# Implementing a Monte Carlo Simulation in Haskell

Griffin N (gcn2106) Anna C (ajc2321) Sparsh B (sb4835)

December 20, 2023

## Contents

**1 Introduction 2**

**2 Background 2**
2.1 Mathematical Background . . . 2
2.1.1 Monte Carlo Method . . . 2
2.1.2 Options Pricing . . . 2
2.1.3 Probability Background . . . 2
2.2 Programming Background . . . 3

**3 Sequential Implementation 4**
3.1 Algorithm . . . 4
3.2 Performance . . . 6

**4 Parallel Implementation 7**
4.1 Algorithm . . . 7
4.2 Parallel Attempt 1 . . . 8
4.3 Parallel Attempt 2 . . . 9
4.3.1 Performance Compared to Sequential . . . 9
4.4 Parallel Attempt 3 (Final) . . . 10
4.4.1 Performance . . . 10
4.4.2 Performance Compared to Parallel Attempt 2 . . . 14

**5 Bonus Implementation: Vector Operations 15**
5.1 Sequential Algorithm . . . 15
5.2 Sequential Performance . . . 16
5.3 Parallel Algorithm . . . 16
5.4 Parallel Performance . . . 17
5.5 Potential Performance Enhancing Alternatives . . . 19

**6 Conclusion: Putting it All Together 19**

**7 Haskell Code Reference 20**

**8 References 26**

# 1 Introduction

For our project, we chose to implement a Monte Carlo simulation for Asian Options pricing in Haskell. There are three ways in which we approached the coding to see which provides the best overall performance gain - straightforward sequential programming, parallelising the sequential code, and, lastly, vectorizing the operations. In the next sections we will provide the necessary background on Monte Carlo method and Options pricing and then describe each implementation technique in detail.

# 2 Background

## 2.1 Mathematical Background

It is necessary to first establish the relevant mathematical context to the Monte Carlo method and how it applies to finance. This is divided into three broad categories, the Monte Carlo method, Options pricing within Financial markets, and Probability.

### 2.1.1 Monte Carlo Method

The Monte Carlo method is a broad class of algorithms that rely on repeated random sampling to obtain a numerical result [6]. By leveraging randomness, it efficiently solves problems that might be deterministic in principle. This is especially useful when there is no straightforward way to use the deterministic nature of the problem to solve it or when using the deterministic nature raises the complexity of the solution.

### 2.1.2 Options Pricing

A Stock Option is the derivative of the stock and is based off the price of the stock over a given time period. For the purpose of this project, we will consider the Asian option. the Asian call option pays off based on how much the average stock price over the time period $T$ exceeds the strike price $K$. If the average stock price is greater then, the payoff is positive and equal to the difference. If the average stock price is less than or equal to $K$, then the payoff is zero [5]. This feature makes Asian options particularly sensitive to the average behavior of the stock price over time rather than its exact value at the expiration time.

Here's a formula for computing the payoff for an Asian call option [3]:

$$C_T = \left( \frac{1}{T} \sum_{i=1}^{T} S_i - K \right)^+$$

- $C_T$: The payoff of the Asian call option at time $T$.
- $\frac{1}{T} \sum_{i=1}^{T} S_i$: Average stock price over the time period from $1 \rightarrow T$.
- $\left( \frac{1}{T} \sum_{i=1}^{T} S_i - K \right)^+$: This ensures that the payoff is non-negative.

- $S_i = S_{i-1}R$: With $R$ being $u$ with probability $p = \frac{(1+r-d)}{(u-d)}$ and $d$ with probability $1-p$
- $u, d$, and $r$ are parameters to the function representing the up market multiplier, the down market multiplier, and the risk free interest rate respectively.

There are many different types of options, beyond the Asian option - American and European options - with the difference being when you are able to exercise the option and how the payoff is calculated (and not related to geographic location). Since the Asian Call Option is one that does not have a closed form solution [3] but rather relies on an average performance over a certain period, simulating the prices works really well for it. The Asian call option differs from a standard call option in that its payoff is based on the average value of the stock over a specified period (from time 1 to time $T$) rather than just the stock price at the expiration time.

### 2.1.3 Probability Background

The method of randomness for the application of Monte Carlo on the Asian option is controlled by a Bernoulli Random variable. A Bernoulli Random Variable is a random variable that yields a success with a probability $p$ and

a failure with a probability of $1 - p$ From this we can derive what the expected value of a Bernoulli is, we know that the expected value of a random variable is defined as follows:

$$E[X] = \sum_{x=0}^{\infty} \mathbb{P}(X = x)x$$

So applying this to the Bernoulli we get:

$$E[X] = \mathbb{P}(X = 0)0 + \mathbb{P}(X = 1)1$$
$$E[X] = (1 - p)0 + (p)1$$
$$E[X] = p$$

From this we can begin describing the role that the Law of Large Numbers plays in these simulations. We will demonstrate this using the classic example of flipping a coin. If we flip a non biased coin a singular time the percentage of flips that are heads are either 0% or 100% The percentage will change as we flip the coin more and more times, why early results may very the more flips we do the closer we get to the expected value.

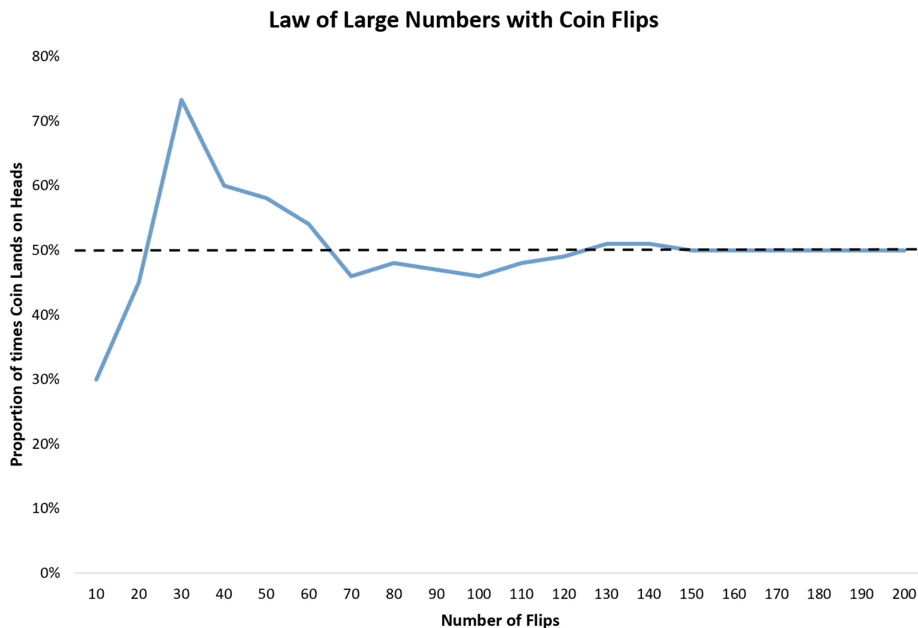



Figure 1: Image displaying the Law of Large Numbers

This example illustrates motivation for the Law of Large Numbers, which states that the result of performing the same experiment a large number of times will have an average value that converges on the theoretical expected value of the experiment.

## 2.2 Programming Background

Our Haskell implementation takes its roots from a sequential Python implementation that is assigned as a homework problem to students enrolled in the economics context in COMS 1002: Computing in Context, solutions are provided to the TAs of the course and since Griffin is a TA for that course, that is how we acquired the solutions. This implementation serves as a baseline for the project and is considered a correct implementation meaning we would know if our Haskell functions were accurate if they lined up well with the output of the python program.

```
def bernoulli(p):
    if random.random() < p:
        return True
    else:
```

```
        return False

def monte_carlo_asian_call(n, t, r, u, d, s_o, k):
    discount = 1 / ((1 + r) ** t)
    p_star = (1 + r - d) / (u - d)
    total = 0

    for i in range(n):
        sum_prices = 0
        price = s_o
        for i in range(t):
            if bernoulli(p_star):
                price=price*u
            else:
                price=price*d
            sum_prices += price
        val = (sum_prices / t) - k
        total += max(val, 0)

    return (total * discount) / n
```

The implementation of the Bernoulli random variable, while naive, is quite standard. From the function implementing the Monte Carlo we can see the mathematical formula play out as well. Two new parameters present are $n$ and $s_0$ which represent the number of trials and the initial stock price. Knowing all of this we can move forward with the Haskell Implementations.

# 3 Sequential Implementation

## 3.1 Algorithm

The sequential implementation of the Monte Carlo Asian option pricing algorithm in Haskell closely follows a Python version distributed in an Economics course. The primary objective is to simulate the pricing of Asian options using a Monte Carlo method, which involves generating random price paths for the underlying asset and computing the corresponding option payoffs.

