**PARALLEL AND DISTRIBUTED COMPUTING PROJECT**

**Monte Carlo Simulation**

**Group members:**

**Ismail Awan (22k-4225)**

**Ayan Ejaz (22k-4222)**

**Ahan Ali (22k-4180)**

**INTRODUCTION:**

Monte Carlo simulations are widely used for modeling complex systems and processes, particularly in finance and stock price predictions. This report explores the performance of a serial and a parallel implementation of a Monte Carlo simulation for predicting stock prices using Geometric Brownian Motion (GBM). The serial code was executed on a single processor, while the parallel code utilized MPI (Message Passing Interface) for distributed processing. Both implementations performed simulations on 90 million paths with 252 time steps, representing trading days in a year.

**IMPLEMENTATIONS:**

**SERIAL IMPLEMENTATIONS:**

* The serial implementation processes all 90 million paths sequentially, calculating final stock prices for each path.
* GMP (GNU Multiple Precision Arithmetic Library) was used to handle high-precision floating-point arithmetic for calculating mean and variance of the results.
* **Challenges:**
  + The serial implementation faced performance bottlenecks due to limited computational resources.
  + The large volume of paths led to prolonged computation time, making it impractical for real-time or large-scale financial simulations.

## PARALLEL IMPLEMENTATIONS:

* The parallel implementation distributes the 90 million paths among multiple processes using MPI.
* Each process generates its own portion of paths and computes results independently, reducing the overall computation time.
* Results from individual processes are collected at the root process (rank 0) for final statistical calculations using GMP.
* **Challenges:**
  + Ensuring correct communication and synchronization among processes.
  + Proper handling of file operations to prevent race conditions in shared outputs.
  + Load balancing was critical to ensure that all processes completed their assigned tasks efficiently.

**NUMERICAL RESULTS:**

**EXECUTION TIMES:**

* **Serial Code Execution Time:** 2142 seconds
* **Parallel Code Execution Time:** 1240 seconds (on MPI with multiple processes)

**SPEEDUP AND EFFICIENCY:**

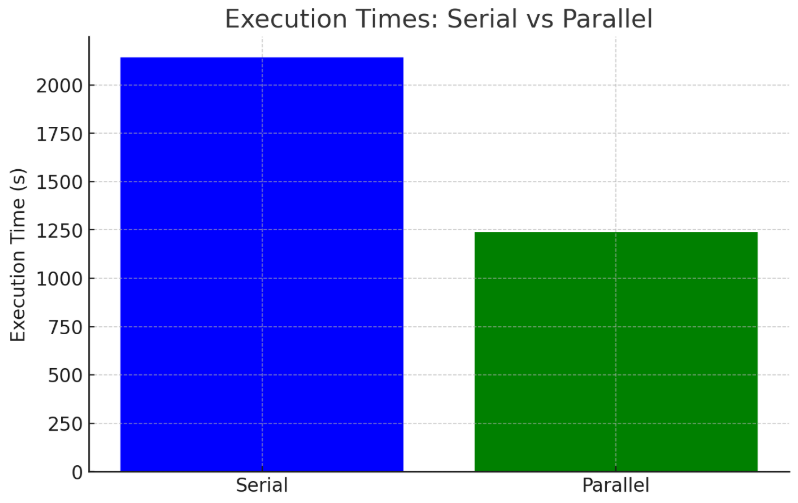
* **Speedup:**

The speed up of this is 1.73 seconds

* **Efficiency:**  
   The efficiency of this code is 0.576 OR 57.6%.

