Multithreaded Battleship: Enhancing Game AI Through Parallelism

Alex Reyes
Undergrad, Dept. Computer Science
University of Central Florida
Orlando, FL, USA
al670064@ucf.edu

Piper Larson
Undergrad, Dept. Computer Science
University of Central Florida
Orlando, FL, USA
pi462054@ucf.edu

Jesús Molina
Undergrad, Dept. Computer Science
University of Central Florida
Orlando, FL, USA
je417003@ucf.edu

Aaron Navarro
Undergrad, Dept. Computer Science
University of Central Florida
Orlando, FL, USA
an32784@ucf.edu

Gael Adames
Undergrad, Dept. Computer Science
University of Central Florida
Orlando, FL, USA
ga193888@ucf.edu

Abstract — Battleship is a strategy game that relies on probability and player decision-making. Traditional AI approaches evaluate moves sequentially, which becomes inefficient when dealing with large-scale simulations. This project aims to develop an efficient Battleship AI using parallel programming techniques to enhance decision-making and targeting. By leveraging parallelism, we seek to accelerate computations while maintaining or improving accuracy through optimized thread management, data sharing, and efficient memory usage. Our main algorithm will be Monte Carlo Tree Search (MCTS), a heuristic search method that utilizes a divide-and-conquer strategy to partition the board, enabling parallel simulations to refine probabilistic models asynchronously. This approach allows the AI to dynamically adapt its strategy, making its moves more precise rather than relying on random guessing. Additionally, MCTS naturally integrates efficient Battleship tactics, such as probability density estimation and ship placement heuristics, further improving targeting efficiency and decision-making. The expected outcome is an optimized Battleship AI that outperforms traditional sequential approaches in both speed and accuracy. By demonstrating the effectiveness of parallelized Monte Carlo Tree Search in game AI development, this project highlights the potential of parallel computing in improving strategic decision-making and real-time adaptability.

I. INTRODUCTION

Battleship is a game of probability and strategy in which players attempt to sink each other's fleets by guessing the locations of their opponent's ships on a hidden grid. While initial moves are largely based on probability, strategy becomes crucial once a player locates part of an opponent's ship. Due to the vast number of possible board configurations and the uncertainty of the opponent's setup, the game heavily relies on both probability and luck. The project we made is about developing an efficient AI specifically for Battleship with implemented parallel

threading for an increased performance boost. Our Battleship AI project addresses the computational challenges involved in targeting and decision-making throughout the game. Traditional Battleship AI approaches evaluate possible moves sequentially, which quickly becomes inefficient when optimizing strategies for large simulations. Our main approach to the AI implementation is using the Monte Carlo Tree Search algorithm, as it seems to be one of the most viable options for a probability-based game like Battleship. For the threading solution, we plan to utilize concurrency, allowing us to use multiple threads simultaneously to search the Monte Carlo Tree in parallel. This approach will help us achieve our desired output faster and more accurately. By leveraging parallel programming techniques, our project aims to enhance the AI's prediction capabilities and strategic decision-making, making it both faster and more effective.

For a better understanding of our project's objectives, I would like to provide some insight into the game of Battleship. "Battleship is a two-player strategy game that involves tactical guessing and probability. The game includes 10 plastic ships, four runners of white pegs, two runners of red pegs, and a label sheet. The fleet consists of five different ships: the Carrier (5 holes), Battleship (4 holes), Cruiser (3 holes), Submarine (3 holes), and Destroyer (2 holes). Each player sits facing their opponent, with the lids of their game units raised to prevent the other from seeing their ocean grid. At the start of the game, both players secretly place their fleet of five ships on their respective grids. To position a ship, its two anchoring pegs must be fitted into the corresponding holes on the grid. Ships can be placed either horizontally or vertically but not diagonally. Additionally, they must not overlap letters, numbers, the edge of the grid, or other ships. Once the game begins, ship positions cannot be changed.

HOW TO PLAY! Decide who will go first. You and your opponent will alternate turns, calling out one shot per turn to try and hit each other's ships.

CALL YOUR SHOT! On your turn, pick a target hole on your upright target grid and call out its location by letter and number. Each target hole has a letter-number coordinate that corresponds with the same coordinate on your opponent's ocean grid. To determine each coordinate, find its corresponding letter on the left side of the target grid and its number on the top of the grid. When you call a shot, your opponent must tell you whether your shot is a hit or a miss.

