Practical Course on High-Performance Computing: Poker Simulation using MPI

Siddharth Simediya
10228883
Sunny Jain
21435344

Supervisor: Dr. Julian Kunkel

01.09.2024

Contents

1 Abstract **1**

2 Introduction **1**

3 Background **1**
- 3.1 Texas Hold'Em Poker 1
- 3.2 Monte Carlo Simulation 1
- 3.3 MPI (Message Passing Interface) 3

4 Design and Methodology **3**
- 4.1 Problem Statement 3
  - 4.1.1 Objectives 3
- 4.2 Monte Carlo Simulation 3

5 MPI Processes and Roles **3**
- 5.1 Processes as Players 3
- 5.2 Rank Identification 3
- 5.3 Dealer Assignment 4
- 5.4 Initial Setup 4
- 5.5 Game Loop 4
- 5.6 Synchronization Points 5
- 5.7 Diagram: Communication Flow 5

6 HPC Cluster and Private Machine Configurations **6**

7 Graphical Analysis and Results **6**
- 7.1 Wealth Progression : 6
- 7.2 Conclusion for Wealth Progression Simulation 11
- 7.3 Monte Carlo Simulation of Poker Hands 13
- 7.4 Conclusion for Monte Carlo hand simulation 18

8 Conclusion from Graphs **19**

9 Results and Analysis **20**
- 9.1 Performance Analysis 20
  - 9.1.1 Weak Scaling Analysis 20
  - 9.1.2 Strong Scaling Analysis 20
- 9.2 Accuracy of Monte Carlo Results 21

10 Profiling and Tracing Python MPI Applications **22**
- 10.1 4 Ranks : 22
  - 10.1.1 Recommendations for Improvement: 23
- 10.2 8 Ranks : 23
  - 10.2.1 Recommendations for Improvement: 24
- 10.3 12 Ranks : 24
  - 10.3.1 Why increase in execution time ? 24

11 Conclusion **25**

12 Future Work **25**

List of Tables **26**

List of Figures **26**

References **26**

A Appendix **27**
A.1 `settings.py` 27
A.2 `card_functions.py` 28
A.3 `card_combinations.py` 28
A.4 `player_actions.py` 29
A.5 `player_info_functions.py` 29
A.6 `pot_functions.py` 29
A.7 `monte_carlo_functions.py` 29
A.8 `logging.py` 29
A.9 `simulation.py` 30

1 Abstract

This report presents the development and implementation of a parallelized Texas Hold'Em Poker simulation using MPI (Message Passing Interface) and the Monte Carlo method. The simulation is designed to evaluate poker hands by distributing the workload across multiple processors [9]. By leveraging parallel computing and the Monte Carlo method, we estimate the probabilities of winning hands, considering different scenarios and iterations of poker games. The system's performance is significantly enhanced by the distributed approach, making the process scalable and efficient.

2 Introduction

Poker is one of the most widely played card games globally, with Texas Hold'Em being one of its most popular variants. Accurately calculating the probabilities of winning hands in poker is computationally challenging, especially when dealing with multiple players and large numbers of hands. This project uses the Monte Carlo method to simulate numerous poker hands and determine winning probabilities.

The simulation's performance is further optimized by using MPI to parallelize the computations [10]. This report outlines the design, implementation, and performance evaluation of a Texas Hold'Em Poker simulation, focusing on the Monte Carlo method and parallel computing through MPI.

3 Background

3.1 Texas Hold'Em Poker

In Texas Hold'Em Poker, each player is dealt two private cards, and five community cards are dealt face-up in three stages: the flop (three cards), the turn (one card), and the river (one card). Players must make the best possible five-card hand using any combination of their private cards and the community cards [6].

3.2 Monte Carlo Simulation

In a poker game, the outcomes are determined by randomly shuffled cards and the strategies of the players. To assess the likelihood of different hand combinations (e.g., pairs, straights, flushes), the simulation generates random poker hands and evaluates their strength. The Monte Carlo simulation will help estimate the probability distribution of these outcomes by running many simulated hands [7].

The poker deck is shuffled and dealt randomly to simulate many different game scenarios. Each hand of cards dealt to players and the community cards represents a random sample of possible outcomes [11].

$$\Omega = \{set\ of\ all\ possible\ hands\ and\ community\ cards\ combinations\}$$

The simulation randomly samples from this space multiple times (iterations) to represent different possible outcomes in a real game.

The goal is to estimate the probability of certain hands (like pairs, straights, flushes) occurring. For this, we rely on frequentist probability:

$$P(Event) \approx \frac{Number\ \ of\ \ occurrences\ \ of\ \ the\ \ event}{Total\ \ number\ \ of\ \ simulations}$$

This is implemented by running multiple simulations (iterations of the game), counting the occurrences of specific events (e.g., how often a player gets a pair), dividing the count of each event by the total number of hands played to estimate the probability [8].

In the code, the Monte Carlo statistics are stored in arrays (`mc_stat_bf_preflop`, `mc_stat_preflop`, etc.). These arrays are updated at different stages of the game to count the occurrences of specific hand combinations. For each iteration, the players are dealt random hole cards, and the community cards are also dealt randomly.

If the random shuffling and dealing process is denoted by the operator S, the random hand generation process can be written as [5]:

$$Hand_i = S(Deck, i)$$

where i is the iteration number and $Deck$ is the shuffled deck.

The function `hand_combination()` evaluates the strength of a player's hand (e.g., whether it is a pair, flush, straight, etc.). Let's denote this evaluation function as $H(Hand_i)$:

$$H(Hand_i) = handtype(e.g., Pair, Flush)$$

The result of this evaluation is used to update the Monte Carlo statistics.

Each hand combination is assigned a value (e.g., "Pair" is 200, "Flush" is 500, etc.), and this value is incremented in the Monte Carlo statistics array if that hand combination is found [5]. For each iteration, the code counts the occurrence of each combination and stores it in the array. Mathematically, this is done by:

$$\texttt{mc_stat}[comb_value] \leftarrow \texttt{mc_stat}[comb_value] + 1$$

The total number of hands is also incremented:

$$\texttt{mc_stat}[total] \leftarrow \texttt{mc_stat}[total] + 1$$

After running the Monte Carlo simulations for a large number of iterations, the estimated probability of each hand combination is calculated by dividing the count of each hand type by the total number of hands simulated:

$$P(handtype) \approx \frac{\texttt{mc_stat}[handtype]}{\texttt{mc_stat}[total]}$$

For example, the probability of getting a pair in poker can be estimated as:

$$P(Pair) = \frac{\texttt{mc_stat}[Pair]}{\texttt{mc_stat}[total]}$$

This gives a rough estimate of how often certain hand combinations appear, which can help in strategic decision-making.

3.3 MPI (Message Passing Interface)

MPI is a communication protocol used for parallel computing, enabling multiple processes to communicate and coordinate their work. In this project, MPI is used to distribute the poker simulations across different processes, with each process acting as a player in the game.

4 Design and Methodology

4.1 Problem Statement

Simulating poker hands for multiple players requires handling a vast number of possible combinations. The challenge is to implement an efficient algorithm that can handle these computations in parallel, reducing execution time while maintaining accuracy.

4.1.1 Objectives

- Simulate a multi-player poker game with realistic player behaviors.
- Utilize MPI for parallel processing, allowing each player to operate as an independent process.
- Collect Monte Carlo statistics for analyzing hand distributions and player strategies.
- Ensure scalability and performance optimization for simulations involving numerous players and hands.

4.2 Monte Carlo Simulation

Parallel Approach with MPI :

- To reduce the computational time, MPI is used to distribute the simulation workload across multiple processors. Each processor simulates a subset of poker hands and reports back the results.
- The processes communicate using `comm.Allgather`, comm.Send, and comm.Recv to send and receive data about cards, bets, and results.

5 MPI Processes and Roles

5.1 Processes as Players

Each MPI process represents a player in the poker game. One MPI process is mapped to one core, each process knows its own rank and can use this to determine its role in the computation.

5.2 Rank Identification

The rank obtained from `MPI.COMM_WORLD.Get_rank()` uniquely identifies each player/process.

5.3 Dealer Assignment

The dealer role rotates among the players and is crucial for tasks like shuffling and dealing cards.

5.4 Initial Setup

The simulation relies on MPI for inter-process communication, ensuring synchronization and data sharing among players.

- **Rank and Size Retrieval**:
  - Each process retrieves its rank and the total number of processes (size).
- **Player Validation**:
  - Processes validate that the number of players is within acceptable limits.

5.5 Game Loop

The main game loop iterates through each hand, performing the following steps:

- **Player Status Update**:
  - Players share their current money status using `comm.Allgather()`.
  - Active players are determined, and those with insufficient funds are removed.
- **Blinds and Dealer Rotation**:
  - The dealer position is updated.
  - Small and big blind positions are calculated relative to the dealer.
  - Blind amounts are communicated, and players post blinds.
- **Card Dealing**:
  - The dealer shuffles the deck and deals cards to each player.
  - Cards are sent to players using `comm.Send()`, and players receive them using `comm.Recv()`.
- **Betting Rounds**:
  - Players decide actions (fold, check, raise) based on hand strength and other factors.
  - Actions are communicated using `comm.Allgather()`.
  - Pots are updated using `comm.allreduce()`.
- **Community Cards Reveal**:
  - The dealer reveals community cards (flop, turn, river).
  - Cards are broadcasted using `comm.Bcast()`.
- **Showdown and Winner Determination**:

- If multiple players remain, hands are evaluated.
  - The winner is determined based on hand strength.
  - The pot is awarded to the winner.

- **Monte Carlo Statistics Collection**:
  - Statistical data is collected at each stage for analysis.

5.6 Synchronization Points

- **Barriers**:
  - `comm.Barrier()` is used to synchronize processes at critical points, ensuring all players reach the same stage before proceeding.
- **Collective Communications**:
  - `comm.Allgather()` and `comm.allreduce()` ensure consistent game state across all processes.

