A Parallel Approach to Least Squares Monte Carlo for Pricing American Options

EECS 587: Parallel Computing

Nikhil Patel

University of Michigan
December 7, 2023

1 Introduction

Options trading stands as a core element of modern financial markets. They offer traders and investors a versatile tool for hedging, speculation, and income generation. Fundamentally, an option is a contract that gives the holder the right, but not the obligation, to buy (in the case of a call option) or sell (in the case of a put option) a specified amount of the underlying asset at a predetermined price (known as the strike price) within a fixed time frame. Due to their widespread use, not just as trading instruments but also in key areas like risk management, price discovery, and market liquidity, the mispricing of options can lead to significant financial consequences.

Broadly speaking, there are two main types of options: European and American. Their main distinction lies in the terms of their exercise. European options can only be exercised at the expiration date. As a consequence, their value is very easy to estimate, since it only depends on the price of the underlying asset on the day of expiry. In contrast, American options offer a much more flexible exercise regime. They allow the holder to exercise the option at any point up to and including the expiration date. This flexibility adds additional complexity to their valuation, as it introduces path-dependency: the option's value is not only dependent on the underlying asset's price on the day of expiry, but on its price path throughout the life of the option.

Beyond the commonly known American and European options, there exists a plethora of other option varieties. While these additional types are not as prevalently used, their valuation often bears similarities to that of American or European options, albeit with some nuanced differences. One such option of particular note to financial engineers attempting to price American options is the bermudan option. The Bermudan option bridges the gap between the European model's simplicity and the American option's flexibility. They allow holders to exercise the contract on specified dates over the life of the option. In essence, it dissects the exercise opportunities of American options into discrete intervals. As such, many financial engineers use bermudan options with a large number of exercise opportunities as a proxy for American options, which vastly simplifies the computational challenge.

This paper will focus on the process of valuing American options. Note that we will only be focusing on pricing the American put option rather than a call option. This is because it is almost never optimal to exercise an American call early, and thus its value is almost always equal to a European call. We aim to approximate their value by using Bermudan options as a practical proxy, coupled with the application of Monte Carlo methods. Specifically, we will explore novel methods for parallelizing this process and conduct a thorough analysis of their efficiency and accuracy relative to established classical methods.

2 Option Pricing

American options are not as simple to price as European options. Their ability to be exercised at any time introduces dependencies on future market conditions, which are unpredictable and dynamic. This feature necessitates a valuation method that can account for numerous possible future scenarios and the corresponding decision-making process at each potential exercise point. So while European options can be valued using

formulas with closed-form solutions – like Black-Scholes – no such formula exists for American options. To address this complexity, Finite Difference Equations are often employed. These numerical methods solve the option pricing problem by discretizing the time and asset price, turning the continuous problem into a solvable grid format. However, the application of FDEs becomes considerably more complex and less efficient in higher-dimensional scenarios, such as options on multiple assets.

In order to combat this limitation of FDEs, the Monte Carlo Least Squares Method (LSM) is increasingly utilized. In this approach, numerous possible paths of the underlying security are simulated. This approach involves simulating numerous potential future paths for the underlying asset's price and then applying a regression technique, such as least squares, to estimate the option's value. As discussed in Jia 2009, Monte Carlo LSM is an effective method for pricing options, as it is highly adaptable to different option types like Asian or Bermudan. Importantly, it manages to accommodate higher-dimensional options without demanding a significantly larger computational effort.

2.1 Least Squares Monte Carlo

2.1.1 Explanation

Developed by Longstaff and Schwartz 2001, the Monte Carlo LSM algorithm itself is relatively simple. The basic intuition behind its approach is the idea that at any exercise time, the holder has only one decision to make: they will compare the payoff from immediate exercise with the expected payoff from continuation, and then exercise if the immediate payoff is higher. Therefore, the optimal exercise strategy is only determined by the conditional expectation of the option's payoff given we hold the option. Calculating an exact value for the conditional expectation is nearly as hard a problem as pricing the asset in general, but the key insight of LSM is that if we work backward through the option's life, then at each timestep, we can use regression trained on future realized payoffs to provide a direct estimate of the conditional expectation.