IT'S A HIT! If you call out a shot location that is occupied by a ship on your opponent's ocean grid, your shot is a hit! Your opponent tells you which ship you have hit (cruiser, submarine, etc.). Record your hit by placing a red peg in the corresponding target hole on your target grid. Your opponent places a red peg in the corresponding hole of the ship you have hit on his or her ocean grid

IT'S A MISS! If you call out a shot location not occupied by a ship on your opponent's ocean grid, it's a miss. Record your miss by placing a white peg in the corresponding target hole on your target grid so you won't call this shot again. It's not necessary for players to record each

other's misses with white pegs on their ocean grids. After a hit or a miss, your turn is over. Play continues in this manner, with you and your opponent calling one shot per turn.

SINKING A SHIP! Once all the holes in any one ship are filled with red pegs, it has been sunk. The owner of the ship must announce which ship was sunk. Keep track of how many of your opponent's ships you sink by placing a red peg for each ship sunk in one of the five holes at the top of your game unit.

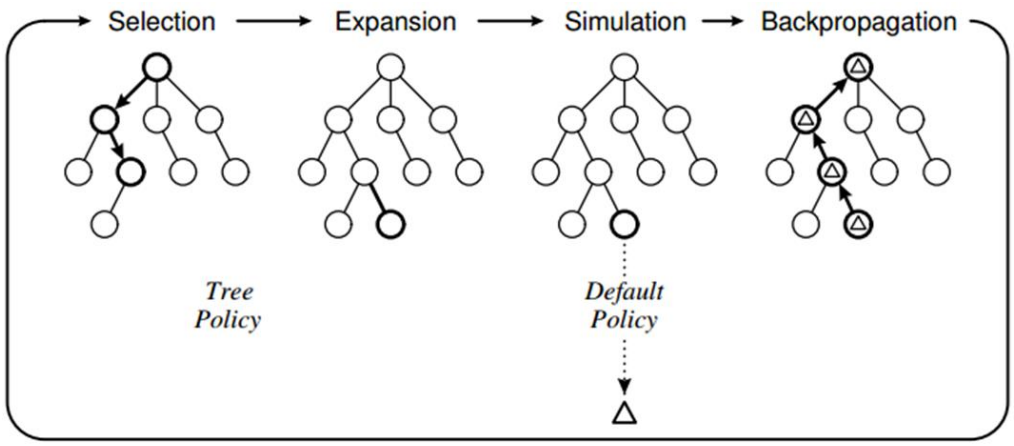
WINNING THE GAME! If you're the first player to sink your opponent's entire fleet of 5 ships, you win the game!" **(Hasbro)**

II. MONTE CARLO ALGORITHM

"Monte Carlo Tree Search (MCTS) is a heuristic algorithm widely recognized in artificial intelligence, particularly in decision-making and game-playing applications. It has been successfully utilized in various domains, including board games like Go, chess, and shogi, as well as card games such as poker and video games. MCTS has demonstrated exceptional performance, often surpassing human expertise in complex gaming scenarios. Additionally, it is highly scalable and can be efficiently parallelized, making it well-suited for distributed computing and multi-core systems. Its strength lies in handling strategic and computationally intensive games where traditional search methods struggle due to the vast number of possible actions.

MCTS combines Monte Carlo techniques—based on random sampling and statistical evaluation—with tree-based search strategies. Instead of exhaustively exploring all possible moves, it focuses on selectively sampling the most promising areas of the search space. The algorithm incrementally builds a search tree by running multiple simulations from the current game state. These simulations continue until a terminal condition or depth limit is met, and their outcomes are then backpropagated through the tree, updating node statistics such as visit counts and win ratios.

One of MCTS's key advantages is its dynamic balance between exploration and exploitation. It evaluates moves by considering both their historical success rates (exploitation) and the potential value of less-explored actions (exploration). This balance is maintained using the Upper Confidence Bound (UCB) formula, specifically the Upper Confidence Bounds for Trees (UCT), which helps guide the search process toward the most promising decisions." **(GeeksforGeeks)**



Why use Monte Carlo Tree Search (MCTS)?