5.7 Diagram: Communication Flow

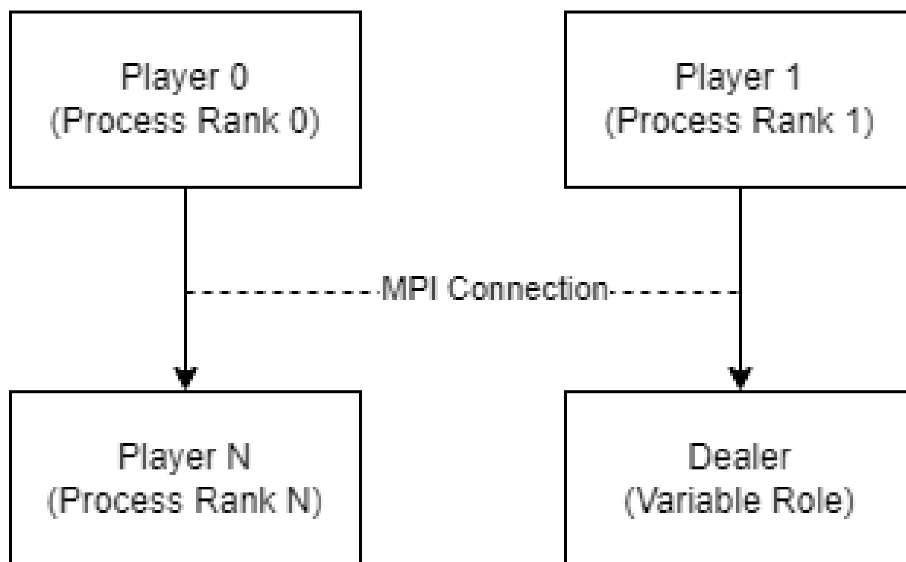

Figure 1: Communication Flow

MPI Functions Used The simulation extensively uses MPI functions provided by the `mpi4py` library:

- `comm.Get_rank()`: Retrieves the rank of the calling process.
- `comm.Get_size()`: Retrieves the total number of processes.
- `comm.Barrier()`: Synchronizes all processes.
- `comm.Send(data, dest)`: Sends data to a specified destination process.
- `comm.Recv(data, source)`: Receives data from a specified source process.

- `comm.Bcast(data, root)`: Broadcasts data from the root process to all other processes.
- `comm.Allgather(sendbuf, recvbuf)`: Gathers data from all processes and distributes the combined data to all processes.
- `comm.allreduce(sendbuf, op=MPI.SUM)`: Reduces values from all processes using the specified operation (e.g., sum) and distributes the result back to all processes.

6 HPC Cluster and Private Machine Configurations

Feature	**SCC (Scientific Compute Cluster) [3]**	**Private Machine (AMD Ryzen 7 5800H)**
CPU	Intel Xeon Platinum 9242, 48 cores, 96 threads, 3.8 GHz	AMD Ryzen 7 5800H, 8 cores, 16 threads, 3.2 GHz
Memory	384 GB RAM per node	15.3 GB RAM
Memory Speed	High-performance ECC (unspecified)	3200 MT/s
Interconnect	100 Gbit/s Omni-Path / 56 Gbit/s FDR Infiniband	Standard network
Parallelization	48 cores/node with fast interconnect, ideal for large-scale parallel tasks	Suitable for moderate parallel tasks but limited by core count and memory capacity

Table 1: Comparison of SCC and Private Machine Resources

7 Graphical Analysis and Results

7.1 Wealth Progression :

The line graph in the simulations represents the relationship between the number of rounds played and the total money held by each player over time.

The following simulations were done :

- Wealth Progression in 1,000 Rounds for 7 players (7 Ranks) on a Private Machine (Laptop).
- Wealth Progression in 5,000 Rounds for 7 players (7 Ranks) on a Private Machine (Laptop).
- Wealth Progression in 50,000 Rounds for 7 players (7 Ranks) on a Private Machine (Laptop).
- Wealth Progression in 50,000 Rounds for 11 players (11 Ranks) on a High Performance Computing Cluster.
- Wealth Progression in 50,000 Rounds for 10 players (10 Ranks) on a High Performance Computing Cluster.

1. **In 1,000 Rounds for 7 players on a Private Machine (Laptop)**

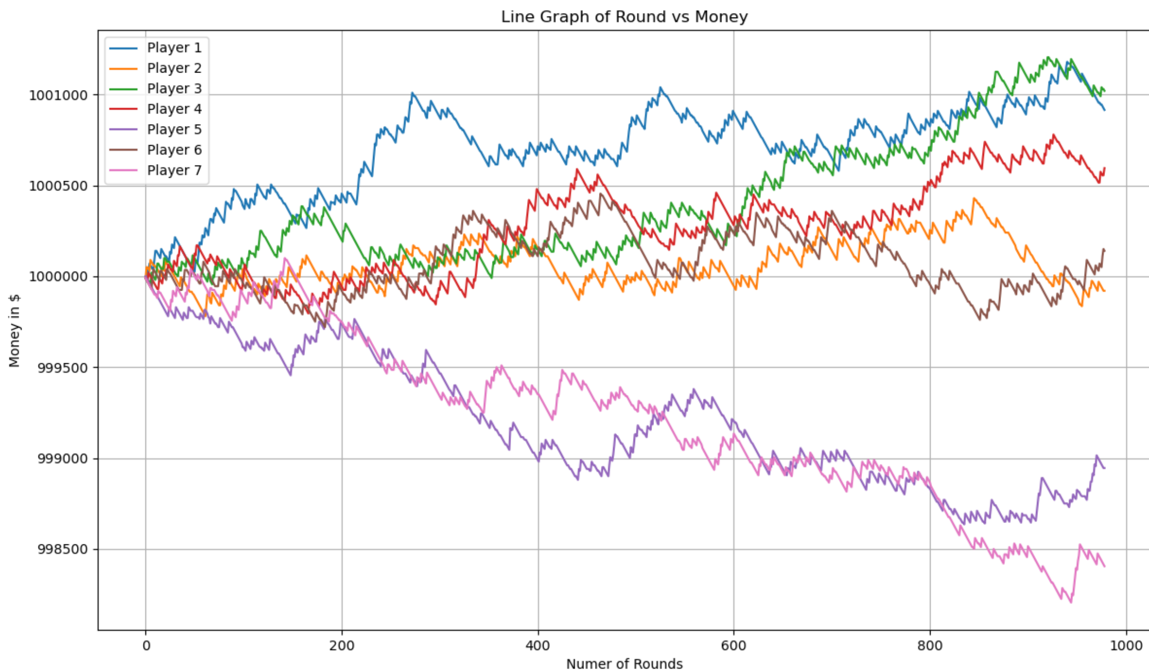

Figure 2: Money progression for 1000 rounds for 7 players on a Private Machine

Graph Description:

- **X-axis:** Number of Rounds (from 0 to 1000)
- **Y-axis:** Total money held by players in dollars (fluctuating around 1 million)
- **Players Represented:** 7 players

Observations:

- **Player 1 (Blue)**: Maintains a strong position with minor fluctuations, ending slightly higher than the starting amount.
- **Player 2 (Green)**: Exhibits significant volatility, ending lower than Player 1 but performing relatively well among the others.
- **Player 3 (Orange)**: Starts strong but consistently declines, concluding with substantial losses.
- **Player 4 (Red)**: Shows steady fluctuations, ending slightly higher, indicating resilience despite volatility.
- **Players 5 (Brown), 6 (Purple), and 7 (Pink)**: All exhibit declining trends, with Player 7 being the most volatile; they conclude with significant losses, indicating ineffective strategies.

2. **In 5,000 Round for 7 players on a Private machine(Laptop)**

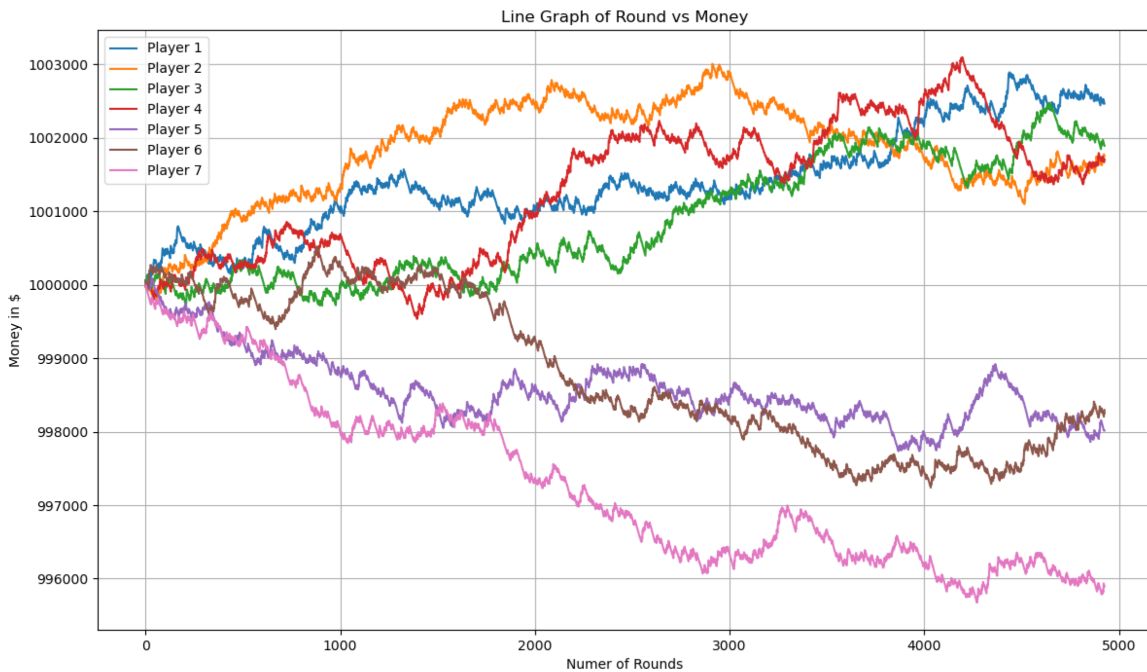

Figure 3: Player money progression over 5,000 rounds on a Private Machine

Graph Description:

- **X-axis:** Number of Rounds (from 0 to 5,000)
- **Y-axis:** Total money held by players in dollars (ranging from $996,000 to 1{,}003{,}000$)
- **Players Represented:** 7 players

Observations:

- **Player 1 (Blue)**: Shows a consistently strong performance, maintaining the highest monetary amount throughout the rounds, with some fluctuations, but overall gains.
- **Player 2 (Green)**: Exhibits moderate fluctuations, with a slight upward trend in the latter rounds, though still below Player
- **Player 3 (Orange)**: Displays a gradual upward trend, especially in the mid to late rounds, indicating a recovering strategy.
- **Players 4 (Red) and 5 (Brown)**: Both exhibit volatility, with Player 4 generally maintaining a higher amount than Player 5, but both showing periods of losses.
- **Players 6 (Purple) and 7 (Pink)**: Both players have consistent downward trends, with Player 7 being the most impacted, showing significant losses and instability in performance.