2.1.2 Challenges in Parallelizing LSM

In general parallelizing a Monte Carlo simulation is trivial, as all one needs to do is split the paths equally over the number of processors. However, with LSM, there are additional considerations introduced by the calculation of the continuation value (also known as the optimal exercise boundary). Since this value must be computed at every timestep and relies on information from all simulation paths, simply splitting the paths over the number of processors will decrease accuracy of the result.

3 Program Development

In order to accurately evaluate the tradeoffs between efficiency and accuracy during my parallelization approach and to determine where speedups originated from, four different approaches were implemented: the classical sequential approach, a trivially parallel approach, a hybrid approach, and a novel approach with a dynamic manager. Each has its own tradeoffs which will be explored in detail over the course of the paper.

Algorithm 1 Monte Carlo Least Squares Method for Option Pricing

Input: Number of paths N, Number of timesteps T, Underlying stock prices, Strike price K, Risk-free rate r

Output: Estimated option price

Simulate N paths of the underlying stock over T timesteps
Initialize matrix V to store option values for all paths at all timesteps
for $t = T$ down to 1 **do**
    Train regression model on values from all simulations to predict option value based on current asset price
    for each simulation n from 1 to N **do**
        Use regression model to calculate continuation value $C_{t,n}$
        Calculate immediate exercise value $E_{t,n}$ at timestep t for path n
        if $E_{t,n} > C_{t,n}$ **then**
            Update value matrix V: Set $V_{t,n} = E_{t,n}$
            for $j = t + 1$ to T **do**
                Set $V_{j,n} = 0$ {Zero out all future values since the option is exercised}
            end for
        end if
    end for
end for
Calculate the final value of the option by averaging the predicted price across all paths
$OptionPrice = \frac{1}{N} \sum_{n=1}^{N} max(V_n)$
return $OptionPrice$

As is standard when developing option pricing models, the paths of the underlying stock are simulated using Geometric Brownian Motion (GBM) as can be seen in Algorithm 2:

Algorithm 2 Simulate Stock Prices using Geometric Brownian Motion

Output: Initial stock price S_0, Volatility σ, Risk-free interest rate r, Dividend yield D, Time step size dt, Number of time steps N, Number of simulation paths $NSim$

Input: Simulated stock price paths S_t

Initialize matrix dB_t to hold Brownian motion increments
Initialize matrix S_t to hold stock prices, set first column to S_0
for $t = 1$ to $N - 1$ **do**
    Calculate drift: drift $= (r - D - 0.5 \times \sigma^2) \times dt$
    Calculate diffusion: diffusion $= \sigma \times dB_t[:, t]$
    Update stock prices: $S_t[:, t] = S_t[:, t-1] \times \exp(\text{drift} + \text{diffusion})$
end for
return S_t

With X representing the current price of the underlying asset, the regression used is a simple least-squares model with an intercept, X, X^1, X^2, and X^3 as features. Recall that the goal of the regression is to predict the value of the option contract given that we hold onto it through the current timestep.

3.1 Sequential Case

For the classical sequential implementation of LSM, the implementation is similar to the pseudocode presented in Algorithm 1 with a few key differences. Predicted payoffs are discounted back to the present value using the risk free rate. Additionally, only paths that are in the money at timestep t are used to train the regression model at timestep t.

Then, in sum, the complete set of parameters necessary to execute the simulation is outlined below. These parameters are summarized in Table 1, providing a clear overview of their values and descriptions.

