University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

FAME: Financial Application with Many-core-on-a-chip architecturE

Weirong Zhu Parimala Thulasiraman†‡ Ruppa K. Thulasiram† Guang R. Gao,

CAPSL Technical Memo 76
February 17th, 2006

†Dept. of Computer Science, University of Manitoba, Winnipeg, MB, Canada R3T 2N2.
Email: {thulasir,tulsi}@cs.umanitoba.ca
‡Author for correspondence: Parimala Thulasiraman, Email: thulasir@cs.umanitoba.ca

Abstract

Research in financial derivatives is one of the important areas in computational finance. The computational requirements for solving models of financial derivatives such as the option pricing problem are huge and demand efficient algorithms and high performance computing capabilities. In this paper, we focus on the development of a Monte-Carlo algorithm with historic volatility and GARCH fitted volatility to price options accurately on a modern multi-core chip architecture. By fabricating hundreds of millions of transistors on a single die, multi-core or many-core-on-a-chip architecture incorporates a complete multiprocessor including the CPU and memory on a single chip. In this paper, we have used Cyclops-64, many-core-on-a-chip architecture, a petaflops supercomputer project under development at IBM T.J. Watson laboratory as the experimental platform for our study in pricing options. In this paper we have shown that the use of incorrect volatilities of the asset prices in the Black-Scholes model would result in moderate to large errors in option prices. The timing results on C64 show that various sets of simulations could be done in a real-time fashion while yielding high performance/price improvement over traditional microprocessors for computational finance applications.

Contents

List of Figures

List of Tables

1 Introduction

We present the motivation and background for the current study in this section.

1.1 Motivation and Contribution

Research in financial derivatives is one of the important areas of computational finance. The investor selects an asset based on the price, the budget and the market trend. Finance models used for evaluation and forecasting purposes to help the investor with the selection process typically lead to large dynamic, nonlinear problems that have to be solved in a short time span to beat the competitors in the market place. The computational requirements for solving such financial models are huge and demand efficient algorithms and high performance computing capabilities [1]. Solution of such systems on a sequential computer will require hours or may be even days depending on the size of the problem. Parallel computing speeds up the computational process by assigning tasks to multiple processors.

In this paper, we focus on development of a Monte-Carlo algorithm with historic volatility and GARCH (Generalized Auto Regression Conditional Heteroskedasticity) fitted volatility to price options accurately on a modern multi-core chip architecture. The purpose is to facilitate pricing of options in a real time fashion. In section 4 we describe the architecture in which our algorithm is studied; we provide some fundamental background on option pricing in the following subsection.

Our contributions from the current work are: (a) devising a means to study a finance application on a many-core-on-a-chip architecture; (b) developing a Monte-Carlo algorithm with two different volatility generation mechanisms; and (c) implementing the algorithm on C64 architecture.

1.2 Background and Related work in Option Pricing

Options on stocks were first traded in an organized exchange in 1973. Since then there has been dramatic increase of trade in the options market. The underlying assets of options include stocks, stock indices, foreign currencies, debt instruments, and commodities [2]. We introduce some basic definitions on financial derivatives here and provide brief survey on various computational techniques.

A *Call Option* [2] is a contract that gives the right to its holder (i.e. buyer) without creating an obligation, to *buy* a pre-specified underlying asset at a predetermined price (*strike price*). Usually this right is created for a specific time period (*maturity date*), e.g. six months. Everything else being same, a *Put Option* gives to its holder the right to *sell*. If the option can be exercised only at its expiration/maturity date (i.e. the underlying asset can be sold only at the end of the life of the option), the option is referred to as an European style Call/Put Option (or *European Call/Put*). If it can be exercised on any date before its maturity, then the option is referred to as an American style Call/Put Option (or *American Call/Put*).

Black and Scholes [3] proposed a model to price option, which has become a classical and celebrated model for pricing options. This Black-Scholes model is basically a stochastic partial differential equation with option price as the unknown and underlying asset price and the time being dependent variables together with various parameters such as volatility of the asset price, expiration date, strike price and

interest rate. While values of most of these parameters could be fixed for the contract, future asset price and volatility of the asset price are two parameters to be determined by forecasting mechanisms. In the current study future asset prices are generated with random number generated in the Monte-Carlo (MC) simulation and volatility is generated by two methods: historic volatility (based on the past changes in the asset price) and GARCH fitted volatility.

Many numerical methods are used to solve the option pricing problem [2], including: (i) MC simulation, for example [4–6]); (ii) lattice or tree based techniques (such as binomial, for example [7, 8]); (iii) fast Fourier transform (FFT) based techniques [9–11]; (iv) finite-difference techniques (for example [12, 13]). Any of these methods can be employed, depending on the user's accuracy requirement and on the availability of computational resources. The binomial lattice and MC methods have been used predominantly.

Clark [13] and Thulasiram et al. [8] developed parallel algorithm for the binomial lattice approach to price options. Their techniques involve constructing a binomial lattice representing possible stock price movement from the present to the expiry date of the option in an intuitive fashion. Further, small machine size is a bottleneck when the problem size is large. On the other hand, multiprocessing does not always provide modest speedup for option pricing problems if inefficient algorithms are used (see, for example [13]).

Srinivasan [6] used the quasi MC simulation technique for option pricing while Rahmail et al. [5] used the traditional MC simulation to study the effect of incorrect volatilities for underlying assets on option pricing errors. In simulations, increasing the sampling size increases the accuracy of the results. However, as the sampling size increases, computational cost also increases.