1. **Complex and Strategic Games:** MCTS excels in games with large search spaces, making strong decisions in games like Go, chess, and poker.

2. **Handling Imperfect Information:** MCTS works well in situations with incomplete or uncertain data, such as card games or real-world problems.

3. **Learning from Simulations:** By running multiple simulations, MCTS improves decision-making over time through trial and error.

4. **Balancing Exploration and Exploitation:** MCTS strategically explores new options while leveraging known good moves for optimal decision-making.

5. **Scalability and Parallelization:** MCTS can efficiently use multi-core or distributed computing to handle complex problems faster.

6. **Applications Beyond Game:** MCTS is useful in planning, scheduling, optimization, and decision-making in real-world scenarios.

7. **Domain Independence:** MCTS does not require domain-specific knowledge, making it adaptable to various problem types.

Advantages of MCTS:

- **Easy to Implement:** MCTS is a straightforward algorithm to set up.

- **Domain-Independent:** It works without prior knowledge, relying only on game rules and random playouts.
- **State Preservation:** Intermediate states can be saved and reused later.
- **Adaptive Growth:** The search tree expands dynamically based on the situation.

Disadvantages of MCTS:

- **High Memory Usage:** The tree grows rapidly, requiring significant memory.
- **Reliability Issues:** Some branches may not be explored enough, leading to poor decisions.
- **Computationally Expensive:** A large number of iterations are needed for optimal performance, impacting speed.

III. IMPLEMENTATION

For our implementation, we will use the Monte Carlo Tree Search (MCTS) algorithm and modify some of its phases to introduce a bias toward preferred moves. First, we need to understand the four phases of MCTS:

- **Selection:** The algorithm starts at the root node and selects a path down the tree based on a policy, typically using Upper Confidence Bound for Trees (UCT):

- **Expansion:** If the selected node is not terminal and has unvisited children, one of them is expanded (i.e., a new node is added to the tree). This new node represents a possible move that has not been explored yet.

- **Simulation:** A random playout (simulation) is performed from the newly expanded node. Moves are chosen randomly (or using a heuristic) until the game reaches a terminal state (win, loss, or draw). The result of this simulation (win, loss, or draw) is recorded.

- **Update:** The result of the simulation is propagated back up the tree. Each node along the path updates its statistics (wins and visit counts). This helps refine the UCT values for future selections.

Now that we understand how the MCTS phases work, we need to determine how to modify the search tree to achieve our desired outcome. First, we must decide on the strategy we will follow to customize our search tree. In Battleship, some strategies are more effective than others, but let's start with the basics. "On average, a game of Battleship played randomly takes around 78 shots to win, which is not ideal considering there are 100 possible moves. The first

strategy that can be employed is the 'Hunt and Target' method, which consists of shooting randomly until a target is found. After finding it, the player selects between up, left, right, or down around the hit mark to find the next valid hit. With this strategy, the average number of shots per game is around 65, which is an improvement compared to random play. Another common strategy is 'Parity,' which involves splitting the board from 100 squares to 50. This method gives us a rough approximation of where to look, as ships tend to extend over multiple squares. It works because the smallest ship, the destroyer, is only 2 squares in size. However, this approach is not optimal as it is only slightly better than random at best. A better implementation of both strategies is called 'Hunt and Target with Parity.' By combining both methods, we achieve an average of 60 shots per game to win. Another addition to common strategies is the 'Guess, Recalibrate, Repeat' strategy. Since we start the game with no information, the best initial move is to pick a random spot in the center of the board, as it is more probable that a ship will be located there. After each shot, the strategy recalibrates and refines the search based on the layout of remaining spaces and the ships that have not yet been sunk. To make accurate guesses, we need to consider the length of the remaining ships and adjust our strategy accordingly, much like creating a heatmap to signal where to hit next. This probability density function approach helps us reduce the average number of shots to win a game, dropping it to between 30 and 40 shots per game." **(Vsauce2)**

Now that we have all our strategies, we can apply the Monte Carlo Tree Search (MCTS). First, we need to understand what the Tree Policy is. The Tree Policy decides how the tree is navigated and grown before simulations occur. It consists of the selection and expansion phases, where we can add our strategy to bias certain moves. To effectively apply a strategy in MCTS, we need to modify the selection and expansion phases by incorporating a bias toward our "Hunt and Target with Parity" and "Guess, Recalibrate, Repeat" strategies. After implementing this bias, our phases will proceed as follows:

1. **Selection Phase:** The tree policy will guide the selection of the best child node.
2. **Expansion Phase:** This phase will consider our strategies, and if the selected node is not terminal and has unvisited children, it will expand one.
3. **Simulation Phase:** A random playthrough will be performed from the new node to generate an outcome.
4. **Update Phase:** The result will be backpropagated to update the node statistics.
5. **Final Phase:** After obtaining the desired UTC (Upper Confidence Bound for Trees), we choose the node with the most simulations and repeat the process until we get our final result.

