Monte Carlo Option Pricing on a SLURM Computer Cluster

Scott Griffin Jr. (scott-griffin@uiowa.edu)
Under the supervisory of John Lewis Jr. (john-lewisjr@uiowa.edu)

For fulfilment of requirements for FIN:9391 Thesis in Finance at the University of Iowa
Henry B. Tippie College of Business
21 E Market St, Iowa City, IA 52245, United States
Tel.: 319-335-0862

Abstract: A Python script template for Monte Carlo option pricing on a SLURM cluster is introduced, and tested scripts for various call option models are provided. A guide for setting up a simple SLURM computer cluster is given. Comparisons of runtime and price error by option S/K ratio, number of SLURM worker computers, and size of simulation are discussed.

Keywords: Slurm, monte carlo, option pricing, hpc, computer clusters, quantitative finance

1. Introduction

Mathematical and statistical models have become increasingly important tools used in financial management, demonstrated by the approximately 210,000 papers on Google Scholar related to ”Quantitative Finance” in 2023 compared to 21,200 in 1990. For ”Finance Machine Learning” there were around 145,000 in 2023 compared to 7,700 in 1990 (see appendix A.1.). Recent models proposed in the subject include an optimized quasi-Monte Carlo pricing model from Xiao & Wang (2019), an artificial neural network for forecasting stock price movements of Brazilian power distribution companies from Laboissiere et. al. (2015), and a chaotic transient-fuzzy deep neuro-oscillatory network for predicting prices of 129 worldwide financial assets concurrently from Lee (2020).

Computation requirements for mathematical models used in quantitative finance can be quite large. Consider a European call option with an initial asset price of 50, a strike price of 50, a time-to-maturity of 20 periods, a risk-free rate of 1.5%, a standard deviation of 2.5%, and a dividend rate of 0.5%. To reliably price this option with Monte Carlo simulation to less than .001 difference from the corresponding Black-Scholes closed-form solution, at least 4,588,820,000 asset path simulations are required[1]. This would take around ten hours to run on a free-tier Google Colab runtime[2].

One way to address the computational requirements of running multiple large Monte Carlo pricing models in a timely and effective matter is to utilize a computer cluster, which is a group of interconnected computers used for high-performance computing (HPC). To reduce total runtime, a single controller computer manages many worker computers to run computations in parallel.

A popular job scheduler for computer clusters is the SLURM Workload Manager, previously known as Simple Linux Utility for Resource Management (SLURM). It's an open-source job scheduler for Linux and Unix-like kernels, and it's used on over half of the world's top 500 largest supercomputers Top500 (2023). It's also the job scheduler used on Stanford, Princeton, Iowa State, and Northeastern university's HPC research clusters, among others Stanford (2023) Princeton (2024) Iowa State (2023) Northeastern (2023).

[1]Boyle (1977) gives a crude Monte Carlo estimate of 17.190 with standard deviation of 0.479 from 5,000 asset path simulations. The number of asset path simulations required to construct 95% confidence interval limits of $\pm 0.001 = (0.479/0.0005)^2 * 5,000 = 4,588,820,000$.

[2]An identical European call option priced with 100,000,000 asset path simulations took around 14 minutes on a free-tier Google Colab Python3 runtime, so a crude time estimate for 4,588,820,000 asset path simulations = 4,588,820,000 / (100,000,000/14) = 642.43 minutes, or 10.71 hours.

2. What are options?

In finance, an option is a contract that rewards to the buyer a potential future payoff dependent on the future price of one or more underlying assets, relative to some strike price or threshold. The buyer of an option maintains the right but not the obligation to exercise the option and receive the payoff. In the first half of 2023, nearly \$1.4 trillion worth of over-the-counter (OTC) financial options were sold globally. At the end of the first half of 2023, the global OTC financial options market had an outstanding notional amount exceeding \$65 trillion BIS (2023).

A simple financial option is the European-style option. A European call option issued on date t gives the buyer the right but not the obligation to purchase an underlying asset S at a predetermined strike price K, on a predetermined maturity date T, where t < T. The payoff for a European call option at maturity T is equal to max(S(T)-K, 0), where S(T) is the underlying asset price on the maturity date T. A European put option is structured in the same way, except it gives the buyer the right to sell the underlying asset at the predetermined strike price instead of the right to buy it at the predetermined strike price. The payoff for a European put option with maturity T is equal to max(K-S(T), 0).

If the S/K ratio of a European option is less than one (in other words the initial underlying asset price is less than the predetermined strike price) that option is said to be out-of-the-money (OTM). If its S/K ratio is equal to one, the option is at-the-money (ATM), and if its S/K ratio is greater than one the option is in-the-money (ITM).

Options are priced by modelling the probability distributions of expected future payoffs. Less-complex options can be priced using closed-form pricing formulas, which rely on assumptions about the distribution of returns of the underlying asset(s). European-style options can be priced with formulas derived from the Black-Scholes partial differential equation (PDE), which was introduced by Black & Scholes (1972) in their paper, *The Pricing of Options and Corporate Liabilities*. The Black-Scholes PDE is given as:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0 \tag{1}$$

Where V is the price of the derivative, S is the price of the underlying asset, t is time, σ is the volatility of the underlying asset, and r is the risk-free interest rate.

The Black-Scholes formula for pricing a European call option is derived from the Black-Scholes partial differential equation (PDE) and is given as:

$$C(S,t) = S_t N(d_1) - Ke^{-r(T-t)} N(d_2) \tag{2}$$

where $\mathrm{d}_1 = \frac{1}{\sigma\sqrt{T-t}} \left[\ln\left(\frac{S_t}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t) \right]$,
$d_2 = d_1 - \sigma\sqrt{T-t}$.

Here, $C(S,t)$ is the price of the call option, S_t is the current price of the underlying asset, K is the strike price, r is the risk-free interest rate, T is the time to maturity, $N()$ is the cumulative distribution function of the standard normal distribution, and σ is the volatility of the underlying asset. More detail on pricing European options can be found in chapter three of the book *A Course in Derivative Securities: Introduction to Theory and Computation* by Kerry Back (2005).

3. What are Monte Carlo option pricing models?

Monte Carlo option pricing models seek to accurately price options through simulation as an alternative to closed-form option pricing formulas. Some financial options cannot be priced with a closed-form formula for a variety of reasons, including the presence of early exercise conditions or a payoff that depends on the path of the underlying asset. Monte Carlo simulation is an applicable alternative for pricing such options; a price can be converged upon for any option given there is a representative underlying distribution for each random variable and enough simulations.

Boyle (1976) introduced the Monte Carlo method for pricing options in his paper, *Options: a Monte Carlo Approach*. Using the Cox & Rox (1976) assumption of risk neutrality where the expected return of all assets is the risk-free rate (r), as well as the assumption from Aitchison & Brown (1963) that stock prices follow a lognormal distribution, the expected return of an asset (S) from time t to T is given by:

$$E\left(\frac{S(T)}{S(t)}\right) = e^{r(T-t)} \tag{3}$$

Where $t \le T$.

And the expected return over one time period can be written as:

$$E\left(\frac{S(t+1)}{S(t)}\right) = \exp(r) = \exp\left(\left(r - \frac{\sigma^2}{2}\right) + \frac{\sigma^2}{2}\right) \tag{4}$$

And the expected log-return over one time period can be written as:

$$E\left[\log\left(\frac{S(t+1)}{S(t)}\right)\right] = r - \frac{\sigma^2}{2} \tag{5}$$

Then the distribution of potential asset prices one period in the future can be simulated by including a normally distributed random variable (X) with mean of zero and variance of one:

$$S(t+1) = S(t)\exp\left[r - \frac{\sigma^2}{2} + X\sigma\right] \tag{6}$$

Then to price a European call option we can simulate a number of asset prices from t to T, calculate the average option payoff, then discount it back to time t by the risk-free rate.

$$C = e^{-r(T-t)} \sum_{i=0}^{M} \max(S_i(T) - K, 0)/M \tag{7}$$

Where M is the number of simulated asset prices, and $S_i(T)$ is the terminal price of the simulated asset i.

Since the introduction of Monte Carlo option pricing programs, techniques to reduce price variance have been proposed. Clewlow and Carverhill (1994) published the first approach to using hedges as control variates in Monte Carlo simulation, reducing asset price variance and therefore the number of simulations required to reach an accurate price. The control variate method consists of constructing a new random variable based on the original asset and a related asset, scaled by a correlation coefficient. If correlation between the two assets is nonzero, the new random variable will have a reduced variance relative to the original asset. In appendix A.2.3 we utilize a control variate to price an Asian-style call option. Control variate methods are described in detail in chapter nine of the book *A Course in Derivative Securities: Introduction to Theory and Computation* by Kerry Back (2005).

4. What is a computer cluster?

A computer cluster is a group of interconnected computers used for high-performance computing. In a simple computer cluster there is one controller computer that manages the parallel processing of a computation across a number of worker computers. The controller computer sends work to each of the worker computers and collects their responses. Computer clusters can easily reduce the computation time of 'embarrassingly parallel' problems; where little or no effort is required to split the problem up into a number of parallel problems. In these sorts of problems worker computers process identical computations simultaneously then combine their results. Monte Carlo option pricing is an 'embarrassingly parallel' kind of problem as it requires a number of asset paths to be independently simulated.

It is unknown when the first cluster computer was built, or who built it. Greg Pfister in his 1998 book *In Search of Clusters*, theorized that computer clusters were invented by computer customers who figured out how to aggregate computing power to solve practical problems. A formal paper on parallel processing was published in 1967 by Gene Amdahl of IBM, which observed the rate of speedup resulting from clustering computers, known as Amdahl's law. The first commercial computer clustering product was the ARCnet networking system, developed in 1977 by Datapoint. The Beowulf cluster, a computer cluster typically made of low-cost computers networked into a local network, was invented in 1995 Pfister (1998) Merkey (2023).

5. What is SLURM?

SLURM is a popular open-source job scheduler for Linux computer clusters. It's used on over half of the world's top 500 largest supercomputers Top500 (2023). It provides an interface for users to schedule, execute, and monitor jobs while managing access to computing resources and load balancing according to administrative policy and computational demand. Since it's open-source, SLURM is customizable. Publicly available plugins include Accounting Storage, which stores additional data about jobs, and Network Topology, which optimizes resource allocation based on network topology SchedMD (2021).