Use of fast Fourier transform (FFT) technique for option pricing was introduced by Carr and Madan [10]. Extending this model, Barua et al. [9] have developed an efficient parallel algorithm to enable quicker and accurate pricing of options by introducing data swapping technique in FFT.

Mayo [14], for example, evaluated American options using the implicit finite-difference method. The algorithm gave fourth order accuracy in the log of the asset price and second order accuracy in time. Thulasiram et al. [15] have designed a second order L_0 stable algorithm for the pricing problem. This algorithm achieves the same error bound as that of the traditional Crank-Nicholson scheme, while at the same time assures that the error will not propagate. Hence, their scheme is better than Crank-Nicholson scheme, which is typically known to be unstable for PDEs. Although finite-difference methods are reasonably straightforward and the discretization of the problem domain is almost uniform, these methods are computationally intensive and produce less speedup as compared to the lattice methods [13]. However, the accuracy of the option values from the finite-difference technique are typically much better than the lattice method.

Valuing options using lattice and finite-difference methods often demands additional computational effort to achieve higher accuracy of results with finer grids or lattices, while MC requires larger number of simulations. Andricopoulos et al. [16] developed a method which curtails the price ranges of the underlying assets for which computations are carried out, achieving considerable savings of computational effort with virtually no loss in accuracy. Such shorter ranges of asset prices implies smaller volatility of the asset, which indicates to a well informed trader/investor the lesser value of the option. In other

words, coarser grids on asset prices are good only for assets that are less volatile. Generally, as the expiration time approaches, the volatility of the asset price decreases, decreasing the value of the option. Therefore, having uneven step sizes in the asset price direction (spatial direction) especially coarser step size near the expiration time should reduce computational cost in all these methods including MC simulation. However, we do not consider uneven step sizes in this study.

Since the finite-difference and finite-element techniques pose additional challenges for parallel computing, as a first step, we use the MC simulation on many-core-on-a-chip architecture for its simplicity in parallelization. In particular, we target an ambitious 160-core-on-a-chip design, Cyclops-64 (C64) architecture, which is a petaflop supercomputer project under development at IBM research laboratory. Unlike other academia projects, a first C64 system is planned to be installed in 2007.

2 Monte Carlo Algorithm

Most commonly used numerical methods in finance include binomial lattice and Monte-Carlo (MC) simulation.Binomial lattice records the asset price movement into a binomial tree, and then uses risk neutral valuation to obtain option prices at all nodes. MC method values the option based on simulated distribution of asset prices. These two methods can be used to effectively solve simple models. When the underlying assets become more complicated, researchers use Monte-Carlo simulation to evaluate complex options with high dimensionality. MC method [2] refers to the use of stochastic techniques to arrive at a solution for a physical problem. MC method generates stock prices repetitively, which are then used to value the option based on the simulated distribution of stock prices. The convergence rate of MC simulation is generally independent of the number of the underlying state variables. Therefore, it is particularly useful for financial problems with high dimensionality.

MC simulation is a forward-based procedure. Option pricing via MC can be divided into three basic steps: (1) simulate the stochastic process underlying stock returns, where each realization is a sample path; (2) evaluate the value of the option in a backward manner in order to find the early exercise point and obtain a sample point estimate; and, (3) average over multiple sample estimates to form an interval estimate that includes some measure of precision (e.g., standard error). Obviously, the existence of the precision measure is an advantage of MC over other numerical methods.

One major difficulty with MC simulation is that it requires a large number of simulations to achieve convergence. Simulations of the order of several millions are not uncommon in practice. However, this is the impetus for the current study, trying to do large simulation in shortest possible time, using many-core-on-a-chip architecture.

3 Many-core-on-a-chip Architecture

As advances in IC processing technology allow the feature size to drop, density of transistors and clock frequency on silicon chips are to continue increasing for the next few years following Moore's Law. At this pace, a billion-transistor chip is approaching [17]. However, the delivered performance versus

number of transistors integrated in a chip for microarchitecture, keeps declining over time [18]. Power consumption and dissipation considerations also place additional obstacles to improve the performance. Due to fundamental circuit limitations, limited amounts of instruction level parallelism, and the memory wall problem, computer architects look for new designs, other than the single-thread wide-issue superscalar approach, to utilize the transistor budget and mitigate the effects of high interconnect delays. On the other hand, by fabricating hundreds of millions of transistors on a single die, it becomes possible to put a complete multiprocessor, including both CPUs and memory, on a single chip, what is known as multi-core or many-core-on-a-chip architecture. Instead of devoting the entire die to a single and complex processor, the many-core-on-a-chip design integrates a large number of simple processors on a single die.

It is believed that the many-core-on-a-chip architecture has many advantages over the single core chip. First, by partitioning the chip resources into individual small, localized simple cores, the effect of the interconnect delay is limited. By enabling multiple cores to share the chip resources, such as on-chip memory (or L2 cache), interconnect network, and on-chip/off-chip memory bandwidth, the resource utilization is improved. Second, given the fact that chip power consumptions drop significantly with reductions in frequency, many-core-on-a-chip architecture alleviates the power dissipation problem without reducing the computation capability by running a large number of cores with moderate clock rate. Finally, many-core-on-a-chip architecture naturally exploits thread-level and process-level parallelism, which are expected to be widespread in future applications and multiprocessor-aware operating system and environments [19]. Not surprisingly, all major microprocessor manufacturers have already begun to move its microarchitecture towards multi-core or many-core-on-a-chip design [20–25]. In this paper, we develop and implement one important financial application, option pricing using the Monte Carlo simulation on one such many-core-on-a-chip architecture, Cyclops 64.

