Multithreaded Algorithms for Pricing a Class of Complex Options

Ruppa K. Thulasiram
Department of Computer Science
University of Manitoba
Winnipeg, MB R3T 2N2, Canada
tulsi@cs.umanitoba.ca

Lubomir Litov
Stern School of Business
New York University
New York, NY 10012, USA
llitov@stern.nyu.edu

Hassan Nojumi
Department of Mathematics
University of Delaware
Newark, DE 19716, USA
nojumi@math.udel.edu

Christopher T. Downing*
Capital Markets Section
Division of Research and Statistics
Federal Reserve Board
Washington, DC, 20551, USA
Christopher.T.Downing@frb.gov

Guang R. Gao
Department of Elec. & Comp. Engineering
University of Delaware
Newark, DE 19716, USA
ggao@capsl.udel.edu

Abstract

In this paper, we study multithreaded algorithms for pricing American Style options. We describe the algorithms, explain their relative complexities, and study their performance. The binomial lattice problem has been formulated in two distinct ways. In the first approach, the recursive algorithm, we establish a parent-child relationship between threads while fully exploiting the inherent parallelism. The second approach, the iterative algorithm, follows a data-flow model based on the producer-consumer style of programming. We implement the algorithms on the EARTH platform. The limitations posed by the problem size on the recursive algorithm and the solution to overcome this problem by the iterative algorithm are explained through the performance results. We have then extended these algorithms to study complicated options with dividend paying underlying assets and reported the performance results.

1. Introduction

In this paper, we study the problem of pricing a class of financial securities using the EARTH (Efficient Architecture for Running THreads) multithreaded computing model.

Economic and financial models used for evaluation and forecasting purposes typically lead to large dynamic, nonlinear problems that have to be solved for a certain time span. In particular, many of the problems in finance demand efficient algorithms and high performance computing capabilities [1]. In part, this is because, in most financial applications, there is a high premium on rapid solutions. Any delay in information processing can be translated into potential financial losses. From an academic point of view, the state of the art in financial modeling is evolving toward ever more complex models, presenting ever more formidable computational challenges.

There are a multitude of financial instruments traded on the markets. A *call option* is a contract that gives the right to its holder (i.e. buyer) without creating an obligation, to *buy* a prespecified underlying asset at a predetermined price. A *put option* is a contract that gives to its holder the right without creating the obligation, to *sell* a prespecified underlying asset at a predetermined price. An *European* (call or put) *option* can be exercised only at its expiration whereas an *American* (call or put) *option* can be exercised on any date before its maturity. The class of instruments known as American options are claims to payoffs at a time chosen by the holder of the security prior to the expiration date in order to achieve maximum profit. Therefore, determining an optimum exercise policy is a key issue in pricing American securities. The solution for the optimal exercise policy must typically be performed numerically, and is usually a computationally intensive problem. To price either an American put or call option we may use a binary tree for the asset price and then roll backward the tree checking if at each node early exercise is optimal. In this paper, we develop multithreaded algorithms for pricing simple American–style options. We extend the algorithms to consider more complex options on dividend paying assets.

*This paper represents the views of the authors and does not necessarily represent the views of the Federal Reserve System or other members of its staff.

The most general technique used, among other, to hide or tolerate both communication and synchronization latencies associated with parallel processing, is *multithreading*. We have implemented our multithreaded algorithms on the Efficient Architecture for Running THreads (EARTH) [5](see section 4).

This paper is organized as follows. The multithreaded pricing algorithms are discussed in the next section. We describe the extension of these algorithms to dividend paying options in section 3. Brief description of EARTH and implementation results are presented in section 4. The final section concludes.

2 Algorithms and Implementation

We concentrate here on the algorithms for pricing of plain vanilla options. The problem at hand is to find the price $F(S,t;T,r,\sigma,K)$ at time t of an American–style Vanilla Put Option on stock (the underlying asset of the option), given the following information: S=the current asset price, T = the time to expiration of the option contract, r=the risk–free rate of interest, σ = the volatility of asset prices, and K=the strike price of the option. A good description of binomial method used in this study which was originally developed in [2] is available in [4].