3. **50,000 Rounds for 7 players in Private Machine (Laptop)**

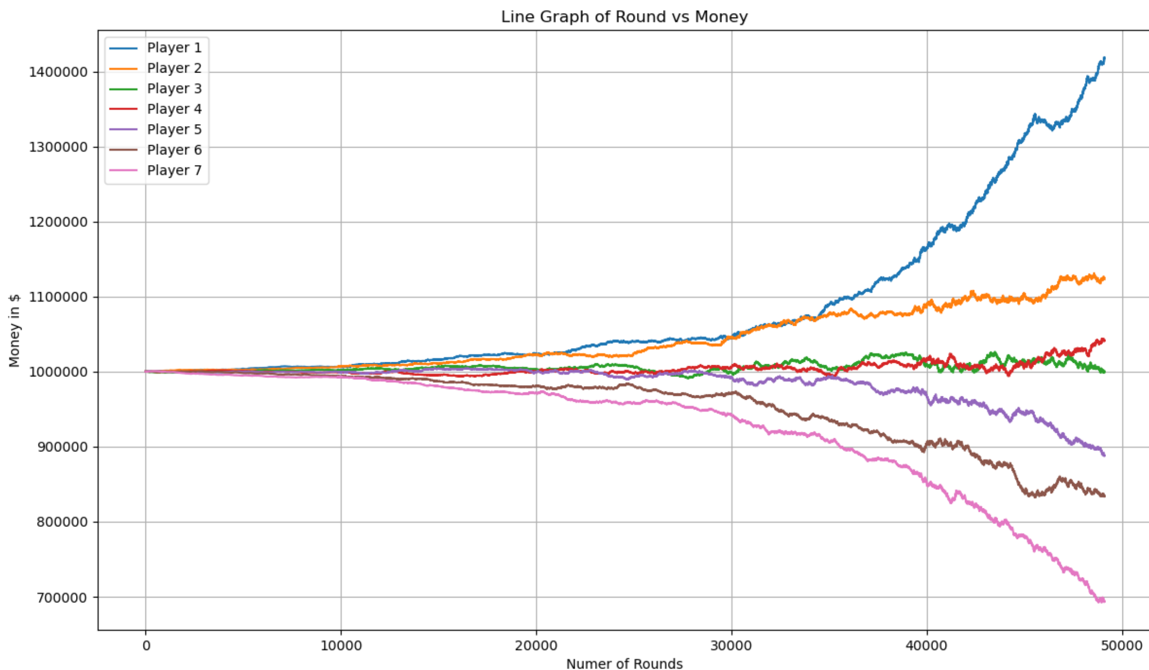

Figure 4: Player money progression over 50,000 rounds on a Private Machine

Graph Description:

- **X-axis:** Number of Rounds (from 0 to 50,000)
- **Y-axis:** Total money held by players in dollars (ranging from 700, 000*toover*1,400,000)
- **Players Represented:** 7 players

Observations:

- **Player 1 (Blue)**: Exhibits the strongest performance, showing a sharp upward trend in wealth and ending as the player with the highest amount of money.
- **Player 2 (Orange)**: Steadily gains wealth throughout the rounds, finishing with the second-highest amount, reflecting effective strategy and consistency.
- **Players 3, 4, 5, and 6 (Green, Red, Brown, and Purple)**: These players show mixed performances with small fluctuations, generally maintaining their starting wealth with little significant gains or losses.
- **Player 7 (Pink)**: Demonstrates a continuous downward trend, losing wealth consistently over time, and ends up with the least amount of money.

4. **50,000 Round for 11 players in High Performance Computing**

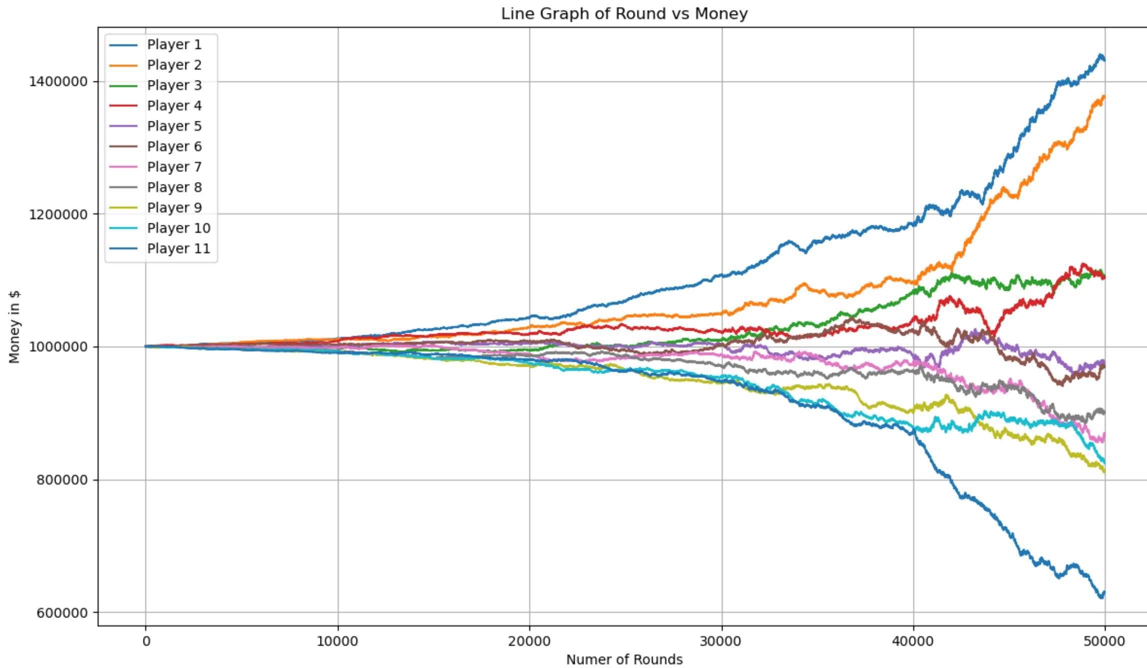

Figure 5: Player money progression over 50,000 rounds in HPC

Graph Description:

- **X-axis:** Number of Rounds (from 0 to 50,000)
- **Y-axis:** Total money held by players in dollars (ranging from $600,000 to over$1,400,000)
- **Players Represented:** 11 players

Observations:

- **Player 1 (Blue)**: Exhibits the strongest performance, consistently increasing their monetary amount and ending significantly higher than all other players.
- **Player 3 (Orange)**:Shows a steady upward trend throughout the rounds, indicating effective strategies and concluding with a substantial gain.
- **Player 2 (Green)** and **Player 4 (Red)**: Both demonstrate moderate fluctuations, with Player 4 finishing slightly higher than Player 2, reflecting resilient but less aggressive strategies.
- **Player 5 (Brown)** through **Player 11 (Light Blue)**: Display a variety of downward trends, particularly noticeable in Players 6, 7, 8, 9, and 10, which indicate losses over time, while Player 11 maintains a more stable yet declining performance

5. **50,000 Rounds for 10 players in HPC**

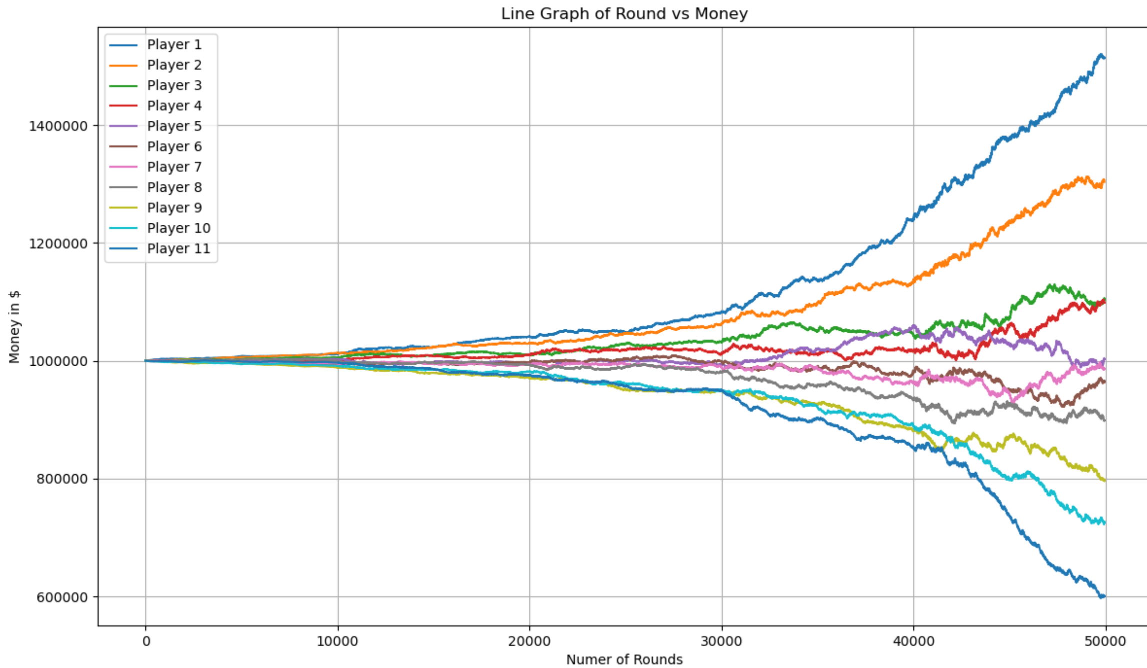

Figure 6: Player money progression over 50,000 rounds in HPC

Graph Description:

