Lab 2 Report - Parallel Minimax

Team Members: Russell Ferrall, Isai Tinoco Gutierrez, Curtis Bradley, Samiksha Gollamudi Karimbil

CSC 364

11 August 2025

1. Problem Definition

We implemented both sequential and parallelized versions of the Minimax algorithm for the game Dots and Boxes.

Game Description:

–Two-player, deterministic, zero-sum game.
–Players take turns drawing edges between adjacent dots. Completing a box scores a point and grants another turn.

Algorithm Goal:

–Given a current game state, determine the optimal move assuming both players play optimally.

Input:

–Current game board (size: 3x3, 4x4, or 5x5).
–Search depth limit (depth = 4 in benchmarks).

Output:

–Best move (edge) according to Minimax evaluation.

2. Parallel Solution Formulation

We used coarse-grained exploratory decomposition:

–The game tree is the search space.
–Each possible root move is treated as an independent task.
–A fixed-size thread pool is used to process all root moves in parallel.
–Each thread runs the standard Minimax search from the child state generated by applying its assigned move.

Communication Model: Shared memory via Java threads (ExecutorService). The only synchronization is during final result aggregation.

Sequential Approach:

–Standard recursive Minimax: explore the tree from the root, evaluating all possible moves sequentially.

Parallel Approach (from ParallelMaxBot.java):

–Enumerate all legal root moves.
–Wrap each move evaluation in a Callable that:
1. Applies the move to create a new MinimaxState.
2. Runs sequential Minimax on that state to the specified depth.
3. Returns the move and its evaluation score.
–Submit all Callable tasks to a fixed thread pool created with Executors.newFixedThreadPool(...).
–Run them in parallel using executor.invokeAll(tasks).
–Wait for all results, compare scores, and select the best move.
–Shutdown the executor after completion.

3. Pseudocode

Sequential Minimax:

function minimax(node, depth,

maximizingPlayer): if depth == 0 or

node is terminal: return

evaluate(node)

if maximizingPlayer:

maxEval = -infinity for each

child in generateMoves(node):

eval = minimax(child, depth-1,

false) maxEval =

max(maxEval, eval) return

maxEval else:

minEval = +infinity for each child in generateMoves(node): eval = minimax(child, depth-1, true) minEval = min(minEval, eval) return minEval

Parallel Minimax (matches ParallelMaxBot.java): function parallel_minimax(state, depth): moves = enumerateRootMoves(state) tasks = []

for move in moves: tasks.add(Callable: childState = state.withMove(move) score = minimax(childState, depth-1, false) return (move, score))

pool = fixedThreadPool(numWorkers = availableProcessors) futures = pool.invokeAll(tasks) // run all tasks in parallel

bestMove, bestScore = argmax(f.get() for f in futures) pool.shutdo

wn()

