Multithreaded Battleship: Enhancing Game AI Through Parallelism

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

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

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

Aaron Navarro
Undergrad, Dept. Computer Science
University of Central Florida
Orlando, USA
aa931330@ucf.edu

Gael Adames
Undergrad, Dept. Computer Science
University of Central FLorida
Orlando, 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 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.**

***Index Terms*—Search Tree, Upper confidence bound, Threading, Artificial Intelligence, Heuristic Algorithm**

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. Although 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 Artificial Intelligence (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 the project's AI objective it is imperative to understand Battleship the game in every light. Hasbro, the company that makes and sells the game bluntly describes it as "a two-player strategy game that involves tactical guessing and probability" [1]. Each player has five ships in their fleet of varying sizes, one five unit ship, one four unit ship, two three unit ships, and one two unit ship. The game is played on two grids 10 by 10 units each. Before the game begins each player is to tactfully place their ships on their respective grids in secret. Ships may not be outside of the grid, overlap, or diagonal, but they can be touching.

Once the game begins each players' ships are locked in place. To start the game, one person will take a turn first. After, each player will alternate turns, itching having one move per turn. On a player's turn they will call out what grid square they would like to take a shot at, and try to hit the other person's ships. The other player will notify them if it is a hit or not. The grid is set up to read with numbers counting vertically and letters labeling horizontally.

If the player calls out a coordinate that corresponds with their opponent's ship, then their opponent must tell them it is a hit as well as which ship it is a hit on. The hit should be

recorded with a red pin on the player's board. If the player calls out a coordinate that does not hit a ship, then it is classified as a miss. A miss is recorded on the board with a white peg so the shot is not called again. After a hit or a miss the player's turn is over. It is also good practice to keep track of which ships a player has sunk on their opponents boards. To win the game a player must be the first to sink all of their opponent's ships.

As mentioned previously, Battleship also has a fair amount of probability in it. If there are five ships per grid, and the ships are five units, four units, three units, three units, and two units, that means there are 17 units occupied on the board by ships. One would assume each square has the same probability of being hit, but that is not the case. According to reference 2's "The Linear Theory of Battleship" the closer to the middle of the board, the more likely a player is to hit a ship. This difference in probability is caused by the way ships can be laid out, a ship placed in a corner can be positioned in two ways, whereas a ship in the middle can be laid out in 10 ways [2]. The chances of getting hit in the center is 20% while the chance of getting hit on the edge is 8% [2].

II. Monte Carlo Algorithm

The Monte Carlo Tree Search (MCTS) is a heuristic algorithm. A heuristic Algorithm is an algorithm that prioritizes time and near-optimal solutions, they focus on using the most favorable part of the tree using rule of thumb and educated guesses. MCTS is well known for combinational game-playing algorithms, games that are two-player, sequential and have a finite amount of defined moves [3]. It is used for games such as GO, chess, poker and various video games as well as being known to handle large numbers of possibilities or actions [4]. The algorithm is also compatible with being parallelized, which would make it more efficient and better for multi-core systems.

MCTS uses random sampling and statistics with tree-based searching in order to avoid going through every possibility and as a result, saving time. The algorithm takes the current game state and simulates multiple options while building a game tree in increments. A game tree is a tree with all of the possible moves and outcomes mapped out from a certain point in the gameplay. The simulations will run until a termination condition or a limitation in depth, after that the outcome is propagated and updated through the tree [4].There are many advantages to using MCTS, one of them being the "balance between exploration and exploitation", exploitation being the success history, and exploration being potential more efficient branches [3]. The Algorithm can achieve these by using the Upper Confidence Bound, which guides the search in one of the phases. There are four phases in the MCTS, selection, expansion, simulation and backpropagation. The selection phase is where the Upper Confidence Bound is used. A formula can be used to "define the weightage between exploitation and exploration" [3]. Which is shown as s_i [3, eq.(1).].

$$UCB(s_i) = v_i + C * \sqrt{\frac{lgN}{n_i}}$$

v_i = exploitation term

C = constant

N = total simulations done for parent node

n_i = exploration term; number of simulations done for child node

At the very beginning of the process, selection is done at random with no information, in order to learn for later. The expansion phase is when the algorithm starts going one node deeper, and the expanded node becomes the parent with the next possible moves. The simulation phase is exactly how it sounds, it simulates the game and depending on the result, a value is assigned to the node. The final phase, back propagation, is where the algorithm updates the result/value found in the simulation, where it will be used for selection again. (4, Fig. 1)

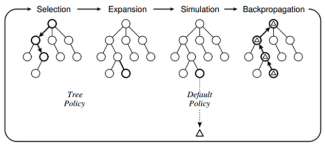

Fig. 1. Phases

Why use Monte Carlo Tree Search (MCTS)? As well as why it is specifically useful to this project it has many more advantages. MCTSs are able to learn from each simulation, improving the decision making through each trial no matter if it was a success or failure, which also means it is ideal for situations that have incomplete data or data that is not concrete. The utilization of trial and error also means that the algorithm uses what it knows to be good decisions in different branches. Other than any game setting MCTS is useful in planning, scheduling, optimization, and decision-making in real-world scenarios. Another benefit to the MCTS is its ease of implementation and domain-independence. The algorithm is easy to set up and relies on absolutely no prior knowledge. It exclusively relies on the game rules written into the rest of the project and random playouts. The MCST also maintains states. In other words, the algorithm restores game states during the simulation and backpropagation phases of the algorithm. This improves the efficiency, correct backpropagation as well as maintaining or improving the learning ability mentioned previously. The last, relevant key feature would be its adaptive growth. The tree expands dynamically based on how the simulation is progressing.