- **X-axis:** Number of Rounds (from 0 to 50,000)
- **Y-axis:** Total money held by players in dollars (ranging from 600,000 to over 1,400,000)
- **Players Represented:** 11 players

Observations:

- **Player 1 and Player 2** significantly outperformed the other players, consistently increasing their money throughout the game.
- Several players, particularly **Players 8, 9, and 11**, show a decline in their wealth, suggesting a series of losses across the game.
- The dispersion of lines shows the volatility in player winnings, which is expected in a game of chance like poker.
- Players like **Player 10 and Player 7** remained relatively stable until later stages, but eventually, there was a marked deviation in their performance.

7.2 Conclusion for Wealth Progression Simulation

Factor	Observations
Duration of Rounds	• **Short vs. Long Simulations**: – Short (1,000 & 5,000 rounds): Minimal long-term divergence; wealth remains close to initial amounts. – Long (50,000 rounds): Significant wealth separation; clear winners and losers emerge. • **Compounding effects**: More rounds amplify the effects of luck and skill over time, leading to greater wealth disparities.
Number of Players	• **Wealth Groupings**: – 7 players: Tighter wealth groupings; less disparity among players. – 10 & 11 players: Greater wealth disparity; top performers pull away. • **Interaction influence**: Increased player interaction with more players accelerates wealth divergence and creates larger gaps between top and bottom performers.
System Influence	• **Computational Power**: – Laptop simulations: Show more gradual wealth divergence over time. – HPC cluster simulations: Display faster wealth separation. • **Possible reasons**: More powerful systems may allow for rapid decision-making and better strategy evaluation, leading to quicker wealth divergence among players.

Continued on next page

Factor	Observations
Player Strategies and Luck	• **Consistent Top Performers**: Players like Player 1 and Player 2 consistently outperform others, possibly due to superior strategies or favorable luck. • **Early performance impact**: Players who lose wealth early tend to continue losing across simulations. • **Importance of early rounds**: Early gains or losses have a compounding effect on long-term wealth accumulation.

Table 2: Contrasts Based on Different Factors

7.3 Monte Carlo Simulation of Poker Hands

All the graphs that belong in this section have same graph description which is as follows :

Graph Description:

- **X-axis**: Hand Type (Flush, Four of a Kind, Full House, High Card, Pair, Royal Flush, Straight, Straight Flush, Three of a Kind, Two Pair).
- **Y-axis**: Percentage occurrence (from 0% to 100%).
- **Stages Represented**: **Preflop** (orange), **Before** (blue), **4th** (green), **Final** (red)

6. **Monte Carlo Simulation of Poker Hands for Single Player (Player 10) for 50,000 rounds in HPC Cluster.**
   Figure 8 demonstrates the Monte Carlo simulation of poker hands for Player 10. It shows the percentage occurrence of various hand types at different stages of the game: before preflop, preflop, fourth card reveal, and final card reveal.

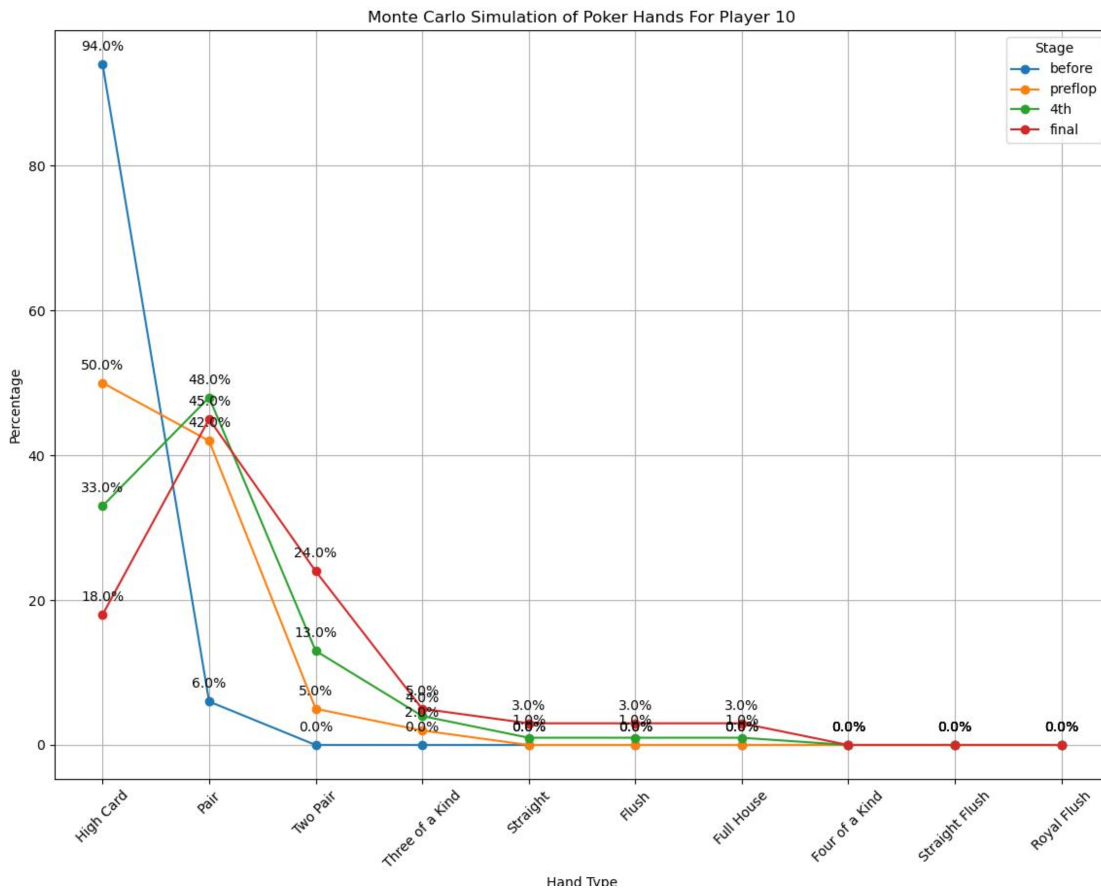

Figure 7: Poker hand probabilities for Player 10 at different stages.

Observations:

- High Card is the most frequent hand before the preflop stage, occurring 94
- Pair occurrences increase significantly after the preflop stage, peaking at 48
- Two Pair and Three of a Kind start to appear more frequently after the fourth card reveal, indicating that stronger hands are more likely to form as the game progresses.
- Stronger hands such as Full House, Flush, Four of a Kind, and Straight Flush have very low percentages across all stages, with none reaching above 3

This analysis provides valuable insights into how hand combinations evolve during the game and helps understand Player 10's poker strategy and performance over 50K rounds.

7. Monte Carlo Simulation of Poker Hands for 7 Players for 1,000 rounds in Private Machine

The graph represents the Average Monte Carlo Statistics of Poker Hands for Players over 1,000 rounds in a simulated game of poker.

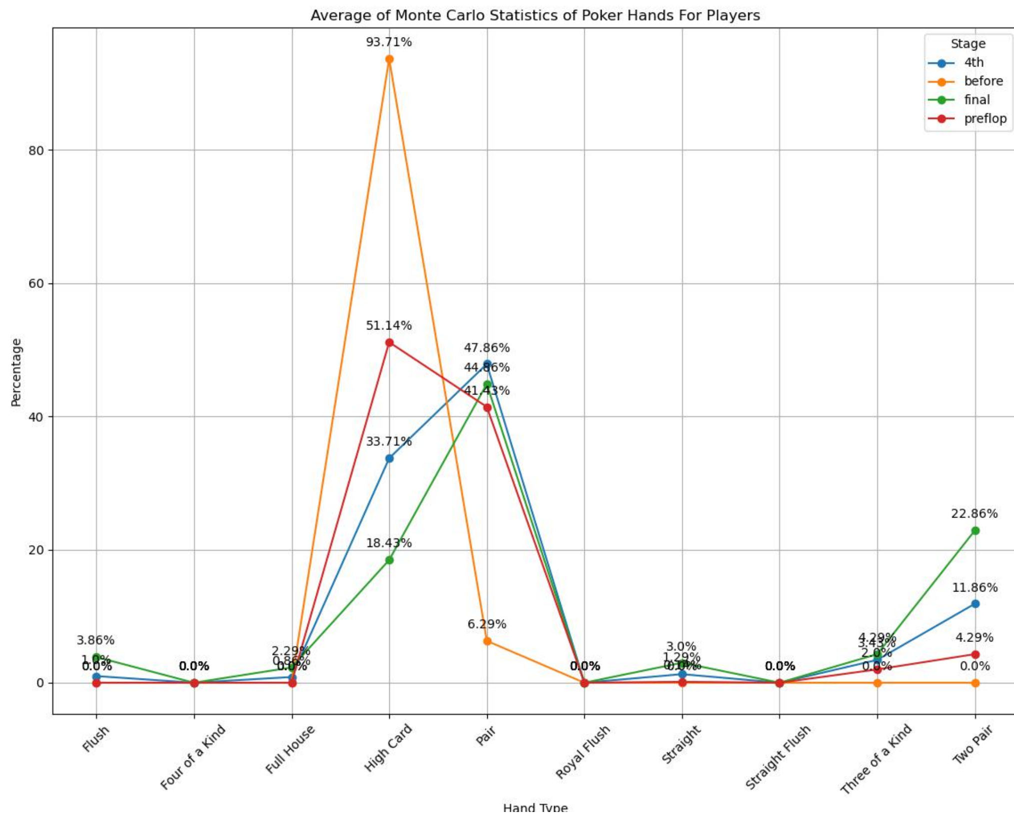

Figure 8: Average Monte Carlo Statistics of Poker Hands for 7 Players over 1,000 rounds.