Parameter	Description	Example Value
`sigma`	Stock volatility	0.2
`S0`	Initial stock price	80.0
`r`	Risk-free interest rate	0.04
`D`	Dividend yield	0.0
`T`	Time to maturity	1
`KP`	Strike price	100.0
`dt`	Time step size	1/50

Table 1: Parameters required for option simulation

Note that a true American option would have a $dt = \infty$ since it can be exercised at

any point over a continuous time frame. However, a dt as low as $\frac{1}{50}$ is often enough for a bermudan option to converge to the price of an American option.

3.2 Trivially Parallel Approach

In this approach, each thread is given an equal amount of simulations to complete. Using openmp, this requires very minimal modifications to the code to run. And no modifications to the actual algorithm are necessary – only to the code calling the simulator:

```
#pragma omp parallel
{
    int thread_id = omp_get_thread_num();

    //Simulate stock paths and run simulations
    vector<vector<double>> SSit =
        simulate_stock_prices(S0, sigma, r, D, dt, N, sims_per_thread);
    vector<double> result =
        american_option_pricing(SSit, KP, r, dt, N, sims_per_thread);

    all_results[thread_id] = result;
}

// Aggregate results from all threads
vector<double> aggregated_results;
for (const auto& thread_result : all_results) {
    aggregated_results.insert(aggregated_results.end(),
        thread_result.begin(), thread_result.end());
}

double price = accumulate(aggregated_results.begin(),
    aggregated_results.end(), 0.0)/NSim;
```

In theory this approach should be a little less accurate than the sequential case, since each regression only has access to at most $1/P$ training samples. Though we do expect to see a speedup by a factor of P.

3.3 Hybrid Approach

Recall that the main simulation in Algorithm 1 has two loops. The outer loop executes code for each timestep. During each timestep, after the regression has been trained, the inner loop handles each simulation's decision-making. In this hyrbid approach to parallelization, we parallelize just this inner loop. This idea is motivated by the fact that each regression will now still have access to the full suite of training data, but we also get to harness the time efficiency of parallelization. In theory, this should result in accuracy equivalent to the sequential approach.

Implementing this with openmp is very simple, as it just requires placing a directive before the relevant loop.

```
#pragma omp parallel for num_threads(numThreads)
```

3.4 Dynamic Approach

This approach presents the most radical departure from the traditional algorithm. It seeks to achieve the accuracy of the sequential algorithm while delivering the speedup of the trivial parallelization. It is based on the idea that training the regression model is very time consuming relative to running the rest of the simulation. Thus, we create two different types of threads: simulators and regressors. Let us first walk through what each type of thread does before we discuss how we decide how many of each thread type to use and where we expect to see speedups.

3.4.1 Simulator Threads

Notice how in Algorithm 1, we only iterate through each timestep once. We compute the option's value for every single path before moving to the next timestep. In the Dynamic approach, this is not the case. Each thread is only responsible for simulating one path's values from start to finish.

For example, say we wanted to simulate $1,000,000$ total paths. Each thread would simulate one path completely from start to finish. When a thread terminates, it is spawned anew and simulates a different path completely from start to finish. This happens until all $1,000,000$ paths have been simulated. This can be seen in Algorithm 3.

Importantly, none of these threads will ever train a regression model. Instead, they assume the regression coefficients have been precomputed and stored in a data structure called the $coefficientsMap$. This allows these simulation threads to simulate paths extremely fast.

The responsibility of training regressions is delegated to the Regressor threads. However, the Regressor threads still need data from the simulator threads to carry out their regressions. As such, each simulator thread is required to periodically send data to a global data structure called the *dataQueue*. The *dataQueue* holds all data used to train regressions. Since sending data to the *dataQueue* must be done in a critical region, we don't necessarily want all threads to have to send all data to the queue. Instead, if a given option is in the money, the thread will be required to send data about the current timestep – and only the current timestep – to the queue with probability $1/P$. A simple overview of this process can be seen in Algorithm 4.

