PFP Final Project: Parallel MCTS on Tic Tac Toe and Go 5 x 5

Xinchen Zhang

December 17, 2025

1 Problem Statement

Monte Carlo Tree Search has become a standard approach for decision making in game AI due to its asymptotic optimality and domain independence. The algorithm is inherently compute-intensive which makes it well-suited for our parallelization task. Most of its computational cost lies in the simulation step, and individual simulations are largely independent. This allows us to run many MCTS searches in parallel with minimal synchronization. In a purely sequential implementation, this computational cost limits real-time decision making, especially for games with larger state spaces. Parallelism can help accelerate MCTS effectively.

The goal of this project is to implement a generic, parallel Monte Carlo Tree Search (MCTS) algorithm in Haskell and evaluate its effectiveness across games. Tic-Tac-Toe is a small, deterministic, perfect-information game played on a 3×3 grid, where players alternate placing marks and the game ends upon achieving three in a row or filling the board. Go 5x5 is a substantially more complex game with a larger state space, capture mechanics based on liberties, and termination conditions involving consecutive passes. By applying the same MCTS framework to both games, this project aims to achieve meaningful parallel speedup while preserving move quality, and to analyze how factors such as computational granularity, memory allocation, and parallel overhead influence scalability and performance.

2 Game Implementations

2.1 Game Typeclass Abstraction

Games are abstracted using a typeclass:

```
class Game s m | s -> m where
  currentPlayer :: s -> Player
  legalMoves :: s -> [m]
  applyMove :: s -> m -> s
  terminal :: s -> Maybe Outcome
```

The `Game` typeclass defines the minimal interface required by the Monte Carlo Tree Search algorithm. With knowing who is the current player, what are possible legal moves, how to applying a move, and how to detect terminal states. This abstraction cleanly separates game logic from search logic and enables a single MCTS implementation to operate over multiple games.

2.2 Tic-Tac-Toe

I choose Tic-Tac-Toe as it is the simplest game to validate correctness and measure parallel overhead in our MCTS implementation.

State Representation The game state is represented by a fixed-size vector of length 9, corresponding to the 3×3 board:

```
data TTT = TTT
  { board :: !(V.Vector (Maybe Player))
  , turn :: !Player
  }
```

Each board cell is either empty or occupied by one of the two players. Instead of hard code the status, I used `Maybe Player` to make sure that the absence of a piece is an explicit and type-safe case. The board is stored in a `Vector` to provide constant-time indexing and improved cache locality during rollouts.

Both fields are marked strict using bang patterns to ensure that the board and current player are evaluated eagerly, preventing the accumulation of unevaluated thunks during deep MCTS simulations.

Move Representation Moves are represented as integers in the range 0–8, directly corresponding to board indices.

```
type Move = Int
legalMoves s =
  [ i | i <- [0..8], board s V.! i == Nothing ]
applyMove s m =
  let b = board s
      p = turn s
      b' = b V.// [(m, Just p)]
      p' = if p == P1 then P2 else P1
  in TTT b' p'
```

Legal moves are computed by selecting all empty board positions, which runs in constant time, negligible compared to the overhead of MCTS control logic. Applying a move is just an indexed update, which creates a new immutable game state. It guarantees that rollouts branching from the same state remain independent and allows states to be safely shared across parallel workers without synchronization.

Terminal State Detection Terminal states are detected by explicitly checking the eight possible winning lines (three rows, three columns, and two diagonals):

```
terminal s =
let b = board s
    line a c d =
      case (b V.! a, b V.! c, b V.! d) of
        (Just x, Just y, Just z) | x==y && y==z -> Just (Win x)
        _ -> Nothing
    wins =
      [ line 0 1 2, line 3 4 5, line 6 7 8
      , line 0 3 6, line 1 4 7, line 2 5 8
      , line 0 4 8, line 2 4 6
      ]
in case [w | Just w <- wins] of
     (w:_) -> Just w
     [] ->
      if null (legalMoves s) then Just Draw else Nothing
```

If no winning configuration is found, the game is declared a draw when no legal moves remain.