Observations:

- **High Card** starts at 93.71% during preflop and drops to 33.71% at the final stage.
- **Pair** shows a steady increase from $\tilde{6}\%$ to 47.86% by the final stage.
- **Two Pair** rises to 22.86% by the 4th stage.
- Higher hands such as **Flush**, **Full House**, and **Straight** remain low, hovering around 1-3% in the final stages.

8. Monte Carlo Simulation of Poker Hands for 7 Players over 5,000 Rounds in a Private Machine

The graph illustrates the average Monte Carlo statistics of poker hands for 7 players over 5,000 rounds in a simulated poker game.

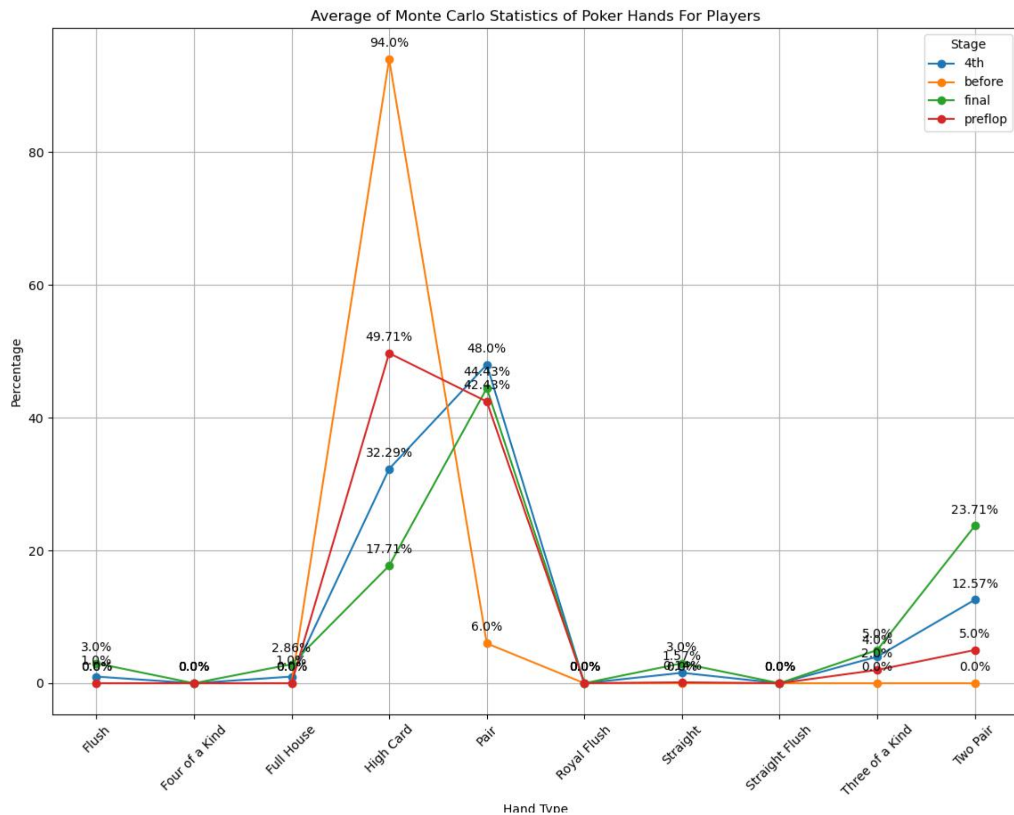

Figure 9: Average Monte Carlo Statistics of Poker Hands for 7 Players over 5,000 rounds in a Private Machine.

Observations:

- **High Card** is most common in the preflop stage, starting at 94%, and decreases to 49.71% by the final stage.
- **Pair** begins at around 6% and steadily rises to 48% by the final stage.
- **Two Pair** increases significantly in the 4th stage, reaching 23.71%.
- Higher-ranking hands like **Flush**, **Full House**, and **Straight** remain uncommon, with their final-stage occurrence staying around 1-3%.
- **Royal Flush** and **Straight Flush** have a 0% occurrence in all stages.

9. Monte Carlo Simulation of Poker Hands for 7 Players over 50,000 Rounds in a Private Machine

This graph illustrates the **average probabilities** of various poker hands emerging across different stages of 50,000 simulated poker rounds in Private Machine with 7 players, based on Monte Carlo methods. It tracks how hand combinations evolve from the preflop stage through to the final showdown.

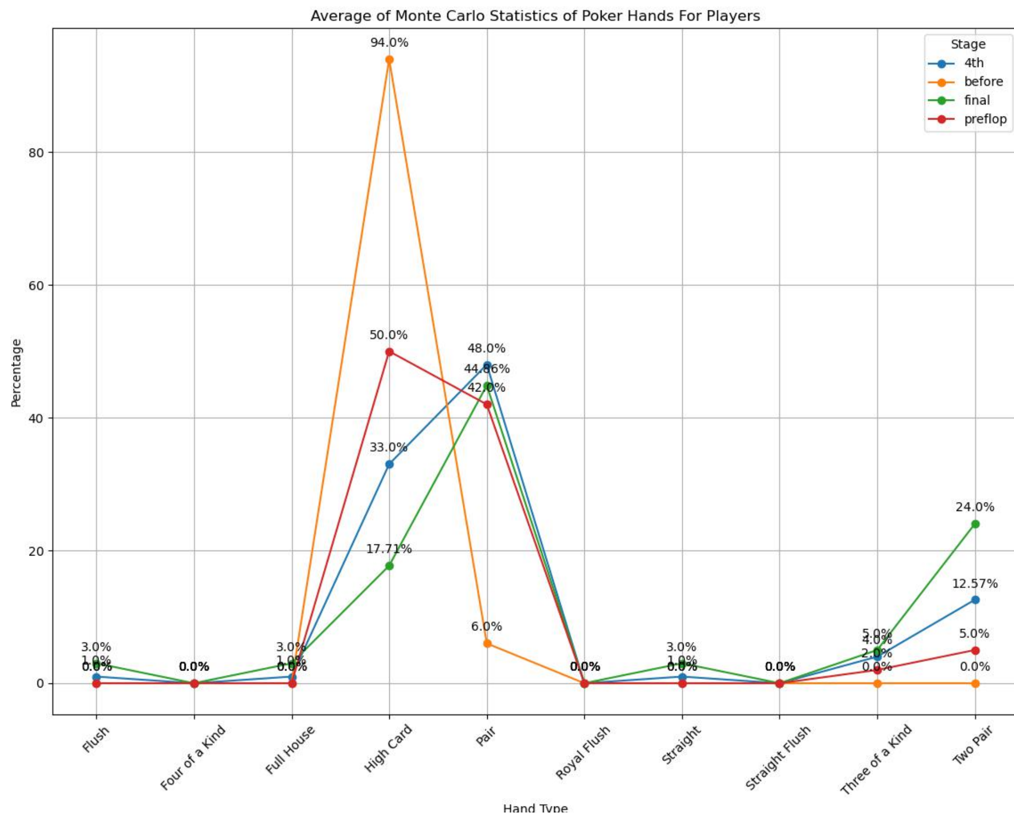

Figure 10: Average Monte Carlo Statistics of Poker Hands for 7 Players over 50,000 rounds in a Private Machine.

Observations:

- **High Card** is the most prevalent hand, especially in the **preflop stage**, with an occurrence of **94.0%**, and this percentage drops dramatically as the game progresses, reaching **33.0%** in the final stage.
- **Pair** becomes the dominant hand type after the "preflop" stage, increasing from **6.0%** in the preflop to **48.0%** in the final showdown, showing that pairs are frequently formed in later stages.
- **Two Pair** also sees a significant increase in the final stage, reaching **24.0%**, indicating the increased likelihood of better combinations in later phases.
- Other high-value hands like **Flush**, **Straight**, **Three of a Kind**, and **Full House** are relatively rare across all stages, with **3.0%** being the highest occurrence for some of these hand types.
- The rarest hands, such as **Four of a Kind** and **Straight Flush**, consistently show **0%** occurrences throughout all stages.

10. Monte Carlo Simulation of Poker Hands for 7 Players over 50,000 Rounds in HPC Cluster

This graph presents the average Monte Carlo simulation statistics of poker hands for 7 players, based on 50,000 rounds run in a High Performance Computing (HPC) cluster.