The system of sending data to the queue with probability $1/P$ every iteration was chosen as it performed the best out of a few different methods. The first method was quite similar to the current method. The thread was still required to send info to the *dataQueue* with probability $1/P$. However, the difference laid with what data the thread was required to send. In this approach, the thread had a temporary *localDataQueue* that held information about every timestep since the last time it uploaded data to the *dataQueue*. Then, with probability $1/P$ every timestep, it had to upload all data from the *localDataQueue* to the global *dataQueue*. The second method is also similar. Instead of possibly sending data every timestep, the thread just keeps track of all data in a *localDataQueue* and uploads everything at once right before it terminates. While both of these alternative methods bolster accuracy a bit, the slow-

down in time they caused was not worth the trade-off. Requiring each thread to gain sole access to the *dataQueue* to do its appending causes too large a bottleneck. This can be seen in Table 2.

Method	Time (ms)	Error (%)
1/P Current Info Only	4112	3.06
1/P All Info Since Last Transmission	32604	2.56
All Info at End of Simulation	37371	2.53

Table 2: Timing results for different *dataQueue* update strategies

Algorithm 3 Parallel Simulation Process

```
#pragma parallel(numthreads)
CompletedSimCount ← 0
while CompletedSimCount < NumSimulations do
    SimulatePath()
    #pragma critical
        CompletedSimCount ← CompletedSimCount +1
    end #pragma critical
end while
end #pragma parallel
```

Algorithm 4 Simulate Path Function

```
S_t ← SimulateStockPrices()
for t = T down to 1 do
    if InMoney then
        if Rand(0,1) < 1/P then
            Send current stock price and payoff to dataQueue
        end if
    end if
    Get coefficients and calculate continuation value
    if immediateExerciseValue > continuationValue then
        Update value matrix
    end if
end for

return  max(value matrix)
```

3.4.2 Regressor Threads

Regressor threads are constantly pulling data from the *dataQueue*, training regressions, and placing the coefficients into the $coefficientsMap$. As simulator threads complete more simulations, there will be more data to train regressions. As such, regressions trained later will be more accurate than ones trained earlier. Since each timestep

requires its own regression model, it is important that each timestep is trained equally often, since we want all regression models to be trained with as much data as possible, and we don't want some to remain old and stale. As such, a priority queue is used to keep track of which timestep should be processed next. Each time a regression is performed, it is added to a priority queue along with the current real world timestamp (i.e. the time that this regression occurred). Then, the timestep at the top of the priority queue will be the one with the smallest timestamp, which happens to be the timestep that was processed the longest ago.

Much of functionality of this thread is allowed to operate completely in parallel, with critical regions guarding only the interactions with the priority queue, $dataQueue$, and $coefficientsMap$.

3.4.3 Thread Manager

If we hold the number of simulations performed constant and just vary the split between the number of simulator and regressor threads, we may be able to draw some insight into what their natural balance should be. This can be seen in Figure 1. For these graphs, we fix the number of simulations and have $P = 8$. That is, we have 8 processors. So for example, if we have 3 Simulator threads, then the other 5 will be Regressors. Thus, we can see that increasing the number of Simulator threads (and therefore decreasing the number of Regressor threads) tends to increase error. Though, this increase seems to become less drastic after a certain point. This suggests that you need a significant number of Regressor threads to start seeing major reductions error. in When all 8 threads are Simulators (and none are Regressors), we can see the error suddenly jump. This is because no regressions at all are being performed. Execution time seems to consistently decrease as we increase the number of simulator threads. This makes sense since the program as a whole terminates when all the simulations are complete, with no regard for how much progress the Regressors have made.

