CSC447: Parallel Programming for Multi-Core and Cluster Computers

A Parallel Monte Carlo Simulation for Black-Scholes Option Valuation Using OpenMP, OpenACC, and CUDA

Due: April 28, 2022

1 Financial Options

The objective of this project is to parallelize a *Monte Carlo simulation of the Black-Scholes* option valuation model using two techniques: *OpenMp* and *OpenACC*. The assignment aims to gain experience in *multithreaded programming using CPUs and GPUs* in addition to benchmarking for performance and accuracy. The assignment is based on the *Black-Scholes* model from finance, although no prior knowledge of finance is necessary to successfully complete the assignment.

2 Background

Consider a scenario in which someone calls you today with the following offer:

> "In 3 months' time you will have the option to purchase *Microsoft Corporation* shares from me at a price of \$120 per share."

The key point is that you have the *option* to buy the shares. Three months from now, you may check the market price and decide whether or not to exercise the option.[1] This deal has no downside for you three months from now you either make a profit or walk away unscathed. The seller, on the other hand, has no potential gain and an unlimited potential loss. To compensate, there will be a cost for you to enter into the option contract. You must pay some money up front. The option valuation problem is to compute a fair value for the option [2]. More precisely, it is to compute a fair value at which the option may be bought and sold on an open market. The option described above is a *European call.* The Microsoft shares are an example of an asset, a financial quantity with a given current value but an uncertain future value. Formalizing the idea and introducing some notation, we have:

> **Definition:** A European call option gives the holder the opportunity to purchase from the writer an asset at an agreed expiry time T at an agreed exercise price E. Given a time t, we will let $S(t)$ denote the asset value at time t, so $S(T)$ is the value

[1] In practice, you would exercise the option if and only if the market price were greater than \$120, in which case you could immediately re-sell for an instant profit.

of the asset at the expiry time. The final payoff to the purchaser is $\max\{S(T)-E,0\}$, because:

- if $S(T) > E$, the option will be exercised for a profit of $S(T) - E$, whereas
- if $S(T) \leq E$, the option will not be exercised.

In 1973, *Robert C. Merton* published a paper presenting a mathematical model which can be used to calculate a rational price for trading options [3], and won later a Nobel prize for his work. In that same year, options were first traded in the open market. Since then, the demand for option contracts has grown to the point that trading options typically far outstrips that for the underlying assets. Merton's work expanded on that of two other researchers, *Fischer Black and and Myron Scholes* [1], and the pricing model became known as the Black-Scholes model. The model depends on a constant sigma, σ, representing how volatile the market is for the given asset, as well as the continuously compounded interest rate r.

3 Monte Carlo methods

Computers are often used to predict the behavior of physical systems such as the airflow around a new automobile or airplane design, the effect of an earthquake on a bride or building, or the behavior of financial markets under specified conditions. One class of algorithms for such simulations are called *Monte Carlo* methods, and are distinguished from other methods in their use of random numbers to select a set of points at which to evaluate a function. The Monte Carlo method of calculating π that was described in the class and in various lectures selects a set of random set of points in a unit square, and counts the fraction of those points that are inside a quadrant of the unit circle. It then uses this ratio and the known formula for the area of a circle to estimate π.

Monte Carlo algorithms are often used to find solutions to mathematical problems that cannot easily be solved by other means and are relatively easy to program on parallel machines, because each processor can evaluate the function independently on a subset of the points.

4 Sequential Algorithm

In this assignment you will be using the Monte Carlo technique to calculate the Black-Scholes pricing model. It will take as input a number of trialsM. As with any Monte Carlo calculation, a higher value will give us a more accurate answer, but take more time. Pseudocode for a sequential algorithm for this problem is given below. In addition to M, the pseudocode refers to the input variables described in Section 2. They are summarized here for your convenience.

- S: asset value function
- E: exercise price
- r: continuously compounded interest rate
- σ: volatility of the asset
- T: expiry time
- M: number of trials

The pseudocode also uses several internal variables:

- trials : array of size M, each element of which is an independent trial (iteration of the Black-Scholes Monte Carlo method)
- mean: arithmetic mean of the M entries in the trials array
- `randomNumber()`, when called, returns successive (pseudo)random numbers chosen from a Gaussian distribution. NOTE: this function will be provided for you. See Section 5.2 for more details.
- mean(a) computes the arithmetic mean of the values in an array a
- stddev(a, μ) computes the standard deviation of the values in an array a whose arithmetic mean is μ.
- confwidth: width of confidence interval
- confmin: lower bound of confidence interval
- confmax: upper bound of confidence interval

Algorithm 1 A sequential Monte Carlo simulation of the Black-Scholes model

```
for $i = 0$ to $M - 1$ do
    t := $S.\exp((r - \frac{1}{2}\sigma^2).T + \sigma\sqrt{T}.randomNumber()$
    trials[i] := $expr(-r \times T).\max\{t - E, 0\}$
end for
mean := mean(trials)
stddev := stddev(trials, mean)
confwidth := 1.96 $\times$ stddev/ $sqrtM$
confmin := mean $-$ confwidth
confmax := mean $+$ confwidth
```

5 Project Requirements

This project involves parallelizing the code using a combination of `OpenMP`, `OpenACC`, and `CUDA C`. This will be done by making changes to the provided working sequential implementation of the *Black- Scholes Monte Carlo* method. You should start out by running the sequential code and getting an understanding for the flow of the program as well as what outputs you should expect for a given set of input parameters.