For our threading implementation, we will use concurrency to traverse all child nodes with multiple threads. The approach with concurrent threading is as follows:

1. **Selection Phase:** We will split the tree into sections, and multiple threads will follow the tree policy to pick the best possible child nodes.

2. **Expansion Phase:** Each threaded section will consider our strategies. If the selected node is not terminal and has unvisited children, it will expand one.
3. **Simulation Phase:** Each threaded section will perform a random playthrough from the new node to generate an outcome.
4. **Update Phase:** Each threaded section will backpropagate the result to update node statistics.
5. **Final Phase:** After obtaining the desired UTC from each threading section, we will join the threads and get the results. Then, we will choose the node with the most simulations and repeat the process until we get the final result.

With this threading implementation, we expect to enhance the performance of MCTS by traversing the tree with multiple threads, leading to better accuracy by generating a higher number of starting nodes. This will force the threads to find more optimal routes, which can yield a higher number of simulations that might otherwise have been missed due to conflicts in the policy of previous nodes.

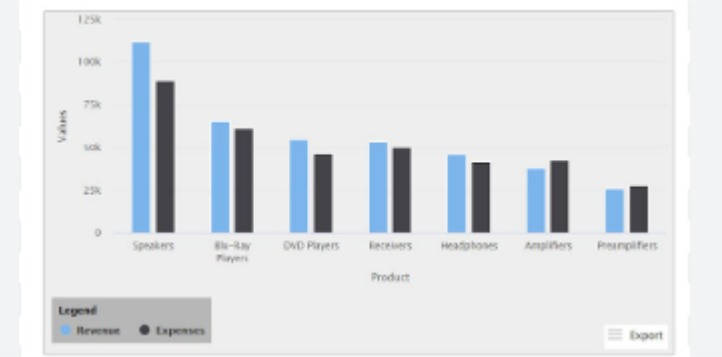
IV. TESTING STRATEGY

- To test our implementation, we ran it multiple times until we achieved a target average number of shots required to win the game.
- We kept our strategies and policies as simple as possible.
- We tested our threading solution with 2, 4, and 6 cores to analyze its scalability and determine any correlation between the number of threads and our concurrent approach.
- We measured the execution time of our base MCTS and compared it to the execution times of MCTS with varying numbers of threads.
- For each implementation, we ran multiple iterations to assess whether increasing the number of threads led to more accurate simulations.
- On average, some games deviated from the expected results for each strategy. However, outliers were anticipated and treated accordingly.
- Each of the five team members conducted these tests on their own hardware, contributing to a total of 100 game simulations.
- Finally, we compared the overall performance averages for each game across all implementations.

V. EVALUATION

The results from our Monte Carlo Tree Search (MCTS) implementation with concurrent threading across multiple cores are satisfactory. We can conclude that the threading

implementation improved the overall performance of MCTS in terms of speed, though with limited improvements in accuracy. From a speed perspective, we can conclude that adding more threads helps us navigate the tree faster, enabling us to reach conclusions in a fraction of the time, thanks to the processing power of multiple threads operating simultaneously. From an accuracy perspective, our findings are mixed. While we can confirm that more threading did, in fact, lower the average number of shots to win, we cannot be 100% certain of this outcome, as our strategy implementation may significantly influence how the final data is handled. However, we can assert that there is a slight improvement in performance accuracy. By considering multiple sections at once and allowing them to try different paths, we enable better decisions to be made. This approach, which allows for a better selection process by exploring different starting conditions, leads to improved outcomes that might otherwise be missed with a single-threaded approach.



Upon further investigation, we can see that the higher the number of cores, the better the performance of MCST. We can conclude that the more threads we have, the more simulations are conducted, and the faster we traverse the entire tree due to the additional threads available.