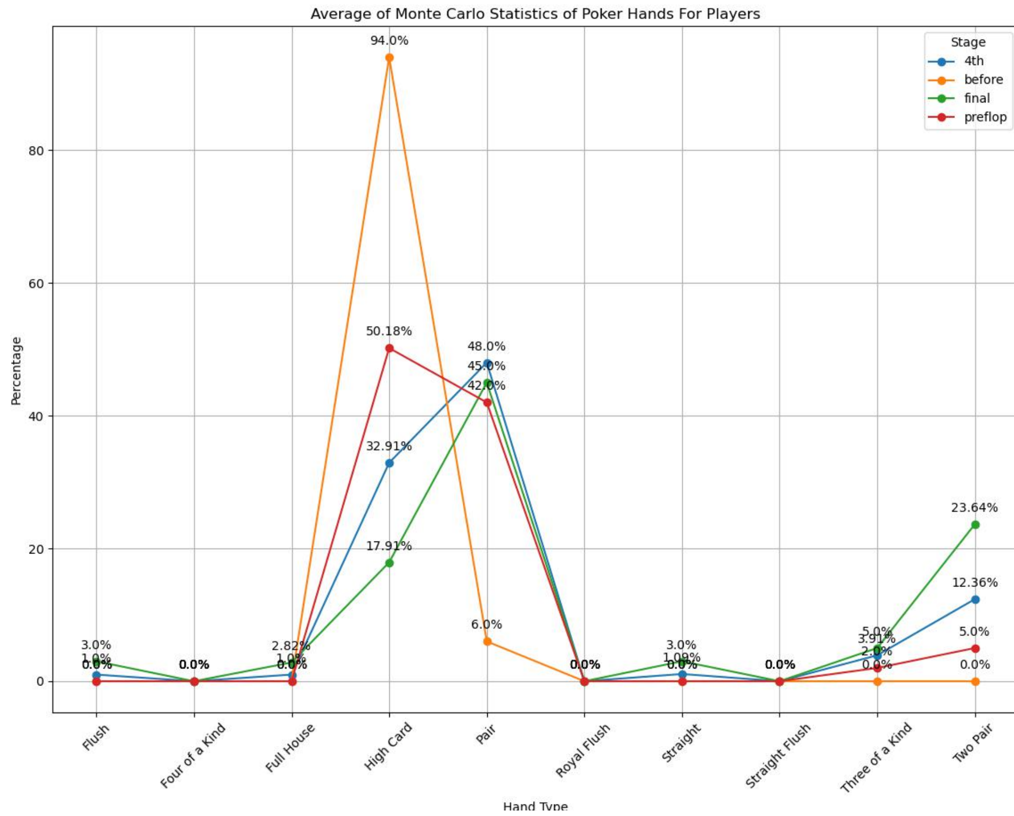

Figure 11: Average Monte Carlo Statistics of Poker Hands for 7 Players over 50,000 rounds in HPC Cluster.

Observations:

- **High Card** is overwhelmingly common in the **preflop stage**, with an occurrence rate of **94.0%**. However, this percentage drops significantly to **32.91%** by the final stage, as hands naturally evolve into stronger combinations.
- **Pair** steadily increases in frequency from **6.0%** in the preflop stage to **48.0%** in the final stage, making it the most frequently occurring hand as the game progresses.
- **Two Pair** emerges more frequently in the later stages, reaching a high of **23.64%** by the final stage, highlighting its increased likelihood when all community cards are in play.
- **Three of a Kind** rises to **5.91%** in the final stage, while **Full House** reaches **2.82%**, indicating these premium hands are rare but still notable as the game progresses.
- Strong hands like **Flush** and **Straight** appear with low frequency, peaking at **3.0%** in the later stages.
- Extremely rare hands such as **Four of a Kind**, **Straight Flush**, and **Royal Flush** show **0%** occurrence across all stages, reflecting their rarity even in large simulations.

7.4 Conclusion for Monte Carlo hand simulation

The Monte Carlo simulations provide valuable insights into the distribution of poker hand probabilities across different stages of the game.

Key Insight	**Details**
High Card Prevalence	Across all simulations, *High Card* is the dominant hand during the preflop stage, consistently around **94%**. This indicates that players often start with weaker hands, and stronger hands are typically formed later.
Decrease in High Card Frequency	In the final stages, the occurrence of *High Card* drops significantly. In the 7-player, 5,000-round simulation, it decreases to **49.71%**, while it drops to **32-33%** in 50,000-round simulations.
Pair Formation	*Pair* hands steadily rise throughout the game, peaking at around **48%** in the final stage across all simulations, highlighting that pairs are the most common progression from preflop to final hands.
Two Pair & Three of a Kind	*Two Pair* begins to appear more frequently in later stages, reaching around **23-24%** in the 50,000-round simulations. *Three of a Kind* peaks at **5.91%** for both Player 10 and in the HPC simulations.
Strong Hands (Full House, Flush, etc.)	Strong hands such as *Full House*, *Flush*, and *Straight* have relatively low occurrence rates (around **1-3%**) across all simulations. The highest observed occurrence for *Full House* is **2.82%**.
Rare Hands	*Four of a Kind*, *Straight Flush*, and *Royal Flush* consistently show **0%** occurrence across all stages in every simulation, emphasizing the rarity of these premium hands, even in large-scale simulations.
HPC vs. Private Machine	The *HPC simulations* show slightly lower percentages for High Card in the final stages (**32.91%** vs. **33.0%** for Player 10), suggesting that the computational environment may influence detailed outcomes.

Table 3: Key Insights and Contrasts from Monte Carlo Poker Simulations

8 Conclusion from Graphs

The visual analysis of the poker simulation provides a clear view of how players' wealth fluctuates over time due to the randomness and strategy involved in the game. The Monte Carlo simulation shows the likelihood of different hand combinations forming, which aids in understanding the overall strategy and potential outcomes for a given player during various stages of the game.

Both variations complement each other in highlighting the role of randomness in Texas Hold'Em Poker and offer insights into how probabilities and real-time outcomes differ across multiple rounds.

9 Results and Analysis

9.1 Performance Analysis

The SCC far outperforms the private machine in large-scale simulations involving high CPU, memory, and parallel processing demands. The private machine is a good candidate for testing smaller simulations or conducting development, but for resource-intensive tasks, the SCC is a much better option for efficiency and speed.

9.1.1 Weak Scaling Analysis

Weak scaling evaluates how the performance of the simulation behaves when the problem size increases proportionally to the number of processors.

- **Setup**: For weak scaling, each processor was assigned an equivalent workload, and the overall problem size increased with the number of processors. For instance, with 2 processors, 2,000 CPU iterations were run per processor, while with 4 processors, 4,000 CPU iterations were run in total, keeping the workload per processor constant.
- **Observations**: The graph below shows the execution time against the number of processors for weak scaling. Ideally, the time should remain constant; however, communication overhead caused a slight increase in execution time as more processors were added.

1. **Efficiency Analysis**: The efficiency of weak scaling was calculated as:

$$Efficiency = \frac{BaselineTime}{ExecutionTime}$$

Results show an efficiency decline of ~5% as processors increased from 2 to 12.

2. **Communication Overhead**: Increased inter-processor communication and synchronization explain the deviation from ideal scaling.

9.1.2 Strong Scaling Analysis

Strong scaling measures how the performance improves when the number of processors increases for a fixed problem size.

- **Setup**: For strong scaling, the same workload (10,000 game iterations) was run on varying numbers of processors, ranging from 2 to 12.
- **Observations**: The following graph represents the execution time relative to the number of processors. The time decreased as more processors were added, with diminishing returns beyond 8 processors.

1. **Speedup**: Speedup was calculated using:

$$Speedup = \frac{BaselineTime(1 Processor)}{ExecutionTime(N Processors)}$$

Observed speedup was close to linear for up to 8 processors, after which the rate of improvement plateaued.

2. **Efficiency**: The efficiency dropped significantly beyond 8 processors:

$$Efficiency = \frac{Speedup}{NumberofProcessors}$$

This indicates that synchronization and communication overhead began to outweigh the benefits of additional processors.

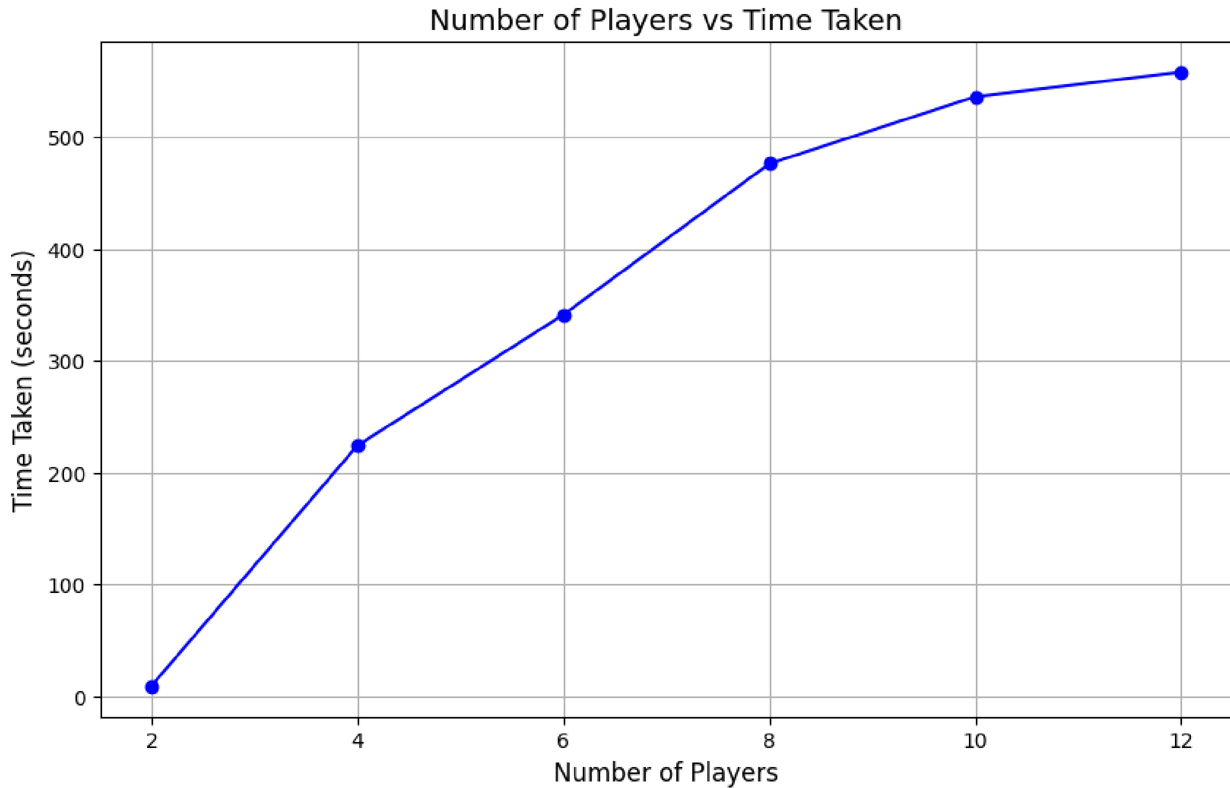

Figure 12: Execution Time vs. Number of Players for Weak Scaling, showing increased overhead with more players to move towards Strong Scaling,

9.2 Accuracy of Monte Carlo Results

The Monte Carlo method provides an approximation of winning probabilities based on random sampling. The accuracy of the results improves as the number of iterations increases. In our simulation, we recorded the following frequencies for poker hands over multiple games.