2.3 Go 5x5

To evaluate MCTS on a more computationally demanding task, we implemented a simplified version of the game Go on a 5×5 board. Compared to Tic-Tac-Toe, Go introduces significantly more complex state transitions and longer rollouts.

State Representation The Go game state consists of a fixed-size board, the current player, and a counter tracking consecutive passes:

```
data Go = Go
  { board :: !(V.Vector Cell)
  , turn :: !Player
  , passes :: !Int
  }
```

```
idx :: Int -> Int -> Int
idx r c = r * 5 + c
data Cell = Empty | Black | White
```

The board is represented as a vector of 25 cells, each of which is typed as `Empty`, `Black`, or `White`. The board is stored as a one-dimensional vector of length 25. To map a two-dimensional board coordinate (r, c) to a vector index, we encode it by row * 5 + column. This mapping assigns each cell a unique index in the range $[0, 24]$, with rows laid out contiguously in memory.

Move Representation Moves are represented as (r,c) in Integer. A `Place` move specifies a board coordinate, amd `Pass` represents the Go-specific action of passing a turn.

```
data Move = Place Int Int | Pass
legalMoves s =
  Pass :
  [ Place r c
  | r <- [0..4], c <- [0..4]
  , board s V.! idx r c == Empty
  , legalPlacement s r c
  ]
```

Legal moves include passing and all valid stone placements. The legalPlacement check ensures that after placing the stone, removing any adjacent opponent groups that become captured, the newly placed stone will have at least one liberty, preventing 'suicide' moves.

```
applyMove s Pass =
s { turn = other (turn s)
  , passes = passes s + 1
  }

applyMove s (Place r c) =
let p = turn s
    b0 = board s
    b1 = b0 V.// [(idx r c, stone p)]
    b2 = removeCaptured b1 (other p) r c
in Go b2 (other p) 0
```

Applying a move including pass and place. A `Pass` move simply switches the current player and increments consecutive passes counter by 1. For a placement move, the current player's stone is first placed on the target cell, producing an intermediate board. The board is then updated by removing any adjacent opponent groups that have lost all liberties. Then the turn is switched to the opposing player and the pass counter is reset.

Terminal Conditions and Scoring The game terminates after two consecutive passes:

```
terminal s
  | passes s >= 2 = Just (scorePosition s)

scorePosition :: Go -> Outcome
scorePosition s =
  case compare black white of
    GT -> Win P1
    LT -> Win P2
    EQ -> Draw
  where
    b = board s
    black = length [ () | x <- V.toList b, x == Black ]
    white = length [ () | x <- V.toList b, x == White ]
```

The winner is the player with more stones on the board at termination.

3 Monte Carlo Tree Search Implementation

The implementation contains four standard step for Monte Carlo Tree Search: selection, expansion, simulation, and backpropagation.

3.1 Selection

The selection phase traverses the search tree by repeatedly choosing the child node with the highest UCT score.

```
selectChild :: Game s m => Node s m -> (m, Node s m)
selectChild node =
  let p = currentPlayer (st node)
      pv = max 1 (visits node)
  in maximumBy
       (comparing (childUctScore p pv . snd))
       (M.toList (kids node))
```

The score is computed by `childUctScore`:

$$\mathrm{UCT}(v) = \begin{cases} \dfrac{w_{P1}}{n} + C\sqrt{\dfrac{\log N}{n}}, & \text{if Player 1 is to move,} \\ \left(1 - \dfrac{w_{P1}}{n}\right) + C\sqrt{\dfrac{\log N}{n}}, & \text{if Player 2 is to move.} \end{cases}$$

where N is the visit count of the parent node, n is the visit count of the child node, and C is a constant controlling the exploration–exploitation trade-off.