In conclusion, MCST benefits greatly from parallel threading solutions, as its nature allows for seamless implementation in such environments. Our approach to implementing MCST with threading has been successful, and the results reflect this. Currently, we are able to make better and faster estimations thanks to our implementations and strategies.

VI. LIMITATIONS OF RESEARCH

We encountered some limitations during our project and research. One limitation was the lack of data for AI implementations in the Battleship game, which constrained our creativity and required us to use only the resources we could gather and piece together. Another limitation was the absence of test data from previously completed games, which would have helped us train a more accurate model. Additionally, we lacked a suitable controlled environment for our research. The devices used for testing had varying specifications, making it difficult to obtain reliable

estimates. These limitations may have artificially affected our data, either positively or negatively, and could have skewed the results. However, the findings we were able to gather are consistent with our expectations. Further testing is necessary to rule out the possibility that our data was skewed or corrupted. A more controlled environment and a deeper understanding of the strategies and implementations are needed to make more accurate conclusions.

VII. CHALLENGES ENCOUNTERED

Some of the challenges we encountered were related to our strategies causing undesired and unpredictable moves due to the nature of how we implemented our concurrent threading solution. At times, the most optimal move was not selected, and instead, a seemingly more viable move was taken, which did not align with our intended strategies. For example, when hitting a ship for the first time, it is optimal to choose between up, left, right, or down to locate the next hit on the same ship in order to destroy it. However, because some threads simulated an incorrect move that seemed better, the wrong move was chosen instead of following the strategy. Another example is when completely destroying a ship: the algorithm kept choosing between up, left, right, or down, unaware that the ship had already been destroyed. We were able to partially resolve these issues by adjusting our strategies and modifying the UTC values accordingly.

Another challenge we encountered was the randomness and uncertain nature of the Battleship game. Since the game relies heavily on luck and limited probabilities, achieving an optimal result was difficult when the ship pieces were positioned in ways that contradicted the strategies we were following. Certain board variations, such as ships being placed in the corners with minimal spacing between them or confined to one side of the board, seemed to have a significant impact on the AI algorithm's effectiveness. These variations nullified the strategy, making it harder to achieve consistent results.

VIII. FUTURE RESEARCH

For future research, it would be worthwhile to evaluate other AI algorithms that could benefit more from parallel threading implementations. A better understanding of strategic implementation would be essential in achieving a more accurate replication of the actual findings, particularly in how the probabilities of these strategies influence the moves being made. Additionally, it would be important to utilize better tools and establish a more stable environment to more accurately calculate the discrepancies between each iteration.

IX. CONCLUSION

Our implementation of the Monte Carlo Search Tree (MCTS) with concurrent threading for Battleship AI has demonstrated promising results. By modifying the selection and expansion phases to incorporate strategic bias—such as "Hunt and Target with Parity" and "Guess,

Recalibrate, Repeat" we successfully reduced the average number of shots needed to win a game. The introduction of parallel threading further enhanced performance, significantly improving the speed of simulations while yielding marginal but noticeable gains in accuracy.

While our approach proved effective, some challenges emerged, including unintended move selections due to threading conflicts and the unpredictable nature of Battleship. Despite these limitations, our findings reinforce the idea that MCTS is highly adaptable to parallel processing, benefiting from increased computing power to explore a larger decision space. Future research should focus on refining strategic implementations, exploring alternative AI methodologies, and improving the testing environment to minimize discrepancies across different hardware. Additionally, gathering more real-world gameplay data would enhance the AI's learning process, leading to more precise decision-making. Ultimately, our work provides a strong foundation for further advancements in AI-driven Battleship strategies and demonstrates the potential of parallelized MCTS in game-playing AI.

REFERENCES

Hasbro. Battleship Game Instructions. Hasbro, https://www.hasbro.com/common/instruct/battleship.pdf. Accessed 25 Feb. 2025.

GeeksforGeeks. "ML | Monte Carlo Tree Search (MCTS)." GeeksforGeeks, 23 May 2023, https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/. Accessed 25 Feb. 2025.

Vsauce2. The Battleship Algorithm. YouTube, 2020, https://youtu.be/LbALFZoRrw8. Accessed 25 Feb. 2025.