Jeremy Newman - jrn2144
Prof. Maxwell Levatich
COMS W4995 - Parallel Functional Programming
12/17/2025

Connect Four Solver Using Monte Carlo Tree Search

Overview

The purpose of this project is to implement an AI player for the Connect 4 game, and then to parallelize the algorithm to improve the performance. The AI player uses the Monte Carlo Tree Search (MCTS) algorithm as its strategy. I use Haskell's built in parallelism libraries to create performance improvements. The goal is to see how much of a speedup the decision-making algorithm can experience using search-tree parallelism. I use Threadscope and other measurement tools to quantify the performance across varying numbers of CPU cores/threads.

Background

Connect 4 is a two-player turn based game that is played in a 6 row x 7 column grid. Each player is assigned a color (red or yellow) and takes turns placing discs into columns that drop all the way down. The objective is to form a line of 4 of your color's discs in a row, either horizontally, vertically, or diagonally. The Monte Carlo Tree Search algorithm is a common algorithm used by AI in turn-based games and searches the game tree to choose the optimal move. This is an interesting choice of an algorithm as the recursive subtree exploration lends itself well to Haskell's ability to evaluate independent subtrees in parallel. Monte Carlo Tree Search balances exploration (trying new moves) and exploitation (choosing moves that are known to be good) when selecting which nodes to explore. The algorithm uses 4 key steps: selection, expansion, simulation, and backpropagation. In the selection step, it traverses the tree using a scoring policy (often the UCT policy shown in Figure 1 below) and chooses which child node to explore based on this policy. This is the step that works to balance the exploration and exploitation features. In the expansion step, the algorithm adds child nodes from the selected node into the list of possible moves. The simulation step plays out the game from that node onward using randomly generated moves and determines which player eventually wins the simulation. Lastly, the backpropagation step propagates the result of the game up the tree. It backtracks the path that led to the current leaf node and updates each node's value of number of visits and number of wins. Unlike other popular algorithms like Minimax, the Monte Carlo Tree Search algorithm is able to explore large search spaces efficiently without having to do full tree traversals. It also doesn't need to use heuristic functions and can benefit more from hardware speedups.

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

In which

- w_i = number of wins after the i-th move
- n_i = number of simulations after the i-th move
- c = exploration parameter (theoretically equal to √2)
- t = total number of simulations for the parent node

Figure 1 - Urticaria Control Test (UCT) Score Equation

Hardware Specifications

Brand	Intel
Model	Core i5-8259U
Cores	4
Hardware Threads	8

Methodology

For this project, I completed four main steps to implement and measure the performance of the Connect Four solver using MCTS: 1. Implement sequential Connect Four solver, 2. Measure performance of sequential solver, 3. Implement parallel Connect Four solver, 4. Measure performance of parallel solver. To create the sequential Connect Four solver, I created 3 files: Board.hs, SequentialMCTS.hs, and Main.hs. Board.hs sets up the Connect Four board and exposes functions like createEmptyBoard, availableCols, placeTile, and checkWin. SequentialMCTS.hs is where the bulk of the solver logic lives and exposes functions like rootNode, mctsAlgo, and bestMove. Main.hs includes the code that calls the solver algorithm with iterations. The sequential solver defines a Node data type that keeps track of the nodeId, the board state, the color, the children nodes, the number of times the node has been visited, and the number of wins the node has yielded. Mirroring the 4 steps of the MCTS algorithm (selection, expansion, simulation, and backpropagation), there are the following functions: selectPath, expand, play, and backpropagate. There are multiple helper functions like score which handles the calculation of the UCT scoring equation and oneMcts which performs one iteration of the MCTS algorithm. The next step was to measure the performance of the solver. I used the command “stack exec connect4 5000 -- +RTS -N1 -l -s” where 5000 is the number of MCTS iterations to run, and -N1 is the number of threads to run it with. I ran this command 8 times (from 1 to 8 threads) and recorded the time it took to run. I wasn’t expecting too much of a speedup since it is a sequential algorithm, not parallel yet. I also ran the command “threadscope connect4.eventlog” to analyze the threadscope graphs in order to see how the threads are splitting up the work. Next, I created ParallelMCTS.hs, a parallelized version of the Connect Four solver. This introduced new functions such as parallelPlay and parallelOneMcts in order to accomplish this. The parallelization strategy I employed was intra-iteration parallelism. Since the selection and backpropagation steps need to be done sequentially, and the expansion step is almost trivial, I decided to focus on the simulation step. Instead of performing one random simulation per iteration, the algorithm performs multiple simulations in parallel from the same selected leaf node. I chose this because simulations are independent and CPU intensive. I then measured the performance of this solver in the same way as the sequential solver and recorded the performance results. The results for both solvers are discussed in the next section.