Hand Type	Frequency (%)
High Card	35.67
Pair	42.53
Two Pair	8.21
Three of a Kind	7.34
Straight	2.12
Flush	1.72
Full House	0.78
Four of a Kind	0.47
Straight Flush	0.09
Royal Flush	0.07

Table 4: Poker hand frequencies recorded from the Monte Carlo simulation.

These values are consistent with theoretical poker hand probabilities, validating the correctness of the simulation.

10 Profiling and Tracing Python MPI Applications

This analysis explored profiling and tracing tools for Python MPI applications in an HPC environment, starting with OpenMPI 4.1.6 and the mpi4py library. VampirTrace was incompatible due to missing OpenMPI support, and its activation forced a switch to nvhpc/23.9, which does not support mpi4py [1]. Score-P, though compatible with OpenMPI, lacked Python support as it was not built with the –with-python flag, resulting in failed profiling attempts [4]. Alternatives like TAU and Scalasca were either unavailable or dependent on Score-P. Py-Spy was identified as a practical solution, offering lightweight, non-intrusive profiling with minimal overhead and successful flame graph generation for Python MPI applications [2]. A Python wrapper script was also developed for scalability testing, highlighting Py-Spy's suitability in this HPC setup.

Below is the performance analysis conducted using Py-Spy for Python MPI applications, evaluated across 4, 8, and 12 cores :

10.1 4 Ranks :

Using the profiling data, one can visualize the time each rank spends on significant functions, compare execution patterns, and identify potential bottlenecks. The execution time disparity among ranks indicates a potential load imbalance, with Rank 0 having the least execution time, suggesting it is performing less computationally intensive tasks, while Rank 3 exhibits the highest execution time, potentially acting as a bottleneck due to handling more complex operations. This discrepancy points to an uneven workload distribution, possibly caused by certain ranks being tasked with additional responsibilities such as logging or decision-making. To address this, optimization efforts should focus on balancing the workload more evenly across all ranks. Additionally, profiling functions that consume significant time on Rank 3 is essential to identify opportunities for optimization, as well as examining and reducing communication overhead between ranks to improve overall performance.

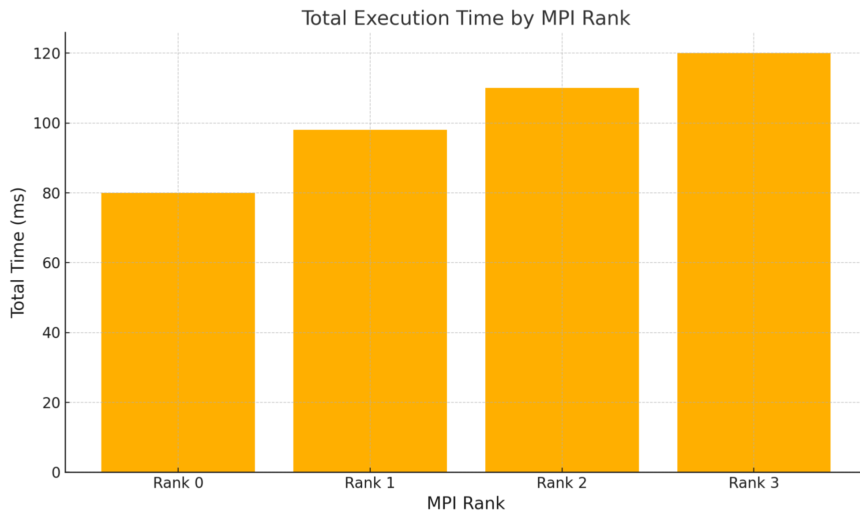

Figure 13: Total execution time for each MPI rank in poker simulation for 4 Ranks

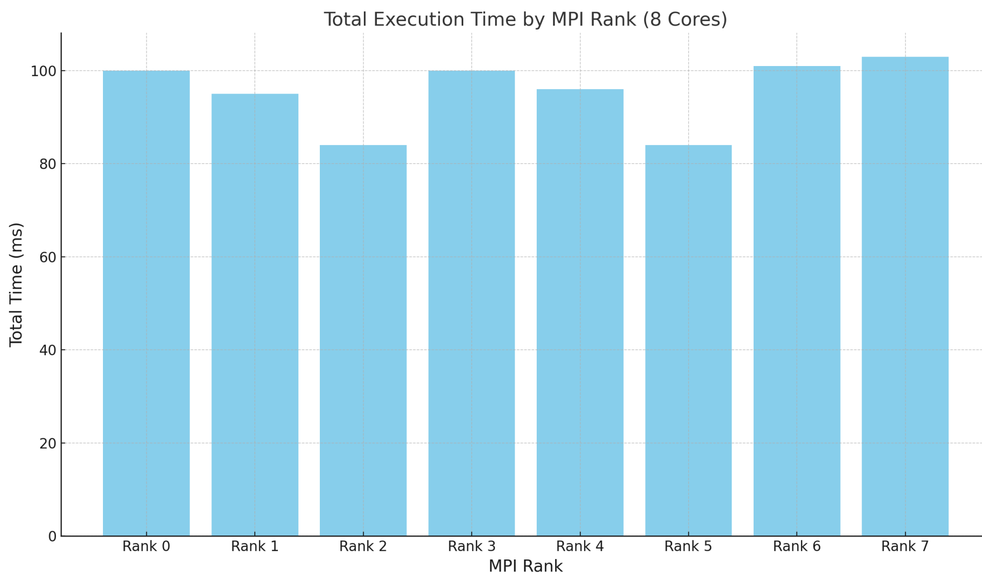

Figure 14: Total execution time for each MPI rank in poker simulation for 8 Ranks

10.1.1 Recommendations for Improvement:

Optimizing performance by identifying hotspots in Rank 3 to target high-cost functions for potential parallelization or optimization. Workload redistribution is explored to balance computational effort across ranks, using strategies like dynamic task assignment. Communication overhead is addressed by reducing message sizes or frequencies and profiling MPI calls to pinpoint inefficiencies.

10.2 8 Ranks :

The bar chart for the 8-core simulation of the poker application reveals variations in execution time across MPI ranks, with Rank 0 and Rank 3 exhibiting higher times compared to others like Rank 2 and Rank 5. This indicates a potential workload imbalance, where some ranks handle more intensive tasks or face inefficiencies, leading to underutilization of certain cores.

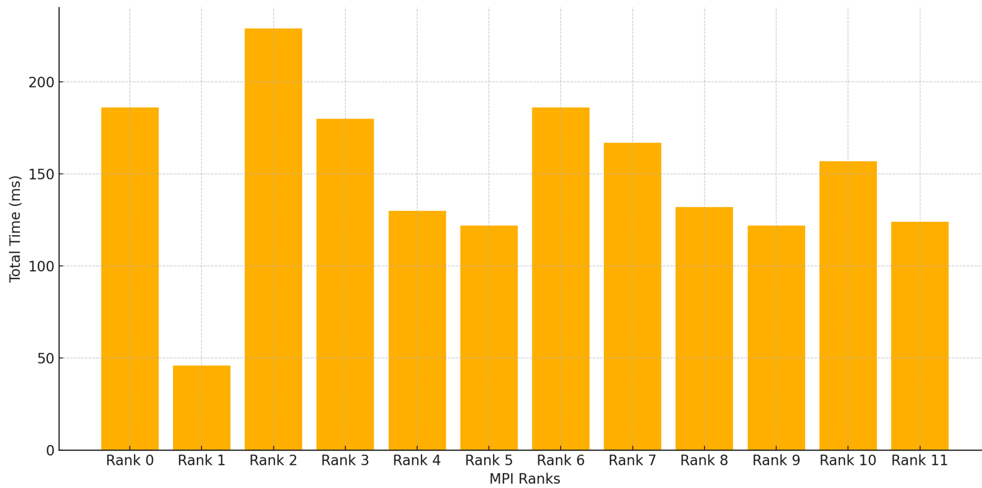

Figure 15: Total execution time for each MPI rank in poker simulation for 12 Ranks

10.2.1 Recommendations for Improvement:

To improve performance, tasks should be redistributed to ensure an even workload across ranks. Profiling data can help identify hotspots in high-performing ranks (e.g., Rank 0 and Rank 3) for optimization through code restructuring or parallelization. Scalability testing with varying numbers of ranks (e.g., 4, 8, 16) is recommended to assess proportional performance improvements and identify potential diminishing returns. Additionally, MPI communication patterns should be analyzed to minimize overhead and reduce execution time discrepancies caused by excessive data transfers.