2.1 Recursive Algorithm

The binomial recursive method proceeds in two phases. First, we construct a lattice of asset prices that approximates the continuous–time risk–neutralized process for asset prices (Phase I). It is assumed that asset prices follow a geometric process $dS = \alpha S dt + \sigma S dB_t$, where α is a constant and B_t is a standard Brownian motion. Second, we work backward through the lattice from time T in order to compute the option value (Phase II). Convergence of the option value computed under the binomial method to the true option value is discussed in [3]. In our algorithm, construction of the lattice proceeds in parallel.

To construct the binomial lattice, we first divide the time interval $[0,T]$ into L smaller intervals, each of length $\Delta t = \frac{T}{L}$. Over each sub–interval, the asset price is assumed to move up from value S to Su, or down to Sd, with probabilities p and $1-p$, respectively. It can be shown that if we set $u = e^{\sigma\sqrt{\Delta t}}$, $d = e^{-\sigma\sqrt{\Delta t}}$ and $p = \frac{e^{r\Delta t}-d}{u-d}$ then over each time interval, the mean and variance of asset price movements will match the mean and variance of the continuous–time risk–neutralized asset price process.

In the first time step of phase I, one thread computes the up–movement to Su, and a second thread computes the down–movement to Sd. In the second time step, four threads operate to produce the values Suu, Sud, Sdu, Sdd. (The notation Sud means S followed by an up–movement, then followed by a down–movement. The rest of the notation is similar). This procedure continues, until at the final step where 2^L threads operate to produce the asset prices at the terminal, or "leaf," nodes of the lattice. It is important to point out that in this implementation of the algorithm, we *do not* make use of the fact that the tree is recombining. In other words, we do not exploit the fact that $Sud = Sdu$ (refer to the equation above) in order to reduce the number of operations.

The second phase in the algorithm is to work backward from the expiration time T to compute the option price. At the expiration time T and leaf–node i, we know that the value of the option (also known as the pay-off) is given by $F(S_i,T) = \max[0, K - S_i]$. At time $T-1$ and node j, the value of the option is given by $F(S_j, T-1) = \max[K - S_j, (pF(S_j u,T) + (1-p)F(S_j d,T))e^{-r\Delta t}]$, the greater of the value under immediate exercise of the option, $K - S_j$, or the expected value of holding the option for another period, $(pF(S_j u,T)+(1-p)F(S_j d,T))e^{-r\Delta t}$. Moving backward through the tree in this way simply amounts to reversing the process of phase one, using the stored memory locations of each node in the tree, and can proceed in parallel.

2.2 Iterative Algorithm

The recursive algorithm is carried out using a non–recombining binomial lattice in the pricing algorithm. The sheer number of nodes created in a non–recombining lattice (2^L) restricts the usefulness of the recursive algorithm (since one can use only small number for L) to options with short expirations. To accurately price options with long expirations, the L must be large and hence it is necessary to implement a recombining lattice so that the number of nodes in the lattice does not grow exponentially. Moving to a recombining lattice raises the issue of how one would partition the data structures so as to optimize performance.

Instead of constructing the lattice recursively from the current date (t=0), as in the recursive algorithm, we proceed from the expiration date (t=T;leaf nodes) for this algorithm. That is, we eliminated the construction of the lattice itself from the iterative algorithm. This feature of the algorithm itself is expected to give rise to an overall better performance. The data is partitioned uniformly (L/P), where L is the number of periods into which the interval $[0,t]$ is divided; and P is the number of processors among the processors (L and P are even and powers of two). The last processor is assigned with the extra data point. At each time level there are two modes in the algorithm: computation and communication. These two operations are performed by two types of threads: producer and consumer threads.

Communication mode: is such that the lower numbered processors send data to higher numbered processors. In this arrangement processor 0 will only send a data to its neighbor processor 1 and processor P will only receive data from its neighboring processor P-1. The intermediate processors will send and receive data. Communication is done in a split phase manner, that is, sending data and receiving acknowledgment for the value sent are atomic operations. Sending and receiving of data requires synchronization between the producing and consuming threads. The consumer thread located in a processor is activated when it receives a synchronization signal from its mate thread. The threads have to determine the location of their mate threads at each iteration and synchronize appropriately.