3.2 Expansion

If the selected node has untried actions, the algorithm expands the tree by selecting one such action and creating a new child node. Each node maintains an explicit list of untried actions and a map of move ⇒ expanded children.

```
case untried node of
  (m:ms) ->
    let s' = applyMove (st node) m
        child0 = newNode s'
        nodeExpanded =
          node { untried = ms
               , kids = M.insert m child0 (kids node)
               }
```

3.3 Simulation

After expansion, a rollout is performed from the newly created child state. The rollout proceeds by repeatedly selecting a random legal move until a terminal state is reached.

```
rollout :: Game s m => s -> StdGen -> (Outcome, StdGen)
rollout s0 g0 =
  case terminal s0 of
    Just out -> (out, g0)
    Nothing ->
      let ms = legalMoves s0
      in if null ms
         then (Draw, g0)
         else
           let (i, g1) = randomR (0, length ms - 1) g0
               m = ms !! i
           in rollout (applyMove s0 m) g1
```

Random moves are sampled uniformly from the set of legal actions by threading `StdGen`.

3.4 Backpropagation

Once a rollout terminates, its outcome is propagated back up the tree. Each node along the traversal path has its visit count incremented and its accumulated win statistic updated.

```
outcomeToP1Score :: Outcome -> Double
outcomeToP1Score (Win P1) = 1
outcomeToP1Score (Win P2) = 0
outcomeToP1Score Draw = 0.5
```

```
child1 =
  child0 { visits = visits child0 + 1
         , winsForP1 = winsForP1 child0 + score
         }

node' =
  nodeExpanded
    { visits = visits nodeExpanded + 1
    , winsForP1 = winsForP1 nodeExpanded + score
  }
```

Outcomes are mapped to numeric rewards from Player 1's perspective and accumulated uniformly throughout the tree.

4 Parallelization Mechanism

Our parallelization allows multiple independent MCTS searches runnning in parallel from the same initial state, and their statistics are merged only at the root.

4.1 Independent Parallel Trees

Rather than sharing a single tree among threads, we run multiple independent MCTS searches in parallel. Each worker performs a fixed number of iterations and produces statistics for the root's immediate children.

Concretely, a single worker executes:

```
runMctsRootStats :: Game s m => Int -> s -> StdGen -> M.Map m Stats
runMctsRootStats iters s g = rootStats (runMcts iters s g)
```

Here `runMcts iters s g` constructs a complete MCTS tree rooted at the initial state `s` using the worker's private random generator `g`. Only the *root statistics* are extracted:

```
rootStats :: Game s m => Node s m -> M.Map m Stats
rootStats t =
  M.map (\c -> (winsForP1 c, visits c)) (kids t)
```

This design avoids synchronization so that workers never share nodes or update common data structures during search.

4.2 Parallel Evaluation with `parList rdeepseq`

Parallelism is implemented using `Control.Parallel.Strategies`. We compute a list of per-spark partial results (statistics maps), and evaluate the list in parallel via `parList rdeepseq`:

```
partialStats =
  withStrategy (parList rdeepseq)
    [ foldl mergeStats M.empty
        [ runMctsRootStats itersPerWorker s0 g
        | g <- gs
        ]
    | gs <- batches
    ]
```

Each element of the outer list is one parallel task. The spark computes multiple workers sequentially (the inner list over `gs`) and merges them into a single `Map` using `mergeStats`. The runtime evaluates different sparks concurrently across available cores.

The `rdeepseq` forces the complete evaluation of each spark's result before it is returned. Without deep evaluation, sparks may only build unevaluated thunks until `maximumBy` when selecting child, causing lost parallelism and increased heap usage.

After all sparks complete, we fold the partial maps into a final map and select the best move:

```
finalStats = foldl mergeStats M.empty partialStats
pure (bestMoveFromStats s0 finalStats)
```

4.3 Chunking

We group workers into chunks to actively control parallel granularity. Each chunk is evaluated as one spark, while workers within a chunk execute sequentially.

We first generate a independent random number generator per worker, then partition them into batches:

```
gens = take workers (splitGens base)
batches = chunk chunkSize gens
```

The chunking function partitions a list into sublists of size `k`:

```
chunk :: Int -> [a] -> [[a]]
chunk _ [] = []
chunk k xs =
  let (a,b) = splitAt k xs
  in a : chunk k b
```

With smaller chunks, it create more sparks which increases scheduling overhead but improving load balancing. With larger chunks, it reduce spark overhead but may underutilize cores if there are too few sparks or imbalanced workload.