SLURM's most basic architecture involves two daemons: slurmctld and slurmd. A daemon is a software program that runs as a background process; it is not in the direct control of any user but constantly runs Fox (1997). The slurmctld daemon monitors resources and queued work, while slurmd daemons await jobs and perform them when requested. Users commands include srun which initiates jobs, scancel which cancels jobs, sinfo which displays system status information, squeue which displays job status information, and scontrol which modifies exising configurations SchedMD (2021).

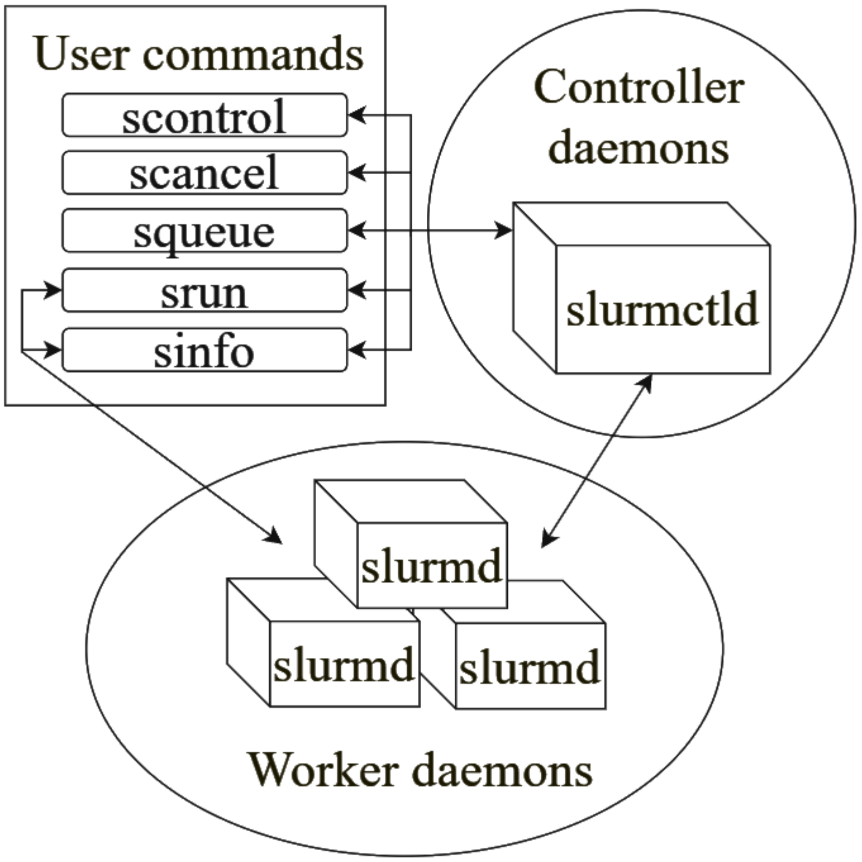


Figure 1: *Basic SLURM architecture*

5.1. SSH

SSH (Secure Shell Protocol) is a protocol for private data transfer between computers in a network. It uses public/private key cryptography, where computers hold both a public key which is shared across the network, and a private key which should be kept private. If computer A wants to send computer B a message, it encrypts the message with computer B's public key before sending it. Computer B can then use its private key to decrypt and read the message. Other computers on the network may be able to intercept the message, but they won't be able to decrypt and read it without computer B's private key Barrett & Silverman (2001) Ylonen & Lonvick (2006).

OpenSSH is an open-source implementation of the Secure Shell (SSH) protocol, and it's used by default by SLURM to communicate between controller and worker computers. Once SLURM initializes a job, its message passing interface (MPI) sends the job to worker computers and collect results through OpenSSH SchedMD (2021) OpenBSD (2023).

5.2. MUNGE

SLURM-specific communication is enabled by a authentication service called MUNGE, which allows clients to create and validate time-sensitive credentials for jobs across a network. MUNGE credentials are encrypted and decrypted with identical private keys located on each computer. The MUNGE authentication process is enabled by communication through SSH, but it is a separate authentication process specific to SLURM jobs. Without the correct private keys on all computers in a SLURM cluster, jobs can't be properly processed Dunlap (2022).

You can read more about SLURM at https://slurm.schedmd.com/overview.html.

6. How to build a SLURM computer cluster

A simple computer cluster may consist of one controller computer and several worker computers. The controller manages and sends jobs to the workers, who complete the jobs and send their output back to the controller. It's simplest for all worker computers to be the same exact model, although its possible with additional configuration to build computer clusters with non-identical worker computers. The controller computer need not be identical to the worker computers. To control multiple computers with a single keyboard, video source (monitor), and mouse you can use a KVM switch.

Once you have computers, a general process for setting up a simple cluster with SLURM is:

1. Configure a private network with DHCP and assign static IP addresses to each computer in your cluster.
2. Install the latest version of Ubuntu on each computer.
3. Setup SSH on each computer.
4. Setup MUNGE on each computer.
5. Setup SLURM on each computer.

A detailed guide made with Sergio Martelo (sergio-martelo@uiowa.edu) on how to set up a SLURM cluster can be found at https://github.com/SergioMEV/slurm-for-dummies.

7. Monte Carlo option pricing with SLURM and Python

7.1. General script format

We can implement a Monte Carlo option pricing model on a SLURM cluster by writing two Python scripts: one for the controller computer, and one for each worker computer.

The controller computer's script initializes the Monte Carlo option pricing simulation, launches a SLURM job in a command line instance using the subproccess library, collects outputs returned from worker computers, then completes any final valuation calculations.

The worker computers' script simulates a number of asset paths and prints the average option payoff value scaled by the number of total simulations to be ran.

The controller and worker computers should have their respective Python scripts copied to their /home directory. Python, along with external libraries (if used) will need to be installed. Once this is done and your SLURM cluster is set up, you can run the controller script in the /home directory of the controller computer to price the option:

```
$ python3 <mc_option_controller >.py
```

7.2. *Example scripts*

Appendix A.2. contains Python scripts for pricing three types of options with Monte Carlo simulation on a SLURM computer cluster. Appendix A.2.1., A.2.2., and A.2.3. are for pricing European call options, European down-and-out call options, and Asian call options with a geometric average control variate, respectively.

Appendix A.3. contains Python scripts for pricing the same three options with Monte Carlo simulation although on an independent computer, without SLURM. Appendix A.3.1., A.3.2., and A.3.3. are for pricing European call options, European down-and-out call options, and Asian call options with a geometric average control variate, respectively.

These scripts were based off those from Clewlow & Strickland (1998) and Back (2015). They are also available at https://github.com/scottgriffinm/Monte-Carlo-Option-Pricing-on-a-SLURM-Cluster.

8. Our computer cluster

8.1. *Our cluster*

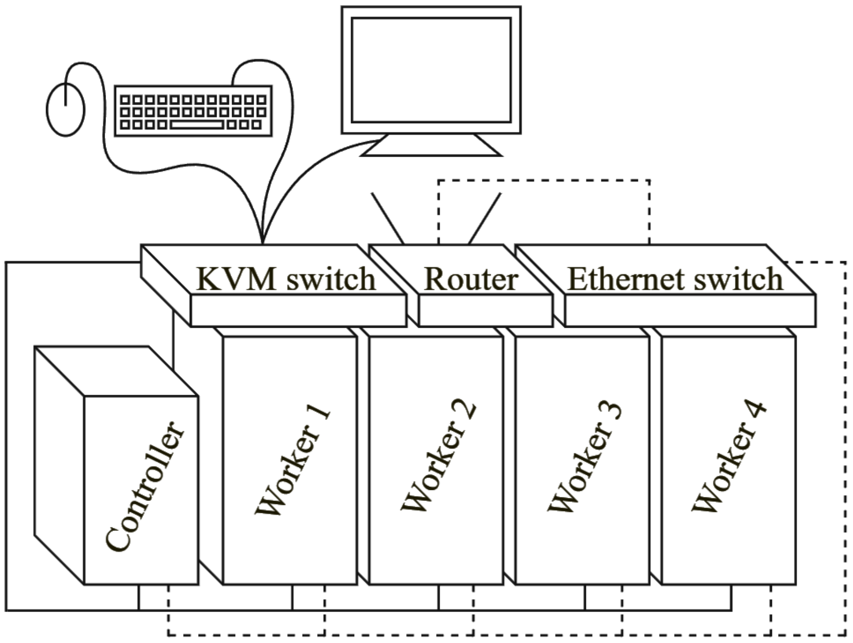


Figure 2: *Diagram of our computer cluster*

Our computer cluster is made up of one controller computer and four worker computers, all running Ubuntu 22.04.03 LTS. Each computer is independently connected to the KVM switch and to the Ethernet switch. The Ethernet switch is connected to the router. The keyboard, mouse, and monitor are connected to the KVM switch.

The controller computer is an HP Slim Desktop 290-p0056 with an 8th Generation Intel(R) Core(TM) i5-8400 processor, an Intel(R) UHD Graphics 630 graphics card, 32GB of RAM and a

1TB HDD.

All four worker computers are identical. Each is an HP Pavilion Desktop TP01-3016 with a 12th Generation Intel(R) Core(TM) I5-12400 processor, an Intel(R) UHD Graphics 730 graphics card, 12GB of RAM and a 500GB SSD.

8.2. Runtime and price error experiment

In this section, runtimes and price errors are compared from Monte Carlo European call option pricing programs ran on an independent computer, as well as on SLURM clusters of varying size. The single, independent computer is identical to one worker computer and is not controlled by a SLURM controller computer.

Out-of-the-money (OTM), at-the-money (ATM), and in-the-money (ITM) European call options are priced by simulating 100 to 100,000,000 asset paths. Ten runs were performed and averaged for each European call option (OTM, ATM, ITM), SLURM cluster size, and number of asset paths simulated. Ten runs were also averaged for each European call option and number of asset paths simulated for the single, independent computer.