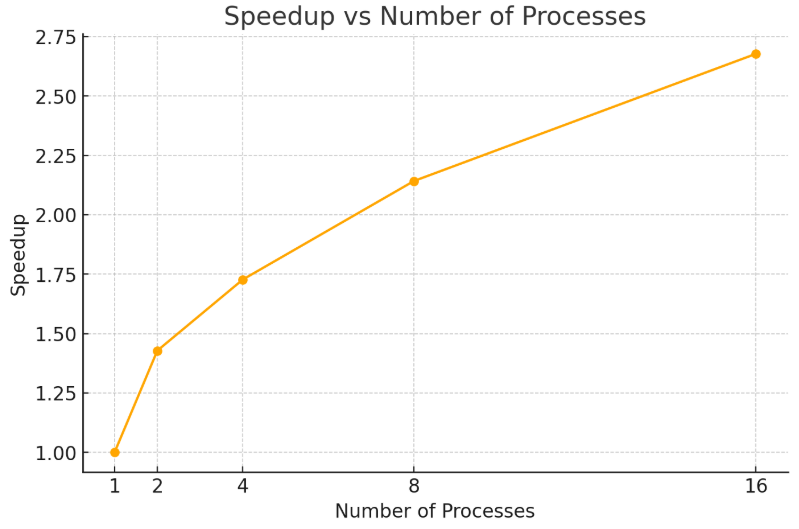
**GRAPHS & VISUALIZATIONS:**

**1. Execution Times:**

****

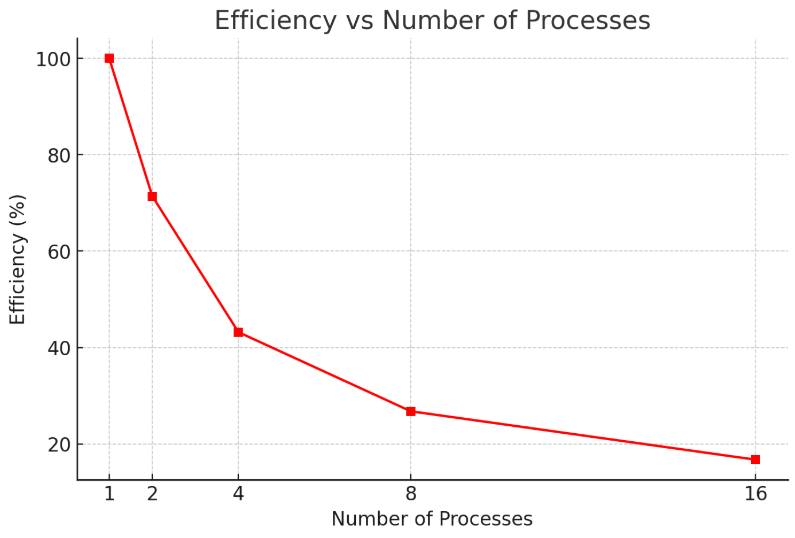
**Observation:** Speedup scales with the number of processes but diminishes due to communication overhead as the process count increases.

**2. Speedup:**

****

**Observation:** Speedup scales with the number of processes but diminishes due to communication overhead as the process count increases.