With the emergence of many-core-on-a-chip architecture, it is important to employ multithreading techniques to avail massive on-chip parallelism provided. Multithreading is a technique that has proved very efficient in exploiting thread-level parallelism and combating the latency problem in parallel computing. Over the past few years, multithreaded algorithms have been developed for problems involving irregular high level data structures arising in traditional and non-traditional areas such as network optimization [26], computational finance [11], and computational medicine [27]. In irregular applications, the data size operated on by each processor changes dynamically [28], which in turn, affects the computational requirements of the problem leading to communication/synchronization latencies and load imbalance. These latencies can be tolerated by multithreading. Load imbalance is handled by (i) repartitioning/remapping data onto processors at runtime (an additional overhead) or (ii) migrating threads between processors (not possible on current clusters). Though multithreading solves the latency problem to some extent by keeping all processors busy exploiting parallelism in an application, it has not been enough. Accessing data in irregular applications [28] greatly affects memory access efficiency due to the non-uniform memory access patterns that are unknown until runtime. In addition, the gap between the processor and memory speeds is widening as processor speed increases more rapidly than memory speed. State-of-the-art many-core-on-a-chip architectures (such as Cyclops-64) provide explicitly visible memory hierarchy, with which techniques, such as manual data placement and prefetching [29], can be used to decrease the effective memory latency. To facilitate multithreading, efficient support for

thread level execution and synchronization is integrated in the design of many-core chip architecture.

4 Cyclops-64 Architecture

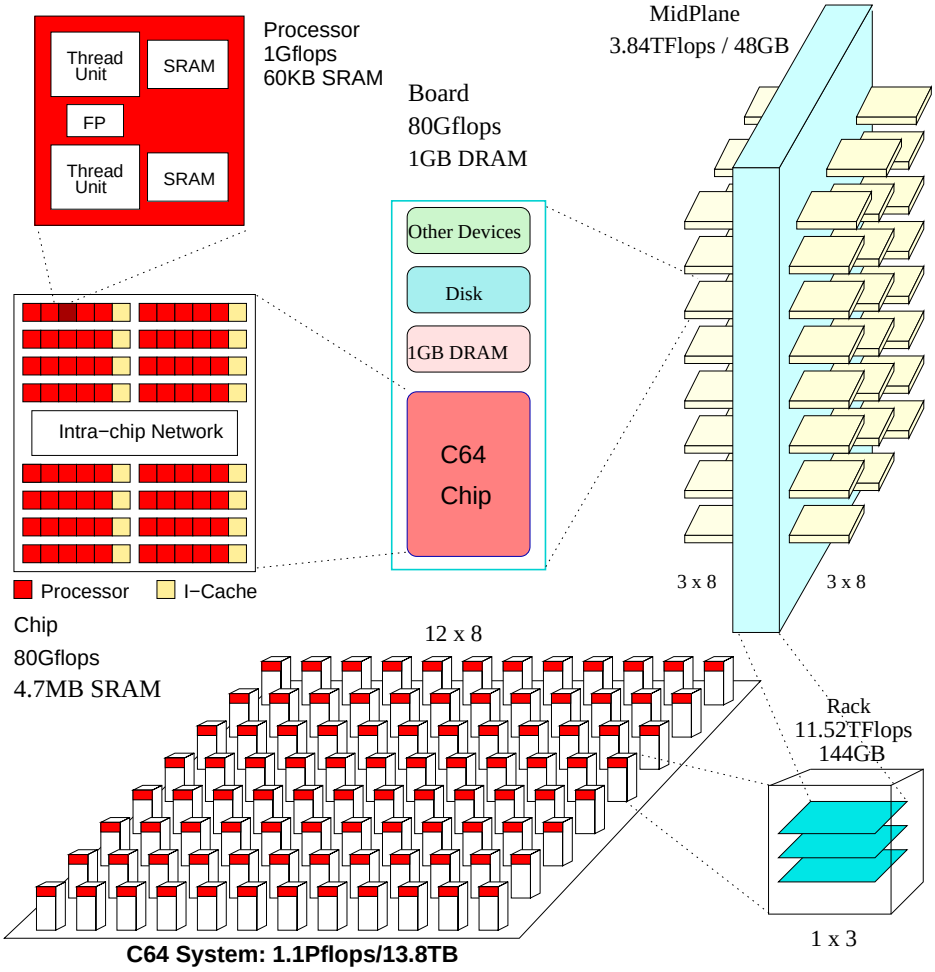

Figure 1: Cyclops 64 Supercomputer

The Cyclops-64 (C64) is a petaflop supercomputer (see Figure 1) project under development at IBM T.J. Watson Laboratory. For its chip architecture (see Figure 2), C64 employs the Many-Core System-on-Chip (SoC) approach, which leverages state-of-the-art VLSI fabrication technology, to achieve high computation rates (even real time performance), low power consumption, and low cost. Based on a cellular architecture, a maximum configuration of a C64 system consisting of 13,824 C64 chip, connected by a 3D mesh network, is expected to achieve over 1 petaflop peak performance. The C64 system is intended to serve as a dedicated compute engine for running high end computing applications, such as molecular dynamics to study protein folding [30], or image processing to support real-time medical procedures. This paper shows that C64 is also an ideal platform for real time finance computation

applications, such as option pricing. To the best of our knowledge, the C64 project is one of the most ambitious petaflop supercomputer projects currently under active development. Unlike other academia projects, a first C64 system is planned to be installed in 2007.