Computation mode: Computation at each node involves (a) Calculating the asset price at the node $t-1$; and (b) Calculating the option value based on the two children nodes at time t. For (a), at the expiration time T and leaf–node i, we calculate the asset price based on the number of up and down movements from the initial node, the node index, and processor index. For an internal node, for example, the stock price at node j and time L_i is calculated as $S_j^{L_i} = S_0 * u^{L_i - 2*(L_i - j)}$. The second step of the iterative algorithm is same as the one in recursive method, as already explained earlier in section 2.1.

2.3 Computational Complexity Analysis

Recursive Approach: Given N points and P processors, a parent-child relationship is established between threads, while fully exploiting the underlying parallelism in the problem. We exploit full parallelism by generating maximum number of threads at the expense of redundancy in the computations at different processing nodes. The performance of the algorithm lies in the balance between the number of remote and local threads and on their overlapped asynchronous scheduling. The total number of threads in the system is $4(2^L - 1)$, since in the phase of building the lattice, $2(2^L - 1)$ threads are created, and a like number are created in the phase of calculating the option value. In the best case, if L is $\log(P)$, the communications complexity is $O(P)$. If $L \geq P$, then the communications complexity is $O(2^L)$. In terms of computations, it can be shown that the complexity is $O(2^L - 1)$.

Iterative Approach: Given L data points and P processors, the iterative algorithm follows a coarse-grained approach, and models the financial derivative problem as the producer-consumer problem. The L data points are partitioned into $\frac{L}{P}$ points and allocated among the various processors, with the exception that the last processor contains one extra data point. The processors perform local computations on $\frac{L}{P}$ points over L iterations. This can be realized as a sequential algorithm over $\frac{L}{P}$ data points using a single thread in each processor. Therefore, there are P local threads per iteration for L iterations. However, the number of computations performed varies at each iteration. The key to understanding the complexity analysis that follows is to realize that the binomial lattice shrinks in size as we move backward through time. We think of the lattice as shrinking from the top. Thus, after each $\frac{L}{P} - 1$ iterations, the lowest indexed processor becomes inactive.

We analyze the algorithm in terms of *local computations* and *remote computations*. At each iteration, each processor (with the exception of the last processor) sends one data point to its (higher numbered) neighboring processor. Each processor sends and communicates until the processor is shed off. There are two types of computations. The computations between two local points performed within the processor and computation performed when a remote data point (boundary point) is received. Both these computations are performed in two different threads. The *remote threads* perform the remote communication/computation.

Processor 0, for $\frac{L}{P}$ points performs local computations over $\frac{L}{P} - 1$ iterations, a total of $\sum_{i=1}^{i=\frac{L}{P}-1}(\frac{L}{P} - i)$ computations before going inactive. Processor 1, performs local computations, in the worst case, on $\frac{L}{P}$ points for $2(\frac{L}{P} - 1)$ iterations and then becomes inactive, and so forth for the rest of the processors. The last processor P, performs the local computations on $\frac{L}{P} + 1$ points. Summarizing, the total number of local computations in all performed in the algorithm is

$$\sum_{j=0}^{P-1} \sum_{i=1}^{\frac{L}{P}-1} (\frac{L}{P} - i) + \sum_{i=1}^{\frac{L}{P}-1} (\frac{L}{P} + 1 - i) + \sum_{i=1}^{P} (\frac{L}{P} - 1) i$$

For large L, the worst case complexity analysis is O(L^2).

We now discuss the number of remote communications (threads) per processor. Processor 0 communicates $\frac{L}{P} - 1$ data points(one option value at each iteration); Processor 1 communicates $2(\frac{L}{P} - 1)$ data points; and so on. Therefore, the total number of remote communications is $(\frac{L}{P} - 1)\sum_{i=1}^{P} i$. For large L, the worst case complexity is O(L).

The ratio of local computations to remote computations is O(L). Compared to the recursive algorithm, the reduction in both the computational and communications complexity follow from the non–recombining nature of the lattice.

3 Extension to Dividend Paying Options

The computational complexity of the algorithms for these options changes, though the communication complexity remains the same. We discuss here only the cause for this additional computational complexity of the algorithms in this paper, rather than the complexity itself.

The occurrence of a dividend payment during the life of the option reduces the price of the asset on the ex-dividend date. We consider three different situations as described briefly below.