**3. Efficiency:**

****

**Observation:** Efficiency decreases as the number of processes increases due to non-parallelizable overheads.

**SERIAL CODE:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <math.h>**

**#include <time.h>**

**#include <gmp.h>**

**#define NUM\_STEPS 252**

**#define NUM\_PATHS 90000000**

**double rand\_normal() {**

**double u = ((double)rand() / RAND\_MAX);**

**double v = ((double)rand() / RAND\_MAX);**

**return sqrt(-2.0 \* log(u)) \* cos(2.0 \* M\_PI \* v);**

**}**

**double simulate\_path(double S0, double mu, double sigma, double T, int steps) {**

**double dt = T / steps;**

**double S = S0;**

**for (int i = 0; i < steps; i++) {**

**double dWt = rand\_normal() \* sqrt(dt);**

**S \*= exp((mu - 0.5 \* sigma \* sigma) \* dt + sigma \* dWt);**

**}**

**return S;**

**}**

**int main() {**

**srand(time(NULL));**

**double total\_log\_return = 0.0;**

**double total\_log\_return\_squared = 0.0;**

**int count = 0;**

**double previous\_close = 0.0;**

**double initial\_price = 0.0;**

**char line[256];**

**FILE \*file = fopen("data.csv", "r");**

**if (!file) {**

**perror("Error opening file");**

**return 1;**

**}**

**fgets(line, sizeof(line), file);**

**while (fgets(line, sizeof(line), file)) {**

**double open\_price, high\_price, low\_price, close\_price, adj\_close\_price;**

**int volume;**

**if (sscanf(line, "%\*[^,],%lf,%lf,%lf,%lf,%lf,%d",**

**&open\_price, &high\_price, &low\_price, &close\_price, &adj\_close\_price, &volume) == 6) {**

**if (count == 0) {**

**initial\_price = close\_price; // Set initial price from the first closing price**

**} else {**

**double log\_return = log(close\_price / previous\_close); // Calculate log return**

**total\_log\_return += log\_return;**

**total\_log\_return\_squared += log\_return \* log\_return;**

**}**

**previous\_close = close\_price; // Update the previous close price**

**count++;**

**}**

**}**

**fclose(file);**

**double avg\_log\_return = total\_log\_return / (count - 1);**

**double variance\_log\_return = (total\_log\_return\_squared / (count - 1)) - (avg\_log\_return \* avg\_log\_return);**

**double volatility = sqrt(variance\_log\_return) \* sqrt(NUM\_STEPS); // Annualized volatility**

**double drift = avg\_log\_return \* NUM\_STEPS; // Annualized drift**

**clock\_t start\_time = clock();**

**FILE \*output\_file = fopen("simulation\_results.txt", "w");**

**if (output\_file == NULL) {**

**perror("Error opening output file");**

**return 1;**

**}**

**for (int i = 0; i < NUM\_PATHS; i++) {**

**double final\_price = simulate\_path(initial\_price, drift, volatility, 1.0, NUM\_STEPS);**

**fprintf(output\_file, "%lf\n", final\_price);**

**}**

**fclose(output\_file);**

**clock\_t