The C64 chip architecture (Figure 2) employs a many-core-on-a-chip design with 160 hardware thread units, half as many floating point units, same amount embedded SRAM memory banks, an interface to off-chip DDR SDRAM memory, and bidirectional inter-chip connection ports on a single silicon chip. A C64 chip consists of 80 processors, each with two thread units, a floating point unit, and two SRAM memory banks of approximately 32KB each. Five processors share a 32KB instruction cache, which is not shown in the figure. Instead of data cache, a portion of each thread unit's corresponding on-chip SRAM bank is configured as the scratchpad memory (SP). Therefore, a thread unit can achieve fast access to its own SP, i.e., one cycle for a store, and two cycles for a load. The remaining sections of all on-chip SRAM banks together form the global memory (GM) that is uniformly addressable from all thread units. The C64 also employs the Network-on-Chip (NoC) concept, all on-chip resources are connected to a 96 ports on-chip crossbar network, which provides a 4GB/s bandwidth per port per direction, 384 GB/s per direction in total. This huge bandwidth sustains all the intra-chip traffic communication and the six routing ports that connect each C64 chip to its nearest neighbors in a 3D-mesh network. Besides the crossbar network, all the thread units within a chip connect to a 16-bit signal bus, which provides a means to efficiently implement barrier. A C64 supercomputer is built out of tens of thousands of such C64 nodes.

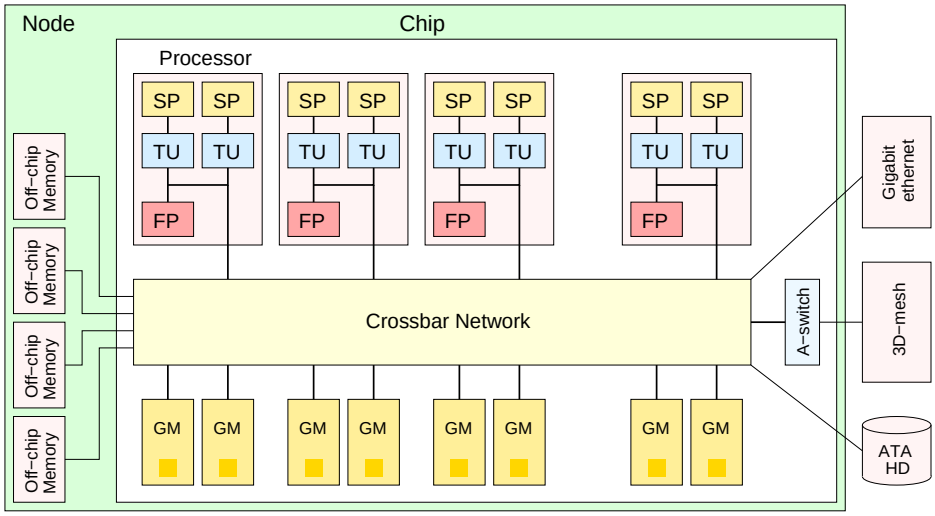

Figure 2: Cyclops 64 Node

In summary, the C64 chip architecture represents a major departure from mainstream microprocessor design in several aspects:

1. The C64 chip integrates a large number of (160) processing elements, embedded memory and communication hardware in the same piece of silicon.

2. A thread unit, the C64 computational cell, is a simple 64-bit, single issue, in-order RISC processor with a small instruction set architecture (60 instruction groups) operating at a moderate clock rate (500MHz). And the execution on a thread unit is non-preemptive.
3. C64 incorporates efficient support for thread level execution. For instance, a thread can stop executing instructions for a number of cycles or indefinitely; and when asleep it can be woken up by another thread through a hardware interrupt.
4. C64 provides no resource virtualization mechanisms. For instance, execution is non preemptive and there is no hardware virtual memory manager. The former means a single application can run at a given time on a set of C64 nodes. Additionally, the OS will not interrupt the user program running on the thread units unless the user explicitly specifies preemption or an exception occurs. The latter means the three-level memory hierarchy of the C64 chip is visible by the programmer.
5. In the C64 chip architecture there is no data cache. Instead, a portion of each SRAM bank can be configured as scratch-pad memory. Such a memory provides a fast temporary storage to exploit locality under software control.
6. The integration of processing logic and memory is further leveraged with a rich set of hardware supported in-memory atomic instructions. Unlike similar instructions on common off-the-shelf microprocessors, atomic instructions in the C64 only block the memory bank where they operate upon while the remaining banks proceed servicing other requests. This functionality facilitates the scalability of multithreading programs with intensive synchronization operations.

5 Experimental Results

5.1 Monte Carlo Experiment Design

Out of all inputs (refer to section 1) into the Black-Scholes option pricing model, it is only volatility that is not observable. In the experiments we consider the following: (a) use of historical volatility of continuously compounded stock returns; (b) use of GARCH-fitted volatility of continuously compounded returns.