The European call options have strike price = 100, risk-free rate = 5%, volatility = 20%, dividend rate = 1%, and time-to-maturity = 1. The initial assets prices for the OTM, ATM, and ITM options are 90, 100, and 110 respectively. S/K ratios for the OTM, ATM, and ITM options are then 0.9, 1, and 1.1. The options were priced using scripts in appendix A.4.1 and A.4.2., and all resulting runtimes and prices from the performance experiment are available at https://github.com/scottgriffinm/Monte-Carlo-Option-Pricing-on-a-SLURM-Cluster.

8.2.1. Runtimes

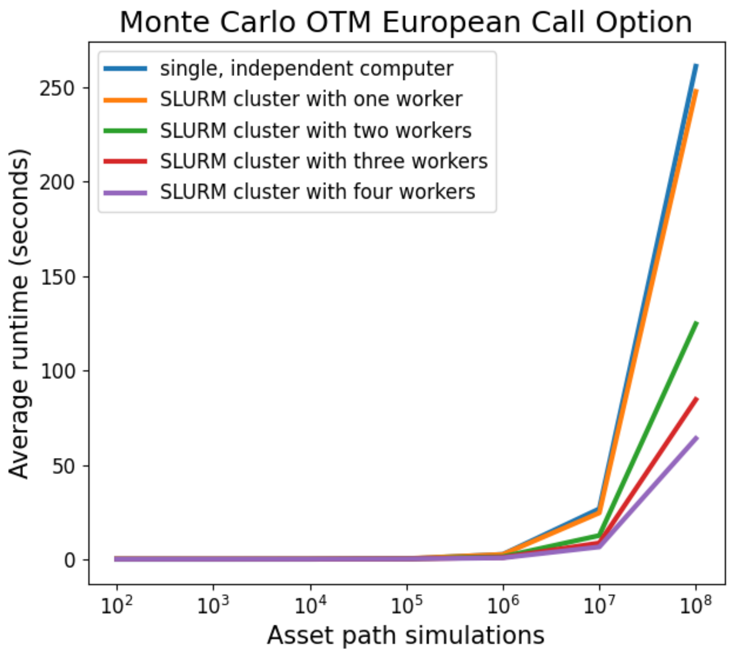


Figure 3: *Graph of average runtimes of ten Monte Carlo European call option pricing simulations with S/K ratio = 0.9*

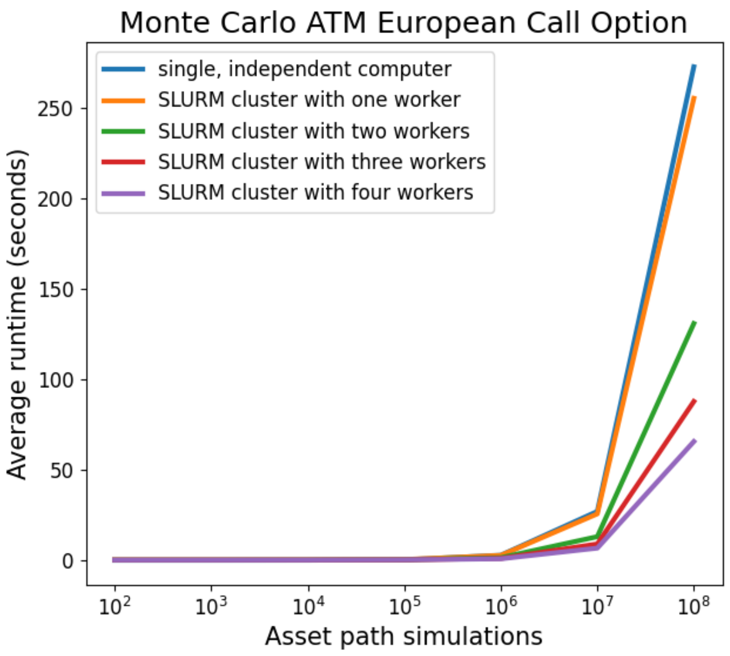


Figure 4: *Graph of average runtimes of ten Monte Carlo European call option pricing simulations with S/K ratio = 1*

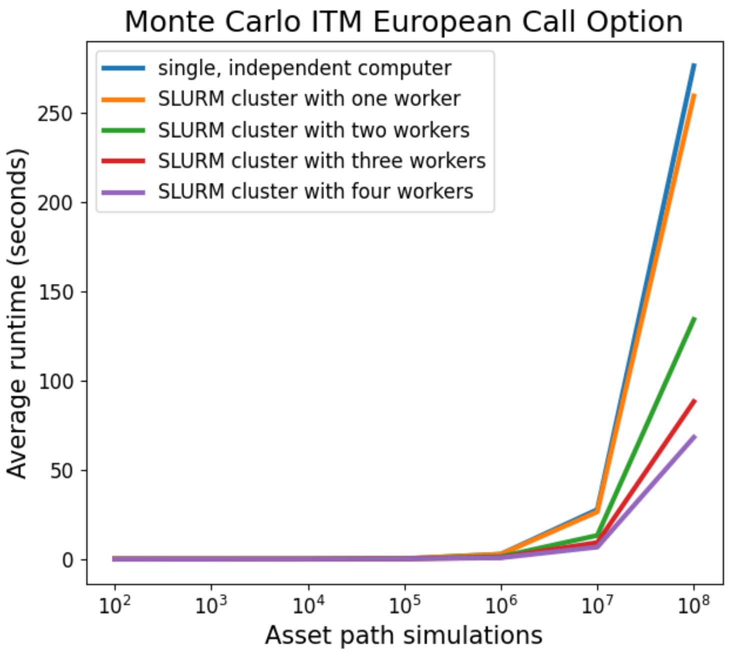


Figure 5: *Graph of average runtimes of ten Monte Carlo European call option pricing simulations with S/K ratio = 1.1*

Runtime in seconds to price a European call option with Monte Carlo (S/K ratio = .9)

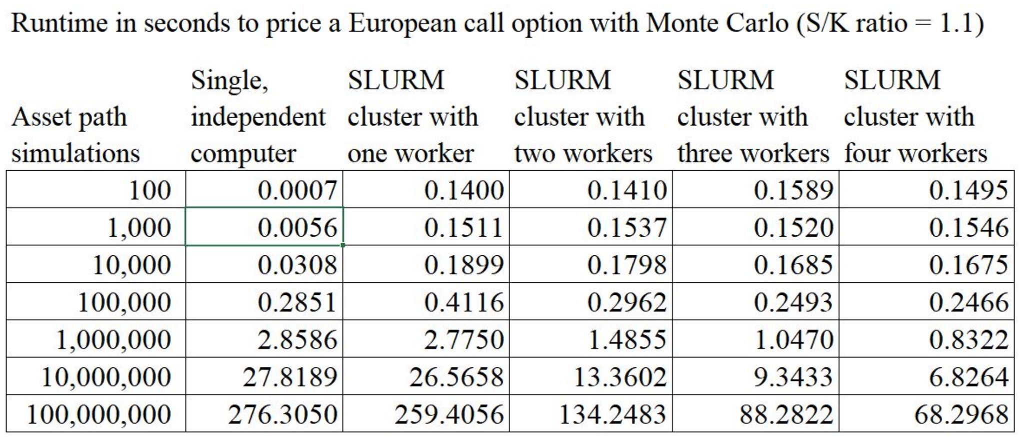
Asset path simulations	Single, independent computer	SLURM cluster with one worker	SLURM cluster with two workers	SLURM cluster with three workers	SLURM cluster with four workers
100	0.0006	0.1454	0.1533	0.1510	0.1430
1,000	0.0053	0.1461	0.1384	0.1416	0.1550
10,000	0.0290	0.1826	0.1554	0.1557	0.1570
100,000	0.2752	0.4014	0.2800	0.2505	0.2279
1,000,000	2.6302	2.5894	1.3934	0.9907	0.7889
10,000,000	26.7461	24.5962	12.6138	8.6434	6.5788
100,000,000	260.9792	247.6045	124.6273	84.5606	63.9617

Figure 6: *Table of average runtimes of ten Monte Carlo European call option pricing simulations with S/K ratio = 0.9*

Runtime in seconds to price a European call option with Monte Carlo (S/K ratio = 1)

Asset path simulations	Single, independent computer	SLURM cluster with one worker	SLURM cluster with two workers	SLURM cluster with three workers	SLURM cluster with four workers
100	0.0006	0.1410	0.1460	0.1433	0.1551
1,000	0.0053	0.1503	0.1499	0.1537	0.1526
10,000	0.0298	0.1858	0.1769	0.1703	0.1623
100,000	0.2718	0.4047	0.2911	0.2472	0.2583
1,000,000	2.7842	2.6941	1.4450	1.0276	0.8212
10,000,000	27.0182	25.6680	13.0610	9.0021	6.7243
100,000,000	272.9284	255.4756	130.9313	87.8276	65.6654

Figure 7: *Table of average runtimes of ten Monte Carlo European call option pricing simulations with S/K ratio = 1*



Runtime in seconds to price a European call option with Monte Carlo (S/K ratio = 1.1)

Asset path simulations	Single, independent computer	SLURM cluster with one worker	SLURM cluster with two workers	SLURM cluster with three workers	SLURM cluster with four workers
100	0.0007	0.1400	0.1410	0.1589	0.1495
1,000	0.0056	0.1511	0.1537	0.1520	0.1546
10,000	0.0308	0.1899	0.1798	0.1685	0.1675
100,000	0.2851	0.4116	0.2962	0.2493	0.2466
1,000,000	2.8586	2.7750	1.4855	1.0470	0.8322
10,000,000	27.8189	26.5658	13.3602	9.3433	6.8264
100,000,000	276.3050	259.4056	134.2483	88.2822	68.2968

Figure 8: *Table of average runtimes of ten Monte Carlo European call option pricing simulations with S/K ratio = 1.1*

Generally, as we added workers to the SLURM cluster runtime decreased at a decreasing rate. A log-linear regression of runtime on the number of SLURM workers estimates that the marginal decrease in runtime from adding any more than 11 workers to our cluster would be less than one second per worker[3]. To further decrease runtime we could aggregate multiple SLURM clusters with multiple controller computers, or increase the computing power of the worker computers.