Some of the disadvantages of the Monte Carlo Tree Search include that it uses a large amount of memory. The tree grows fast, and that requires rapidly increasing the memory. In some cases, branches may not be explored enough or as thoroughly as they probably should be. This results in a rare, but possible poor decision through the tree. And finally its speed is profoundly impacted. Since there is a lot of memory being used by all of the branches, and many iterations are

needed for the best outcome, it results in the program being computationally expensive.

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:

A. Basic Phases

1) 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)
2) 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.
3) Simulation: A random playout (simulation) is performed from the newly expanded node. Moves are chosen randomly (or heuristically) until the game reaches a terminal state (win, loss, or draw). The result of this simulation (win, loss, or draw) is recorded.
4) 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.

After understanding 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 that we will follow to customize our search tree. In Battleship, some strategies are more effective than others, but let us start with the basics. There are also a few pre-existing theorized strategies for winning. There is the random strategy, which is just as it seems, picking squares at random to hit until the player has won. Adding to the random strategy, once a ship is hit, the player goes into a 'hunt' mode where they have to choose a spot next to the previous option. This is referred to as the Hunt and Target method. To further this, if each square on the battleship grid is labeled as odd and even, similar to a checkerboard (5, Fig. 2), then every ship has to be touching both an odd (white) and an even (blue) square. Reference 3 suggests that for the random search, the player only picks either the odd or even squares and for the hunt portion to choose the other around it [5]. Putting these two together results in an even more efficient strategy, one that can be used.

As stated previously, there are a few strategies that can be employed, one of them being the Hunt and Target method. The hunt and target method takes the average moves to win a battleship game from 78 to 65 [6]. This is a significant improvement on random play. The other strategy, that has been mentioned previously, is Parity. When Parity is used in combination with Hunt and Target, the average amount of moves per game decreases to 60 [6]. Another addition to common strategies is the 'Guess, Recalibrate, Repeat' strategy. Since we start the game with no information, the best initial

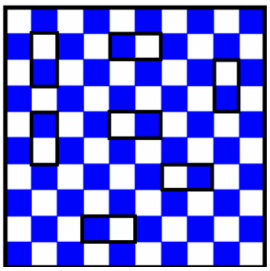

Fig. 2. Labeled Grid

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 [6].

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:

B. New Phases

1) Selection: The tree policy will guide the selection of the best child node.
2) Expansion: This phase will consider our strategies, and if the selected node is not terminal and has unvisited children, it will expand one
3) Simulation: This phase will consider our strategies, and if the selected node is not terminal and has unvisited children, it will expand one.
4) Update: The result will be back-propagated to update the node statistics.
5) Final: 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 and locking details where we implemented threading.

C. Threading Implementation

1) Parallelizing MCTS Iterations in runMCTSPhases: We modified the runMCTSPhases function to run multiple MCTS iterations concurrently. Instead of processing each iteration sequentially, we spawn multiple threads

(using C++'s ¡thread¿ library). The total number of iterations is divided among these threads so that each thread works independently on a portion of the search space.

2) Concurrent Execution of MCTS Phases: In each thread, the AI performs the four essential phases of MCTS:
   a) Selection: The thread traverses down the tree to select a node by comparing UCB (Upper Confidence Bound) scores.
   b) Expansion: Once a leaf or non-terminal node is reached, the thread expands the node by generating child nodes representing potential moves.
   c) Simulation: The thread simulates a random playout from the current node to determine a possible outcome.
   d) Backpropagation: The result of the simulation is then propagated back up the tree, updating win and visit counts.
3) Global Mutex (treeMutex): Where and Why We Use Locks. Since the MCTS tree is a shared data structure accessed and modified by all threads, we introduced a global mutex. This mutex acts like a "gatekeeper" to prevent multiple threads from accessing critical sections simultaneously. Locking Critical Sections:
   a) Selection Phase: In the selection function, before a thread iterates through the children vector of a node to calculate UCB scores, it acquires the mutex. This ensures that no other thread is modifying the vector (e.g., adding new children) while the iteration is in progress.
   b) Expansion Phase: When expanding a node, we lock the section where we check if the node's children are empty and then add new child nodes. This prevents multiple threads from attempting to expand the same node at the same time.
   c) Backpropagation Phase: While updating the win/visit counts up the tree, the mutex is used to ensure that these updates happen atomically. This avoids conflicting updates from different threads that could lead to incorrect statistics.
   d) Inside the Thread Lambda: Within the lambda function in runMCTSPhases, we wrap key operations (such as checking if a node is terminal, accessing or randomly selecting a child node) with locks. This guarantees that when a thread is accessing shared data, it is not simultaneously modified by another thread.