During the experiments we generated stock price series under the assumption that prices follow a random walk with drift. We generated increments using the normal probability distribution function. In various stages of the experiment volatility of increments was: (i) Constant; (ii) Decreasing; (iii) Increasing; (iv) Stochastic; (v) Decreasing and stochastic; and, (vi) Increasing and stochastic. The whole idea about using different volatility schemes stems from two earlier studies [31, 32], where it was found that volatilities of individual stock returns increased over the period 1962-1997, while the market volatility did not exhibit any significant pattern. The parameters of the normal distribution used in simulations are taken from the real data set, which we describe below.

Using these generated volatilities σ_t and a pseudo-random number generator, we generate stock price series that follow the geometric Brownian motion process:

$$lnS_t = \gamma + \delta lnS_{t-1} + \nu_t \tag{1}$$

where γ is the drift in the stock price, σ is the variance rate (volatility) of the stock price, In equation 1 we have $\gamma > 0$ to make sure that prices do not fall below zero and that the increment is normally distributed:

$$\nu_t \approx N(0, \sigma_t^2) \tag{2}$$

We assumed the continuously compounded interest rate of 5% and the flat and deterministic yield curve, as it is assumed in the Black-Scholes model. Expiration date was set at 3 months from the starting point (time t), and strike price was varying from 5 to 105 with the step size of 20. The starting prices P_0 in all cases were \$5.00. Using these parameters, we calculated call prices for the non-dividend paying stocks for each point in time t using all inputs as known. The formula used in calculations is the classical option pricing formula (see [2]). Next, we calculated option prices with all the same inputs but measured volatility.

In the first run of the experiment we estimated conditional volatilities and used them in the option pricing formula. In the second run of the experiment we estimated historical and GARCH-fitted volatilities of continuously compounded stock returns [33]. After calculating option prices (C^{TRUE}) using known data and option prices using observable and measured data ($C^{MEASURED}$), we calculated the option pricing error E in the following way:

$$E = C^{MEASURED} - C^{TRUE} \tag{3}$$

This error would give us a dollar estimate of the mistake in case of using an improper measure of volatility in the Black-Scholes option pricing formula).

5.1.1 Option Pricing Results

Figure 3 depicts one of the many sets of pricing errors that results from the experimental study. This graph shows, which of the data generating processes creates larger errors when we use GARCH-fitted volatility of continuously compounded returns. The x-axis in all these figures corresponds to the strike prices ranging from \$5 -\$105 and the y-axis corresponds to the option pricing errors: -0.004 to 0.003 in fig 3 (a); -0.05 to 0.03 in fig 3 (b); -0.06 to 0.01 in fig 3 (c); -0.04 to 0.04 in fig 3 (d); -20.0 to 140.0 in fig 3 (e); -0.20 to 0.06 in fig 3 (f).

Figure 3 (a) corresponds to constant volatility, Figure 3 (b) corresponds to decreasing volatility, Figure 3 (c) corresponds to increasing volatility, Figure 3 (d) corresponds to stochastic volatility, Figure 3 (e) corresponds to decreasing and stochastic volatility, Figure 3 (f) corresponds to increasing and stochastic volatility. The six legends below each of these figures identify the prices starting from \$5 to \$105 in steps of \$20 for respective figures.

During the experiments the drift component of the stock price is \$0.006t, where t stands for the number of days. Therefore, we are able to plot call pricing error against various exercise prices and unconditional expectations of stock prices ($E[S_t] = 0.006 * t$). In this part of the experiment, $GARCH$ model is estimated for the sample size k with the mean equation that regresses continuously compounded returns on a constant. We generate fitted volatility and record the last value h_k. This is our

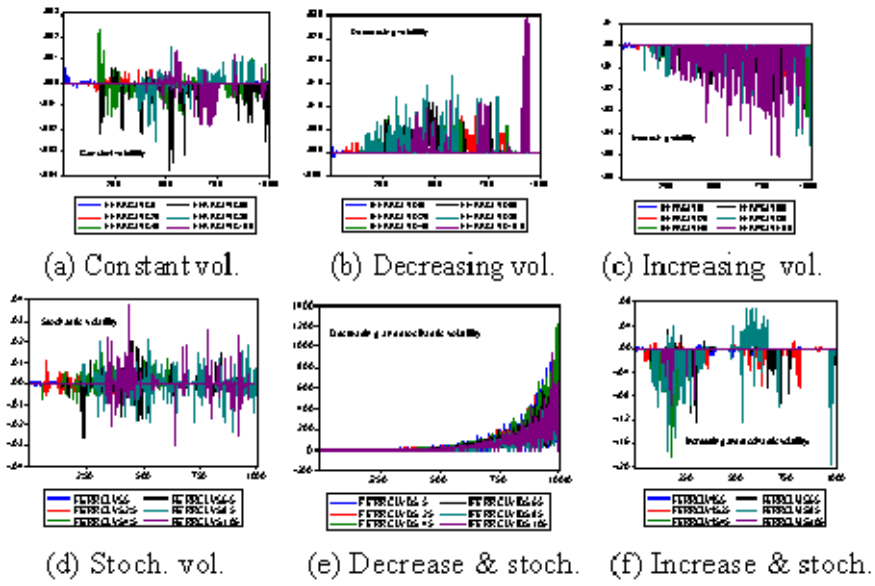

Figure 3: Option pricing errors for stock prices generated using various patterns of volatility, average across series. GARCH-fitted volatility estimates based on continuously compounded returns are used to generate option prices.

input into the Black-Scholes formula for calculating the measured call price, $C_k^{MEASURED}$. Next, we add one more data point to the stock price series, estimate the GARCH model for the sample of $k+1$ observations, and use h_{k+1} to calculate $C_{k+1}^{MEASURED}$. Using this process, we generate the time series of measured call prices $C_t^{MEASURED}$.