The main function, `monteCarloAsian`, accepts parameters such as the number of trials (`n`), the number of time steps (`t`), the risk-free interest rate (`r`), the up movement factor (`u`), the down movement factor (`d`), the initial stock price (`s0`), and the strike price (`k`). It performs a Monte Carlo simulation by generating random price paths for the underlying asset, calculating the corresponding option payoffs, and then averaging the results over multiple trials.

```haskell
monteCarloAsian :: Int -> Int -> Double -> Double -> Double -> Double -> Double -> IO Double
monteCarloAsian n t r u d s0 k = do

  let discount = 1 / ((1 + r) ^ t)

  -- implementation details

  -- Perform 'n' trials and compute the average
  total <- sum <$> replicateM n seqTrial
  return $ (total * discount) / fromIntegral n
```

The simulation involves recursively calculating the sum of prices along a path in the financial market. This is achieved through the `seqTrial` function, which uses a helper function `seqCalcPrice` to traverse the price path and accumulate the sum.

```haskell
-- ... (other code)
```

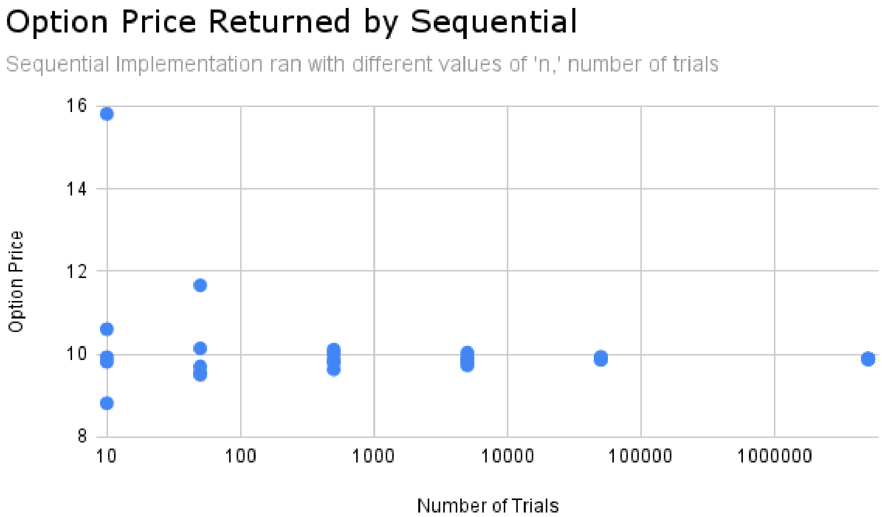
```haskell
 -- Perform a single trial of the simulation
  let seqTrial = do
        -- Recursively calculate the sum of prices along a path
        let seqCalcPrice i sumPrices price
              | i == t = return sumPrices
              | otherwise = do
                b <- bernoulli pStar
                if b == 1
                    then seqCalcPrice (i + 1) (sumPrices + (price * u)) (price * u)
                    else seqCalcPrice (i + 1) (sumPrices + (price * d)) (price * d)

        -- Calculate the difference between average simulate price and strike price
        sumPrices <- seqCalcPrice 0 0.0 s0
        let diffVal = (sumPrices / fromIntegral t) - k
        return $ diffVal `max` 0

-- ... (other code)
```

To perform each step in each trial we use randomIO to get a random value between 0 and 1, which is used to mimic the Bernoulli distribution

```haskell
bernoulli :: Double -> IO Int
bernoulli p = do
    randomVal <- randomIO -- :: IO Double
    return $ if randomVal < p then 1 else 0
```

Within `seqCalcPrice`, random up or down movements are determined using a Bernoulli distribution. The difference between the average simulated price and the strike price is then computed, and the option payoff is calculated as the maximum of this difference and zero. The chart seen below shows our sequential implementation run on different values of n, the number of Monte Carlo trials. The other values i the function were set at t = 10, r = 0.05, u = 1.15, d = 1.01, s0 = 50, k = 70.

The function is run for different values of n 5 times. As seen in the scatter plot, the output of the program approaches the value 9.88 as the number of trials is increased, which is the expected behavior of the Monte Carlo simulation following the law of large numbers.

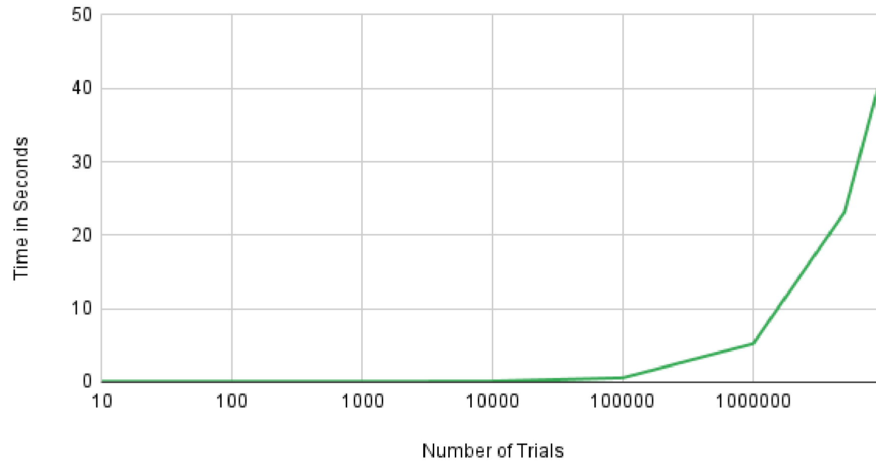



In our test file (test/Spec.hs), we confirm that the value our implementation is approaching is within range of the value expected, as determined by the orginal Python solution for the Asian Option.

### 3.2 Performance

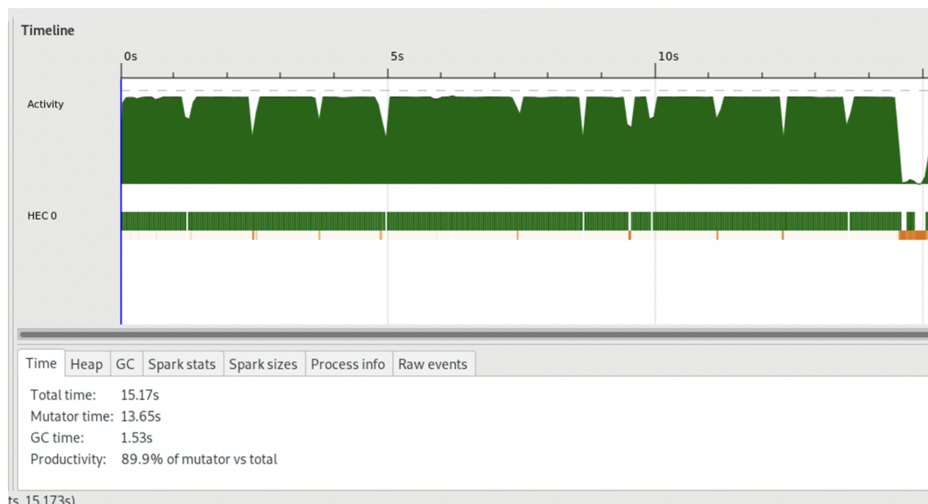
We conducted several tests of the sequential implementation with the following parameters controlled as:
t = 100
r = 0.05
u = 1.15
d = 1.01
s0 = 50
k = 70

We varied n, the number of trials and recorded the program's runtime:




The runtime is relatively low, but after increasing the number of trials past 100000, runtime increases significantly, with the program taking 15.46 seconds for n = 5000000, and 42.80 seconds for n = 10000000. Increasing n past 10000000 led to the sequential implementation failing due to overflow.

The ThreadScope activity seen below is for the trial ran with n = 5000000.




The eventlog shows consistent activity near 1 throughout the runtime of the program, and with a lot of garbage collection left for the end of the program. The consistent activity of the program indicates the program is doing effective work, but it is limited by its sequential nature, as it is forced to do all of these computations one after another.

These metrics demonstrate how the sequential evaluation of our current program is severely limited as we increase the number of trials, which is not ideal considering the Monte Carlo Output will be more precise with more trials. In turn, we have reason to turn to Parallel strategies to create a more effective Monte Carlo Option Pricing Method.

# 4 Parallel Implementation

Each trial of our Monte Carlo Simulation is doing a computation of similar complexity- each trial runs the same calculation to determine an accumulated price along a time path, and only vary in random Bernoulli numbers. The similarity in computational complexity between trials makes the function an excellent candidate for parallelism.

## 4.1 Algorithm

Before parallelism could occur, we had to significantly change the sequential algorithm, specifically the way we were getting random Doubles for the Bernoulli random numbers. The sequential algorithm used IORandom to return random doubles for the Monte Carlo simulations; however, we quickly discovered that the IO Monad forces evaluation into sequential order. As an alternative, we decided to use a Pseudorandom Number Generator, or PRNG. Generating truly random numbers is slow; alternatively, PRNGs use an initial seed to arithmetically generate random-like sequences of numbers. Haskell's built in PRNG is slow [2], so we needed to find an alternative.