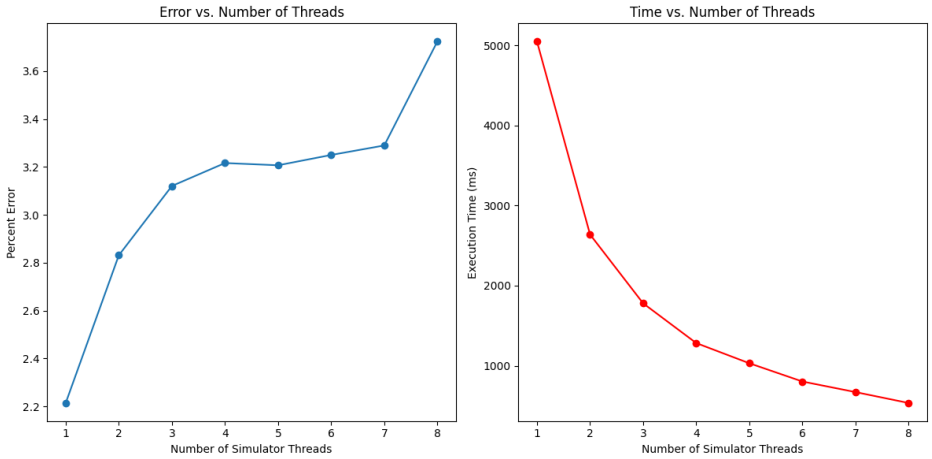

Figure 1: Error and Execution Time vs. Simulator/Regressor Thread Split for P=8

Expanding this to see $P = 16$ processors in Figure 2 lets us see the trend more clearly. We can roughly break the error graph into 3 regions. Let P_s denote the number of Simulator threads and P_r denote the number of Regressor threads. In region 1

($P_s \in [1, 4]$), P_s is low and P_r is high. We can see that increasing P_r in this region does not really contribute to much reduction in error. This suggests that at this point, Regressor threads are able to do regressions as fast as data is being generated by the simulators, so there is no point to adding more. In region 2 ($P_s \in [5, 10]$), we see massive reductions in error as we add more Regressor threads. In region 3 ($P_s \in [11, 16]$), we see that again adding more Regressor threads does not lower error. This suggests that the simulators are producing way more data than can be handled by the few Regressor threads, and they are simply not able to perform enough regressions before the program terminates.

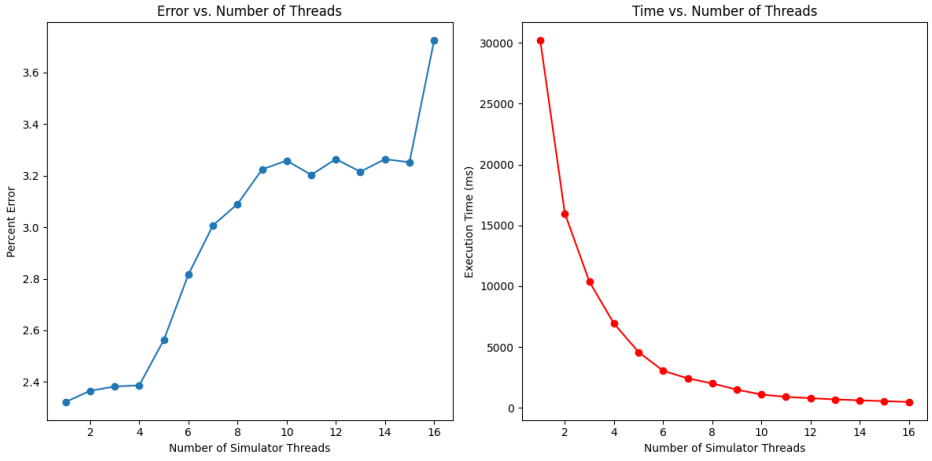

Figure 2: Error and Execution Time vs. Simulator/Regressor Thread Split for P=16

One way to capitalize on this behavior is to implement a thread manager that controls the number of Simulator and Regressor threads in order to target some specified ratio of completed simulations to completed regressions. This would allow the user to fine tune the simulator to prioritize either accuracy, speed, or some mix of both. In industry, financial engineers may want to use regression models more complicated than a simple least squares, or predict options on stocks that follow paths more complicated to simulate than Geometric Brownian Motion. These changes would require less and more Simulator threads to achieve the same accuracy, respectively. By specifying the desired ratio of completed simulations to completed regressions, the manager can ensure that the required changes to P_n and P_s are made automatically. A simplified example of how the manager works can be seen in Algorithm 5.