Continuous Dividend Yield: If the underlying asset pays a continuous dividend yield at a rate of δ per unit time then the data generating process in the risk-neutral process becomes (refer to section 2.1) $dS = (r - \delta)Sdt + \sigma SdB_t$ As can be realized from this small change, we price these options as the plain vanilla American puts and calls, with the difference in the formulas for the jump sizes and the probabilities. Hence, the class of options is a very simple variation of the vanilla options. They produce exact same performance results as mentioned later in section 4.

A Known Discrete Proportional Dividend: If there is a one-time proportional dividend payment of δ during the life of the option that occurs at time τ, then the price at all nodes of the tree preceding the dividend payment date will be $Su^j d^{i-j}$ (for $i\Delta t < \tau$) and $(1-\delta)Su^j d^{i-j}$ for all times after the dividend payment date (i.e. for $i\Delta t \geq \tau$). To price an option of this type we use a tree with a one time jump of the tree structure. At the node i where $(i-1)\Delta t \leq \tau < i\Delta t$ we send a token with input value $(1-\delta)Su$ instead of Su. Correspondingly, we modify the formulas for the probabilities of up and down movements and the jump sizes as well. Hence, the natural load of computation before and after the ex-dividend date changes slightly from the earlier option.

A Known Discrete Cash Dividend: It is a more plausible assumption that the dividend is a known dollar amount rather than a percentage yield. Therefore, we consider the case of a known discrete cash dividend. The problem here is that the cash dividend changes the structure of the tree. Suppose we have a one time dividend payment at time τ where $i\Delta t \leq \tau < (i+1)\Delta t$. We assume that we have a recombining tree to start with. At time $(i+1)\Delta t$ the price of the asset will decrease by the amount of the dividend payment, D. This will make the tree at the subsequent nodes non-recombining.

One way to cope with the non-recombining tree is to use the recursive algorithm. Another is to consider the asset price S as composed of two parts, certain (the discounted dividend payments) and uncertain S*, where $S* = S$ for $i\Delta t \geq \tau$ and $S* = S - De^{-r(\tau - i\Delta t)}$ for $i\Delta t \leq \tau$. We model S* instead of S, and thus we obtain a recombining tree. We can construct the tree for S from the tree for S* and thus obtain the value of the option. In this study, we follow the first approach, that is treating the rest of the tree as non–recombining to keep the implementation simple. That is, we start with a non-recombining tree and continue to construct a non-recombining tree after the dividend payment date.

4 Experimental Results

EARTH Platform: EARTH supports a multithreaded program execution model in which code is divided into threads that are scheduled atomically using dataflow - like synchronization operations. Once a thread is started, it runs to completion, and instructions within it are executed in sequential order. Conceptually, each EARTH node consists of an execution unit (EU) which executes the threads and a synchronization unit (SU) which performs the EARTH operations requested by the threads. The implementations reported in this paper have been developed using Threaded-C and performance results have been obtained on the EARTH-MANNA simulator SEMi.

We present here the details of the results from the recursive and iterative algorithms on both the Vanilla Options and a class of complex options.

4.1 Vanilla Options

In our performance experiments, the pricing problem is parameterized as follows: the current asset price is twenty dollars (S=\$20), and the option is slightly in-the-money, with a strike price of twenty–one dollars (K=\$21); the option has six months to expiration (T=0.5 years), the risk–free rate of interest is twelve percent (r=12%), and the volatility of the asset price is set at forty percent (σ=40%). Using these parameters, we measure the time to compute an option price for different combinations of processors (P) and time steps (L).

At twenty time steps (L=20) for the short dated American option, the speed–up attained with 120 processors is 94.

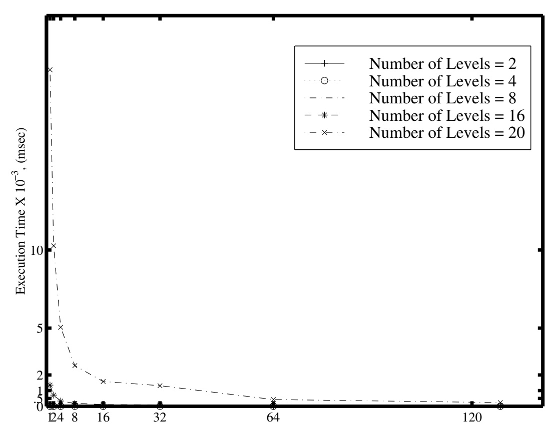

Figure 1. Recursive Speedup Graphs:

The performance results imply that the communication and synchronization latencies associated with conventional distributed computing are effectively hidden by multithreading which overlaps the computation and communications operations using threads. The threads used to build the binomial lattice (recursive algorithm), and to compute the option values, are fine-grained. Since there are two units in EARTH, the execution of threads in the EU is never interrupted as the SU handles all the synchronization and network communications.