The effect of SLURM's job management overhead on the computation time of simulation levels below 100,000 is noticeable. For a call option with S/K ratio = 0.9, the single, independent computer is 23,733% faster than the SLURM cluster with four workers at 100 simulations, 2,825% faster at 1,000 simulations, and 441% faster at 10,000 simulations. At asset path simulations above 100,000, all SLURM clusters priced options faster than the independent computer.

There's a potential positive relationship observed between the S/K ratio of the European call options and the runtime required to price them with Monte Carlo simulation[4]. At 100,000,000 asset path simulations, the average runtimes to price European call options with S/K ratios = 0.9, 1, and 1.1 were 156.35, 162.57, and 165.31 seconds respectively.

[3]A log-linear regression of runtime on the number of SLURM workers was conducted using the average runtimes of ten pricing programs simulating 100,000,000 asset paths to price the previously described OTM European call option. With four observations, the regression's adjusted R^2 was 0.959 with a regression standard error of 0.146. The intercept coefficient was 5.84 with a p-value of 0.0009, and the coefficient for an additional SLURM worker was -0.444 with a p-value of 0.02. A plot of residuals showed no discernible patterns.

[4]H0: E[R(.9)] == E[R(1.1)], H1: E[R(.9)] != E[R(1.1)], where R() is a function that inputs the S/K ratio of the European call option described previously, and outputs the runtime to price that option with a Monte Carlo simulation of 100,000,000 asset price paths ran on a single, independent computer. In a sample of ten runs, E[R(.9)] = 260.98, Var[R(.9)] = 0.06, E[R(1.1)] = 276.30, Var[R(1.1)] = 1.57. The t-value of an unequal variance t-test is 37.94, with nine degrees of freedom. The two-sided t-value distribution table value for nine degrees of freedom and a 95% confidence level is 2.262. The computed t-value is higher, so H0 is rejected.

8.2.2. Price errors

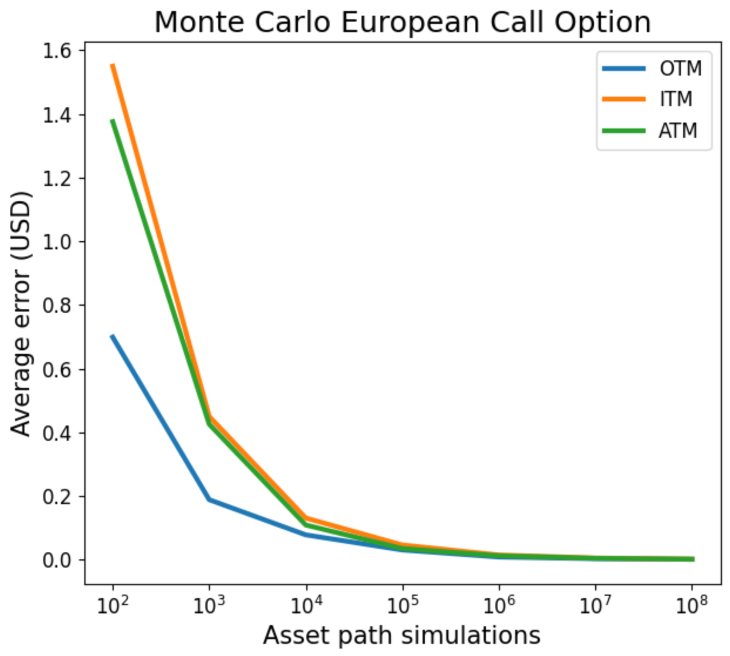
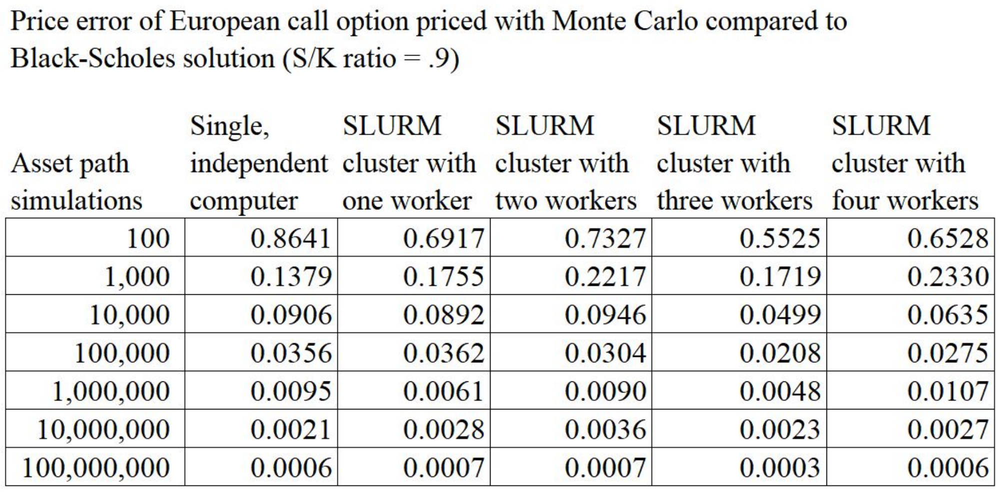


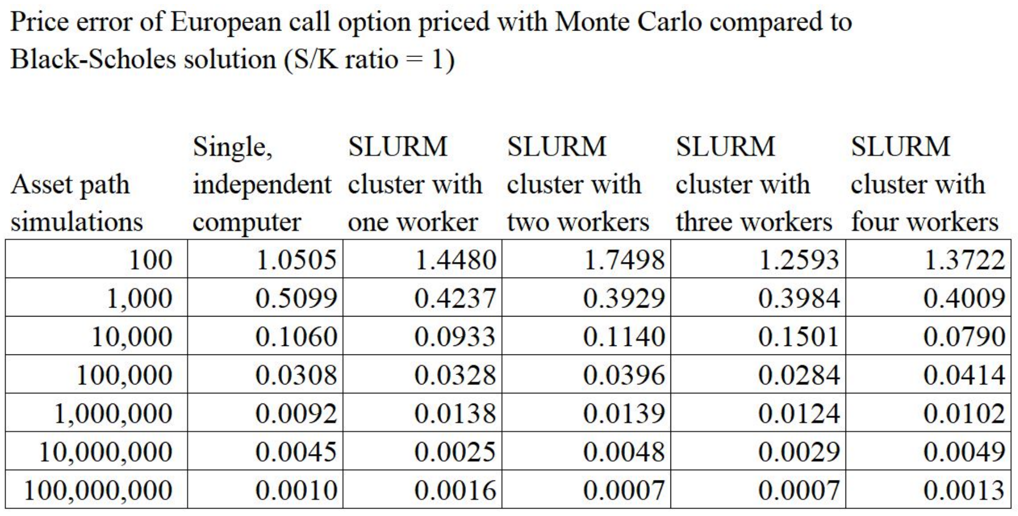
Figure 9: *Graph of average price error of fifty Monte Carlo European call option pricing simulations for varying S/K ratios*



Price error of European call option priced with Monte Carlo compared to Black-Scholes solution (S/K ratio = .9)

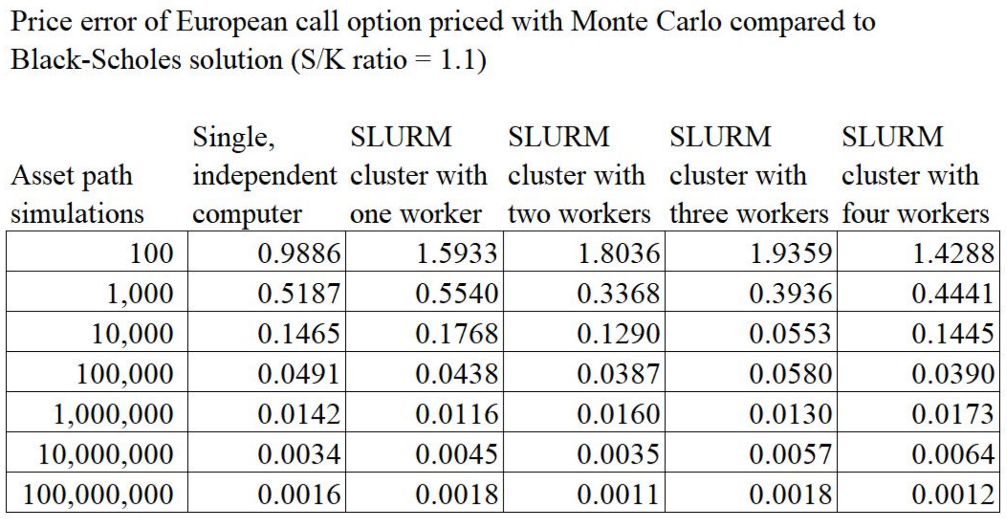
Asset path simulations	Single, independent computer	SLURM cluster with one worker	SLURM cluster with two workers	SLURM cluster with three workers	SLURM cluster with four workers
100	0.8641	0.6917	0.7327	0.5525	0.6528
1,000	0.1379	0.1755	0.2217	0.1719	0.2330
10,000	0.0906	0.0892	0.0946	0.0499	0.0635
100,000	0.0356	0.0362	0.0304	0.0208	0.0275
1,000,000	0.0095	0.0061	0.0090	0.0048	0.0107
10,000,000	0.0021	0.0028	0.0036	0.0023	0.0027
100,000,000	0.0006	0.0007	0.0007	0.0003	0.0006

Figure 10: *Table of average price error of ten Monte Carlo European call option pricing simulations with S/K ratio = 0.9*


Price error of European call option priced with Monte Carlo compared to Black-Scholes solution (S/K ratio = 1)

Asset path simulations	Single, independent computer	SLURM cluster with one worker	SLURM cluster with two workers	SLURM cluster with three workers	SLURM cluster with four workers
100	1.0505	1.4480	1.7498	1.2593	1.3722
1,000	0.5099	0.4237	0.3929	0.3984	0.4009
10,000	0.1060	0.0933	0.1140	0.1501	0.0790
100,000	0.0308	0.0328	0.0396	0.0284	0.0414
1,000,000	0.0092	0.0138	0.0139	0.0124	0.0102
10,000,000	0.0045	0.0025	0.0048	0.0029	0.0049
100,000,000	0.0010	0.0016	0.0007	0.0007	0.0013

Figure 11: *Table of average price error of ten Monte Carlo European call option pricing simulations with S/K ratio = 1*


Price error of European call option priced with Monte Carlo compared to Black-Scholes solution (S/K ratio = 1.1)

Asset path simulations	Single, independent computer	SLURM cluster with one worker	SLURM cluster with two workers	SLURM cluster with three workers	SLURM cluster with four workers
100	0.9886	1.5933	1.8036	1.9359	1.4288
1,000	0.5187	0.5540	0.3368	0.3936	0.4441
10,000	0.1465	0.1768	0.1290	0.0553	0.1445
100,000	0.0491	0.0438	0.0387	0.0580	0.0390
1,000,000	0.0142	0.0116	0.0160	0.0130	0.0173
10,000,000	0.0034	0.0045	0.0035	0.0057	0.0064
100,000,000	0.0016	0.0018	0.0011	0.0018	0.0012

Figure 12: *Table of average price error of ten Monte Carlo European call option pricing simulations with S/K ratio = 1.1*

As the number of asset path simulations increased, price error decreased at a decreasing rate. There's no observed effect on price accuracy from varying the size of the SLURM cluster, or from not using one.