In a blog post, Alexey Kuleshevi conducted a series of benchmark tests of different Haskell PRNGs and found splitmix to be the best parallel performer for generating Doubles.[4]

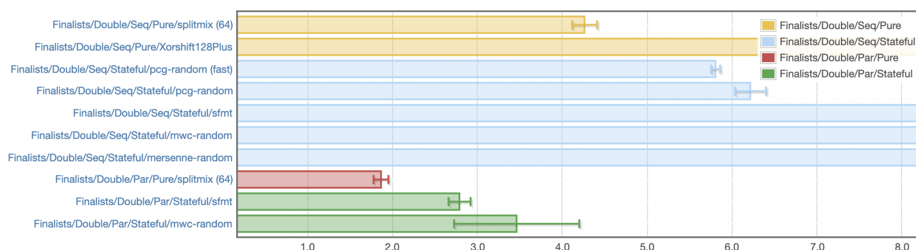



Figure 2: Chart of the Performance of Different PRNGs

Shown above in figure 2, splitmix (the red bar) performs much faster than its competitors (mwc-random, sfmt, pcg-random, mersenne-random) in a parallel setting. Consequently, we chose splitmix for our implementation, since it provides a performance-optimal way to generate random Doubles. We incorporated splitmix into the parallel implementation of the Asian Option Monte Carlo, monteCarloAsianParallel, in the following way:

- In our main executor we generate one truly random generator, and pass that generator as an argument to monteCarloAsian Parallel

```
    initGen <- initSMGen
-- ... (other code)
    let resultPar = monteCarloAsianParallel coreCount n t r u d s0 k initGen
```

- We created a helper function used to split the initial generator into 'n' independent generators.

```
-- Helper function to unfold SMGen into a list of n generators
unfoldsSMGen :: SMGen -> Int -> [SMGen]
unfoldsSMGen gen n = take n $ iterate (snd . splitSMGen) gen
```

- Within monteCarloAsian Parallel, unfoldsSMGen is used to split the initial generator into n generators, where n is equal to the number of monte carlo trials to be conducted. a trial is run with each generator.

- Within each trial, the nextDouble function is used to determine the bernoulli random numbers, which returns a random Double and a new generator.

```
bernoulliParallel :: Double -> SMGen -> (Int, SMGen)
bernoulliParallel p gen = let (!random_val, gen') = nextDouble gen
                          in (if random_val < p then 1 else 0, gen')
```

After incorporating splitmix, we were able to get monteCarloAsianParallel out of the IO Monad, allowing us to begin parallelizing with Strategies.

## 4.2 Parallel Attempt 1

Our initial attempt was to simply apply `rdeepseq` to each trial with `parMap`.

```
monteCarloAsianParallel n t r u d s0 k initialGen =
    sum (parMap rdeeseq trial (unfoldsSMGen initialGen n)) * discount) / fromIntegral n
-- ... (other code)
```

After testing this version, it was clear the initial attempt failed. The Threadscope eventlog showed Activity remaining around 1, indicating that the program was still executing processes sequentially; the cores weren't working in parallel as we first expected. Tested on the same parameters, the our 'parallel' version managed to be about 1.6 times slower than the sequential version.

We believe this initial failure was due to how `parMap` works. Reading the documentation,

```
parMap strat f
```

is equivalent to

```
withStrategy (parList strat) . map f
```

Rewritten in the `withStrategy` form, our code looks like this:

```
monteCarloAsianParallel n t r u d s0 k initialGen =
    sum (withStrategy (parList rdeepseq) . map trial
        (unfoldsSMGen initialGen n)) * discount) / fromIntegral n
-- ... (other code)
```

The code above more clearing indicates how the strategy was being applied to the trial. However, the work that occurs in the trial (as seen below) is a sequential calculation (`calcPrice` as shown below), since the accumulated price along the path is dependent on the output of the previous step.

```
trial p_star u d s0 k t genTrial = do
  let (!sum_prices, _) = calcPrice 0 t p_star u d 0 s0 genTrial
      -- ... (other code)
```

```haskell
-- Helper function to calculate the price in a given trial
calcPrice :: Int -> Int -> Double -> Double -> Double -> Double -> Double ->
            SMGen -> (Double, SMGen)
calcPrice i t p_star u d !sum_prices !price genCalc
  | i == t    = (sum_prices, genCalc)
  | otherwise = let (b, genNext) = bernoulliParallel p_star genCalc
                    (!newPrice, newGen) = if b == 1
                                          then (price * u, genNext)
                                          else (price * d, genNext)
                in calcPrice (i + 1) t p_star u d (sum_prices + newPrice) newPrice newGen
```

Rather than simply applying a strategy to the trials' computation, what we truly wanted was for the trials themselves to be executed in parallel. This motivated our second attempt at parallelism.

## 4.3 Parallel Attempt 2

```haskell
monteCarloAsianParallel n t r u d s0 k gen =
  let chunkSize = n `div` 4
      gens = unfoldsSMGen gen n
      -- ... (other code)
      trials = parMap rdeepseq (trial p_star u d s0 k t)
                    gens `using` parListChunk chunkSize rdeepseq
      result = sum trials * discount / fromIntegral n
   in result
```

Instead of simply apply `parMap rdeepseq`, we incorporated `parListChunk`. As shown in the code above, `parListChunk` encases the original `parMap rdeepseq` statement, and sparks the list of evaluations in chunks of a predetermined size. We chose to have `chunkSize` simply equal n, the number of trials divided by 4, which is equivalent to saying we want around 4 chunks of size n/4 when grouping the trials in `parListChunk`. Each chunk is evaluated with the strategy `rdeepseq`.

The 'using' statement is equivalent to runEval (s x)

so the trials statement from above could be interpreted as

```haskell
trials = runEval (parListChunk chunkSize rdeepseq) (parMap rdeepseq (trial p_star u d s0 k t) gens)
```

By encasing the `parMap rdeesq` within another strategy for chunking, we are able to force the program out of the sequential behavior exhibited in Attempt 1, and tell it to run the trials in chunks with `rdeepseq`.

In addition, we added BangPatterns throughout (see code in section 7.1) to specify strict instead of lazy evaluation within the trial helper functions.

### 4.3.1 Performance Compared to Sequential

By adding the `parListChunk` strategy, we were able to increase performance dramatically compared to the sequential version. With sequential and parallel on 4 cores run with the parameters

n = 5000000
t = 100
r = 0.05
u = 1.15
d = 1.01
s0 = 50
k = 70

We saw an improvement from 15.46 seconds to 3.36 seconds, which is a 4.6x speedup. More performance figures for Attempt 2 are found in section 4.4.2, which compares our final parallel performance with Attempt 2.

## 4.4 Parallel Attempt 3 (Final)

Despite the increased performance in Attempt 2, we were able to improve further by calculating the `chunkSize` differently. We modified our original `chunkSize` calculation from

```
let chunkSize = n `div` 4
```

to

```
let chunkSize = n `div` (10 * numCores)
```

Our motivation for the new equation is to determine the `chunkSize` more carefully. The number of chunks that will be run by `parListChunk` will end up being,

$$\text{numChunk} = \frac{n}{\text{chunkSize}}$$

where n is the number of trials.

By setting `chunkSize` n, the number of trials we have to run, divided by 10 * number of cores, we are saying that the number of chunks will be equal to 10*numCores

$$\text{numChunks} = \frac{n}{\frac{n}{10\times\text{numCores}}}$$

Which simplifies to

$$\text{numChunks} = 10 \times \text{numCores}$$

By setting the number of chunks to 10 * number of cores, we hope to divide the computational work evenly so that each core will ideally run 10 equal chunks of trials in parallel.

To facilitate this new `chunkSize` calculation, we added an additional parameter to the `monteCarloAsianParallel`, called `numCores`, which is determined by `getNumCapabilities` in main. `getNumCapabilities` returns the number of Haskell threads that can run truly simultaneously (on separate physical processors) at any given time. [1]

```
    coreCount <- getNumCapabilities
    initGen <- initSMGen
    let resultPar = monteCarloAsianParallel coreCount n t r u d s0 k initGen
```

### 4.4.1 Performance

Performance tests were conducted on an Ubuntu VM with 4 CPUs.
The new method for determining, along with the other changes from attempts 1 and 2 (splitmix, strategies, BangPatterns), led to significant performance improvements.
Compared to the sequential implementation, the parallel implementation performed significantly faster when performing trials $\geq 100000$. For instance, when the parallel was run on 4 cores and with the parameters

n = 5000000
t = 100
r = 0.05
u = 1.15
d = 1.01
s0 = 50
k = 70

The parallel completed in 1.63 seconds, which is a 9.5x speedup compared to the sequential on the same parameters.