D. Benefits of This Threading and Locking Approach

   a) Enhanced Throughput: Multiple threads allow the AI to perform many more simulations in parallel, meaning the AI can explore a larger search space in the same time period. This leads to faster convergence on an optimal move.
   b) Efficient Use of Multi-Core Processors: Modern CPUs have several cores. By threading the simulation process, the AI leverages these cores, potentially reducing decision time significantly.
   c) Data Integrity: Locks (mutexes) ensure that updates to the shared MCTS tree are done safely. Without these locks, race conditions could corrupt the tree or cause segmentation faults.

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 thoroughly test our implementation, we ran the program multiple times until we achieved our target average number of shots required to win the game. Throughout the testing process, we ensured that our strategies and policies remained as simple and efficient as possible to minimize unnecessary complexity. This includes testing our threading solution with 1, 2, 4, and 8 threads in order to analyze its scalability and determine any correlation between the number of threads and our concurrent approach.

During each test run we measured the execution time of our base MCTS implementation and compared it to the execution times of MCTS with varying numbers of threads. This also allowed us to assess the impact of threading on the performance. For each implementation, we ran multiple iterations to determine whether increasing the number of threads led to more accurate simulations. On average the majority of games yielded results within our expected range but there were some games that deviated from the expected results for each strategy. However, outliers were anticipated and treated accordingly.

In addition to just testing our algorithm 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, allowing us to draw further conclusions on the effectiveness of different strategies and threading configurations.

V. Evaluation

The results from our Monte Carlo Tree Search (MCTS) implementation with concurrent threading across multiple cores are satisfactory with 2 threads and the same with more than 2. We can conclude that the threading implementation improved the overall performance of MCTS in terms of speed by around 45% to 55%, although with limited improvements in accuracy. From a speed perspective, we can conclude that adding 2 threads helps us navigate the tree faster. It enables us to reach conclusions in a half a fraction of the time, due 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. We have averaged 60 to 90 moves to win with the occasional outlier of 100 moves. The outlier can be attributed to the randomness and was expected. 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.

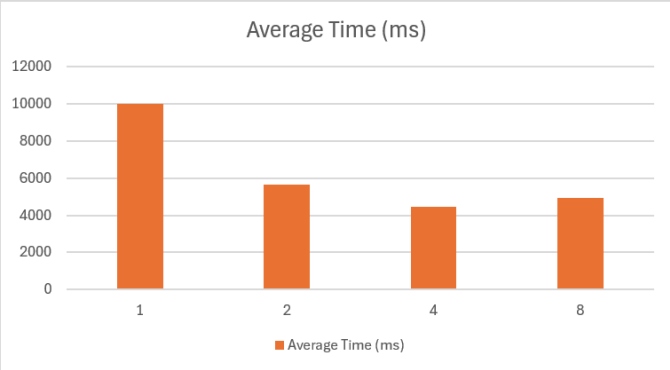

Fig. 3. Phases

Upon further investigation, we can see that the higher the number of threads achieve the same performance as 2 threads due to bottleneck in our implementation. 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 the use of only the finite amount of 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.

Threading limitations included. Lock Contention, Although threading increases throughput, the global mutex can become a bottleneck. If many threads are waiting to access critical sections, only a few threads can do useful work at a time, limiting performance improvements. Amdahl's Law, The portions of the algorithm that require locking are inherently sequential. This limits the maximum possible speedup, even when many threads are available. Thread Management Overhead, Creating and managing threads incurs overhead. If the work between lock acquisitions is minimal, the benefits of parallelism might be reduced by the overhead of context switching and scheduling. Physical Core Limitations, The actual performance boost depends on the number of physical cores available. If your machine only has a few cores, adding more threads won't proportionally increase performance.

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 less 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 relative to the previous move in order 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 occurs after 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, as mentioned previously. 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. In other words, human decision making cannot always be predicted or follow a pattern. 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 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 the AI needs 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

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

[2] Alemi. "The Linear Theory of Battleship." The Virtuosi, 3 Oct. 2011, thephysicsvirtuosi.com/posts/old/the-linear-theory-of-battleship/. Accessed 26 Feb. 2025.

[3] Radke, Parag. "Monte Carlo Tree Search: A Guide — Built In." Builtin.com, 20 July 2023, builtin.com/machine-learning/monte-carlo-tree-search. Accessed 23 Feb. 2025.

[4] 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.

[5] Barry, Nick. "Battleship." Www.datagenetics.com, 2011, www.datagenetics.com/blog/december32011/.

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