There's also a potential positive relationship observed between the S/K ratio of the European call options and their price error relative to their corresponding Black-Scholes closed-form solutions[5].

[5]H0: E[Err(.9)] == E[Err(1.1)], H1: E[Err(.9)] != E[Err(1.1)], where Err() is a function that inputs the S/K ratio of the European call option described previously and outputs the price error compared to the Black-Scholes closed-form solution of a Monte Carlo European call option pricing simulation of 100,000,000 asset price paths. In a sample of 50 runs, E[Err(.9)] = 0.0006, Var[Err(.9)] = 0.0000003, E[Err(1.1)] = 0.0015, Var[Err(1.1)] = 0.000001. The t-value of an unequal variance t-test is 5.59, with 55 degrees of freedom. The two-sided t-value distribution table value for 40 degrees of freedom and a 95% confidence level is 2.021. The computed t-value is higher, so H0 is rejected.

8.2.3. SLURM cluster with one worker ran faster than single, independent computer

Consistently observed at large simulation sizes, the SLURM cluster with one worker ran faster than a single, independent computer. This is an intuitively unexpected result, as SLURM clusters require computational overhead to communicate jobs and transfer data. One could easily hypothesize then that a computer without SLURM overhead would be able to process computations faster.

Running a Python script that calculates the average of 100,000,000 random numbers on the independent computer and on the SLURM cluster with one worker, the independent computer ran the script slightly slower (22.3949 seconds) compared to the SLURM cluster with one worker (22.1204 seconds). This result was supported by a subsequent experiment calculating the average of 1,000,000,000 random numbers, where the independent computer again ran the script slightly slower (217.5624 seconds) than the SLURM cluster with one worker (216.8136 seconds).

Further research into SLURM specifications and processes is needed to determine reasoning for this result, although its impact on the implementation of a large SLURM cluster seems insignificant.

9. Conclusion

A guide was developed for setting up a simple SLURM HPC cluster, and a variety of Monte Carlo option pricing models were implemented on our SLURM HPC cluster with Python. A Python script template for Monte Carlo option pricing on a SLURM cluster was introduced, and tested scripts for European call, European down-and-out call, and Asian call options with a geometric control variate are provided. It was observed that above a number of simulations, adding worker computers to a SLURM cluster increases its processing speed at a decreasing rate. A single worker computer controlled by a SLURM controller computer was found to consistently process computations faster than a single, independent computer. A potential positive relationship was found between the S/K ratio of a European call option and the runtime required to price it with Monte Carlo simulation, as well as between the S/K ratio and price error relative to the corresponding Black-Scholes closed-form solution.

A. Appendix

A.1. ***Additional figures***

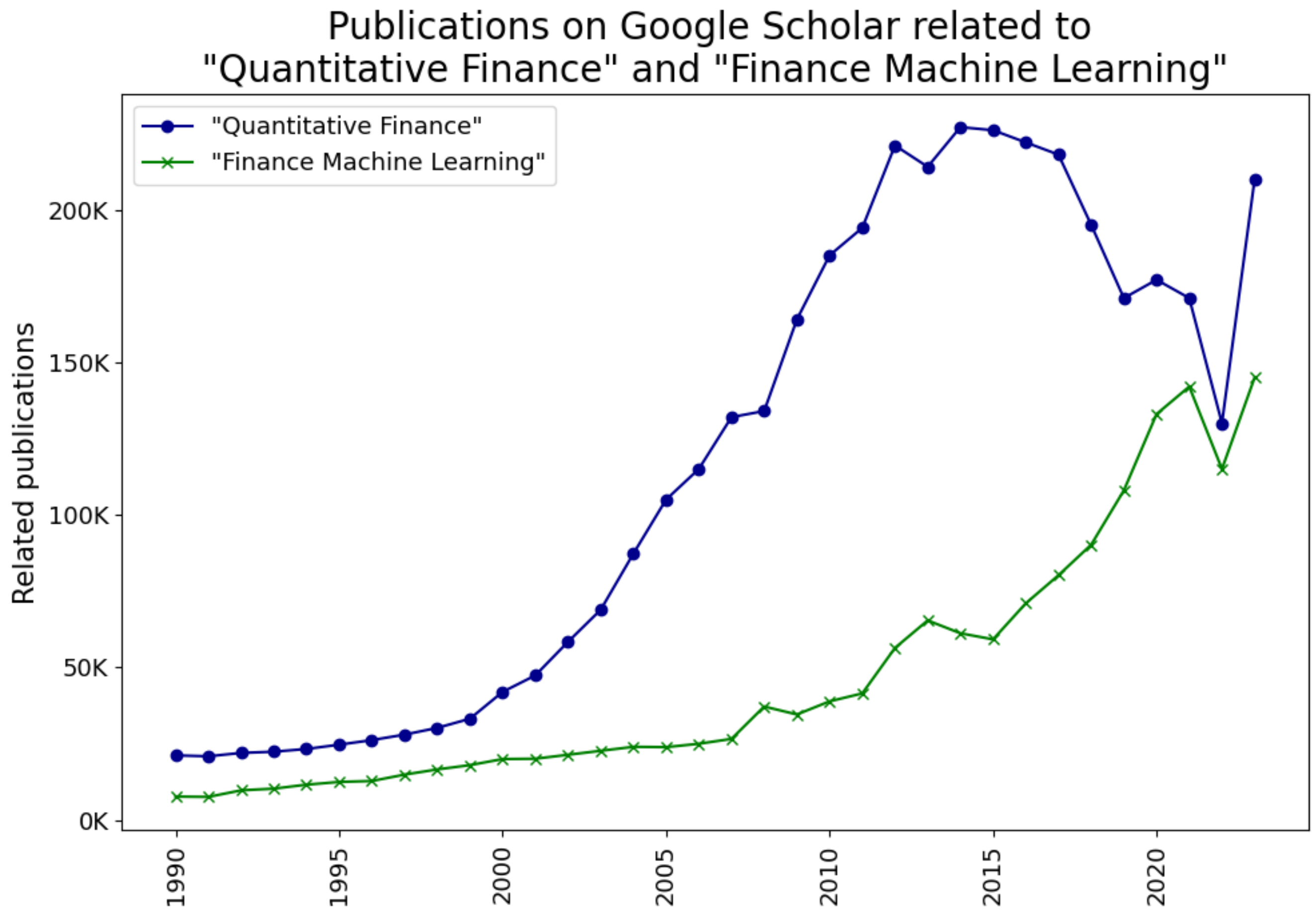


Figure 13: *Publications on Google Scholar related to ”Quantitative Finance” and ”Finance Machine Learning”*

A.2. *Python scripts for pricing options with Monte Carlo simulation on a SLURM computer cluster*

A.2.1. *European call*

`mc_euro_call_controller.py`

```
import numpy as np
import subprocess

'''Controller computer script for pricing a European call option using
Monte Carlo simulation. This script should be ran in the /home directory
of the SLURM controller computer.'''

def mc_euro_call_controller(S, K, r, sigma, q, T, total_simulations, workers):
        '''Controller computer function for pricing a European call option using
        Monte Carlo simulation.

        S: float, initial stock price
        K: float, strike price
        r: float, risk-free interest rate
        sigma: float, volatility
        q: float, dividend yield
        T: int, time to maturity
        total_simulations: int, total number of simulations
        workers: int, number of workers to employ'''
        if total_simulations % workers != 0:
                total_simulations += (workers - total_simulations % workers)
                print(f"Total number of simulations adjusted to {total_simulations}
                                    to be evenly divisible by {workers} workers.")
        worker_simulations = int(total_simulations/workers)
        # Build SLURM job command
        worker_commands = [S, K, r, sigma, q, T, worker_simulations,
                                                        total_simulations]
        command_list = ['srun', f"-N{workers}",'python3','mc_euro_call_worker.py']
        for i in range(len(worker_commands)):
                command_list.append(str(worker_commands[i]))
        # Launch SLURM job and collect results
        result = subprocess.run(command_list, capture_output=True, text=True,
                                                  check=True)
        result = result.stdout.strip()
        result = list(result.splitlines())
        for i in range(len(result)):
                result[i] = float(result[i])
        # Sum values and discount to present time
        sum_call = sum(result)
        call_value = np.exp(-r * T) * sum_call
        return call_value

if __name__ == "__main__":
        # Example usage
        S = 90
        K = 100
        r = 0.05
        sigma = 0.2
        q = 0.01
        T = 1
        total_simulations = 1_000_000
        workers = 1
        price = mc_euro_call_controller(S, K, r, sigma, q, T, total_simulations,
                                                                           workers)
        print(f"Workers = {workers}")
        print(f"Total Simulations = {total_simulations}")
        print(f"Price = {price}")
```