In Figure 3 we can see that error and time change how we would expect based on the optimal ratio chosen. In the next section we will analyze the speedups introduced by the manager system as well as test whether or not this system is faster and more accurate than the other approaches.

4 Analysis

4.1 Ensuring Simulated Values are Correct

Since Finite Difference Methods are very accurate at determining the price of one dimensional American options, it is sufficient to test the accuracy of these Monte Carlo

Algorithm 5 Manager

```
1: CompletedSimCount ← 0
2: CompletedRegCount ← 0
3: numSimulators ← P/2
4: OptimalRatio ← 3
5: #pragma omp parallel
6: while CompletedSimCount < NumSimulations do
7:    if omp_get_thread_num() < numSimulators then
8:       SimulatePath()
9:       #pragma omp critical
10:          CompletedSimCount ← CompletedSimCount + 1
11:      end #pragma omp critical
12:   else
13:      Regression_Worker()
14:      #pragma omp critical
15:         CompletedRegCount ← CompletedRegCount + 1
16:      end #pragma omp critical
17:   end if
18:   #pragma omp master
19:   ratio ← CompletedSimCount/CompletedRegCount
20:   if ratio > OptimalRatio then
21:      numSimulators ← max(1,numSimulators−1)
22:   else
23:      numSimulators ← min( numSimulators +1, omp_get_max_threads())
24:   end if
25:   end #pragma omp master
26: end while
27: end #pragma omp parallel
```

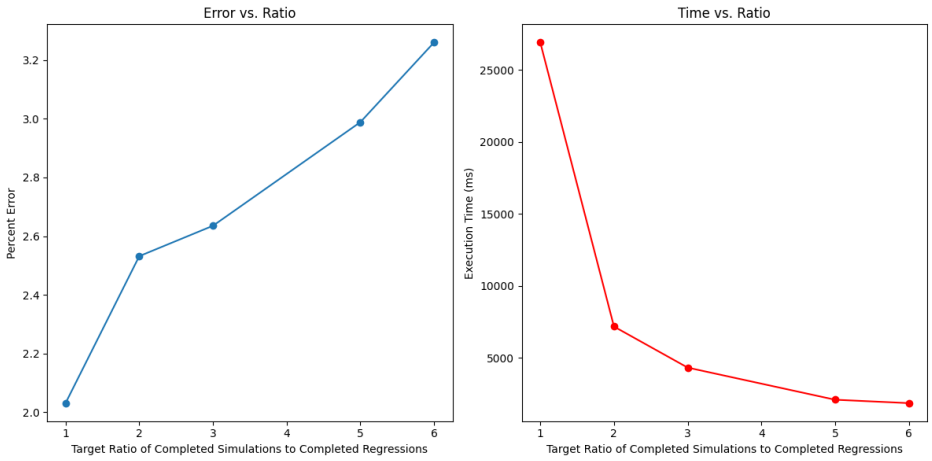

Figure 3: Error and Execution Time vs. Target Ratio for P=32

simulations by comparing them to values computed using Finite Difference methods.

4.2 Comparing the Approaches

Parameter	Description	Value
`sigma`	Stock volatility	0.2
`S0`	Initial stock price	36.0
`r`	Risk-free interest rate	0.06
`D`	Dividend yield	0.0
`T`	Time to maturity	1
`KP`	Strike price	40.0
`dt`	Time step size	1/50
`P`	Number of threads	32

Table 3: Parameters used to compare approaches

Consider the set of parameters shown in Table 3. We will use these parameters to test and compare all four approaches. According to Finite Difference Methods, the true price of this option should be \$4.478. The results of this test are shown in table 4.