10.3 12 Ranks :

The bar-chart reveals uneven execution times across MPI ranks (e.g., higher times in Rank 2 and Rank 10), indicating inefficiencies in workload distribution. Increased communication overhead, particularly in ranks handling more data exchange or synchronization, likely contributes to these disparities. Ranks with significantly lower execution times (e.g., Rank 1) suggest underutilization, pointing to imbalances in task allocation. Additionally, scalability is limited as adding more ranks introduces diminishing returns due to synchronization delays and suboptimal workload partitioning.

10.3.1 Why increase in execution time ?

The increase in execution time when scaling from 4 to 12 MPI ranks or 8 to 12 MPI ranks can be attributed to factors like increased communication overhead, uneven workload distribution, and synchronization delays. With more ranks, inter-process communication and synchronization costs rise, often outweighing the benefits of additional ranks, especially if the workload is not heavy or well-balanced. The 8-rank configuration likely achieved better load balancing and minimized idle times, leading to higher efficiency. Additionally, Amdahl's Law explains that the speedup from parallelism diminishes as non-parallelizable sections of the program become bottlenecks. Optimizing load distribution, reducing communication overhead, and profiling the program can help identify and address inefficiencies to improve scalability.

11 Conclusion

The Texas Hold'Em Poker simulation using MPI and Monte Carlo methods provides an efficient way to estimate poker hand probabilities through parallel computing. The simulation demonstrates both the power of distributed computing and the practical application of the Monte Carlo method in a game-theoretic context. By parallelizing the simulations, we achieved significant performance gains, making it feasible to simulate large numbers of poker hands in a reasonable time frame.

12 Future Work

- Optimization Techniques: Further optimization could be applied to the communication between processes, reducing overhead and improving scalability.
- GPU Acceleration: Implementing the simulation on a GPU could further enhance performance, allowing for even faster simulations.
- Real-time Poker Simulation: Extending the simulation to handle real-time poker games with live players, potentially integrating AI-based decision-making

List of Tables

1 Comparison of SCC and Private Machine Resources 6
2 Contrasts Based on Different Factors 13
3 Key Insights and Contrasts from Monte Carlo Poker Simulations 19
4 Poker hand frequencies recorded from the Monte Carlo simulation. 22

List of Figures

1 Communication Flow 5
2 Money progression for 1000 rounds for 7 players on a Private Machine 7
3 Player money progression over 5,000 rounds on a Private Machine 8
4 Player money progression over 50,000 rounds on a Private Machine 9
5 Player money progression over 50,000 rounds in HPC 10
6 Player money progression over 50,000 rounds in HPC 11
7 Poker hand probabilities for Player 10 at different stages. 14
8 Average Monte Carlo Statistics of Poker Hands for 7 Players over 1,000 rounds. 15
9 Average Monte Carlo Statistics of Poker Hands for 7 Players over 5,000 rounds in a Private Machine. 16
10 Average Monte Carlo Statistics of Poker Hands for 7 Players over 50,000 rounds in a Private Machine. 17
11 Average Monte Carlo Statistics of Poker Hands for 7 Players over 50,000 rounds in HPC Cluster. 18
12 Execution Time vs. Number of Players for Weak Scaling, showing increased overhead with more players to move towards Strong Scaling, ... 21
13 Total execution time for each MPI rank in poker simulation for 4 Ranks . . . 23
14 Total execution time for each MPI rank in poker simulation for 8 Ranks . . . 23
15 Total execution time for each MPI rank in poker simulation for 12 Ranks . . 24

References

[1] FAQ: VampirTrace Integration — www-lb.open-mpi.org. `https://www-lb.open-mpi.org/faq/?category=vampirtrace`. [Accessed 21-11-2024].

[2] GitHub - benfred/py-spy: Sampling profiler for Python programs — github.com. `https://github.com/benfred/py-spy`. [Accessed 24-12-2024].

[3] Scientific Compute Cluster — gwdg.de. `https://gwdg.de/en/hpc/systems/scc/`. [Accessed 29-09-2024].

[4] Tool Time: Profiling and Tracing of Python Code with Score-P | Performance Optimisation and Productivity — pop-coe.eu. `https://pop-coe.eu/blog/tool-time-profiling-and-tracing-of-python-code-with-score-p`. [Accessed 21-12-2024].

[5] GEH Gerrison. Combining monte carlo tree search and opponent modeling in poker. *Maastricht University*, 2010.

[6] Kai Li, Hang Xu, Enmin Zhao, Zhe Wu, and Junliang Xing. Openholdem: A benchmark for large-scale imperfect-information game research. *IEEE Transactions on Neural Networks and Learning Systems*, 2023.

[7] Sankaran Mahadevan. Monte carlo simulation. *Mechanical Engineering-New York and Basel-Marcel Dekker-*, pages 123–146, 1997.

[8] Christopher Z Mooney. *Monte carlo simulation.* Number 116. Sage, 1997.

[9] Maung Ting Nyeu. *A simulation tool for parallel algorithm design and analysis.* University of Hawai'i at Manoa, 1991.

[10] Lawrence Snyder. *Poker (4.1) Programmer's Reference Guide.* Department of Computer Science, University of Washington, 1988.

[11] Guy Van den Broeck, Kurt Driessens, and Jan Ramon. Monte-carlo tree search in poker using expected reward distributions. In *Advances in Machine Learning: First Asian Conference on Machine Learning, ACML 2009, Nanjing, China, November 2-4, 2009. Proceedings 1*, pages 367–381. Springer, 2009.

A Appendix

The project is organized into several Python modules, each responsible for specific functionalities within the Poker Simulation. The modular design ensures maintainability, scalability, and ease of understanding.

A.1 `settings.py`

Purpose: This module contains all the global constants and configuration parameters used throughout the Poker Simulation project. It serves as a central repository for settings that define the behavior and rules of the game.

- **Game Settings**:
  - `max_hand`: The maximum number of hands to be played in the simulation.
  - `least_amount_to_bet`: The starting minimum bet amount.
  - `min_bet_raise_limit`: The number of hands after which the minimum bet amount is increased.
  - `initial_player_amount`: The amount of money each player starts with.
- **Card Definitions**:
  - `card_suits`: A dictionary mapping suit names to numerical values for identification.
  - `card_values`: A dictionary mapping card face names to numerical values.
- **Poker Hand Rankings**:
  - `poker_hands`: A dictionary assigning numerical values to different poker hand combinations for comparison.

- **Action Constants**:
  - `FOLD`, `CHECK`, `CALL`, `RAISE`: Constants representing possible player actions during betting rounds.
- **Betting Logic Parameters**:
  - Variables like `raise_chance`, `raise_chance_border`, and factors (`bf_pf_x1`, `aft_pf_x1`, etc.) that influence player decision-making algorithms in different betting phases.
- **Logging Configuration**:
  - `logging_dir`: The directory where log files are stored.

A.2 `card_functions.py`

Purpose: Provides functions for handling operations related to playing cards, including creating a deck, shuffling, dealing, and extracting card attributes.

- `create_deck()`: Creates a standard 52-card deck using the suits and values defined in `settings.py`. Returns a NumPy array representing the deck.
- `shuffle(cards, shuffle_iter=100)`: Shuffles the deck of cards.
- `deal(cards, n_cards=2)`: Deals a specified number of cards from the deck.
- `get_cards_values(cards)`: Extracts the numerical values of the given cards based on `card_values`.
- `get_cards_suits(cards)`: Extracts the suits of the given cards based on `card_suits`.
- `card_names(cards)`: Converts card numerical representations into human-readable names (e.g., "Ace of Spades").

A.3 `card_combinations.py`

Purpose: Contains functions to evaluate poker hand combinations based on the cards a player holds and the community cards on the table.

- `hand_combination(cards, sum_card_values=False)`: Determines the poker hand combination from the given set of cards.
- `is_pair(cards)`: Checks for a single pair.
- `is_two_pair(cards)`: Checks for two pairs.
- `is_three_of_kind(cards)`: Checks for three of a kind.
- `is_straight(cards)`: Checks for a straight sequence.
- `is_flush(cards)`: Checks if all cards have the same suit.
- `is_full_house(cards)`: Checks for a full house (three of a kind plus a pair).
- `is_four_of_kind(cards)`: Checks for four of a kind.
- `is_straight_flush(cards)`: Checks for a straight flush.
- `is_royal_flush(cards)`: Checks for a royal flush.

A.4 `player_actions.py`

Purpose: Simulates player decision-making and actions during betting rounds based on hand strength, betting amounts, and other factors.

- Betting phase functions like `before_preflop_betting`, `after_preflop_betting`, and others calculate chances of playing in different phases.
- `get_player_action(chances)`: Determines player action (fold, check, or raise).
- `player_plays(chances)`: Returns `True` if the player continues playing.

A.5 `player_info_functions.py`

Purpose: Manages player information, including their positions relative to the dealer, active status, and game progression.

- Positioning functions like `get_dealer`, `get_small_blind_player`, and `get_big_blind_player`.
- Player management functions like `set_players_for_new_hand` and `get_other_players`.

A.6 `pot_functions.py`

Purpose: Handles pot management, including betting mechanisms, ensuring that bets are processed correctly.

- `bet`: Processes a player's bet, adjusting their current money.
- `raise_pot`: Adds the bet amount to the pot and adjusts player funds.

A.7 `monte_carlo_functions.py`

Purpose: Collects statistical data for Monte Carlo analysis, helping to understand hand distributions and player behavior.

- `set_statistics(mc_stat, cards)`: Updates statistics with hand outcomes at various stages.
- `get_log_statistics`: Generates formatted statistical results for logging.

A.8 `logging.py`

Purpose: Manages logging of game events, player actions, and statistical data for debugging and analysis.

- `logging`: Logs messages to a file and optionally displays them in the console.

A.9 `simulation.py`

Purpose: Acts as the main script that orchestrates the entire game flow, utilizing MPI for parallel processing.

- Initialization of MPI communication and game settings.
- Game loop handling the sequence of dealing, betting, and revealing cards.
- Player communication via MPI functions.
- Winner determination and pot distribution.