Parallel Implementation of Basket Option Pricing

Ya-Wei Tsai

November 29, 2024

1 Introduction

1.1 Overview

This project implements a computational system for **Basket Option Pricing** using Monte Carlo simulations. Basket options are financial derivatives whose payoff depends on the weighted average performance of a portfolio of underlying assets. These derivatives are widely used in financial markets for hedging and speculative purposes.

The system supports both **call options** and **put options**:

- **Call Option:** The holder has the right (but not the obligation) to buy the basket of assets at a specified strike price (K).
- **Put Option:** The holder has the right (but not the obligation) to sell the basket of assets at a specified strike price (K).

The program calculates the fair price of these options using Monte Carlo simulations, employing both sequential and parallel implementations to optimize the computational process.

1.2 Theoretical Background

The payoff of a basket option depends on the weighted average of the prices of m assets in the basket at maturity T. For a **call option**, the payoff P is:

$$P = \max\left(W_T - K, 0\right),$$

and for a **put option**, the payoff is:

$$P = \max\left(K - W_T, 0\right),$$

where W_T is the weighted basket value at maturity:

$$W_T = \sum_{i=1}^{m} w_i S_i(T),$$

with w_i being the weight of asset i in the basket and $S_i(T)$ being the price of asset i at maturity T.

The fair price of the basket option is the discounted expected value of the payoff:

$$\text{Option Price} = e^{-rT}\mathbb{E}[P],$$

where r is the risk-free rate, and $\mathbb{E}[P]$ is the expected payoff computed using Monte Carlo simulations.

1.3 Why Monte Carlo Simulation?

Closed-form solutions (e.g., the Black-Scholes formula) exist for simple European options but become infeasible for basket options due to:

- The weighted average introduces dependencies between multiple assets.
- Different volatilities (σ_i) and expected returns (μ_i) for each asset.
- Non-linear payoffs (max function) and path dependencies.

Monte Carlo simulations are well-suited for such scenarios as they approximate the expected payoff through random sampling, allowing flexibility in modeling complex distributions.

1.4 Why Parallel Computation?

Monte Carlo simulations involve a large number of independent trials (N), where each trial calculates a payoff based on a random price path. This independence makes the problem *embarrassingly parallel*, as each trial can be computed in isolation. Parallelizing the computation:

- Reduces overall execution time by distributing tasks across multiple threads.
- Makes the method scalable for large simulations ($N > 10^6$).

1.5 Assumptions in the Model

To simplify the implementation:

- We assume the assets in the basket are **independent**. This eliminates the need for a correlation matrix and the associated Cholesky decomposition.
- Asset prices follow a **geometric Brownian motion**:

$$S_i(t + \Delta t) = S_i(t) \exp\left[\left(\mu_i - \frac{\sigma_i^2}{2}\right)\Delta t + \sigma_i\sqrt{\Delta t}Z\right],$$

  where $Z \sim \mathcal{N}(0, 1)$ is a standard normal random variable.

2 Parallelization Approach and Justification

2.0.1 Basic Parallel Implementation

The basic parallel implementation uses a hybrid approach combining the **BSP (Bulk Synchronous Parallel)** model and **pipelining**. The key design aspects are:

- **Task Distribution:** Tasks (simulations) are distributed across threads using a shared task channel. Each thread computes its assigned simulations and sends results back to the main thread using a result channel.
- **BSP Characteristics:** The implementation is structured into supersteps:

1. Task Assignment: Simulations are distributed to threads via a channel.
2. Local Computation: Each thread independently computes price paths and payoffs for its subset of tasks.
3. Result Aggregation: Results are collected and combined by the main thread after all threads finish their computations.

Synchronization occurs at the end of each superstep, ensuring that all threads complete their current tasks before moving to the next stage (e.g., final aggregation).

- **Pipelining Aspects:** The use of channels enables task distribution and result collection to overlap with computation, optimizing throughput.
- **Why BSP and Pipelining are Appropriate:**
  - **Embarrassingly Parallel Nature:** The task of simulating price paths is highly independent, making BSP an ideal choice for dividing work into supersteps.
  - **Low Communication Requirements:** Channels effectively reduce contention by allowing threads to fetch tasks and submit results asynchronously.
  - **Simple Load Balancing:** Static allocation of N/T tasks per thread ensures balanced workloads for uniform simulations.
- **Limitations:** Static task allocation is inefficient for non-uniform workloads, leading to idle threads when task complexities vary.