`mc_euro_call_worker.py`

```
import numpy as np
import sys

'''Worker computer script for pricing a European call option using Monte Carlo
simulation. This script should be located in the /home directory of all SLURM
worker computers.'''

def mc_euro_call_worker(S, K, r, sigma, q, T, worker_simulations, total_simulations):
    '''Worker computer function for pricing a European call option using Monte Carlo
    simulation.

    S: float, initial stock price
    K: float, strike price
    r: float, risk-free interest rate
    sigma: float, volatility
    q: float, dividend yield
    T: int, time to maturity
    worker_simulations: int, number of simulations to run on this worker
    total_simulations: int, total number of simulations'''
    # Precompute constants
    drift = (r - q - 0.5 * sigma**2) * T
    sig_sqrt_t = sigma * np.sqrt(T)
    up_change = np.log(1.01)
    down_change = np.log(0.99)
    sum_call = 0
    sum_call_change = 0
    sum_pathwise = 0
    random_numbers = np.random.randn(worker_simulations)
    # Simulate asset paths
    for i in range(worker_simulations):
        log_st = np.log(S) + drift + sig_sqrt_t * random_numbers[i]
        call_val = max(0, np.exp(log_st) - K)
        sum_call += call_val
        log_su = log_st + up_change
        call_vu = max(0, np.exp(log_su) - K)
        log_sd = log_st + down_change
        call_vd = max(0, np.exp(log_sd) - K)
        sum_call_change += call_vu - call_vd
        if np.exp(log_st) > K:
            sum_pathwise += (np.exp(log_st) / S)
    # Average payoffs by total simulations
    return sum_call/total_simulations
```

```
if __name__ == "__main__":
    # Collect arguments from SLURM job command
    S = float(sys.argv[1])
    K = float(sys.argv[2])
    r = float(sys.argv[3])
    sigma = float(sys.argv[4])
    q = float(sys.argv[5])
    T = int(sys.argv[6])
    worker_simulations = int(sys.argv[7])
    total_simulations = int(sys.argv[8])
    # Return partial average payoff to controller computer
    print(mc_euro_call_worker(S, K, r, sigma, q, T, worker_simulations,
                                             total_simulations))
```

```
Price with current inputs = 4.7129
```

A.2.2. ***European down-and-out call***

`mc_euro_down_and_out_call_controller.py`

```
import numpy as np
import subprocess

'''Controller computer script for pricing a European down-and-out call option using
Monte Carlo simulation. This script should be ran in the /home directory of the
SLURM controller computer.'''

def mc_euro_down_and_out_call_controller(S, K, r, sigma, q, T, H, N,
                                                    total_simulations, workers):
        '''Controller computer function for pricing a European down-and-out
        call option using Monte Carlo simulation.

        S: float, initial stock price
        K: float, strike price
        r: float, risk-free interest rate
        sigma: float, volatility
        q: float, dividend yield
        T: int, time to maturity
        H: float, barrier
        N: int, number of monitoring points
        total_simulations: int, total number of simulations
        workers: int, number of workers to employ'''
        if total_simulations % workers != 0:
                total_simulations += (workers - total_simulations % workers)
                print(f"Total number of simulations adjusted to {total_simulations}
                                    to be evenly divisible by {workers} workers.")
        worker_simulations = int(total_simulations/workers)
        # Build SLURM job command
        worker_commands = [S, K, r, sigma, q, T, H, N, worker_simulations,
                                                    total_simulations]
        command_list = ['srun', f"-N{workers}",'python3',
                                    'mc_euro_down_and_out_call_worker.py']
        for i in range(len(worker_commands)):
                command_list.append(str(worker_commands[i]))
        # Launch SLURM job and collect results
        result = subprocess.run(command_list, capture_output=True, text=True,
                                                            check=True)
        result = result.stdout.strip()
        result = list(result.splitlines())
        for i in range(len(result)):
                result[i] = float(result[i])
        # Sum values and discount to present time
        sum_call = sum(result)
        call_value = np.exp(-r * T) * sum_call
        return call_value

if __name__ == "__main__":
        # Example usage
        S = 100
        K = 100
        T = 1
        sigma = 0.2
        r = 0.06
        q = 0.03
        H = 99
        N = 10
        total_simulations = 1_000_000
        workers = 1
        price = mc_euro_down_and_out_call_controller(S, K, r, sigma, q, T, H, N,
                                                total_simulations, workers)
        print(f"Workers = {workers}")
        print(f"Total Simulations = {total_simulations}")
        print(f"Price = {price}")
```

`mc_euro_down_and_out_call_worker.py`

```
import numpy as np
import sys

'''Worker computer script for pricing a European down-and-out call option using
Monte Carlo simulation. This script should be located in the /home directory of
all SLURM worker computers.'''

def mc_euro_down_and_out_call_worker(S, K, r, sigma, q, T, H, N, worker_simulations,
                                                            total_simulations):
    '''Worker computer function for pricing a European down-and-out call option
    using Monte Carlo simulation.

    S: float, initial stock price
    K: float, strike price
    r: float, risk-free interest rate
    sigma: float, volatility
    q: float, dividend yield
    T: int, time to maturity
    worker_simulations: int, number of simulations to run on this worker
    total_simulations: int, total number of simulations'''
        # Precompute constants
        dt = T/N
        nudt = (r - q - 0.5*sigma*sigma)*dt
        sigsdt = sigma * np.sqrt(dt)
        sum_CT = 0
        # Simulate asset paths
        for i in range(worker_simulations):
                St = S
                BARRIER_CROSSED = False
                for j in range(N):
                        e = np.random.randn()
                        St = St * np.exp(nudt + sigsdt*e)
                        if St <= H:
                                BARRIER_CROSSED = True
                                break
                if BARRIER_CROSSED:
                        CT = 0
                else:
                        CT = np.maximum(0, St - K)
                sum_CT += CT
        # Average payoffs by total simulations
        return sum_CT/total_simulations

if __name__ == "__main__":
        # Collect arguments from SLURM job command
        S = float(sys.argv[1])
        K = float(sys.argv[2])
        r = float(sys.argv[3])
        sigma = float(sys.argv[4])
        q = float(sys.argv[5])
        T = int(sys.argv[6])
        H = float(sys.argv[7])
        N = int(sys.argv[8])
        worker_simulations = int(sys.argv[9])
        total_simulations = int(sys.argv[10])
        # Return partial average payoff to controller computer
        print(mc_euro_down_and_out_call_worker(S, K, r, sigma, q, T, H, N,
                                        worker_simulations, total_simulations))
```

```
Price with current inputs = 5.0486
```

A.2.3. ***Asian call with geometric control variate***

`mc_asian_call_control_variate_controller.py`

```
import numpy as np
import subprocess
from scipy.stats import norm

'''Controller computer script for pricing an Asian call option using Monte Carlo
simulation with a geometric control variate. This script should be ran in the
/home directory of the SLURM controller computer.'''

def black_scholes_euro_call(S, K, r, sigma, q, T):
        '''Prices a European call option using the Black-Scholes formula.

        S: float, initial stock price
        K: float, strike price
        r: float, risk-free interest rate
        sigma: float, volatility
        q: float, dividend yield
        T: int, time to maturity'''
        d1 = (np.log(S/K) + (r - q + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
        d2 = d1 - sigma * np.sqrt(T)
        call_value = S * np.exp(-q * T) * norm.cdf(d1) - K * np.exp(-r * T)
                                                            * norm.cdf(d2)
        return call_value

def geometric_asian_call(S, K, sigma, r, q, T, N):
        '''Prices a geometric Asian call option using the Black-Scholes formula.

        S: float, initial stock price
        K: float, strike price
        sigma: float, volatility
        r: float, risk-free interest rate
        q: float, dividend yield
        T: int, time to maturity
        N: int, number of monitoring points'''
        dt = T/N
        nu = r - q - 0.5 * sigma * sigma
        a = N * (N + 1) * (2 * N + 1) / 6
        V = np.exp(-r*T)*S*np.exp(((N+1)*nu/2 + sigma*sigma*a/(2*N*N))*dt)
        sigavg = sigma * np.sqrt(a) / (N**1.5)
        call_value = black_scholes_euro_call(V, K, r, sigavg, 0, T)
        return call_value

def mc_asian_call_control_variate_controller(S, K, r, sigma, q, T,
                                                total_simulations, workers):
        '''Controller computer function for pricing an Asian call option
        using Monte Carlo simulation with a geometric control variate.

        S: float, initial stock price
        K: float, strike price
        r: float, risk-free interest rate
        sigma: float, volatility
        q: float, dividend yield
        T: int, time to maturity
        total_simulations: int, total number of simulations
        workers: int, number of workers to employ'''
        if total_simulations % workers != 0:
                total_simulations += (workers - total_simulations % workers)
                print(f"Total number of simulations adjusted to {total_simulations}
                                to be evenly divisible by {workers} workers.")
        worker_simulations = int(total_simulations/workers)
        # Build SLURM job command
        worker_commands = [S, K, sigma, r, q, T, N, worker_simulations,
                                                            total_simulations]
        command_list = ['srun', f"-N{workers}",'python3',
                                        'mc_asian_call_control_variate_worker.py']
        for i in range(len(worker_commands)):
                command_list.append(str(worker_commands[i]))
        # Launch SLURM job and collect results
        result = subprocess.run(command_list, capture_output=True, text=True,
                                                                check=True)
        result = result.stdout.strip()
        result = list(result.splitlines())
        for i in range(len(result)):
                result[i] = float(result[i])
        # Sum values and discount to present time
        portfolio_value = sum(result) * np.exp(-r * T)
        # Add control variate
        call_price = portfolio_value + geometric_asian_call(S, K, sigma, r, q, T, N)
        return call_price

if __name__ == "__main__":
        # Example usage
        K = 100
        T = 1
        S = 100
        sigma = 0.2
        r = 0.06
        q = 0.03
        N = 10
        total_simulations = 1_000_000
        workers = 1
        price = mc_asian_call_control_variate_controller(S, K, r, sigma, q, T,
                                                        total_simulations, workers)
        print(f"Workers = {workers}")
        print(f"Total Simulations = {total_simulations}")
        print(f"Price = {price}")
```