Figure 2 presents the performance results for the iterative algorithm. At low values of L, the computational load is so light that there is little benefit from additional processors. The algorithm achieves linear speedups for high values of L, as illustrated in figure 2. One of the main reasons for such a high performance is due to the inherent parallelism in the problem.

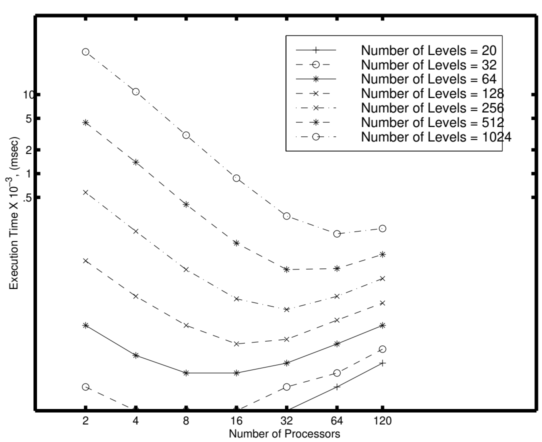

Figure 2. Iterative Algorithm: Scalability with respect to machine size, varying problem size (log-log plot)

4.2 Complex Options

We present here results for the case of complex options. The parameter for all the three different options are set as follows: S=\$100;T=1 year;r=6%;σ=20%;δ=3%;and K=\$100. The options considered for these experiments are at-the-money. Setting the parameters different from case of vanilla options will only change option values. The amount of computation is not affected by these general parameters, and hence, a comparison of the performance results with plain vanilla options is possible as long as the the tree levels are same.

Continuous Dividend Yield: In figure 3 we present the experimental results using the recursive algorithm for the case of the binomial model for an American put option on an asset paying a continuous dividend yield, δ.

The number of tokens generated in the lattice with 20 divisions of given expiration time is large and hence the corresponding execution time as in the plain vanilla case. Increasing the number of processors beyond 64 does not seem to scale the performance using the recursive algorithm. This problem could be overcome using the iterative algorithm which handles larger trees (larger division of given period T by recombining the nodes), thereby generating less number of threads and nodes for computation. The result for this case is the same as the plain vanilla option as presented in figure 2. The largest tree, however, (with 1024 steps) in the current iterative experiments still seems to be small in the iterative algorithm since the scalability of the performance is limited to 64 processors. Comparing with the results from other problem sizes (512, 256 etc. steps) we can infer that with the iterative algorithm we can go beyond 1024 to get good scalability. This also implies that complex options with long expiration date, which require larger division of the given expiration time, could easily be experimented. One important observation here is that if the underlying asset in the option is paying the dividend continuously through the period of the option, using the iterative algorithm, the computation complexity of the problem does not change from the simple vanilla option as shown in figure 2.

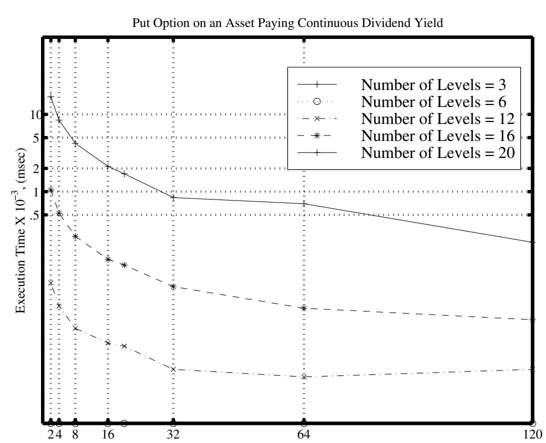

Figure 3. American Put Option with a Continuous Dividend Yield (Recursive)

Known Discrete Dividend: The actual execution times for the largest problem (with 20 periods) in the experiments in the earlier cases with recursive algorithm are, in general, larger than the corresponding execution times in the present case of known discrete one-time proportional dividend. Figure 4 presents the experimental results for the case of the binomial model for an American-style put option on an asset with a known discrete one-time proportional dividend, δ. This difference in execution time is due to the amount of computation in these two cases. The former case has to handle little more computation at each node than the latter and hence the longer execution time using the same algorithm.