Hybrid Parallelization, while providing a small speedup, does not nearly scale with P. This is because not all of the code is performed in parallel. Importantly, every regression training – which is the most time consuming step – is still performed in sequence. We do see accuracy close to the sequential approach. This makes sense since each regression has access to the full suite of training data.

When we run the Dynamic approach on the same number of simulations as the sequential method, we see a speedup greater than a factor of P. However, we see a noticeable decrease in accuracy. In order to remedy this, we can run the Dynamic approach on more simulations. When we do this, we can even raise the ratio of simulations to regressions. Though this makes the model less accurate, we can expect the increased number of simulations to offset this. Running this approach, we do see accuracy that beats the sequential method. However, most of our speedup gains are gone.

Approach	NSimulations	Ratio (Dynamic Only)	Time (ms)	Error ($)
Classical Sequential	100,000	-	13,989	0.056
Trivial Parallelization	100,000	-	441	0.091
Trivial Parallelization	3,200,000	-	14,868	0.049
Hybrid Parallelization	100,000	-	10,240	0.058
Dynamic	100,000	8	383	0.081
Dynamic	3,200,000	16	12,106	0.042

Table 4: Timing results for multiple strategies

As we increase the number of simulations, regressions take longer to train. Therefore, it gets harder for the manager to maintain the target ratio. So as the program runs, the manager will have to dedicate more and more threads to being Regressor threads, increasing the run-time of the program. By the numbers, the dynamic approach is still faster and more accurate than the trivial parallelization, so this approach still offers some merit in practical use.

4.3 Analysis of the Dynamic Approach

The core advantage of the Dynamic Approach lies in its separation of the simulation and regression processes. This methodology enables simulations to run at their maximum speed, unencumbered by the large demands of regression tasks. As a result, simulations can be completed significantly faster than traditional methods that integrate regression within the simulation. Empirical evidence supports this, showing that a simulation conducted without embedded regression can be completed twice as fast as one that includes regression training. When considering a system with P processors, divided into P_r for regression and P_s for simulation, this separation approach can lead to a performance enhancement by a factor of $2 \cdot P_s$. The trivially parallelized program enjoys a theoretical speedup by a factor of P. Therefore, in theory, as long as $P_s > \frac{P}{2}$, we would expect the manager to perform better.

It is important to also consider accuracy. In the trivial approach, each regression is trained on $NS \cdot \frac{1}{P}$ samples, where NS represents the number of simulations. In the Dynamic approach, each simulation has a $1/P_s$ chance of contributing data to the regression's dataset. There are NS simulations, so we might find that each regression is trained on $NS \cdot \frac{1}{P_s}$ samples. However, this is only the case for regressions trained near the end of the NS simulations, since the data pool is built up over time. So in practice, each regression is trained on $NS \cdot \frac{1}{P_s} \cdot \frac{CurSimCount}{NS}$, where $CurSimCount$ represents the

current number of completed simulations. Then,

$$\mathbb{E}[\text{Training Samples}] = \mathbb{E}\left[NS \cdot \frac{1}{P_s} \cdot \frac{\text{CurSimCount}}{NS}\right] \tag{1}$$

$$= NS \cdot \frac{1}{P_s} \cdot \mathbb{E}\left[\frac{\text{CurSimCount}}{NS}\right] \tag{2}$$

$$= NS \cdot \frac{1}{P_s} \cdot \frac{1}{2} \tag{3}$$

$$= NS \cdot \frac{1}{2P_s} \tag{4}$$

This leads to an interesting scenario where both the speed an accuracy of the Dynamic approach are inversely proportional to $2P_s$. When $P_s = \frac{P}{2}$, we would expect the Dynamic approach to have the same speedup and training samples as the trivial program. As we increase P_s, its speed increases, but number of training samples decreases, which will decrease accuracy. This behavior can be seen in Figure 4 where $P = 16$ and $NS = 100$.