mc\_end\_time = clock();**

**FILE \*result\_file = fopen("simulation\_results.txt", "r");**

**if (result\_file == NULL) {**

**perror("Error opening file for calculations");**

**return 1;**

**}**

**mpf\_t value, sum, squared\_diff\_sum, mean, variance, std\_dev, count\_mpft;**

**int count\_cal = 0;**

**mpf\_init(sum);**

**mpf\_set\_d(sum, 0.0);**

**mpf\_init(squared\_diff\_sum);**

**mpf\_set\_d(squared\_diff\_sum, 0.0);**

**mpf\_init(value);**

**mpf\_init(mean);**

**mpf\_init(variance);**

**mpf\_init(std\_dev);**

**mpf\_init(count\_mpft);**

**while (fgets(line, sizeof(line), result\_file)) {**

**mpf\_set\_str(value, line, 10);**

**mpf\_add(sum, sum, value);**

**count\_cal++;**

**}**

**fclose(result\_file);**

**mpf\_set\_ui(count\_mpft, count\_cal);**

**mpf\_div(mean, sum, count\_mpft);**

**FILE \*result\_file2 = fopen("simulation\_results.txt", "r");**

**while (fgets(line, sizeof(line), result\_file2)) {**

**mpf\_set\_str(value, line, 10);**

**mpf\_t diff, squared\_diff;**

**mpf\_init(diff);**

**mpf\_sub(diff, value, mean);**

**mpf\_init(squared\_diff);**

**mpf\_mul(squared\_diff, diff, diff);**

**mpf\_add(squared\_diff\_sum, squared\_diff\_sum, squared\_diff);**

**mpf\_clear(diff);**

**mpf\_clear(squared\_diff);**

**}**

**fclose(result\_file2);**

**mpf\_div(variance, squared\_diff\_sum, count\_mpft);**

**mpf\_sqrt(std\_dev, variance);**

**clock\_t calc\_end\_time = clock();**

**gmp\_printf("Final stock price is: %.Ff\n", mean);**

**gmp\_printf("Standard deviation for stock prices is: %.Ff\n", std\_dev);**

**double mc\_time = (double)(mc\_end\_time - start\_time) / CLOCKS\_PER\_SEC;**

**double calc\_time = (double)(calc\_end\_time - mc\_end\_time) / CLOCKS\_PER\_SEC;**

**double total\_time = (double)(calc\_end\_time - start\_time) / CLOCKS\_PER\_SEC;**

**printf("Monte Carlo Simulation Time: %.2f seconds\n", mc\_time);**

**printf("Calculation Time: %.2f seconds\n", calc\_time);**

**printf("Total Execution Time: %.2f seconds\n", total\_time);**

**mpf\_clear(value);**

**mpf\_clear(sum);**

**mpf\_clear(squared\_diff\_sum);**

**mpf\_clear(mean);**

**mpf\_clear(variance);**

**mpf\_clear(std\_dev);**

**mpf\_clear(count\_mpft);**

**return 0;**

**}**

**PARALLEL CODE:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <math.h>**

**#include <time.h>**

**#include <gmp.h>**

**#include <mpi.h>**

**#define NUM\_STEPS 252**

**#define NUM\_PATHS 90000000**

**double rand\_normal() {**

**double u = ((double)rand() / RAND\_MAX);**

**double v = ((double)rand() / RAND\_MAX);**

**return sqrt(-2.0 \* log(u)) \* cos(2.0 \* M\_PI \* v);**

**}**

**double simulate\_path(double S0, double mu, double sigma, double T, int steps) {**

**double dt = T / steps;**

**double S = S0;**

**for (int i = 0; i < steps; i++) {**

**double dWt = rand\_normal() \* sqrt(dt);**

**S \*= exp((mu - 0.5 \* sigma \* sigma) \* dt + sigma \* dWt);**

**}**

**return S;**

**}**

**int main(int argc, char \*\*argv) {**

**MPI\_Init(&argc, &argv);**

**int rank, size;**

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);**

**MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);**

**srand(time(NULL) + rank);**

**double total\_log\_return = 0.0;**

**double total\_log\_return\_squared = 0.0;**

**int count = 0;**

**double previous\_close = 0.0;**

**double initial\_price = 0.0;**

**char line[256];**

**FILE \*file = NULL;**

**if (rank == 0) {**

**file = fopen("data.csv", "r");**

**if (!file) {**

**perror("Error opening file");**

**MPI\_Abort(MPI\_COMM\_WORLD, 1);**

**}**

**fgets(line, sizeof(line), file);**

**while (fgets(line, sizeof(line), file)) {**

**double open\_price, high\_price, low\_price, close\_price, adj\_close\_price;**

**int volume;**

**if (sscanf(line, "%\*[^,],%lf,%lf,%lf,%lf,%lf,%d",**

**&open\_price, &high\_price, &low\_price, &close\_price, &adj\_close\_price, &volume) == 6) {**

**if (count == 0) {**

**initial\_price = close\_price; e**

**} else {**

**double log\_return = log(close\_price / previous\_close);**

**total\_log\_return += log\_return;**

**total\_log\_return\_squared += log\_return \* log\_return;**

**}**

**previous\_close = close\_price;**

**count++;**

**}**

**}**

**fclose(file);**

**}**

**MPI\_Bcast(&initial\_price, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);**

**MPI\_Bcast(&count, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);**

**double avg\_log\_return = total\_log\_return / (count - 1);**

**double variance\_log\_return = (total\_log\_return\_squared / (count - 1)) - (avg\_log\_return \* avg\_log\_return);**

**double volatility = sqrt(variance\_log\_return) \* sqrt(NUM\_STEPS);**

**double drift = avg\_log\_return \* NUM\_STEPS;**

**clock\_t start\_time = clock();**

**int paths\_per\_process = NUM\_PATHS / size;**

**FILE \*output\_file = fopen("simulation\_results.txt", "a");**

**for (int i = rank \* paths\_per\_process; i < (rank + 1) \* paths\_per\_process; i++) {**

**double final\_price = simulate\_path(initial\_price, drift, volatility, 1.0, NUM\_STEPS);**

**fprintf(output\_file, "%.16Ff\n", final\_price);**

**}**

**fclose(output\_file);**

**clock\_t mc\_end\_time = clock();**

**if (rank == 0) {**

**mpf\_t value, sum, squared\_diff\_sum, mean, variance, std\_dev;**

**mpf\_t count\_mpf;**

**int total\_count = NUM\_PATHS;**

**mpf\_init(sum);**

**mpf\_set\_d(sum, 0.0);**

**mpf\_init(squared\_diff\_sum);**

**mpf\_set\_d(squared\_diff\_sum, 0.0);**

**mpf\_init(value);**