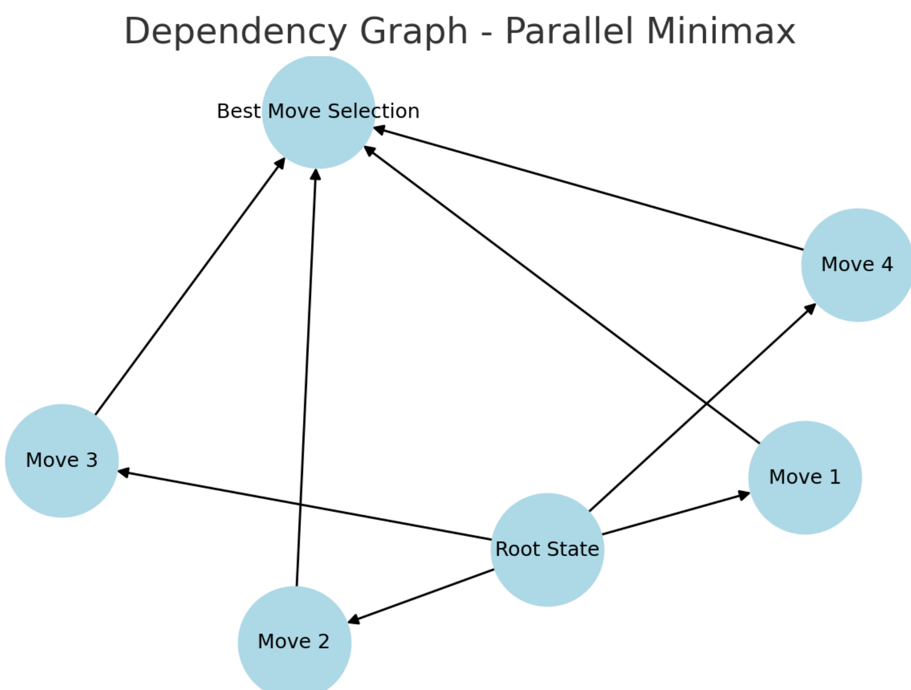
return

bestMove

4. Dependency & Interaction Graphs

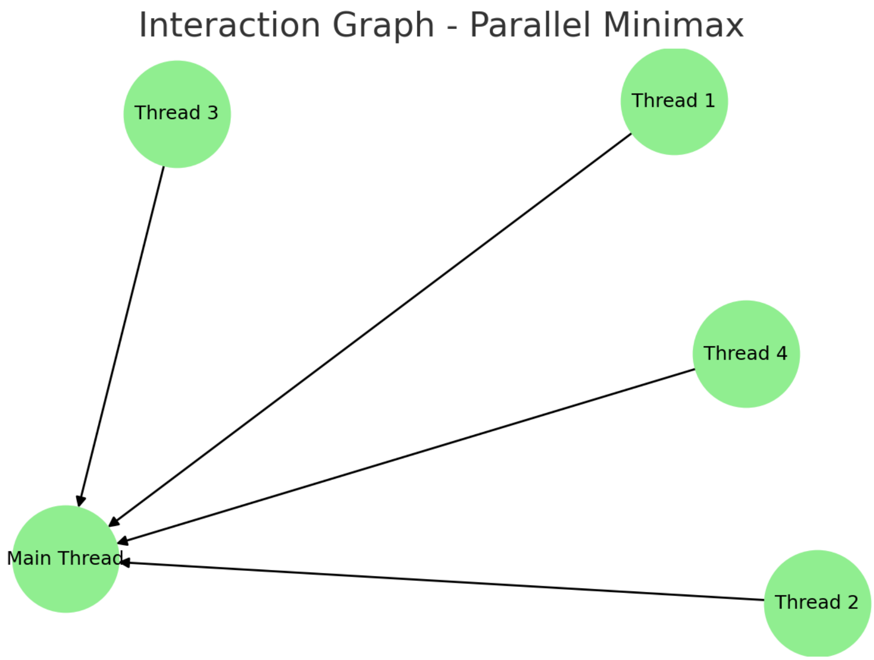
Dependency Graph:

– Each root move evaluation is independent until results are merged.
– No shared state except the final aggregation step.



Interaction Graph:

– Sequential: One process explores all branches in order.
– Parallel: Root node spawns independent tasks, each evaluating a branch to full depth, and returns results to the main thread for comparison.