2.0.2 Work-Stealing Implementation

The work-stealing implementation introduces dynamic load balancing to address the limitations of static task allocation:

- **Task Distribution:** Each thread maintains a local double-ended queue (*deque*) of tasks. Threads prioritize tasks from their own queue but can "steal" tasks from other threads when their queue is empty.
- **Pipelining Aspects:** Channels are used to collect results from threads dynamically, overlapping task execution and result aggregation.
- **Why Work-Stealing is Appropriate:**
  - **Dynamic Load Balancing:** The ability to redistribute tasks among threads ensures that all threads remain active, even for non-uniform workloads.
  - **Improved Resource Utilization:** By reducing idle time, work-stealing maximizes throughput and achieves better performance for larger simulations.
- **Challenges:**
  - Increased synchronization overhead from mutexes and task stealing.
  - Slightly higher latency due to contention for shared resources (e.g., deques).

2.1 Load Balancing, Latency, and Throughput

- **Load Balancing:**
  - The basic parallel implementation provides simple load balancing by evenly dividing tasks among threads. However, this static approach can result in idle threads if workloads are non-uniform.
  - The work-stealing implementation dynamically redistributes tasks, achieving better load balancing, especially for large-scale simulations.
- **Latency:**
  - Channels introduce minimal latency for communication between threads in the basic parallel implementation.
  - In work-stealing, mutexes and deque operations add synchronization overhead, slightly increasing latency.
- **Throughput:**
  - The basic parallel implementation achieves high throughput for uniform workloads but struggles with imbalanced task complexities.
  - The work-stealing implementation improves throughput by keeping all threads active, even for non-uniform tasks.

2.2 Challenges Faced

- **Random Number Generation:** Ensuring thread-safe and independent random number streams was critical. Sharing a single random generator could lead to data races. The solution was to assign a unique random generator to each thread using independent seeds.
- **Task Distribution:** Static task allocation in the basic parallel implementation led to imbalances for non-uniform workloads. Work-stealing resolved this issue but introduced additional synchronization overhead.
- **Synchronization Overhead:** Both implementations used channels and mutexes, which introduced latency, particularly for small simulation sizes where thread overhead dominated.
- **Hotspots and Bottlenecks:** The sequential code's bottlenecks included:
  - Initialization of random number generators.
  - Aggregation of results, which required sequential accumulation of payoffs.

  Despite these bottlenecks, the primary hotspot (simulation of price paths and payoff calculation) was successfully parallelized.

3 Program Execution

3.1 Input Portfolio File

The system takes a CSV file containing portfolio details as input. The file specifies the assets in the portfolio and their associated parameters, such as initial price, weight, mean return, and volatility. Below is an example of the `portfolio_test.csv` file used in the experiments:

```
Asset,InitialPrice,Weight,MeanReturn,Volatility
AAPL,150,0.4,0.08,0.2
MSFT,320,0.3,0.07,0.18
TSLA,800,0.2,0.1,0.3
GOOGL,2800,0.1,0.06,0.15
```

3.1.1 Description of Columns

- **Asset:** The name or ticker symbol of the asset (e.g., AAPL for Apple, MSFT for Microsoft).
- **InitialPrice:** The starting price of the asset.
- **Weight:** The proportion of the portfolio allocated to this asset (should sum to 1 for all assets).
- **MeanReturn:** The expected annual return of the asset (in decimal form, e.g., 0.08 for 8%).
- **Volatility:** The annualized standard deviation of the asset's returns (in decimal form).

The portfolio file allows the system to simulate asset price paths and calculate the basket option payoff based on the specified parameters.

3.2 Cluster Execution

To execute the program on a cluster, use the following command:

```
sbatch run_experiments.sh
```

This command will:

- Generate speedup graphs and save them in the `results` folder.
- Write the simulation output and execution logs to the `slurm/out` folder.

The `generate_time_plots.py` script can be modified to adjust simulation parameters for generating different speedup graphs.

3.3 Standalone Execution

To execute the program locally or individually (not using the batch script), you can directly run the Go program by specifying the required parameters. The Go program supports the following parameters:

- `mode`: Execution mode. Options are `sequential`, `parallel`, or `parallel-stealing`.
- `portfolio`: Path to the portfolio CSV file.
- `K`: Strike price (float).
- `r`: Risk-free interest rate (float).
- `T`: Time to maturity in years (float).
- `steps`: Number of simulation steps (integer).
- `simulations`: Number of Monte Carlo simulations (integer).
- `threads`: Number of threads for parallel execution (integer).
- `type`: Option type. Options are `call` or `put`.