As we can infer from Figure 3, in some cases (for example, cases b and e) option pricing errors grow with higher sample size. This can be attributed to the non-stationarity of option prices: as sample size increases, sample volatility of data approaches infinity. Therefore, we get upward-biased estimates for the volatility of stock prices. The comparative statistics of Black-Scholes model show that call price increases when stock price volatility increases. Therefore, upward-biased estimates of stock prices result in upward-biased estimates of option prices. This situation could result in a false belief that there exists a Put-Call-Parity arbitrage strategy based on erroneously calculated call prices. In case of constant and stochastic volatility of prices (Fig. 3 (a and d)) the situation is not as clear. Option pricing errors seem to be fluctuating around zero on the average.

These results were obtained first on a statistical package called *E-Views*. We reproduced the results first with the in-house developed sequential code before parallelizing our code and implementing on the many-core-on-a-chip architecture. On a AMD-K7-II processor, with 6 possible exercise prices, 6 patterns of volatility, and 1000 data points, one run of the experiment using the E-Views package for only 20 iterations took 4 hours and 45 minutes.

In the next section, we demonstrate the performance results of running the simulation on the C64 architecture.

5.2 Monte Carlo Simulation on C64

Table 1: Processor Configurations

Processor	Clock Rate	Cache	off-chip Memory	Compiler
Cyclops-64 Thread Unit	500MHz	No data cache 5MB on-chip SRAM memory	1GB DRAM	gcc-3.2.3 for C64
AMD Opteron	2.4GHz	1MB L2 cache	3GB	gcc-3.2.3 for x86_64
Intel Centrino	1.86GHz	2MB L2 cache	512MB	gcc-3.3.6
Intel Pentium4	3.2GHz	512KB L2 cache	1GB	gcc-3.4.3

The performance of the Monte Carlo algorithm on C64 is compared to different representative off-the-shelf processors: AMD Opteron 250, Intel Centrino, and Intel Pentium 4. The basic configurations of those processors are shown in Table 1. The computation conducted on C64 is simulated with the FAST simulator [34], which is a functionally accurate simulation tool set for the C64 cellular architecture. The parameter setting for 4 different simulations is shown in Table 2

For C64, a portion of each SRAM bank can be configured as the scratchpad memory (16KB, in this case), which guarantees fast and predictable access latency for the corresponding owner thread unit. For the Monte Carlo simulation, we carefully design the code, such that the intermediate results of the computation can completely fit into a thread unit's scratchpad memory. Only the latest ith simulation results were stored on-chip for the next (i+1)th simulation. The previous 1, ... , $i-1$ results were stored off-chip. For the parallel version, since no synchronization is needed, the performance increase is proportional to the number of threads employed.

We performed on average ten runs on each of the machine and obtained the execution times as shown in Figure 4 for the simulation parameters chosen in Table 2. Please note that only one C64 thread unit is used for this comparison.

The Monte Carlo simulation is known to be embarrassingly parallel, i.e., all the thread units can work independently without involving synchronization during the simulation. Since all the intermediate data needed for a thread unit's computation can fit into its own scratchpad memory, there is no runtime

Table 2: Parameters for Monte Carlo Simulation

Parameter	begprice	step	variety	ARSIZE
sim-1	5	20	6	1000
sim-2	5	40	8	1000
sim-3	5	40	8	2000
sim-4	5	100	10	5000