Results

After measuring the times the sequential solver took using a varying number of threads, I created a graph showing speedup vs. threads (as shown in Figure 2 below). The results show a very slight speedup trend as more threads are increased but range from 1.280 (1 thread) seconds to 1.136 seconds (8 threads). I also looked at the threadscope graph across the 8 threaded version (shown in Figure 3 below). The results show one thread doing the vast majority of the work and extremely little overlap in the thread usage across the 8 threads. This minimal speedup and unequal thread usage was expected as a sequential algorithm shouldn't benefit very much from adding threads. After measuring the times the parallel solver took using a varying number of threads, I created a graph showing speedup vs. threads (as shown in Figure 4 below). The results show a considerable speedup trend as more threads are increased. They range from 4.365 seconds (1 thread) to 2.175 seconds (8 threads). I also looked at the threadscope graph across the 8 threaded version (shown in Figure 5 below). The results show a really equal distribution of work across the threads for around 75% of the time, and then falls back to sequential behavior. These results are expected for the parallel solver as the simulations are able to happen much faster in parallel. Since I only added parallelization to that portion of the algorithm, it did not speed up the whole process. One interesting thing that sticks out is that the parallelized version takes much longer than the sequential version. At first, I thought this was due to the added overhead of parallelizing the threads, but I realized that each iteration is performing 8 simulations rather than just 1 random one. So even though the overall time taken to complete 5000 iterations is 2 to 4 times longer than the sequential version, the time per simulation is actually much faster. Another thing to note is that the parallelized threadscope graph has two clear sections of time. The first is extremely parallelized, and the second is entirely sequential. I had trouble identifying what this sequential part of the code was, but I believe it was related to IO.