3.4 Example Usage

Here are some example commands for standalone execution:

```
# Sequential execution
go run main.go --mode=sequential --portfolio=data/portfolio_test.csv --K=700 --r=0.05
               --T=1.0 --steps=252 --simulations=10000 --type=call

# Parallel execution with 4 threads
go run main.go --mode=parallel --portfolio=data/portfolio_test.csv --K=700 --r=0.05 \
               --T=1.0 --steps=252 --simulations=10000 --threads=4 --type=put

# Parallel work-stealing execution with 8 threads
go run main.go --mode=parallel-stealing --portfolio=data/portfolio_test.csv --K=700 \
               --r=0.05 --T=1.0 --steps=252 --simulations=1000000 --threads=8 \
               --type=call
```

3.5 Output Details

- The speedup graphs will be saved as PNG files in the `results` folder.
- Execution logs and runtime information will be stored in the `slurm/out` folder if executed on a cluster.

4 Performance Evaluation

4.1 Experimental Setup

- **System:** Experiments were run on the Peanut cluster.
- **Simulations:** $N = \{100,000; 1,000,000; 10,000,000\}$.
- **Threads:** $T = \{2, 4, 6, 8, 12\}$.
- **Implementations:** Sequential, Basic Parallel, and Work-Stealing Parallel.

4.2 Speedup Analysis

4.2.1 Speedup for 100,000 Simulations

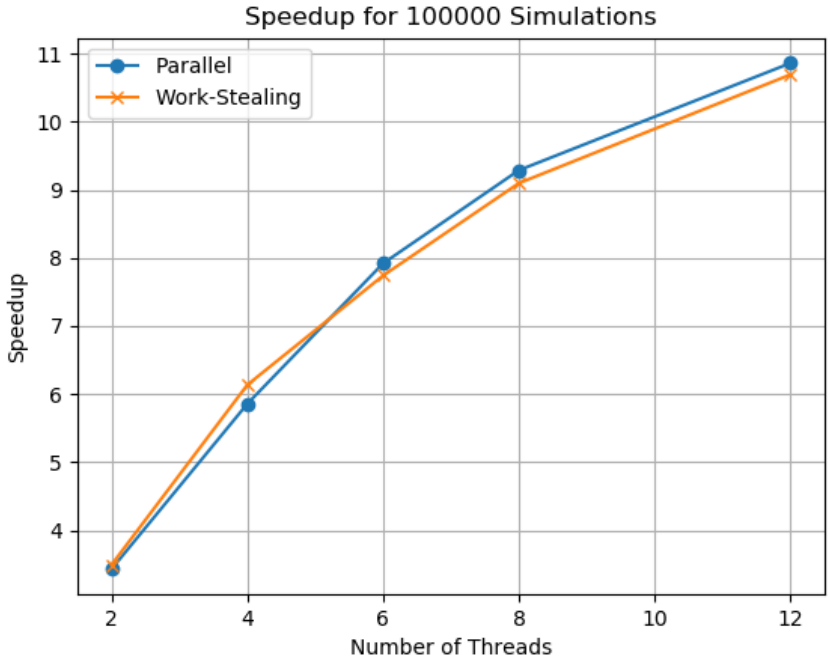

Figure 1: Speedup for 100,000 simulations.

Observation: Both the basic parallel implementation and work-stealing achieve near-linear speedup with an increasing number of threads. However, there is no significant difference between the two implementations for this smaller workload.

Analysis: The computational overhead of task stealing is negligible when the workload is light and well-balanced. Both implementations distribute tasks efficiently across threads, leading to similar performance.

4.2.2 Speedup for 1,000,000 Simulations

Observation: Both implementations achieve linear scalability, with work-stealing slightly outperforming basic parallelism as the thread count increases.