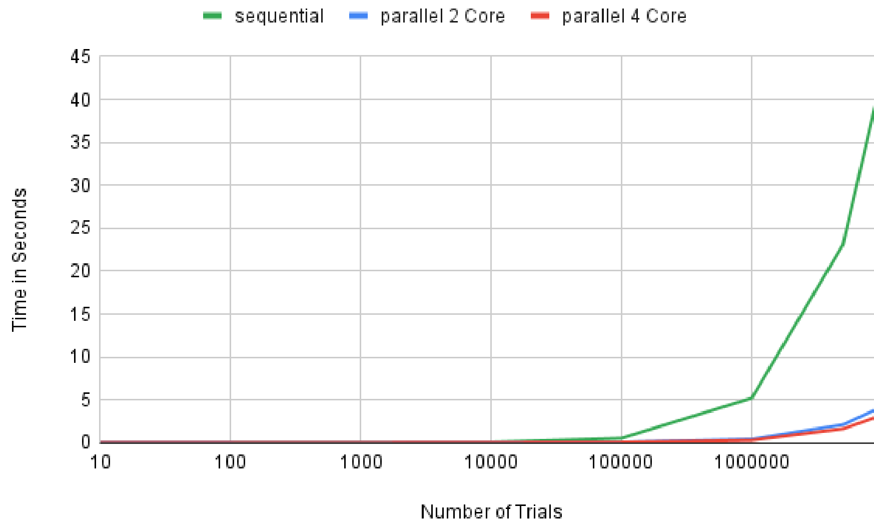
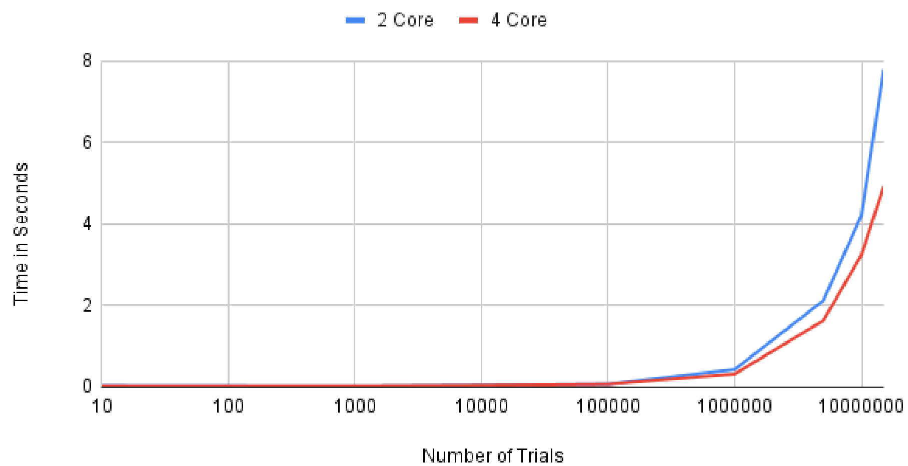
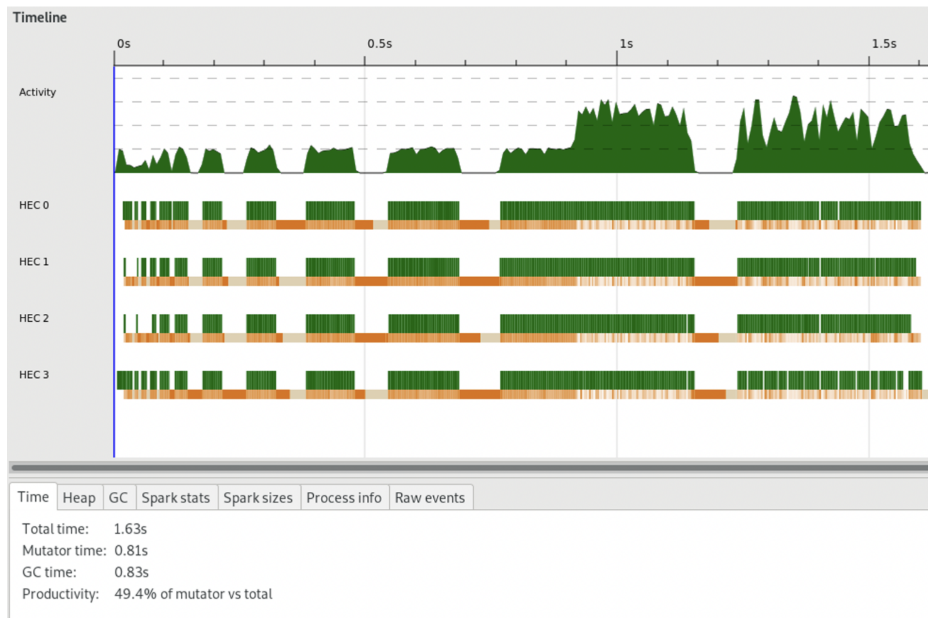



Figure 3: Performance of Sequential versus Parallel




Further, when we look closer at the performance of the parallel implementation on different cores, we saw the program ran faster on more cores. This result was expected considering our intent was to spread the work evenly across different cores, so using more cores was expected to improve performance.
The Threadscope results indicate a significant amount of activity in the later part of execution, indicating when the trials are being run in parallel with the `parListChunk` strategy.

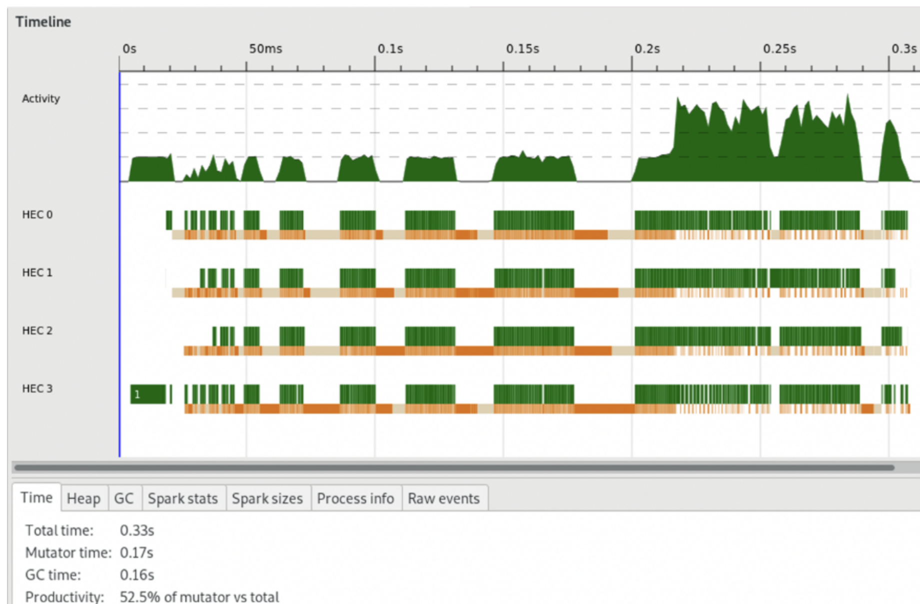



| HEC | Total | Converted | Overflowed | Dud | GC'd | Fizzled |
|---|---|---|---|---|---|---|
| Total | 40 | 40 | 0 | 0 | 0 | 0 |
| HEC 0 | 0 | 13 | 0 | 0 | 0 | 0 |
| HEC 1 | 0 | 11 | 0 | 0 | 0 | 0 |
| HEC 2 | 0 | 12 | 0 | 0 | 0 | 0 |
| HEC 3 | 40 | 4 | 0 | 0 | 0 | 0 |

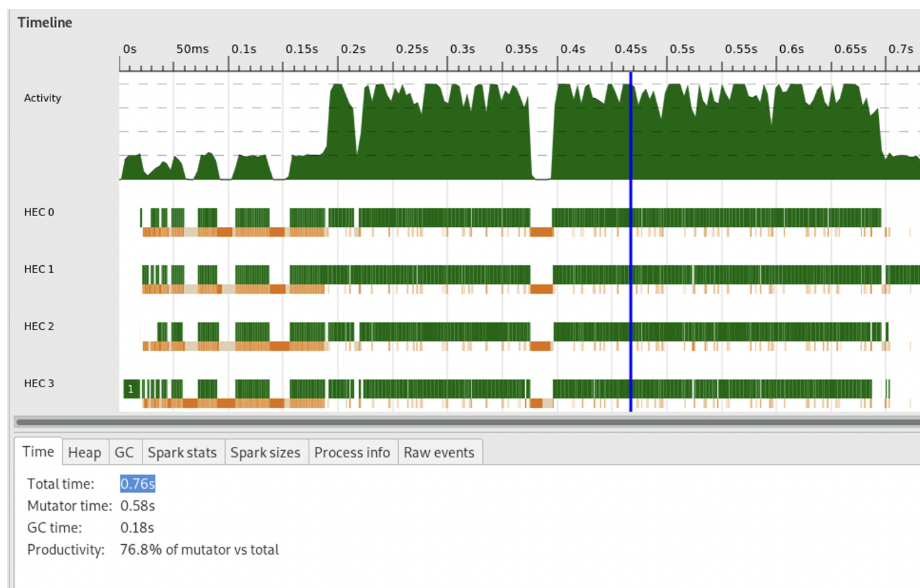
When analyzing the spark stats, all sparks were converted, indicating that the amount of sparks being created are balanced and not adding unnecessary overhead to the program.