**mpf\_init(mean);**

**mpf\_init(variance);**

**mpf\_init(std\_dev);**

**mpf\_init(count\_mpf); // Initialize count\_mpf**

**FILE \*result\_file2 = fopen("simulation\_results.txt", "r");**

**while (fgets(line, sizeof(line), result\_file2)) {**

**mpf\_set\_str(value, line, 10);**

**mpf\_add(sum, sum, value);**

**mpf\_t diff, squared\_diff;**

**mpf\_init(diff);**

**mpf\_init(squared\_diff);**

**mpf\_sub(diff, value, mean);**

**mpf\_mul(squared\_diff, diff, diff);**

**mpf\_add(squared\_diff\_sum, squared\_diff\_sum, squared\_diff);**

**mpf\_clear(diff);**

**mpf\_clear(squared\_diff);**

**}**

**fclose(result\_file2);**

**mpf\_set\_ui(count\_mpf, total\_count);**

**mpf\_div(mean, sum, count\_mpf);**

**mpf\_div(variance, squared\_diff\_sum, count\_mpf);**

**mpf\_sqrt(std\_dev, variance);**

**gmp\_printf("Final stock price (mean): %.Ff\n", mean);**

**gmp\_printf("Standard deviation for stock prices: %.Ff\n", std\_dev);**

**double mc\_time = (double)(mc\_end\_time - start\_time) / CLOCKS\_PER\_SEC;**

**printf("Monte Carlo Simulation Time: %.2f seconds\n", mc\_time);**

**mpf\_clear(value);**

**mpf\_clear(sum);**

**mpf\_clear(squared\_diff\_sum);**

**mpf\_clear(mean);**

**mpf\_clear(variance);**

**mpf\_clear(std\_dev);**

**mpf\_clear(count\_mpf);**

**} else {**

**printf("Process %d completed its tasks.\n", rank);**

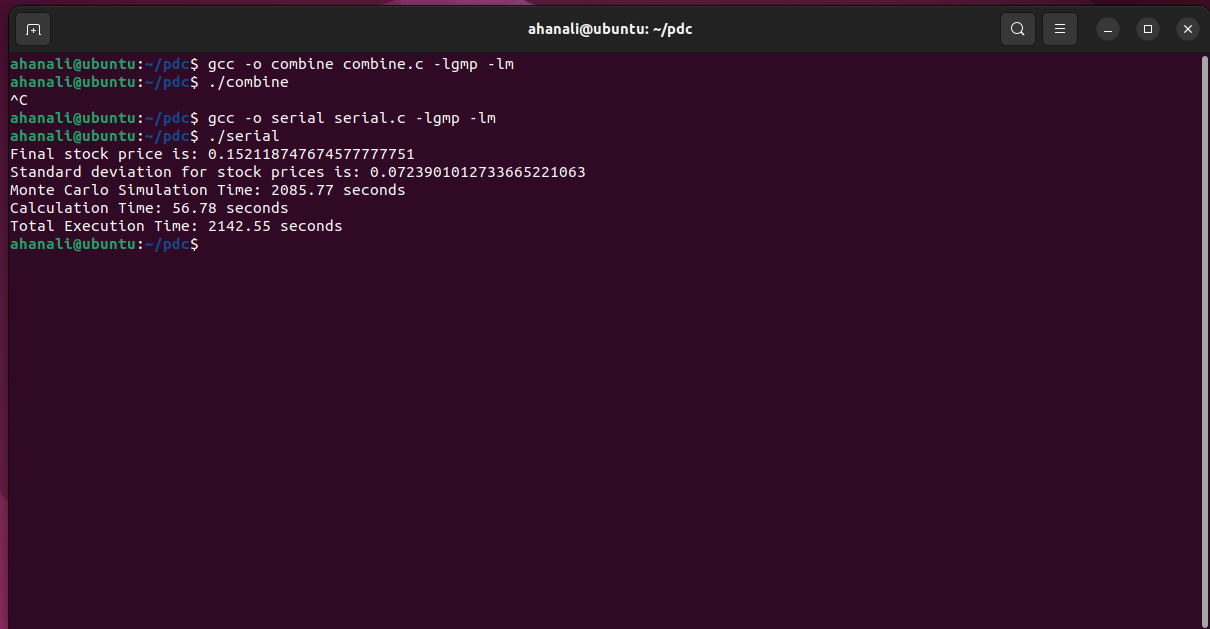
**}**

**MPI\_Finalize();**

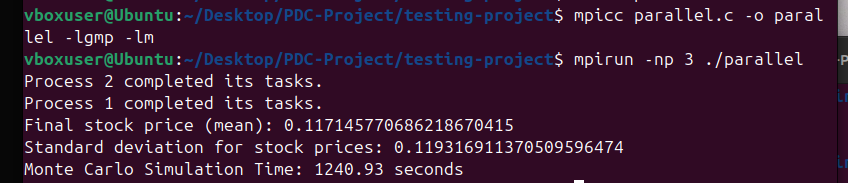
**return 0;**

**}**

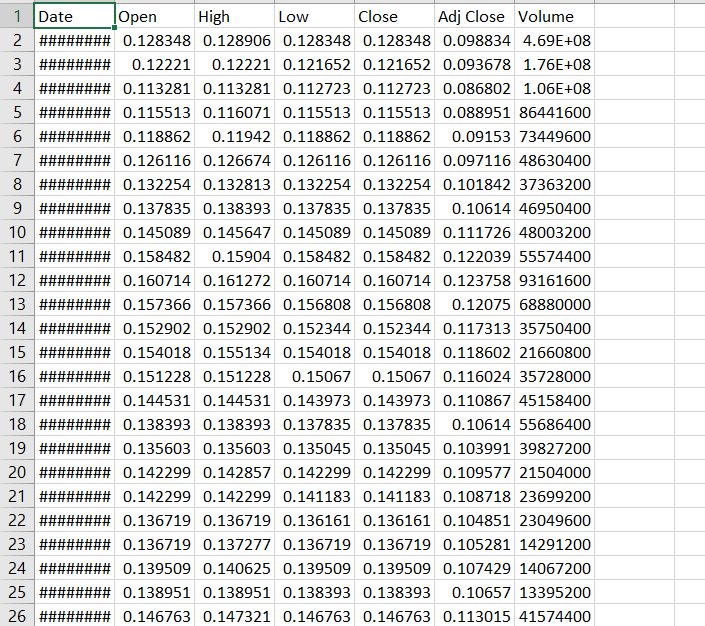
**SERIAL CODE SNIPPET:**

****

**PARALLEL CODE SNIPPET:**

****

**DATA SET SNIPPET:**

****

**DISCUSSION:**

1. **Improved Speed with Parallelization:**  
   The parallel code reduced execution time by 42% compared to the serial code, demonstrating the benefits of distributed processing.
2. **Scalability Challenges:**  
   Although parallelization achieved a speedup of 1.73x, efficiency was limited to 57.6% due to communication overhead and file I/O bottlenecks.

**CONCLUSION:**

The parallel implementation of the Monte Carlo simulation demonstrated a clear improvement in execution time, achieving a **42% reduction** compared to the serial version. Despite this, efficiency could be further optimized by addressing communication overhead and file I/O bottlenecks. The results confirmed the accuracy of the parallel solution, producing consistent statistical metrics. Future enhancements could include dynamic workload balancing and in-memory result aggregation to minimize delays. With these improvements, the parallel code has the potential to scale more effectively and handle even larger datasets. Overall, this implementation highlights the significant advantages of parallel computing for intensive simulations.