A working Makefile that uses `pthreads` has been included that will build the program. Instructions on running it can be found within `main.c`. Note that there is an input file named `params` which passes the values needed to evaluate the Black-Scholes confidence intervals. These input variables are the same as the pseudocode: asset value function S; exercise price E; continuously compounded interest rate r; volatility of the asset σ; expiry time T; and number of trials M. You will want to vary these values once you get a working parallel version of the code and see its effect on the runtime of your program. The main functions that you will be editing can be found in black `scholes.c`.

The output of the program consists of the input parameters, the time it took for `black_scholes ()` to finish executing, and the lower/upper bounds of the Black-Scholes confidence intervals. These confidence intervals should be nearly the same in both sequential and parallel execution, assuming all input parameters except *nthreads* are the same. However, your execution time should be less in parallel. This may not be true if you did not choose a good parallel strategy or if the cost of thread creation overruns the need for parallelism in certain cases with a small number of trials.

5.1 Provided for you

A working, serial implementation of the Black-Scholes Monte Carlo method has been provided for you. You should only have to modify some of the functions found within black_scholes.c and also learn to use the PRNG `gaussrand1()` function successfully. The code was written so that to be parallelized using `pthreads`. Make sure you update these.

5.2 Hints:

The following are characteristics of a correctly working program:

- *Parallel random number generation:* The Monte Carlo method depends on having a high-quality pseudorandom number generator (PRNG). Imagine, for example, that in the π program example, that all of the random points (darts on our boards) end up in a narrow area of the unit square. Then our calculation of π would not be accurate. Computers cannot generate truly random numbers, but can produce a stream of 'pseudorandom' numbers that behave 'random enough' for our purposes. Parallelism adds further complications, as the random number generator function must behavior correctly when two or more threads call it simultaneously, i.e., it must be *thread-safe.*

- *Thread-safe PRNG API:* The provided code is based on pthreads (which needs to be modified for OpenMP/OpenACC). The interface to the thread-safe pseudorandom number generator can be found in random.h and gaussian.h. First, before spawning any threads, call `init_prng ()` with a seed argument either the same seed each time (for debugging) or a random seed generated by random `seed()` (in random.h). Then, in each thread, call spawn `prng_stream()` with the ID number of that thread, and use the returned opaque object as the f state parameter of `gaussrand1()` (in gaussian.h). The function pointer parameter of `gaussrand1()` should be uniform random double. This needs to be modified!

- As the number of trials increase, the confidence interval becomes smaller.

- For a given number of trials, the time it takes to calculate the confidence interval should decrease as you increase the number of threads, up to a certain number of threads which often (but not always!) corresponds to the number of processors. Beyond that number, the time starts increasing due to factors like thread creation and destruction overhead overwhelming potential parallelism, and possible saturation of memory bandwidth.

6 What to turn in

1. Turn in all source and header files, along with the Makefile. Any command-line arguments that we need to supply should be clearly documented.

2. Comment your code, as your grade does not depend solely on whether it runs in parallel.

3. Submit a report on the performance results from your completed project including speedup for various number of threads.

(a) Document the experimental setup: what type of processor? How many processors (cores) are available on this machine? Is your solution scalable.

(b) Experiment with the Black-Scholes parameters to see if they affect the rate of trials per second that you can do. Do this for various numbers of threads.

(c) Experiment with the number of threads, and report the best speedup you see with your code (given a fixed number of trials M).

(d) Produce a speedup table or graph, showing the speedup for various numbers of threads created (from 1 to 8 at least).

(e) Explain why you believe the performance behaves as it does.

(f) Feel free to include any additional benchmarking or test results that you find interesting, as well as anything you think we should know about your code. Here is where you should document any special instructions for building and/or running your code.

4. Your written report should be in a file starting with your last name and preferably in LaTeX.

5. Your project will be graded based on your written report, running the code, the **speed-up percentage**, and looking at selected parts of the code you have written.

7 Deadline

This project will be due on May 10, 2021 by midnight on `GitHub`. Please commit often to show your progress. Presentations will be during lab hours.

References

[1] F. Black and M. S. Scholes, The pricing of options and corporate liabilities, Journal of Political Economy, 81 (1973), pp. 637–654. Available at http://ideas.repec.org/a/ucp/jpolec/v81y1973i3p637-54.html.

[2] D. J. Higham, Black-Scholes for scientific computing students, Computing in Science and Engineering, 6 (2004), pp. 72–79.

[3] R. C. Merton, Rational theory of option pricing, Bell Journal of Economics and Management Science, (1973).