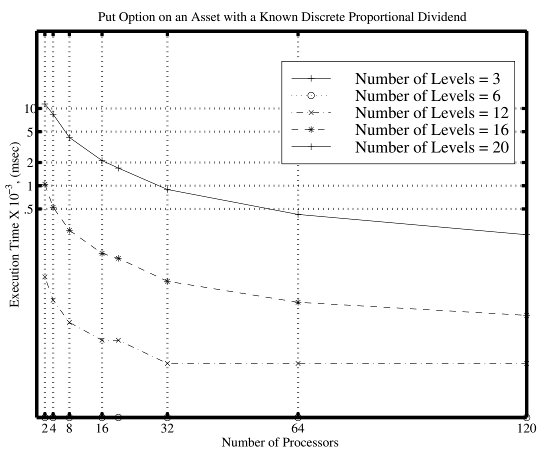

Figure 4. American Put Option with a Known Discrete Proportional Dividend

Known Discrete Cash Dividend: In figure 5 we present the experimental results for an option on an asset with a known discrete cash dividend using the recursive algorithm. As explained in section 3, the dividend payment at date τ makes an originally recombining tree into a non-recombining tree after this date. By choosing a non-recombining tree after the dividend date, we have employed the recursive algorithm in our experiments in this paper. The other possibility of transforming the reminder of the tree into a recombining tree is beyond the scope of this paper and hence we do not discuss here.

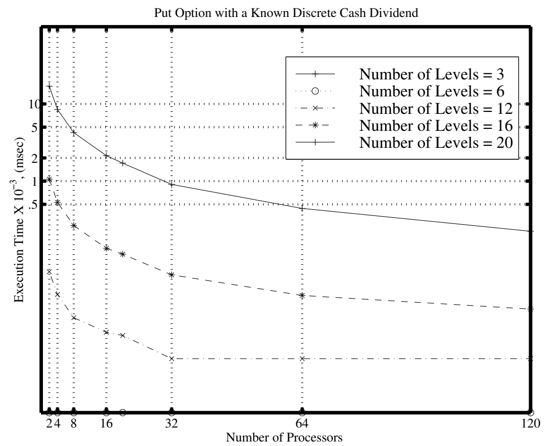

Figure 5. American Put Option with a Discrete Known Cash Dividend

5 Conclusions

In this paper, we discussed the design, implementation and performance of recursive and iterative multithreaded algorithms for pricing American–style options both vanilla and complicated options with dividend paying underlying assets. The multithreaded algorithms allowed us to exploit inherent, parallelism in the binomial lattice pricing approach. We showed, in the context of our pricing algorithms, that the latencies inherent in distributed computation models could be effectively hidden under the multithreading paradigm. The binomial lattice method is applicable to a wide variety of asset pricing problems, particularly when non–recombining lattices are implemented. Moreover, our initial algorithm were shown to be general enough to be used for a class of complex options with corresponding modifications to the algorithm. The speedups attained by our iterative algorithm illustrate clearly that large portfolios of long-dated options (even with complications) can be rapidly priced.

References

[1] H. M. Amman. *Supercomputing in Economics: A New Field of Research?* High Performance Computing, (Ed: J.L. Delhaye, E. Gelenbe) Elsevier Science Publishers B.V. (North-Holland), 1989.

[2] J. C. Cox, S. A. Ross, and M. Rubinstein. Option pricing: A simplified approach. *Journal of Financial Economics*, 7:229–263, 1979.

[3] D. Duffie. *Dynamic Asset Pricing Theory*. Princeton University Press, Princeton, NJ, 1996.

[4] J. C. Hull. *Options, Futures and Other Derivatives*. Prentice Hall, Upper Saddle River, NJ, 3 edition, 1997.

[5] K. B. Theobald. *EARTH: An Efficient Architecture for Running Threads*. PhD thesis, McGill University, Montreal, Canada, May 1999.