5 Results

5.1 Tic-Tac-Toe Results

Cores	Elapsed (s)	Speedup vs. N1	MUT (s)	GC (s)	Productivity (%)	Sparks Converted
N1	0.131	1.00×	0.088	0.030	67.7	0/8
N2	0.083	1.58×	0.052	0.024	62.5	7/8
N4	0.058	2.26×	0.029	0.020	49.9	7/8
N8	0.060	2.18×	0.021	0.019	35.7	7/8

Table 1: Tic-Tac-Toe parallel performance (workers=8, chunk size=1, 5000 iterations per worker).

Table 5.1 shows the parallel performance of Tic-Tac-Toe under 8 workers, chunk size of 1, and 5000 iterations per worker. The best performance is achieved at $N = 4$, with a speedup of 2.26×. We could find that increase the number of cores to 8 does not lead to further improvement, instead the performance slightly degrades compared to $N = 4$.

This might be explained that each simulation involves only a small number of moves and minimal computation due to the easiness of Tic Tac Toe, causing parallel overheads to dominate at higher core counts. As the number of cores increases, productivity declines from 67.7% at $N = 1$ to 35.7% at $N = 8$, indicating that a growing fraction of execution time is spent on garbage collection, scheduling, and runtime coordination rather than useful computation.

Workers	Chunk	Cores	Elapsed (s)	Total Iters	Productivity (%)	Speedup
4	1	N2	0.049	20,000	59.9	1.00×
4	1	N4	0.024	20,000	52.2	2.04×
4	1	N8	0.034	20,000	43.0	1.44×
4	2	N2	0.048	20,000	51.3	1.00×
4	2	N4	0.038	20,000	62.1	1.26×
4	2	N8	0.046	20,000	54.1	1.04×
4	4	N2	0.069	20,000	67.5	1.00×
4	4	N4	0.073	20,000	66.9	0.95×
4	4	N8	0.082	20,000	61.1	0.84×
8	1	N2	0.070	40,000	67.4	1.00×
8	1	N4	0.048	40,000	59.0	1.46×
8	1	N8	0.056	40,000	42.3	1.25×
8	2	N2	0.081	40,000	57.5	1.00×
8	2	N4	0.049	40,000	54.3	1.65×
8	2	N8	0.047	40,000	55.2	1.72×
8	4	N2	0.095	40,000	50.6	1.00×
8	4	N4	0.083	40,000	56.4	1.14×
8	4	N8	0.089	40,000	56.2	1.07×

Table 2: Tic-Tac-Toe performance under varying worker counts and chunk sizes. Speedup is relative to $N = 2$ case

5.2 Worker and Chunk Size Analysis (Tic-Tac-Toe)

To better understand the interaction between worker count, chunk size, and available cores, we conducted a evaluation where each worker executed 5000 iterations, resulting in total workloads of 20,000 iterations for 4 workers and 40,000 iterations for 8 workers. Experiments were run on 2, 4, and 8 cores, with chunk sizes of 1, 2, and 4.

Several trends emerge from Table 5.2. First, for smaller workloads (4 workers, 20,000 total iterations), the best performance is achieved with fine-grained parallelism (chunk size 1) on four cores, with a speedup of 2.04×. Increasing the number of cores to eight does not improve performance and instead degrades speedup due to parallel overhead and reduced per-core utilization.

For larger workloads (8 workers, 40,000 total iterations), moderate chunking becomes beneficial. With chunk size 2, the configuration achieves its best performance on eight cores, reaching a speedup of 1.72×. This suggests that coarser-grained tasks better amortize scheduling and runtime overheads when more workers are active.

5.3 Go 5x5 Results

To further evaluate the performance of our program under a larger workload, we conduct experiment on Go 5x5 with configuration of 8 workers and chunk size 1. Each worker performs 20,000 iterations.

