

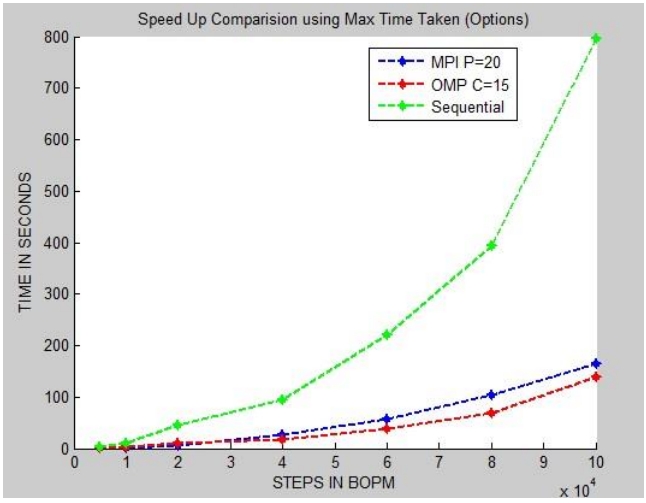

Fig.2:Speed up comparison using Max Time

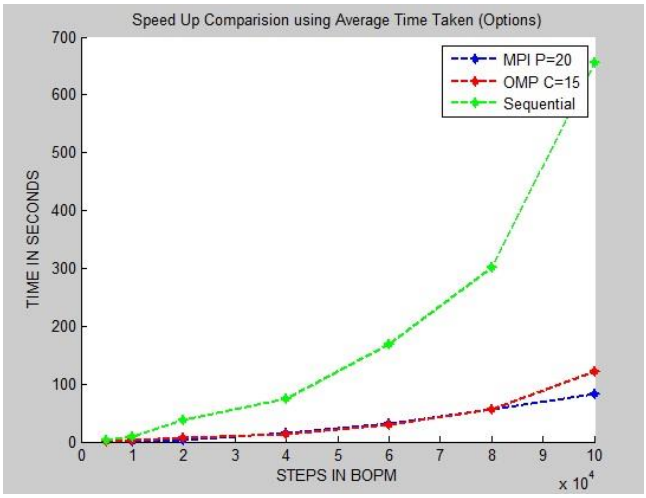

Fig.3: Speed up comparison using Avg Time

Analysis--

Tables 1 and 2 given in Fig.1 indicate the increasing performance of Binomial Option Pricing Method using OpenMP and MPI Implementations for increasing problem sizes. The best performance of the algorithm is obtained with MPI implementation using 20 processor nodes in Table 1. Chunk Size (Steps / p) < 1000 gave twice the speed up than Size > 1000. Therefore with increasing problem size, increasing processor nodes improves the performance of our implementation and also parallelizing on processor nodes such that Chunk Size < 1000 gives optimal speedup.

Compare the speedup obtained with timing results obtained using Max Time i.e. the max time of all 4 options(American call, American Put, European Call and European Put) to the timing results obtained using Average time of all 4 options.

Speed up obtained with Average Time is significantly higher than Max Time. The Communication Overhead i.e delay generated while sending/broadcasting data between nodes in MPI_COMM_SIZE when calculating American and European PUT Options is significantly less when compared to CALL Options. Therefore Avg time taken is reduced compared to Max time.

Increasing number of threads improved the performance of our OpenMP implementation as shown in Table 2. Comparing OpenMp and MPI, with Max Time there is no significant difference whereas with Avg Time MPI gives better performance as time taken for PUT options is very less compared to OpenMP

Some preliminary timing results for Sequential,OpenMP and MPI implementations are in Appendix A. If we compare between OpenMP, MPI and sequential timing results of 100000 nodes, we find that the speedup of OpenMP as compared to sequential for a EUROPUT option is 3.26 and MPI as compared to sequential for the same is 32.76. We see that our MPI implementation performs 10.05 times better in this case.

Fig 1. graph depicts the timing results obtained using Max Time i.e. the max time of all 4 options(American call, American Put, European Call and European Put). American Call option takes more time as compared to the other three. And fig 2. depicts the timing results obtained using Average time of all 4 options. On comparing the two graphs we come to the following conclusions. Looking at only sequential results, Max time is greater than Avg time which was to be expected. Looking at the OpenMP implementation, we see that the results are almost the same for both max time and avg time. For MPI on the other hand, max time is significantly higher than avg time.

Conclusion-

Thus from our results and observations we come to the following conclusions. Our MPI implementation when compared to existing implementations (refer to paper mentioned in step 5 of references column) achieved better speed up and accurate results due to unambiguous blocking scheme. The performance of our implementation directly depends on the number of processor nodes in the computing cluster. This can be a setback for organizations with less parallel computing power. By accommodating bounded inaccuracy while calculating options, we can reduce communication overhead and improve speed up.

6.ACKNOWLEDGMENTS

The authors would like to thank Professor Vipin Chaudhary for the excellent guidance given for this project and CCR for providing us the resources for implementation and guidance regarding the same.

7. REFERENCES

1. http://en.wikipedia.org/wiki/Binomial_options_pricing_model

2. http://www.investopedia.com/terms/b/binomialoptionpricing.asp

3. http://en.wikipedia.org/wiki/OpenMP

4. http://en.wikipedia.org/wiki/Message_Passing_Interface

5. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.8.7051

Appendix A

Note:

1. For steps less than 40000, Results are taken from my laptop instead of submitting on ccr cluster
2. Only portion of the results are mentioned below for reference.

OpenMP with 10 threads::

40000: EUROPUT: 35.39 EUROCALL:45.51 AMERPUT: 23.84 AMERCALL:45.78

60000: EUROPUT: 53.39 EUROCALL:102.81 AMERPUT: 53.10 AMERCALL:103.37

80000: EUROPUT: 96.39 EUROCALL:184.10 AMERPUT: 93.98 AMERCALL:184.86

100000: EUROPUT: 157.39 EUROCALL:286.10 AMERPUT: 145.28 AMERCALL:288.86

Sequential::

40000: EUROPUT: 63.39 EUROCALL:93.51 AMERPUT: 65.84 AMERCALL:94.78

60000: EUROPUT: 137.03 EUROCALL:213.53 AMERPUT: 148.79 AMERCALL:220.76

80000: EUROPUT: 257.00 EUROCALL:383.50 AMERPUT: 265.98 AMERCALL:394.03

100000: EUROPUT: 512.39 EUROCALL:777.10 AMERPUT: 538.28 AMERCALL:795.86

MPI with 20

100000: EUROPUT: 15.64 EUROCALL:185.06 AMERPUT: 16.92 AMERCALL:186.92

60000: EUROPUT: 6.06 EUROCALL:65.06 AMERPUT: 6.92 AMERCALL:66.199

80000: EUROPUT: 10.29 EUROCALL:117.06 AMERPUT: 11.92 AMERCALL:118.79