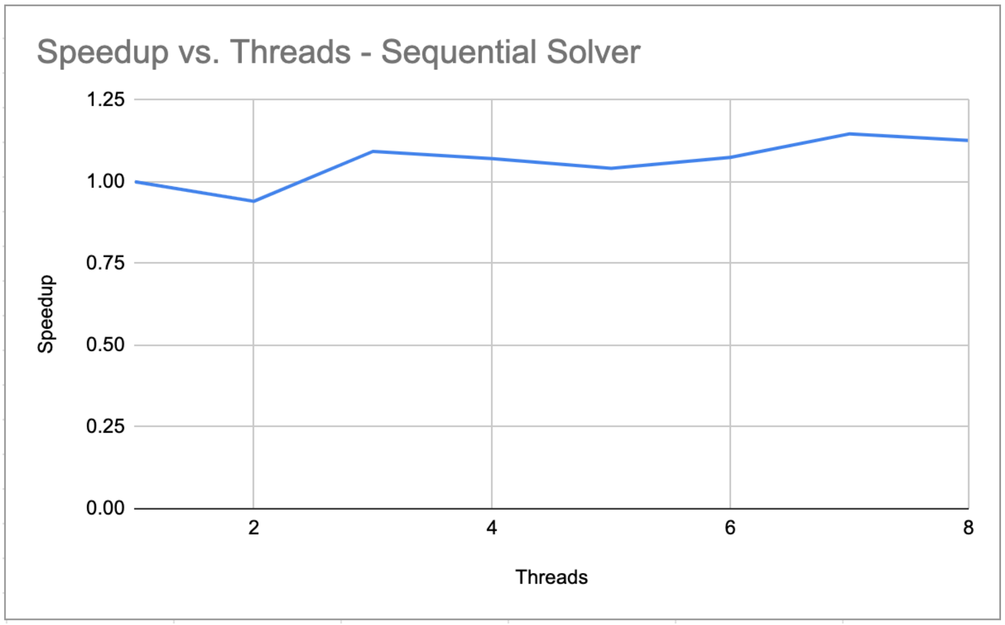
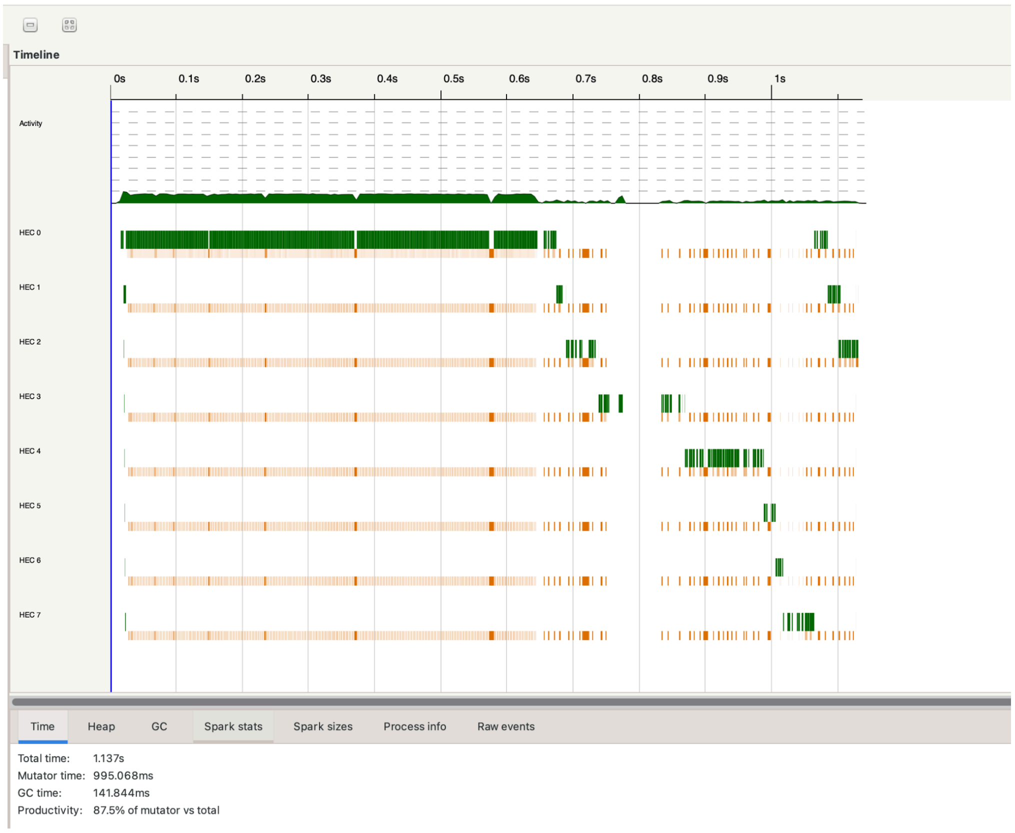


Figure 2 - Speedup vs. Threads Graph for Sequential MCTS Solver



```
Total time:     1.137s
Mutator time:   995.068ms
GC time:        141.844ms
Productivity:   87.5% of mutator vs total
```