Compared to the sequential Threadscope data when run on trials = 50000000 and time steps = 100, the productivity of the parallel version decreased to 49.4%. However, we took a look at what happens when we increase the number of time steps instead of trials, and saw that productivity increased for more time steps.

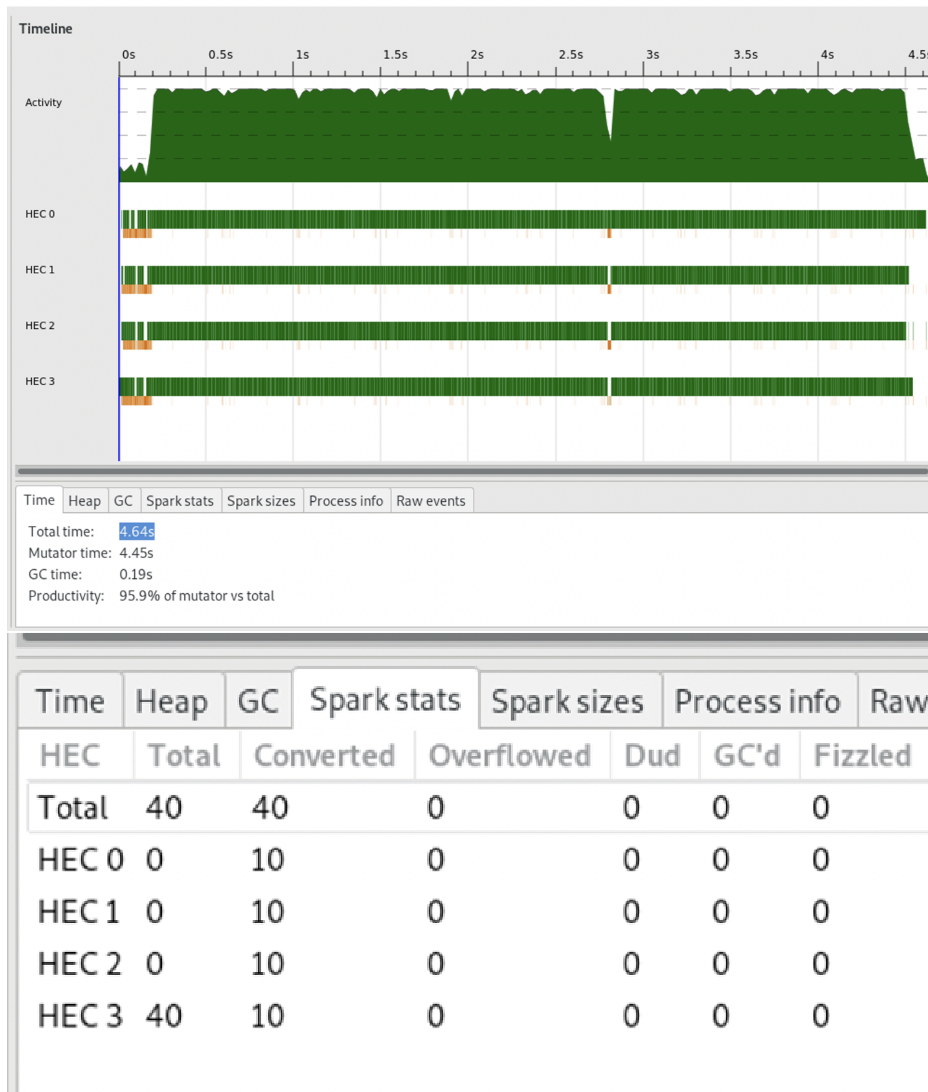
We kept the parameters for r ,u ,d ,s0, and k the same as all previous tests, but we kept n fixed at 1000000 trials and instead modified t, the number of time steps per trial, from 100 to 1000 to 10000. Each test was run on 4 cores.
First we tested t =100




As expected, the productivity of t at 100 is 52.5%, matching the previous test which also used t=100.
Next, we increased time steps, t, to 1000




The results for t=1000 got more interesting. We saw a lot more Activity, peaking at 4 compared to when t = 100. Further, productivity increased from 52.5% to 76.8%. This illustrates how despite each trial having to conduct a more lengthy computation (since the recursive calls to `calcPrice` increase with more time steps), the cost of garbage collection remains relatively the same between the trials.
This relationship continued when we increased t from 1000 to 10000. At t=10000, we were able to take full advantage of the parallelism provided by the strategies




| HEC | Total | Converted | Overflowed | Dud | GC'd | Fizzled |
|---|---|---|---|---|---|---|
| Total | 40 | 40 | 0 | 0 | 0 | 0 |
| HEC 0 | 0 | 10 | 0 | 0 | 0 | 0 |
| HEC 1 | 0 | 10 | 0 | 0 | 0 | 0 |
| HEC 2 | 0 | 10 | 0 | 0 | 0 | 0 |
| HEC 3 | 40 | 10 | 0 | 0 | 0 | 0 |

Activity remained around 4 for the execution of the program, and the spark stats show that the sparks were all converted and evenly distributed between the cores.
By looking at the effect of varying n, the number of trials, and t, the number of steps per trial, we are able to better understand the algorithmic properties of the Monte Carlo Simulation for Options that lend well to parallelism. Since each trial is a similar computation in terms of complexity, they can be effectively distributed and run across cores. Further, increasing t, the depth of the trial path, is what what originally makes Asian options difficult to compute in a deterministic manner; yet, in terms of parallelism, increasing t most effectively takes advantage of the `parChunk` strategy, and led to greater observed productivity in the program.

#### 4.4.2 Performance Compared to Parallel Attempt 2

Finally, it is interesting to consider how a small change in computing the chunk size passed to `parListChunk` changed the performance of the parallel algorithm.
As stated, between parallelization Attempts 2 and 3, we changed the evaluation of `chunkSize` from

```
let chunkSize = n `div` 4
```

to

```
let chunkSize = n `div` (10 * numCores)
```

The Attempt 2 version of the parallel method and the Final version of the parallel method were run with usual stock parameters, trials n = 5000000, and time steps t = 100.

| | |
|---|---|
| Total time: | 3.36s |
| Mutator time: | 2.28s |
| GC time: | 1.07s |
| Productivity: | 68.0% of mutator vs total |

| | |
|---|---|
| Total time: | 1.63s |
| Mutator time: | 0.81s |
| GC time: | 0.83s |
| Productivity: | 49.4% of mutator vs total |

The old parallel implementation (left) took 3.36 seconds while the new implementation took 1.63 seconds, indicating a 2.6x speedup. We believe the poorer performance of the previous implementation was due to a poor balancing of sparks.

Time | Heap | GC | Spark stats | Spark sizes | Process info | Raw eve

| HEC | Total | Converted | Overflowed | Dud | GC'd | Fizzled |
|---|---|---|---|---|---|---|
| Total | 5000004 | 100449 | 4899555 | 0 | 0 | 0 |
| HEC 0 | 0 | 34794 | 0 | 0 | 0 | 0 |
| HEC 1 | 0 | 33167 | 0 | 0 | 0 | 0 |
| HEC 2 | 0 | 32488 | 0 | 0 | 0 | 0 |
| HEC 3 | 5000004 | 0 | 4899555 | 0 | 0 | 0 |

Time | Heap | GC | Spark stats | Spark sizes | Process info | Raw events

| HEC | Total | Converted | Overflowed | Dud | GC'd | Fizzled |
|---|---|---|---|---|---|---|
| Total | 40 | 40 | 0 | 0 | 0 | 0 |
| HEC 0 | 0 | 13 | 0 | 0 | 0 | 0 |
| HEC 1 | 0 | 11 | 0 | 0 | 0 | 0 |
| HEC 2 | 0 | 12 | 0 | 0 | 0 | 0 |
| HEC 3 | 40 | 4 | 0 | 0 | 0 | 0 |

When looking at the spark activity for the old implementation (left), we can see millions of sparks were initialized and most overflowed. In comparison, the new implementation (right) initialized 40 sparks, which is exactly how many chunks we specified the program to divide the trial into. By divides the work into 40 chunks, we were able to more carefully spark the work in parallel, and all 40 sparks were converted.
From these tests, the importance of having a more thoughtful sparking strategy became more clear. While running the strategy on a somewhat arbitrary chunk size increased the speed of the parallel Monte Carlo substantially compared to the sequential, determining a smaller chunk size and based on the number of cores led to run time being halved. A final note about the performance of our parallel implementation, there is clearly two distinct phases of the operation, an initial garbage collection intensive phase and then a more work intensive but less garbage collection phase. This initial phase is the sole bottleneck for performance as while the initialisation of the values of discount and pStar are there it is also definitely the way we are handling the random numbers, while less strictly sequential than using the IO Monad there is still an element there that is holding back performance by a decent bit.