Interestingly, an $n\%$ decrease in training samples does not cause an $n\%$ decrease in accuracy, as accuracy is dependent on a wide array of factors including number of simulations, P_s/P_n split, and the parameters of the simulation.Therefore, we expect the dynamic manager to perform better than the trivial parallelization in empirical trials.

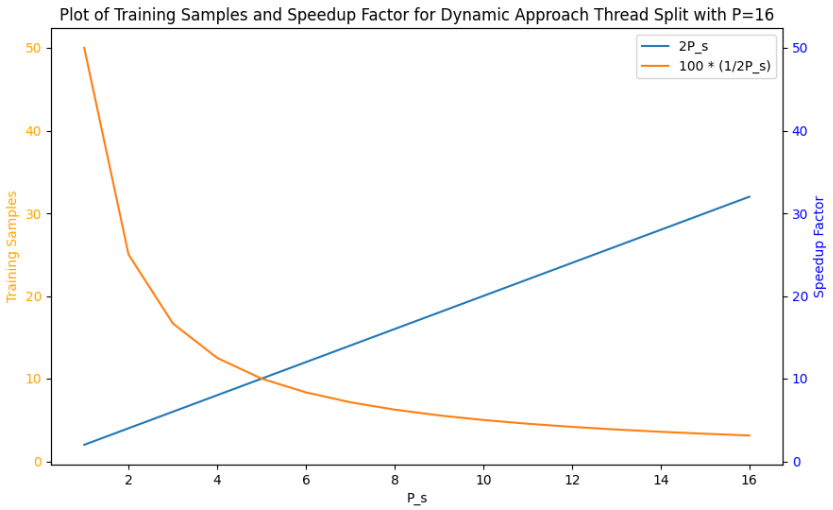

Figure 4: Theoretical Training Samples and Speedup Factor for Dynamic Approach for P=16

5 Conclusion

This project has sought to examine the application of parallel computing techniques to the Least Squares Monte Carlo (LSM) method for pricing American options. We explored and analyzed different parallelization strategies to strike a balance between computational efficiency and accuracy in option pricing.

The widely-used sequential approach, is highly-accurate and offers a baseline for performance. The trivially parallel approach, despite its simplicity and improved speed, compromises on accuracy due to the limited data available for each regression model. The hybrid approach, while offering a balance between speed and accuracy, still does not fully exploit the potential of parallel computing, as the regression training remains a sequential process.

The Dynamic approach is a radical shift from traditional approaches. By moving simulation and regression tasks into separate threads and dynamically managing these based on the desired ratio of completed simulations to regressions, this approach effectively harnesses the power of parallel computing. It's flexibility is particularly noteworthy, as it allows for customization according to the complexity of the underlying stock model or the sophistication of the regression method, adapting the number of Simulator and Regressor threads accordingly. This adaptability makes it a robust solution for a variety of scenarios in financial computation.

It is a known flaw that the Dynamic approach's performance degrades as the number of simulations is increased, since it gets harder for the manager to maintain the target ratio as regressions take longer to train. It would be an interesting next step to test and validate a system that relaxes the target ratio over time to account for this. Additionally, much work is being done on choosing appropriate basis functions for the regression. In some works, researchers have used the Hermite polynomials to achieve more accurate results. It would be insightful to test the parallel approaches presented with alternate regression models to see how performance changes. It would be interesting to determine whether the increased accuracy offsets the additional computational complexity of training more complicated regression models, especially in a system like the Dynamic approach.

Overall, this work presents a strong baseline for an interesting new approach to parallelizing LSM with a dynamic thread management system, offering a novel and efficient way to balance speed and accuracy in financial computations.

References

Jia, Quiyi (2009). "Pricing American Options using Monte Carlo Methods". In: *Uppsala University Department of Mathematics*.

Longstaff, Francis A. and Eduardo S. Schwartz (2001). "The TeXbook". In: *The Review of Financial Studies*.