Figure 3 - Threadscope Graph for 8 Threaded Version of Sequential MCTS Solver

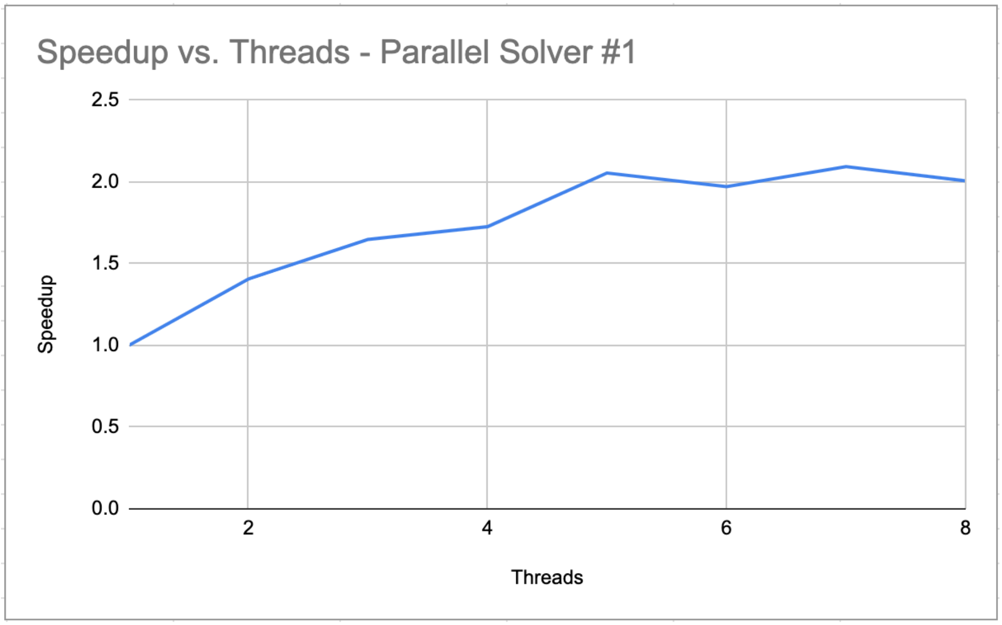
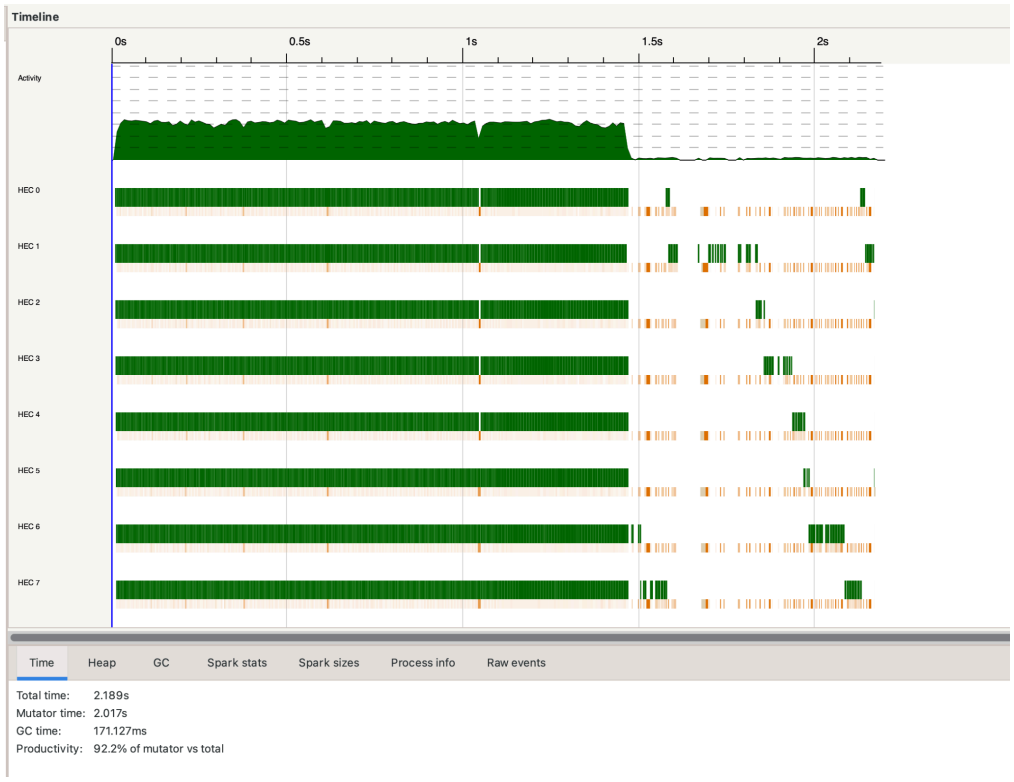


Figure 4 - Speedup vs. Threads Graph for Parallel MCTS Solver



```
Total time:     2.189s
Mutator time:   2.017s
GC time:        171.127ms
Productivity:   92.2% of mutator vs total
```

Figure 5 - Threadscope Graph for 8 Threaded Version of Parallel MCTS Solver

Future Work

Unfortunately, I ran out of time before completing all that I would have liked to on this project. There are a few key ideas that I wish I had time to explore, and I would consider as steps for future work. First of all, I would like to investigate the performance of the parallelized solver a bit more on a per-simulation basis. I would normalize the performance metrics by simulation count rather than iteration count. It is problematic to have two different loads of work being done by the different algorithms when comparing the two. I would also like to test the performance of the parallel version to see if it performs "better" in the game to see if it benefits from having more simulations done per iteration. Another piece of future work would be to try another parallelization strategy. I would like to try doing root parallelization which would allow the algorithm to run independent MCTS iterations. Each of the iterations would have its own tree from the root. They would run in parallel and merge the statistics to choose the best move. Lastly, if this helped speed up the algorithm, I would want to measure the performance differences of having 1. No parallelization, 2. Intra-iteration parallelization only, 3. Root parallelization only, 4. Combining both strategies of parallelization.