The results demonstrate excellent scaling behavior. Compared to one core, execution time is reduced by 42% on 2 cores, 68% on 4 cores, and 75% on 8 cores. The configuration with 8 cores achieves a near-ideal 3.99× speedup, representing the best performance.

The consistently high productivity confirms that the majority of execution time is spent in computation rather than runtime overhead or garbage collection. Even at 8 cores, GC overhead remains below 10%, demonstrating that the heavy computational cost of Go rollouts effectively

Cores	Elapsed (s)	Speedup	Parallel Eff. (%)	Productivity (%)	Sparks
N1	89.288	1.00×	100.0	98.5	0/8
N2	51.691	1.73×	86.5	96.1	7/8
N4	28.563	3.13×	78.3	93.9	7/8
N8	22.401	3.99×	49.9	91.3	7/8

Table 3: Go 5x5 parallel performance (workers=8, chunk size=1, 20,000 iterations per worker)

dominates parallel runtime costs.

Compared to Tic-Tac-Toe, Go 5x5 demonstrates substantially better scalability While Tic-Tac-Toe saturates at around 2.26× speedup on 4 cores, Go 5x5 achieves nearly 4× speedup under the same parallel execution model. This highlights the importance of sufficient per-iteration computation for effective parallel Monte Carlo Tree Search.

5.4 Worker and Chunk Size Analysis (Go)

Workers	Chunk	Cores	Elapsed (s)	Speedup	Productivity (%)
4	1	N2	1.361	1.00×	94.8
4	1	N4	0.781	1.74×	92.6
4	1	N8	0.787	1.73×	89.0
4	2	N2	1.306	1.00×	95.9
4	2	N4	1.357	0.96×	94.9
4	2	N8	1.421	0.92×	92.0
4	4	N2	2.406	1.00×	97.6
4	4	N4	2.706	0.89×	95.3
4	4	N8	2.939	0.82×	92.4
8	1	N2	2.641	1.00×	95.2
8	1	N4	1.465	1.80×	92.6
8	1	N8	1.229	2.15×	89.5
8	2	N2	2.483	1.00×	96.3
8	2	N4	1.433	1.73×	93.5
8	2	N8	1.625	1.53×	91.1
8	4	N2	2.605	1.00×	96.2
8	4	N4	2.558	1.02×	94.8
8	4	N8	2.976	0.88×	92.1

Table 4: Go 5x5 performance under varying worker counts and chunk sizes

Across all configurations, the best overall runtime is achieved with **4 workers, chunk size 1, on 4 cores**, completing the workload in 0.781 seconds with a speedup of 1.74× and a productivity of 92.6%. The highest speedup is observed with **8 workers, chunk size 1, on 8 cores**, achieving a speedup of 2.15×.

We can find some different trends from Go 5x5 compared with the Tic-Tac-Toe results. First, chunk size 1 consistently outperforms larger chunk sizes across all worker counts and core configurations. Unlike Tic-Tac-Toe, Go rollouts are sufficiently expensive that fine-grained parallelism does not incur prohibitive overhead. Second, productivity remains exceptionally high across all configurations, never dropping below 89%, which indicates that the majority of execution time is spent in useful computation. The heavy computation required for liberty checks, group traversal, and capture logic effectively masks parallel overhead.

6 Conclusion

In this project, we implement a generic parallel Monte Carlo Tree Search implementation in Haskell and evaluate its performance across Tic Tac Toe and Go 5x5. By applying the same MCTS framework to Tic-Tac-Toe and Go 5x5 and, we found that effective parallelization depends critically on the granularity of per-iteration work.

For Tic-Tac-Toe, the small state space and inexpensive rollouts limit achievable speedup, with parallel overhead and garbage collection starts dominating beyond four cores. Differently, Go 5x5 exhibits substantial and sustained parallel speedup, with consistently high productivity and low runtime overhead.

Overall, the results show that our Haskell MCTS program achieves strong parallel performance on multicore systems when paired with careful evaluation strategies and well-chosen workload granularity.