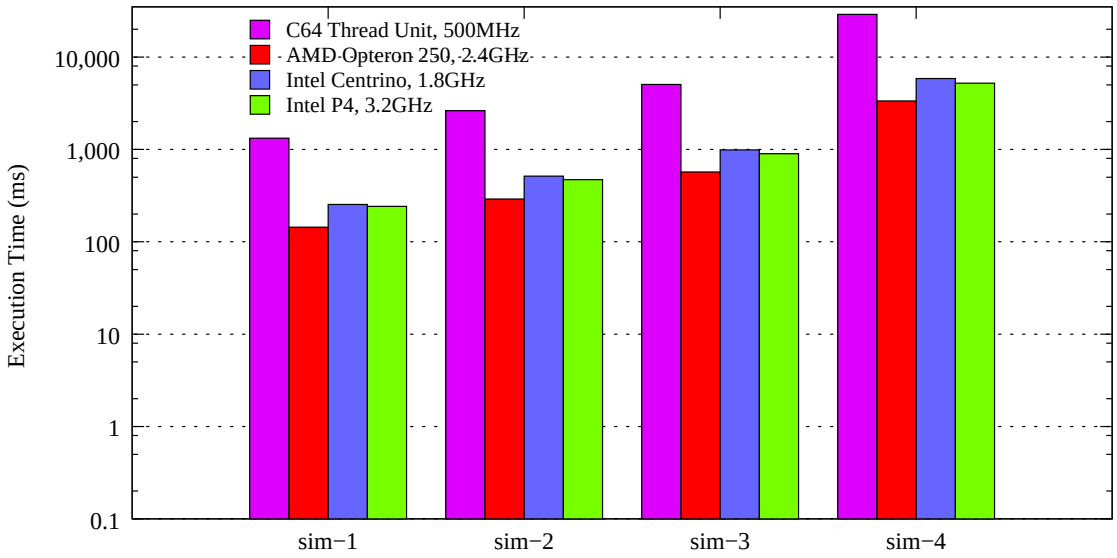

Figure 4: Execution Time of Monte Carlo Simulation

competition and conflict for shared resources, such as global on-chip memory. As a result, the speedup of the parallel version can increase linearly with the number of threads used (This is also demonstrated with the simulation). By conducting all 160 thread units for the Monte Carlo simulation, a C64 chip can complete 160 times simulation task with the same amount of execution time shown in Figure 4. For all other three processors, the execution time increases proportionally. Through calculations for all four groups of simulations, for the Monte Carlo option pricing simulation, we can conclude that *1 C64 node delivers the performance equivalent to 18 Opteron 250 CPU, 32 Intel Centrino CPU, and 28 Intel P4 3.2GHz CPU*. The approximate price for a C64 node would be quite similar to machines built with those CPUs compared. Therefore, the C64 delivers huge *performance/price* improvement over traditional microprocessors for computational finance applications. Moreover, for the Monte Carlo simulation, since all data fits into the scratchpad memory, all the computation is performed locally for each thread units and the power on the very long wires going to and from the crossbar is saved. In such a situation, the power consumption of a C64 node is lower than or close to machine built with conventional microprocessor. Given a C64 node delivers tens of times performance, the C64's *performance/power consumption* ratio is much higher compared to other microprocessors. Unlike a traditional microprocessor, which dies if any parts on the chip is broken, the C64 chip can still be in working condition, even if one or more thread units/memory banks fail.

6 Conclusions

The current work has paved a way to study a finance problem on many-core-on-a-chip architecture. As a first attempt we have analyzed a embarrassingly parallel problem of MC simulation for an important option pricing problem.

In this paper we showed that the use of incorrect volatilities of the asset prices in the Black-Scholes model would result in moderate to large errors in option prices.

We could achieve this conclusion by conducting many experiments for various asset price ranges, strike price ranges and 3 different volatilities. The timing results on C64 show that these various sets of simulations could be done in a real-time fashion. For the Monte-Carlo option pricing simulation done in the current study, we can conclude that 1 C64 node delivers the performance equivalent to 18 Opteron 250 CPU, 32 Intel Centrino CPU, and 28 Intel P4 3.2GHz CPU. This translates to high *performance/price* and *performance/power consumption* improvement over traditional microprocessors for computational finance applications.

With the promising results from the current study we are developing a FFT algorithm for C64 architecture to study option pricing problem based on an earlier study [9].

Acknowledgement

Part of this work was conducted while the second and third authors were visiting University of Delaware in Summer 2005. These authors acknowledge the financial support from Natural Sciences and Engineering Research Council of Canada and the University Research Grant Program of the University of Manitoba. The first and last authors would like to acknowledge the support from IBM, in particular, Monty Denneau, who is the architect of IBM Cyclops-64 architecture, ETI, the Department of Defense, the Department of Energy (DE-FC02-01ER25503), the National Science Foundation (CNS-0509332), and other government sponsors. We would also like to acknowledge other members of the CAPSL group at University of Delaware, who provide a stimulus environment for scientific discussions and collaborations, in particular Juan del Cuvillo, and Ziang Hu.

References

[1] E. J. Kontoghiorghes, A. Nagurnec, and B. Rustem, "Parallel Computing in Economics, Finance and Decision-making," *Parallel Computing*, vol. 26, pp. 207–509, 2000.

[2] J. C. Hull, *Options, Futures and Other Derivatives*, 5th ed. Upper Saddle River, NJ: Prentice Hall, 2002.

[3] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *J.Political Economy*, vol. 81, pp. 637–654, Jan. 1973.

[4] P. Boyle, "Options: A Monte Carlo Approach," *Journal of Financial Economics*, vol. 4, pp. 323–338, 1977.

[5] S. Rahmail, I. Shiller, and R. K. Thulasiram, "Different Estimators of the Underlying Asset's Volatility and Option Pricing Errors: Parallel Monte Carlo Simulation," in *Proc. Intl. Conf. on Computational Finance and its Applications*, Bologna, Italy, April 2004, pp. 121–131.

[6] A. Srinivasan, "Parallel an Distributed Computing Issues in Pricing Financial Derivatives through Quasi Monte Carlo," in *Proc. Intl. Parallel and Distributed Processing Symp. (IPDPS02)*, Fort Lauderdale, FL, April 2002.

[7] J. Cox, S. Ross, and M. Rubinstein, "Option pricing: A simplified approach," *J. Financial Economics*, vol. 7, pp. 229–263, 1979.

[8] R. K. Thulasiram, L. Litov, H. Nojumi, C. T. Downing, and G. R. Gao, "Multithreaded Algorithms for Pricing a Class of Complex Options," in *Proc. Intl. Parallel and Distributed Processing Symp. (IPDPS01)*, San Francisco, CA, Apr. 2001.

[9] S. Barua, R. K. Thulasiram, and P. Thulasiraman, "High Performance Computing for a Financial Application using Fast Fourier Transform," in *LNCS Vol. 3648, Proc. European Parallel Computing Conference (EuroPar05)*, Lisbon, Portugal, aug-sep 2005, pp. 1246–1253.