## 5 Bonus Implementation: Vector Operations

Another way to look at these mathematical operations is from the perspective of vectors, since we have a state vector - being the price - that is continually changing throughout the duration of the program. Haskell provides a library full of vector operations and the vector as a data structure called ”Data.Vector”.

### 5.1 Sequential Algorithm

The algorithm for the vectorized sequential version is quite similar to the original sequential version with the main difference being simply that we are using vectors:

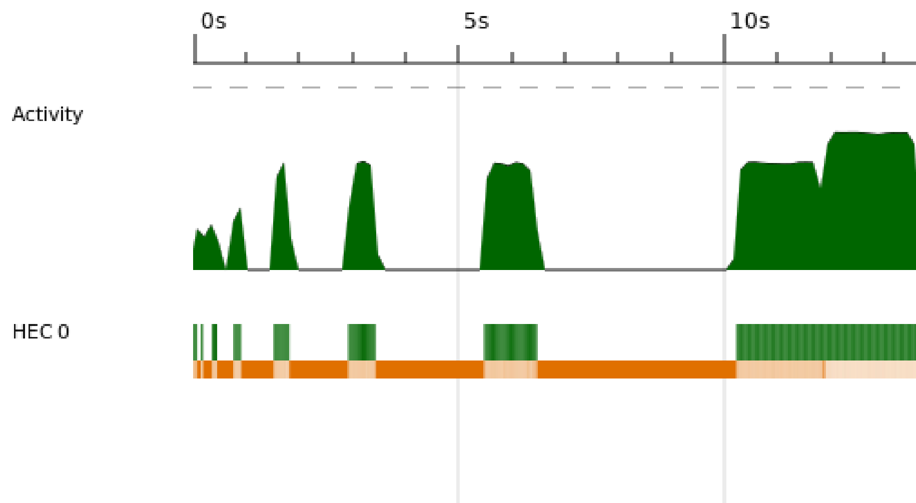
```
monteCarloAsianVector :: Int -> Int -> Double -> Double -> Double -> Double -> Double -> IO Double
monteCarloAsianVector n t r u d s0 k = do
  let discount = 1 / ((1 + r) ** fromIntegral t)
      pStar    = (1 + r - d) / (u - d)

  let vectorTrialSeq = do
        steps <- V.replicateM t (bernoulli pStar)
        let priceVector = V.scanl' (\price step -> price * (if step == 1 then u else d)) s0 steps
            sumPrices   = V.sum priceVector - V.head priceVector
            avgPrice    = sumPrices / fromIntegral t
```

```
        diffVal      = avgPrice - k
      return $ max diffVal 0

  total <- sum <$> replicateM n vectorTrialSeq
  return $ (total * discount) / fromIntegral n
```

The computation of the discount and the `pStar` variables are identical as in the original implementation, we start by generating a vector containing each of the Bernoulli values needed and then we operate on the price, using the values within the steps vector, we then perform the same operations as before, summing up the results, taking the average and calculating the payout.

## 5.2 Sequential Performance

The performance of the vector implementation seems to vary depending on the system and other factors like version of Haskell you are using, for reference the performance figures that will be presented came from a virtual machine running Ubuntu 20.04 rather than the native operating system of the executing machine which was Windows 11. In general it was observed that for a relatively small amount of computations, the vectorized version bested its sequential counterpart however this was a trend that did not last and it will become quite evident why in a moment.




Looking at this Threadscope image the problem becomes quite clear, there is too much overhead and garbage collection takes up a significant amount of the processing time. This is significance is quite large as productivity is only 27.6%.

| | |
|---|---|
| Total time: | 13.76s |
| Mutator time: | 3.80s |
| GC time: | 9.96s |
| Productivity: | 27.6% of mutator vs total |

The function call that resulted in the following information had the following parameters passed along to it. $n = 100000, t = 100, r = 0.05, u = 1.15, d = 1.01, s_0 = 50, k = 70$ There is much to be had when it comes to performance, but with the current implementation those garbage collections instances are not going to go down by much. It is perhaps these garbage collection moments that give reason as to why the algorithm performs noticeably different on different systems, if different operating systems have more elegant ways of handling garbage collection then this waste can be minimized.

## 5.3 Parallel Algorithm

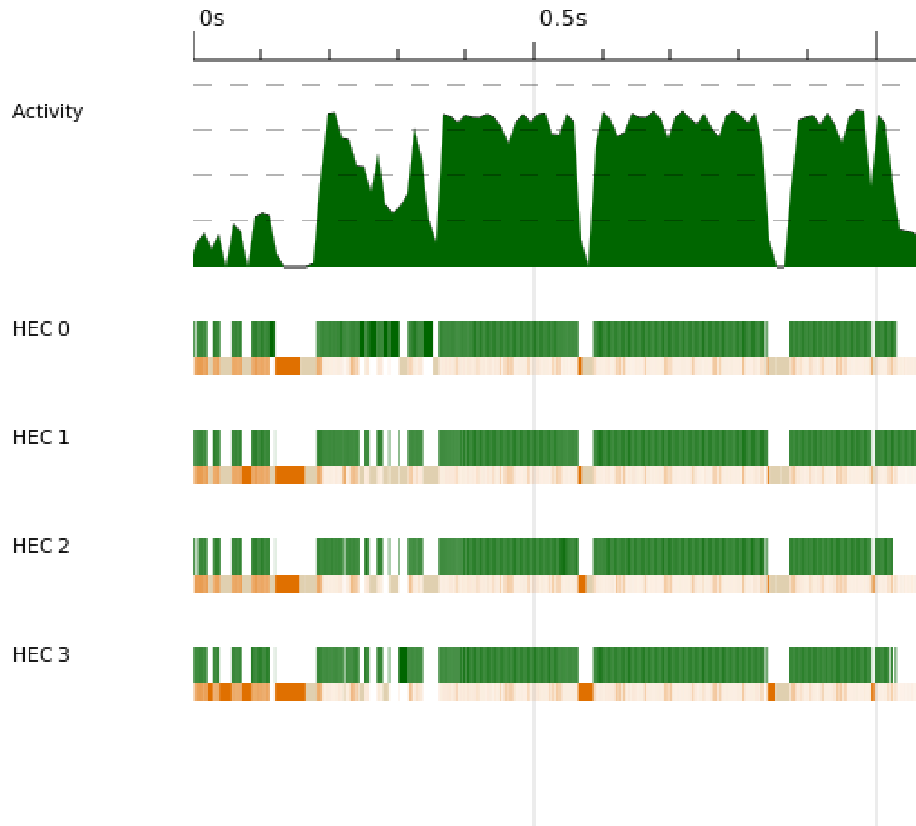
The parallel algorithm takes much inspiration from the previous parallel implementation when vectors were not concerned. This is present in both the modularization of the code and the particular parallelization strategies used to achieve the enhanced performance.

```haskell
monteCarloAsianParallelVector :: Int -> Int -> Int -> Double -> Double -> Double -> Double -> \
Double -> SMGen -> Double
monteCarloAsianParallelVector numCores n t r u d s0 k init_gen =
  let !discount = 1 / ((1 + r) ** fromIntegral t)
      !pStar = (1 + r - d) / (u - d)
      chunkSize = n `div` (10 * numCores)
      gens = unfoldsSMGen init_gen n
      trials = withStrategy (parListChunk chunkSize rdeepseq) $
               map (runEval . vectorTrial pStar u d s0 k t) gens
      !result = sum trials * discount / fromIntegral n
  in result

vectorTrial :: Double -> Double -> Double -> Double -> Double -> Int -> SMGen -> Eval Double
vectorTrial pStar u d s0 k t gen = do
  let steps = V.unfoldrN t (unfoldBernoulli pStar) gen
      priceVector = V.scanl' (\price step -> price * (if step == 1 then u else d)) s0 steps
      sumPrices = V.sum priceVector - V.head priceVector
      avgPrice = sumPrices / fromIntegral t
      diffVal = avgPrice - k
  return $ max diffVal 0

unfoldBernoulli :: Double -> SMGen -> Maybe (Int, SMGen)
unfoldBernoulli p gen = Just $ bernoulliParallel p gen
```

We used the same chunking strategy that we used to implement the original parallel implementation and this runs the vector trials in chunks that are dependent on the number of cores, as mentioned earlier this strategy of deriving the chunk size based on the core count rather than a static value helps with balancing when more cores are introduced.

### 5.4 Parallel Performance

As soon as we ran a single multi-threaded test with this new parallelized algorithm we knew that there was a massive speed up and this is supported by the details presented in threadscope.