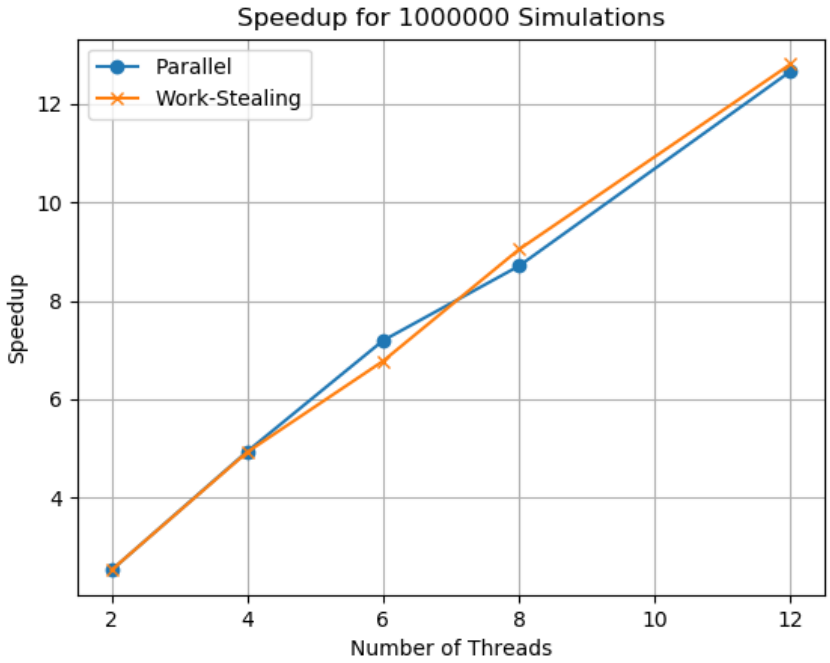

Figure 2: Speedup for 1,000,000 simulations.

Analysis: The larger workload allows work-stealing to shine due to dynamic load balancing, especially in scenarios where certain threads complete their tasks early and can steal tasks from others. This ensures that all threads remain active throughout the simulation process.

4.2.3 Speedup for 10,000,000 Simulations

Observation: Basic parallel implementation exhibits higher speedup with 12 threads, while work-stealing slightly lags behind, especially with 8 or more threads.

Analysis: The overhead of managing task queues and synchronization in the work-stealing implementation becomes more pronounced as the thread count increases. For a massive workload, basic parallelism's simplicity and static task allocation reduce communication overhead, making it more efficient. However, work-stealing remains competitive due to its robustness in dynamically balancing uneven workloads.

4.3 Conclusion from Speedup Analysis

The experimental results show that:

- **Smaller Workloads:** Both parallel implementations perform similarly, as the workload is evenly distributed and the computational overhead is minimal.
- **Medium Workloads:** Work-stealing outperforms basic parallelism due to better load balancing and reduced idle time for threads.
- **Large Workloads:** Basic parallelism slightly outperforms work-stealing, as the additional synchronization cost in work-stealing becomes a limiting factor.

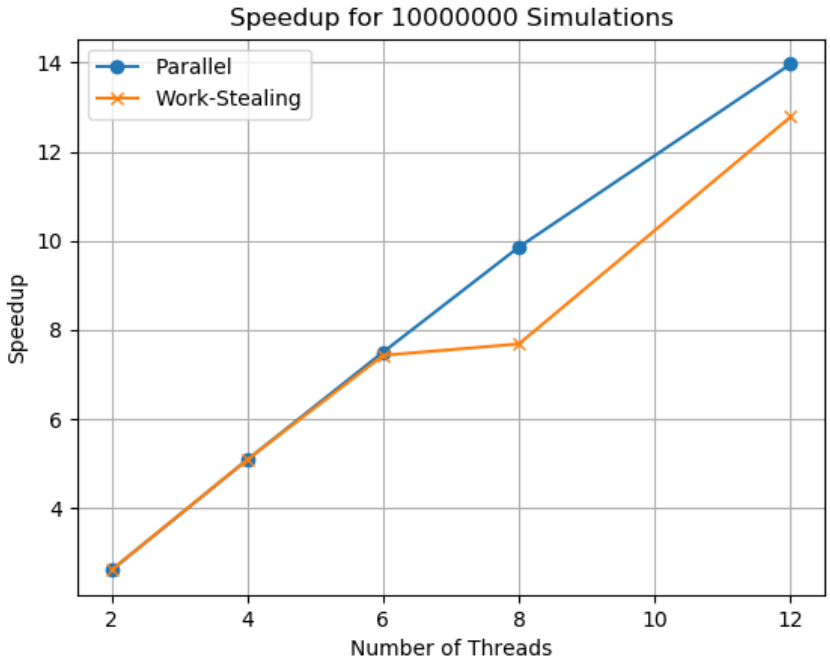

Figure 3: Speedup for 10,000,000 simulations.

These findings align with the expectation that the choice of parallelization strategy depends on workload size and computational complexity.

5 Conclusions

- Monte Carlo simulations for basket option pricing benefit significantly from parallelization, particularly for large workloads.
- The basic parallel implementation is effective for uniform workloads but suffers from load imbalance for non-uniform tasks.
- The work-stealing implementation addresses these limitations, achieving better load balancing and higher speedups for large-scale simulations.
- Future improvements could include reducing synchronization overhead, exploring hybrid parallelization techniques, and leveraging GPU acceleration for further optimization.