[10] P. Carr and D. B. Madan, "Option Valuation using the Fast Fourier Transform," *The Journal of Computational Finance*, vol. 2, no. 4, pp. 61–73, 1999.

[11] R. K. Thulasiram and P. Thulasiraman, "Performance Evaluation of a Multithreaded Fast Fourier Transform Algorithm for Derivative Pricing," *The Journal of Supercomputing*, vol. 26, no. 1, pp. 43–58, Aug. 2003.

[12] D. Tavalla and C. Randall, *Pricing Financial Instruments: The Finite Difference Method.* New York, NY: John Wiley and Sons, 2000.

[13] I. J. Clark, "Option Pricing Algorithms for the Cray T3D Supercomputer," *Proceedings of the first National Conference on Computational and Quantitative Finance*, Sept. 1998.

[14] A. Mayo, "Fourth Order Accurate Implicit Finite Difference Method for Evaluating American Options," in *Proc. Intl. Conf. on Computational Finance 2000*, London, England, June 2000.

[15] R. K. Thulasiram, C. Zhen, and A. Gumel, "A Second Order L_0 Stable Algorithm for Evaluating European Options," in *Proc. 18th Intl. Symp. on High Performance Computing Systems and Applications*, Winnipeg, MB, Canada, May 2004, pp. 17–23.

[16] A. Andricopoulos, M. Widdicks, P. Duck, and D. Newton, "Curtailing the Range of Lattice and Grid Methods," *Journal of Derivatives*, vol. 11, no. 4, pp. 55–61, June 2004.

[17] D. Burger and J. R. Goodman, "Billion-transistor architectures - guest editors' introduction," *IEEE Computer*, vol. 30, no. 9, pp. 46–49, 1997.

[18] L. Spracklen and S. G. Abraham, "Chip multithreading: Opportunities and challenges," in *the 11th Intl. Symp. on High-Performance Computer Architecture (HPCA'05)*, 2005.

[19] L. Hammond, B. A. Nayfeh, and K. Olukotun, "A single-chip multiprocessor," *Computer*, vol. 30, no. 9, pp. 79–85, 1997.

[20] S. Y. Borkar, H. Mulder, P. Dubey, S. S. Pawlowski, K. C. Kahn, J. R. Rattner, and D. J. Kuck, "Platform 2015: Intel processor and platform evolution for the next decade," 2005.

[21] The CELL project at IBM research. [Online]. Available: http://www.research.ibm.com/cell/

[22] P. Kongetira, "A 32-way multithreaded SPARC processor," Hot Chips 16, 2004. [Online]. Available: http://www.hotchips.org/archive/

[23] ARM11 MPCore Processor. [Online]. Available: http://www.arm.com/

[24] T. Maruyama, "SPARC64 VI: Fujitsu's next generation processor," in Mircroprocessor Forum 2003, 2003.

[25] ClearSpeed CSX600. [Online]. Available: http://www.clearspeed.com/products/si.php

[26] M. T. Islam, P. Thulasiraman, and R. K. Thulasiram, "Implementation of Ant Colony Optimization Algorithm for Mobile Ad hoc Network Applications: OpenMP Experiences," *Parallel and Distributed Computing Practices*, vol. 2, no. 5, pp. 177–191, 2002.

[27] D. Martin, P. Thulasiraman, and R. Gordon, "Local Independence in Computed Tomography as a Basis for Parallel Computing," *Technology in Cancer Research and Treatment*, vol. 4, no. 2, April 2005.

[28] P. Thulasiraman, G. Heber, A. A. Khokhar, and G. R. Gao, "Load Adaptive Algorithms and Implementations for the 2D Discrete Wavelet Transform on Fine-Grain Multithreaded Architectures," *Journal of Parallel and Distributed Computing*, vol. 64, no. 1, pp. 68–78, Jan. 2004.

[29] Z. Zhang and J. Torrellas, "Speeding up Irregular Applications is Shared-Memory Multiprocessors: Memory Binding and Group Prefetching," in *ACM Intl. Symp. on Computer Architecture*, Santa Margherita, Italy, 1995, pp. 188–199.

[30] G. Almasi, C. Cascaval, J. Castanos, and M. D. et al., "Demonstrating the scalability of a molecular dynamics application on a petaflops computer," in *Proc. Intl. Conf. on Supercomputing*, Sorrento, Italy, June, 2001, pp. 393–406.

[31] G. Schwert, "Why Does Stock Market Volatility Change Over Time?" *J. of Finance*, vol. 43, pp. 1095–1112, 1989.

[32] J. Campbell, M. Lettau, B. Malkiel, and Y.Xu, "Have Individual Stocks Become More Volatilie? An Empirical Exploration of Idiosyncratic Risk," *J. Finance*, vol. 56, pp. 1–43, 2001.

[33] M. Chesney and L. Scott, "Pricing European Currency Options: A Comparision of the Modified Black-Scholes and a Random Variance Model," *Journal of Financial and Quantitative Analysis*, vol. 24, pp. 267–284, 1989.

[34] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "FAST: A functionally accurate simulation toolset for the Cyclops64 cellular architecture," in *Workshop on Modeling, Benchmarking, and Simulation (MoBS2005)*, Madison, WI, June 2005.