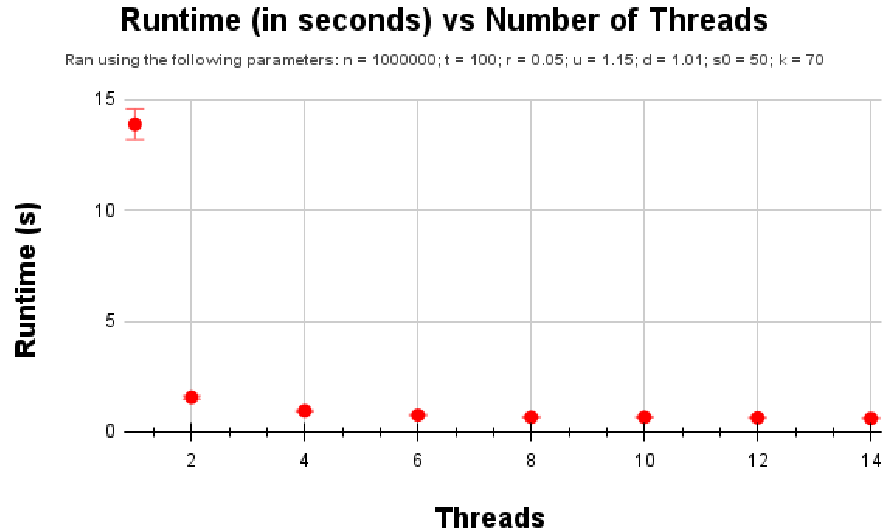



The figure illustrates a massive decrease in runtime as well as a very noticeable decrease in the time spent handling garbage collection, it is important to note that this is purely algorithmic as the test that produced the above figure was ran immediately after the test for the sequential vector algorithm. We can also see that there is a massive increase in productivity as well from the following figure:

Time Heap GC Spark stats Spark size

Total time: 1.07s
Mutator time: 0.76s
GC time: 0.31s
Productivity: 71.2% of mutator vs total

A jump to 71.2% percent in productivity from 27.6% is a 2.6x increase in productivity, when it comes to the actual runtime there was a 14.6x speedup. The following graphic shows the runtime against the number of threads and it can be seen that there is a massive speedup from the initial parallelization but as more threads are added the benefits tapper off, which is expected. There is also the consideration of substantial algorithmic differences which are also definitely contributing to the increased performance.




The associated error on the figures comes from running a few iterations of each test and then determining the spread of those values and determining the median and the low and high values and what percentage of the median value the difference was.

### 5.5 Potential Performance Enhancing Alternatives

Of course this is not the absolute best solution for parallelizing vector operations in Haskell. Viable alternative approaches to parallelization include using the Repa library (Regular Parallel Arrays) contained with Data.Array.Repa to model the simulation and perform operations on those. It is a good option but not the approach we went with for this analysis. Repa comes with builtin parallelization functions and when used properly could provide some significant performance enhancements. The potential peak for performance with these vector operations is found within the Accelerate library that is associated with Haskell, when using accelerate at its best which is with using a cuda supported GPU as the backend for your code, the number of threads jumps from ˜16 to over 1000 and maybe even more depending on the specific NVIDIA GPU you have within your system. The only problem is that accelerate is not kept up to date with the standard Haskell programming language or any of its main associated tool like stack and cabal, so in order to use it you need to have the conditions set just right within your system to get it to even compile properly. This very issue is what led us to creating VMs in the first place as we did attempt to experiment with what Accelerate had to offer and we got as far as enabling it to compile with an enhanced CPU backend. The new syntax, lengthy documentation, and the limit on the available time we had to spend on this particular portion led to us abandoning it, the decision to cease development of an accelerated version of our algorithm was made even more rational considering the fact that the cuda development kit necessary to even access the GPU in the first place was simply not working well with one of our VM environments, so much so that the VM had to be recreated 2 times due to attempts to install the kit corrupting vital parts of the system.

## 6 Conclusion: Putting it All Together

At this point we have seen 4 distinct implementations of the Monte Carlo simulations involving pricing Asian call options: A Standard Sequential Algorithm, A Standard Parallel Algorithm, A Vectorized Sequential Algorithm, and A Vectorized Parallel Algorithm. We saw excellent performance gains when we transitioned from both sequential algorithms to their parallel counterparts. The gain in performance was the highest in the vectorized form but despite these gains it still didn't pull ahead of the standard parallel algorithm, we believe this to be due to the overhead involved with Haskell data structures as those require extra memory that will eventually need to be reclaimed by the computer at some point in the life span of our program. Chances are had we managed to actually program the GPU using the accelerate library then that implementation would have blown the other implementations out of the water. Overall Haskell does make it relatively straightforward to parallelize your programs in order to make a better use of your computer's processing power, but it also very much helped that Monte Carlo simulations by design are able to be parallelized to such a significant extent that it makes the efforts put forth while in our attempts to parallelize the algorithm very much worth it.