mc_asian_call_control_variate_worker.py

```
import numpy as np
import sys

'''Worker computer script for pricing an Asian call option with a geometric control
variate using Monte Carlo simulation. This script should be located in the /home
directory of all SLURM worker computers.'''

def mc_asian_call_control_variate_worker(S, K, sigma, r, q, T, N,
                                        worker_simulations, total_simulations):
        '''Worker computer function for pricing an Asian call option using
        Monte Carlo simulation with a geometric control variate.

        S: float, initial stock price
        K: float, strike price
        r: float, risk-free interest rate
        sigma: float, volatility
        q: float, dividend yield
        T: int, time to maturity
        N: int, number of monitoring points
        worker_simulations: int, number of simulations to run on this worker
        total_simulations: int, total number of simulations
        '''
        # Precompute constants
        dt = T/N
        nudt = (r - q - 0.5 * sigma * sigma) * dt
        sigsdt = sigma * np.sqrt(dt)
        sum_CT = 0
        normal_samples = np.random.normal(size=N)
        # Simulate asset paths
        for i in range(worker_simulations):
                St = S
                sumSt = 0
                productSt = 1
                for j in range(N):
                        e = normal_samples[j]
                        St = St * np.exp(nudt + sigsdt * e)
                        sumSt += St
                        productSt = productSt * St
                A = sumSt / N
                G = productSt ** (1 / N)
                CT = max(A - K, 0) - max(G - K, 0)
                sum_CT = sum_CT + CT
        # Average payoffs by total simulations
        return sum_CT / total_simulations

if __name__ == "__main__":
        # Collect arguments from SLURM job command
    S = float(sys.argv[1])
    K = float(sys.argv[2])
    r = float(sys.argv[3])
    sigma = float(sys.argv[4])
    q = float(sys.argv[5])
    T = int(sys.argv[6])
    N = int(sys.argv[7])
    worker_simulations = int(sys.argv[8])
    total_simulations = int(sys.argv[9])
        # Return partial average payoff to controller computer
    print(mc_asian_call_control_variate_worker(S, K, sigma, r, q, T, N,
                                    worker_simulations, total_simulations))
```

```
Price with current inputs = 5.5867
```

A.3. *Python scripts for pricing options with Monte Carlo simulation on a single, independent computer*

A.3.1. *European call*

`mc_euro_call_no_slurm.py`

```
from time import time
import numpy as np

'''Single, independent computer script for pricing a European call option
with Monte Carlo simulation.'''

def mc_euro_call(S, K, r, sigma, q, T, total_simulations):
        '''Prices a European call option using Monte Carlo simulation.

        S: float, initial stock price
        K: float, strike price
        r: float, risk-free rate
        sigma: float, volatility
        q: float, dividend yield
        T: int, time to maturity in years
        total_simulations: int, number of simulations'''
        # Precompute constants
        sum_call = 0
        drift = (r - q - 0.5 * sigma**2) * T
        sig_sqrt_t = sigma * np.sqrt(T)
        up_change = np.log(1.01)
        down_change = np.log(0.99)
        sum_call = 0
        sum_call_change = 0
        sum_pathwise = 0
        random_numbers = np.random.randn(total_simulations)
        # Simulate asset paths
        for i in range(total_simulations):
                log_st = np.log(S) + drift + sig_sqrt_t * random_numbers[i]
                call_val = max(0, np.exp(log_st) - K)
                sum_call += call_val
                log_su = log_st + up_change
                call_vu = max(0, np.exp(log_su) - K)
                log_sd = log_st + down_change
                call_vd = max(0, np.exp(log_sd) - K)
                sum_call_change += call_vu - call_vd
                if np.exp(log_st) > K:
                        sum_pathwise += (np.exp(log_st) / S)
        # Discount average call value to present time
        call_value = np.exp(-r * T) * sum_call/total_simulations
        return call_value

if __name__ == "__main__":
    # Example usage
    S = 90
    K = 100
    r = 0.05
    sigma = 0.2
    q = 0.01
    T = 1
    M = 1_000_000
    price = mc_euro_call(S, K, r, sigma, q, T, M)
    print(f"Simulations = {M}")
    print(f"Price = {price}")
```

```
Price with current inputs = 4.7129
```

A.3.2. ***European down-and-out call***

`mc_euro_down_and_out_call_no_slurm.py`

```
import numpy as np

'''Single, independent computer script for pricing a European down-and-out
call option using Monte Carlo simulation.'''

def mc_euro_down_and_out_call(S, K, r, sigma, q, T, H, N, total_simulations):
        '''Prices a European down-and-out call option using Monte Carlo
        simulation.

        S: float, initial stock price
        K: float, strike price
        r: float, risk-free interest rate
        sigma: float, volatility
        q: float, dividend yield
        T: int, time to maturity
        H: float, barrier price
        N: int, number of monitoring points
        total_simulations: int, total number of simulations'''
        # Precompute constants
        dt = T/N
        nudt = (r - q - 0.5*sigma*sigma)*dt
        sigsdt = sigma * np.sqrt(dt)
        sum_CT = 0
        # Simulate asset paths
        for i in range(total_simulations):
                St = S
                BARRIER_CROSSED = False
                for j in range(N):
                        e = np.random.randn()
                        St = St * np.exp(nudt + sigsdt*e)
                        if St <= H:
                                BARRIER_CROSSED = True
                                break
                if BARRIER_CROSSED:
                        CT = 0
                else:
                        CT = np.maximum(0, St - K)
                sum_CT += CT
        # Average call value and discount to present time
        call_value = sum_CT/total_simulations*np.exp(-r*T)
        return call_value

if __name__ == "__main__":
        # Example usage
        K = 100
        T = 1
        S = 100
        sigma = 0.2
        r = 0.06
        q = 0.03
        H = 99
        N = 10
        total_simulations = 1_000_000
        print(f"Simulations = {total_simulations}")
        print("Price = ", mc_euro_down_and_out_call(S, K, r, sigma, q, T, H, N,
                                                     total_simulations))
```

```
Price with current inputs = 5.0486
```

A.3.3. ***Asian call with geometric control variate***

`mc_asian_call_control_variate_no_slurm.py`

```
import numpy as np
from scipy.stats import norm

'''Single, independent computer script for pricing an Asian call option using
Monte Carlo simulation with a geometric control variate.'''

def black_scholes_euro_call(S, K, r, sigma, q, T):
        '''Prices a European call option using the Black-Scholes formula.

        S: float, initial stock price
        K: float, strike price
        r: float, risk-free interest rate
        sigma: float, volatility
        q: float, dividend yield
        T: int, time to maturity'''
        d1 = (np.log(S/K) + (r - q + 0.5 * sigma**2) * T) / (sigma * np.sqrt(T))
        d2 = d1 - sigma * np.sqrt(T)
        call_value = S * np.exp(-q * T) * norm.cdf(d1) - K * np.exp(-r * T)
                                                           * norm.cdf(d2)
        return call_value

def geometric_asian_call(S, K, sigma, r, q, T, N):
        '''Prices a geometric Asian call option using the Black-Scholes formula.

        S: float, initial stock price
        K: float, strike price
        sigma: float, volatility
        r: float, risk-free interest rate
        q: float, dividend yield
        T: int, time to maturity
        N: int, number of monitoring points'''
        dt = T/N
        nu = r - q - 0.5 * sigma * sigma
        a = N * (N + 1) * (2 * N + 1) / 6
        V = np.exp(-r*T)*S*np.exp(((N+1)*nu/2 + sigma*sigma*a/(2*N*N))*dt)
        sigavg = sigma * np.sqrt(a) / (N**1.5)
        call_value = black_scholes_euro_call(V, K, r, sigavg, 0, T)
        return call_value

def mc_asian_call_control_variate(S, K, r, sigma, q, T, N, total_simulations):
        '''Prices an Asian call option using Monte Carlo simulation with a
        geometric control variate.

        S: float, initial stock price
        K: float, strike price
        r: float, risk-free interest rate
        sigma: float, volatility
        q: float, dividend yield
        T: int, time to maturity
        N: int, number of monitoring points
        total_simulations: int, total number of simulations'''
        # Precompute constants
        dt = T/N
        nudt = (r - q - 0.5 * sigma * sigma) * dt
        sigsdt = sigma * np.sqrt(dt)
        sum_CT = 0
        normal_samples = np.random.normal(size=N)
        # Simulate asset paths
        for i in range(total_simulations):
                St = S
                sumSt = 0
                productSt = 1
                for j in range(N):
                        e = normal_samples[j]
                        St = St * np.exp(nudt + sigsdt * e)
                        sumSt += St
                        productSt = productSt * St
                A = sumSt / N
                G = productSt ** (1 / N)
                CT = max(A - K, 0) - max(G - K, 0)
                sum_CT = sum_CT + CT
        # Average payoffs and discount to present time
        portfolio_value = sum_CT / total_simulations * np.exp(-r * T)
        # Add control variate
        call_value = portfolio_value + geometric_asian_call(S, K, r, sigma, q, T, N)
        return call_value

if __name__ == "__main__":
        # Example usage
        K = 100
        T = 1
        S = 100
        sigma = 0.2
```

```
    r = 0.06
    q = 0.03
    N = 10
    total_simulations = 1_000_000
    print(f"Total Simulations = {total_simulations}")
    print("Price = ", mc_asian_call_control_variate(S, K, r, sigma, q, T, N,
                                                    total_simulations))
```

```
Price with current inputs = 5.5867
```

*A.4. **Runtime and price error experiment results***

*A.4.1. **Python scripts ran on SLURM clusters***

`mc_euro_call_slurm_runtimes_prices.py`

```
import numpy as np
from time import time
from statistics import mean
import mc_euro_call_controller

'''Controller computer script for pricing a number of European call options with
Monte Carlo for a number of worker computers and asset path simulations.
This script should be ran in the /home directory of the SLURM controller
computer.'''

if __name__ == "__main__":
        # Option Parameters
        S = 100
        K = 100
        r = 0.05
        sigma = 0.2
        q = 0.01
        T = 1
        # Experiment parameters
        total_simulations = 100000000
        workers = 4
        bs_price = 16.79983686 # Black-Scholes price to compare to
        runs = 10
        average_runtime = 0
        runtimes = []
        prices = []
        # Run runtime and pricing experiment
        for i in range(runs):
                print(f"run {i}/{runs}")
                start = time()
                price = mc_euro_call_controller(S, K, r, sigma, q, T,
                                                    total_simulations, workers)
                end = time()
                runtime = end-start
                runtimes.append(runtime)
                prices.append(price)
                average_runtime += runtime/runs
                print(f"\nworkers = {workers}")
                print(f"sims = {total_simulations}")
                print(f"price = {price}")
                print(f"time = {round(runtime,6)} seconds")
                print(f"running_average = {round((average_runtime*runs)/(i+1),6)}")
        # Print runtimes and prices
        average_runtime = round(mean(runtimes),5)
        average_error = float(round(abs(mean(prices)-bs_price),5))
        print("---------------------------------------------")
        print(f"\nworkers = {workers}")
        print(f"sims = {total_simulations}")
        print(f"\nRUNTIMES")
        for i in range(len(runtimes)):
                print(f"run {i}, {runtimes[i]} seconds.")
        print(f"\nPRICES")
        for i in range(len(prices)):
                print(f"run {i}, {prices[i]}")
        print(f"\nAVERAGE RUNTIME = {average_runtime}")
        print(f"AVERAGE ERROR = {average_error}")
```

`simple_slurm_speed_test_controller.py`

```
import subprocess
from time import time

'''Controller computer script for calculating the average of a number of random
numbers. This script should be run in the \home directory of the SLURM
controller computer.'''

if __name__ == "__main__":
        start = time()  # Start timer
        runs = 1_000_000_000
        # Build command list
        command_list = ['srun','python3','simple_slurm_speed_test_partial.py',
                                                                    f"{runs}"]
        # Launch SLURM job and collect results
        result = subprocess.run(command_list, capture_output=True, text=True,
                                                                    check=True)
        result = result.stdout.strip()
        end = time()    # End timer
        # Print results
        print(f"random_avg = {result}")
        print(f"time = {round(end-start,6)} seconds")
```

`simple_slurm_speed_test_worker.py`

```
import numpy as np
import sys

'''Worker computer script for calculating the average of a number of random
numbers. This script should be located in the \home directory of all SLURM
worker computers.'''

if __name__ == "__main__":
        # Collect runs argument from SLURM job command
        runs = int(sys.argv[1])
        random_avg = 0
        for i in range(runs): # Calculate average of random numbers
                random_avg += np.random.randn()/runs
        print(random_avg) # Return to controller computer
```

A.4.2. Python scripts ran on a single, independent computer

`mc_euro_call_no_slurm_runtimes_prices.py`

```
from time import time
import numpy as np
from statistics import mean
import mc_euro_call_no_slurm

'''Single, independent computer script for pricing a number of European
call options with Monte Carlo for a number of worker computers and
asset path simulations.'''

if __name__ == "__main__":
        # Option parameters
        S = 110
        K = 100
        r = 0.05
        sigma = 0.2
        q = 0.01
        T = 1
        # Experiment parameters
        total_simulations = 100000000
        bs_price = 16.79983686 # Black-Scholes price to compare to
        runs = 10
        average_runtime = 0
        runtimes = []
        prices = []
        # Run runtime and pricing experiment
        for i in range(runs):
                print(f"\nrun {i}/{runs}")
                start = time()
                price = mc_euro_call_no_slurm(S, K, r, sigma, q, T,
                                                        total_simulations)
                end = time()
                runtime = end-start
                runtimes.append(runtime)
                prices.append(price)
                average_runtime += runtime/runs
                print(f"sims = {total_simulations}")
                print(f"price = {price}")
                print(f"time = {round(runtime,6)} seconds")
                print(f"running_average = {round((average_runtime*runs)/(i+1),6)}")
        # Print runtimes and prices
        average_runtime = round(mean(runtimes),5)
        average_error = float(round(abs(mean(prices)-bs_price),5))
        print("--------------------------------------------------")
        print(f"\nsingle, independent computer")
        print(f"sims = {total_simulations}")
        print(f"\nRUNTIMES")
        for i in range(len(runtimes)):
                print(f"run {i}, {runtimes[i]} seconds.")
        print(f"\nPRICES")
        for i in range(len(prices)):
                print(f"run {i}, {prices[i]}")
        print(f"\nAVERAGE RUNTIME = {average_runtime}")
        print(f"AVERAGE ERROR = {average_error}")
```

simple_slurm_speed_test_no_slurm.py

```
import numpy as np
import sys
from time import time

'''Single, independent computer script for calculating the average of a number
of random numbers.'''

if __name__ == "__main__":
        start = time() # Start timer
        iters = 1_000_000_000
        random_avg = 0
        # Calculate average of random numbers
        for i in range(iters):
                random_avg += np.random.randn()/iters
        end = time() # End timer
        # Print results
        print(f"random_avg = {random_avg}")
        print(f"time: {round(end-start,6)}")
```

References

[Aitchison & Brown (1963)] Aitchison, J. & Brown, J.M.C. (1963). *The lognormal distribution*. Cambridge University Press, Cambridge.

[Amdahl (1967)] Amdahl, Gene M. (1967). *Validity of the single processor approach to achieving large scale computing capabilities*. In Proceedings of the April 18-20, 1967, spring joint computer conference (AFIPS '67 (Spring)). Association for Computing Machinery, New York, NY, USA, 483–485. https://doi.org/10.1145/1465482.1465560.

[Back (2005)] Back, K. (2005). *A Course in Derivative Securities: Introduction to Theory and Computation*. Springer Berlin Heidelberg.

[Barrett & Silverman (2001)] Barrett, D. J., & Silverman, R. E. (2001). *SSH, the Secure Shell: The definitive guide*. O'Reilly.

[BIS (2023)] Bank for International Settlements. (2023). *Global OTC derivatives market*. https://www.bis.org/statistics/derivatives/ra.htm.

[Black & Scholes (1972)] Black, F., & Scholes, M. (1972). *The Pricing of Options and Corporate Liabilities*. https://www.journals.uchicago.edu/doi/10.1086/260062.

[Boyle (1977)] Boyle, Phelim P. (1977). *Options: A Monte Carlo approach*. Journal of Financial Economics, Elsevier, vol. 4(3), pages 323-338, May.

[Cox & Ross (1976)] Cox, J.C. & Ross, S.A. (1976). *The valuation of options for alternative stochastic processes*. Journal of Financial Economics 3, 145-166.

[Clewlow & Carverhill (1994)] Clewlow, L.J. and A. Carverhill. (1994). *On the Simulation of Contingent Claims*, Journal of Derivatives, 2, 66-74.

[Clewlow & Strickland (1998)] Clewlow, L., & Strickland, C. (1998). *Implementing Derivative Models*. Wiley.

[Dunlap (2022)] Dunlap, Chris. (2022). *MUNGE by Chris Dunlap*. https://dun.github.io/munge/.

[Fox (1997)] Fox, E. A. (1997). *The Origin of the Word Daemon*. https://ei.cs.vt.edu/ history/-Daemon.html.

[Iowa State (2023)] Iowa State University. (2023). *SLURM | High Performance Computing*. https://www.hpc.iastate.edu/guides/introduction-to-hpc-clusters/slurm.

[Laboissiere et. al. (2015)] Leonel A. Laboissiere, Ricardo A.S. Fernandes, Guilherme G. Lage. (2015). *Maximum and minimum stock price forecasting of Brazilian power distribution companies based on artificial neural networks*. Applied Soft Computing, Volume 35, 2015, Pages 66-74, ISSN 1568-4946, https://doi.org/10.1016/j.asoc.2015.06.005.

[Lee (2020)] R. S. T. Lee. (2020). *Chaotic Type-2 Transient-Fuzzy Deep Neuro-Oscillatory Network (CT2TFDNN) for Worldwide Financial Prediction*. IEEE Transactions on Fuzzy Systems vol. 28, no. 4, pp. 731-745, April 2020, doi: 10.1109/TFUZZ.2019.2914642.

[Merkey (2023)] Merkey, P. (2023). *The Beowulf Archives* . https://www.beowulf.org/pipermail/beowulf/.

[Northeastern (2023)] Northeastern University. (2023). *Introduction to HPC and Slurm - RC RTD*. https://rc-docs.northeastern.edu/en/latest/welcome/introtocluster.html.

[OpenBSD (2023)] OpenBSD Project. (2023). *OpenSSH*. https://www.openssh.com/.

[Pfister (1998)] Pfister, Gregory. (1998). *In Search of Clusters (2nd ed.)*. Upper Saddle River, NJ: Prentice Hall PTR. p. 36. ISBN 0-13-899709-8.

[Princeton (2024)] Princeton University. (2023). *What is a cluster? | princeton research computing*. Princeton University. https://researchcomputing.princeton.edu/faq/what-is-a-cluster.

[SchedMD (2021)] SchedMD. (2021). *Slurm Workload Manager - Overview*. https://slurm.schedmd.com/overview.html.

SLURM: Simple Linux Utility for Resource Management. Job Scheduling Strategies for Parallel Processing. JSSPP 2003. Lecture Notes in Computer Science, vol 2862. Springer, Berlin, Heidelberg. https://doi.org/10.1007/10968987_3.

[Stanford (2023)] Stanford University. (2023). *Why use slurm?*. https://login.scg.stanford.edu/tutorials/job_scripts/.

[Top500 (2023)] Top500. (2023). *November 2023|TOP500*. https://top500.org/lists/top500/2023/11/.

[Xiao & Wang (2019)] Xiao, Y. & Wang, X. (2019) *19Enhancing Quasi-Monte Carlo Simulation by Minimizing Effective Dimension for Derivative Pricing*. Computational Economics 54, 343–366. https://doi.org/10.1007/s10614-017-9732-2.

[Ylonen & Lonvick (2006)] Ylonen, & Lonvick. (2006). *The Secure Shell (SSH) Protocol Architecture*. https://datatracker.ietf.org/doc/html/rfc4251.