# 7 Haskell Code Reference

```haskell
-- Library.hs

{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE ScopedTypeVariables #-}

module Library (bernoulli, monteCarloAsian, validateInputs, monteCarloAsianParallel,
bernoulliParallel, unfoldsSMGen, monteCarloAsianVector, monteCarloAsianParallelVector,
unfoldBernoulli, vectorTrial) where

import System.Random
import System.Random.SplitMix
import qualified Data.Vector as V
import Control.Monad (replicateM, unless, when)
import Control.Parallel.Strategies (rdeepseq, Eval, parListChunk, runEval, withStrategy)

-- Sequential Content Begins --
monteCarloAsian :: Int -> Int -> Double -> Double -> Double -> Double -> Double -> IO Double
monteCarloAsian n t r u d s0 k = do
  -- Calculate discount factor and probability of an up movement
  let discount = 1 / ((1 + r) ^ t)
      pStar    = (1 + r - d) / (u - d)

  -- Perform a single trial of the simulation
  let seqTrial = do
        -- Recursively calculate the sum of prices along a path
        let seqCalcPrice i sumPrices price
              | i == t = return sumPrices
              | otherwise = do
                b <- bernoulli pStar
                if b == 1
                    then seqCalcPrice (i + 1) (sumPrices + (price * u)) (price * u)
                    else seqCalcPrice (i + 1) (sumPrices + (price * d)) (price * d)

        -- Calculate the difference between average simulate price and strike price
        sumPrices <- seqCalcPrice 0 0.0 s0
        let diffVal = (sumPrices / fromIntegral t) - k
        return $ diffVal `max` 0

  -- Perform 'n' trials and compute the average
  total <- sum <$> replicateM n seqTrial
  return $ (total * discount) / fromIntegral n

-- Main function for sequential vectors
monteCarloAsianVector :: Int -> Int -> Double -> Double -> Double -> Double -> Double -> IO Double
monteCarloAsianVector n t r u d s0 k = do
  let discount = 1 / ((1 + r) ** fromIntegral t)
      pStar    = (1 + r - d) / (u - d)

  let vectorTrialSeq = do
        steps <- V.replicateM t (bernoulli pStar)
        let priceVector = V.scanl' (\price step -> price * (if step == 1 then u else d)) s0 steps
            sumPrices   = V.sum priceVector - V.head priceVector
            avgPrice    = sumPrices / fromIntegral t
            diffVal     = avgPrice - k
```

```
      return $ max diffVal 0

  total <- sum <$> replicateM n vectorTrialSeq
  return $ (total * discount) / fromIntegral n

-- Helper function for sequential algorithm to generate random values
bernoulli :: Double -> IO Int
bernoulli p = do
    randomVal <- randomIO -- :: IO Double
    return $ if randomVal < p then 1 else 0

-- Sequential Content Ends --
-- Parallel Content Begins --

-- Main function for the parallel simulations
monteCarloAsianParallel :: Int -> Int -> Int -> Double -> Double -> Double -> Double -> Double -> SMGen ->
monteCarloAsianParallel numCores n t r u d s0 k init_gen =
  let !discount = 1 / ((1 + r) ^ t)
      !pStar = (1 + r - d) / (u - d)
      chunkSize = n `div` (10 * numCores)
      gens = unfoldsSMGen init_gen n
      trials = withStrategy (parListChunk chunkSize rdeepseq) $
               map (runEval . trial pStar u d s0 k t) gens
      !result = sum trials * discount / fromIntegral n
  in result

-- Helper function to unfold SMGen into a list of n generators
unfoldsSMGen :: SMGen -> Int -> [SMGen]
unfoldsSMGen gen n = take n $ iterate (snd . splitSMGen) gen

-- Helper function to complete a single trial
trial :: Double -> Double -> Double -> Double -> Double -> Int -> SMGen -> Eval Double
trial pStar u d s0 k t genTrial = do
  let (!sumPrices, _) = calcPrice 0 t pStar u d 0 s0 genTrial
      !diffVal = (sumPrices / fromIntegral t) - k
  return $ max diffVal 0

-- Helper function to calculate the price in a given trial
calcPrice :: Int -> Int -> Double -> Double -> Double -> Double -> Double -> SMGen -> (Double, SMGen)
calcPrice i t pStar u d !sumPrices !price genCalc
  | i == t    = (sumPrices, genCalc)
  | otherwise = let (b, genNext) = bernoulliParallel pStar genCalc
                    (!newPrice, newGen) = if b == 1
                                          then (price * u, genNext)
                                          else (price * d, genNext)
                in calcPrice (i + 1) t pStar u d (sumPrices + newPrice) newPrice newGen

-- Helper function to generate a Bernoulli trial result given a probability and a generator
bernoulliParallel :: Double -> SMGen -> (Int, SMGen)
bernoulliParallel p gen = let (!randomVal, gen') = nextDouble gen
                          in (if randomVal < p then 1 else 0, gen')

-- Main function for vectors in parallel
monteCarloAsianParallelVector :: Int -> Int -> Int -> Double -> Double -> Double -> Double -> Double -> SM
monteCarloAsianParallelVector numCores n t r u d s0 k init_gen =
  let !discount = 1 / ((1 + r) ** fromIntegral t)
```

```
      !pStar = (1 + r - d) / (u - d)
      chunkSize = n `div` (10 * numCores)
      gens = unfoldsSMGen init_gen n
      trials = withStrategy (parListChunk chunkSize rdeepseq) $
               map (runEval . vectorTrial pStar u d s0 k t) gens
      !result = sum trials * discount / fromIntegral n
  in result

-- Helper function to complete a single vector trial
vectorTrial :: Double -> Double -> Double -> Double -> Double -> Int -> SMGen -> Eval Double
vectorTrial pStar u d s0 k t gen = do
  let steps = V.unfoldrN t (unfoldBernoulli pStar) gen
      priceVector = V.scanl' (\price step -> price * (if step == 1 then u else d)) s0 steps
      sumPrices = V.sum priceVector - V.head priceVector
      avgPrice = sumPrices / fromIntegral t
      diffVal = avgPrice - k
  return $ max diffVal 0

-- Helper function to generate the bernoullis
unfoldBernoulli :: Double -> SMGen -> Maybe (Int, SMGen)
unfoldBernoulli p gen = Just $ bernoulliParallel p gen
-- Parallel Content Ends --

-- General All purpose helper functions below --
-- Helper function to validate the provided inputs from the user
validateInputs :: Int -> Int -> Double -> Double -> Double -> Double -> Double -> IO ()
validateInputs n t r u d s0 k = do
  when (n <= 0) $ error "Invalid value for n. Number of trials (n) must be greater than 0."
  when (t < 1) $ error "Invalid value for t. Number of time steps (t) must be greater than or equal to 1."
  when (r <= 0) $ error "Invalid value for r. The interest rate (r) must be greater than 0."
  when (u <= 0) $ error "Invalid value for u. The up factor (u) must be greater than 0."
  when (d <= 0) $ error "Invalid value for d. The down factor (d) must be greater than 0."
  when (s0 <= 0) $ error "Invalid value for s0. Initial stock price (s0) must be greater than 0."
  when (k <= 0) $ error "Invalid value for k. Strike price (k) must be greater than 0."
  unless (0 < d && d < 1 + r && 1 + r < u) $
    error "Invalid values for r, u, and d entered.\nThe relationship 0 < d < r < u must be maintained to g

-- Main.hs
module Main (main) where

import Library
import Control.Concurrent (getNumCapabilities)
import System.Random.SplitMix
import Data.Time

{- |
Entry point for the system asks the user to enter different quantities
and then validates the input prior to execution.
-}
main :: IO ()
main = do
    putStrLn "Enter the number of simulations (n):"
    n <- read <$> getLine

    putStrLn "Enter the number of time steps (t):"
    t <- read <$> getLine
```

```haskell
putStrLn "Enter the interest rate (r):"
r <- read <$> getLine

putStrLn "Enter the up factor (u):"
u <- read <$> getLine

putStrLn "Enter the down factor (d):"
d <- read <$> getLine

putStrLn "Enter the initial stock price (s0):"
s0 <- read <$> getLine

putStrLn "Enter the strike price (k):"
k <- read <$> getLine

validateInputs n t r u d s0 k

-- sequential non vector
putStrLn "Sequential Monte Carlo Simulation:"
start <- getCurrentTime
resultAsian <- monteCarloAsian n t r u d s0 k
stop <- getCurrentTime
let timeDiff = diffUTCTime stop start
putStrLn $ "Result Monte Carlo Asian Option [Sequential]: " ++ show resultAsian
putStrLn $ "Time to Run: " ++ show timeDiff

-- parallel non vector
coreCount <- getNumCapabilities
initGen <- initSMGen
start' <- getCurrentTime
let resultPar = monteCarloAsianParallel coreCount n t r u d s0 k initGen
stop' <- getCurrentTime
let timeDiff' = diffUTCTime stop' start'
putStrLn $ "Result Monte Carlo Asian Option [Parallel]: " ++ show resultPar
putStrLn $ "Time to Run: " ++ show timeDiff'

-- sequential vector
putStrLn "Sequential Monte Carlo Simulation Vector:"
start'' <- getCurrentTime
resultAsianVec <- monteCarloAsianVector n t r u d s0 k
stop'' <- getCurrentTime
let timeDiff'' = diffUTCTime stop'' start''
putStrLn $ "Result Monte Carlo Asian Option [Sequential Vector]: " ++ show resultAsianVec
putStrLn $ "Time to Run: " ++ show timeDiff''

-- parallel vector
putStrLn "Monte Carlo Simulation Parallel Vector:"
start''' <- getCurrentTime
let resultAsianPA = monteCarloAsianParallelVector coreCount n t r u d s0 k initGen
stop''' <- getCurrentTime
let timeDiff''' = diffUTCTime stop''' start'''
putStrLn $ "Result Monte Carlo Asian Option [Parallel Vector]: " ++ show resultAsianPA
putStrLn $ "Time to Run: " ++ show timeDiff'''
```

```haskell
-- Spec.hs
import Library
import Test.HUnit
import System.Random.SplitMix
import Control.Concurrent (getNumCapabilities)


{- |
Test Suite for the Monte Carlo Simulations functions contained within the Library module.

Precondition: HUnit library is installed, use stack install hunit to confirm installation or proceed with
Postcondition: All Tests run successfully.
-}

test1 :: Test
test1 = TestCase $ do
  let n = 10000
      t = 10
      r = 0.05
      u = 1.15
      d = 1.01
      s0 = 50
      k = 70

  result <- monteCarloAsian n t r u d s0 k

  let lowerBound = 0.9
      upperBound = 1.4

  assertBool "Result is within bounds" (result >= lowerBound && result <= upperBound)

test2 :: Test
test2 = TestCase $ do
  let n = 10000
      t = 10
      r = 0.05
      u = 1.15
      d = 1.01
      s0 = 50
      k = 70

  result <- monteCarloAsianVector n t r u d s0 k

  let lowerBound = 0.9
      upperBound = 1.4

  assertBool "Result is within bounds" (result >= lowerBound && result <= upperBound)

test3 :: Test
test3 = TestCase $ do
  let n = 10000
      t = 10
      r = 0.05
      u = 1.15
      d = 1.01
      s0 = 50
```

```
      k = 70

  numCores <- getNumCapabilities
  initGen <- initSMGen
  let result = monteCarloAsianParallel numCores n t r u d s0 k initGen

  let lowerBound = 0.9
      upperBound = 1.4

  assertBool "Result is within bounds" (result >= lowerBound && result <= upperBound)

test4 :: Test
test4 = TestCase $ do
  let n = 10000
      t = 10
      r = 0.05
      u = 1.15
      d = 1.01
      s0 = 50
      k = 70

  numCores' <- getNumCapabilities
  initGen' <- initSMGen
  let result = monteCarloAsianParallelVector numCores' n t r u d s0 k initGen'

  let lowerBound = 0.9
      upperBound = 1.4

  assertBool "Result is within bounds" (result >= lowerBound && result <= upperBound)


tests :: Test
tests = TestList [TestLabel "Sequential Test [Non Vector]" test1, TestLabel "Sequential Test [Vector]" tes

main :: IO ()
main = do
  results <- runTestTT tests
  print results
```

# 8 References

[1] *Control-Concurrent*. Hackage. URL: https://hackage.haskell.org/package/base-4.19.0.0/docs/Control-Concurrent.html#v:getNumCapabilities.

[2] Bryan O'Sullivan. *Real World Haskell*. O'Reilly, 2008, pp. 531–560.

[3] Nicholas Privault. *Asian Options*. 2023, pp. 1–3.

[4] *Random Benchmarks*. Alexey Kuleshevi. URL: https://alexey.kuleshevi.ch/blog/2019/12/21/random-benchmarks/.

[5] *What Are Asian Options and How Are They Priced?* SoFi. URL: https://www.sofi.com/learn/content/asian-option/#:~:text=How%20Asian%20Options%20Work,a%20specified%20period%20of%20time..

[6] *What is The Monte Carlo Simulation?* Amazon Web Services. URL: https://aws.amazon.com/what-is/monte-carlo-simulation/#:~:text=The%20Monte%20Carlo%20simulation%20is,on%20a%20choice%20of%20action.