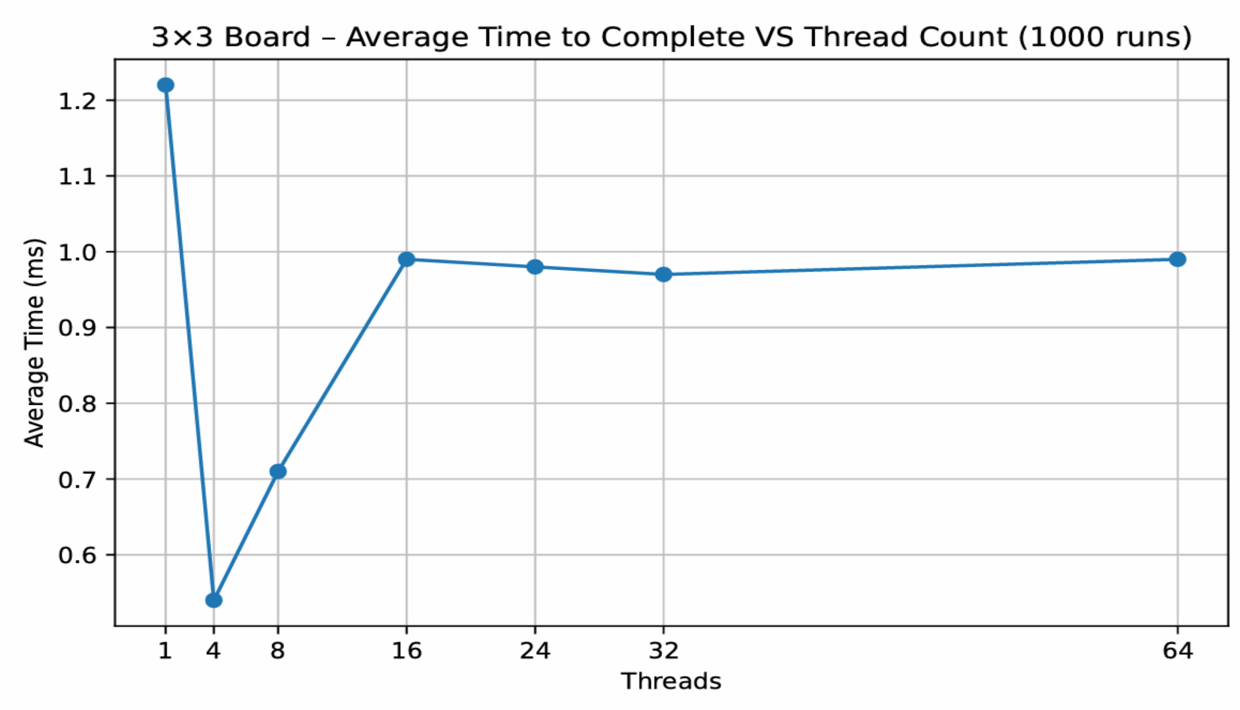


5. Runtime Analysis

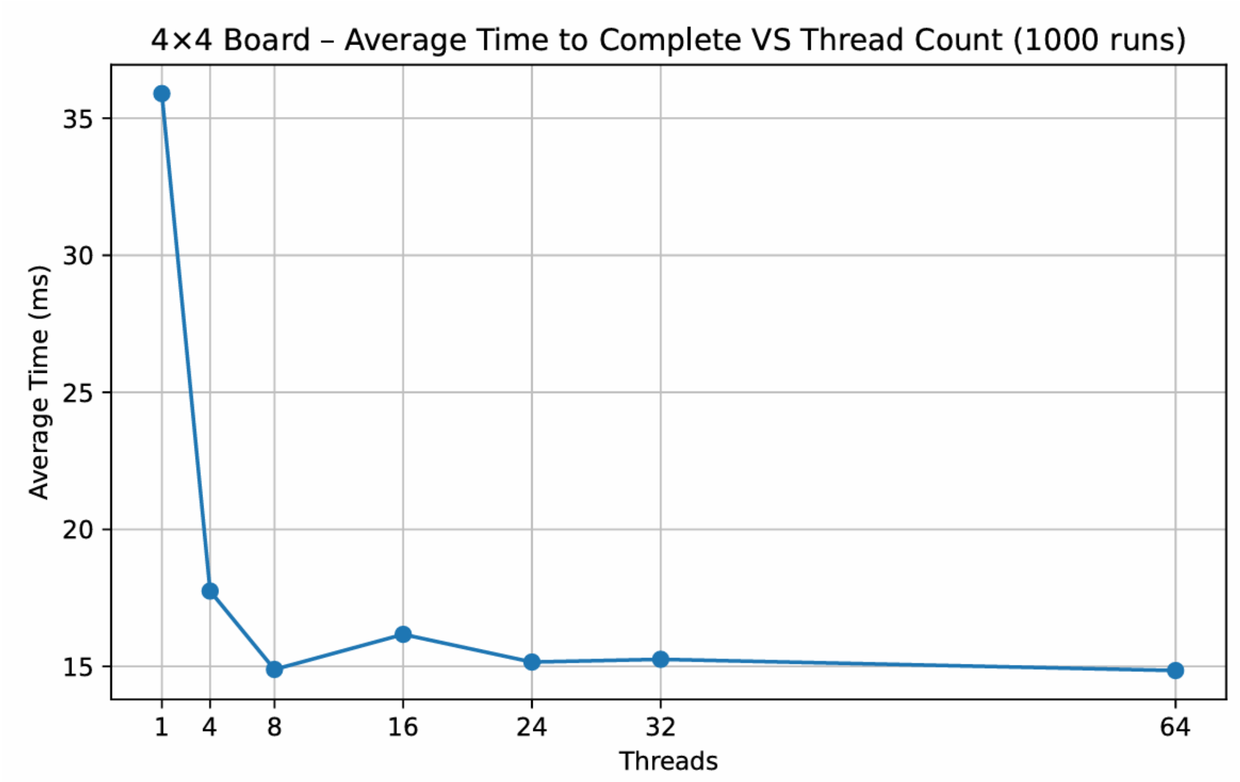
Benchmark Parameters:

–Runs: 1000 per configuration
–Depth: 4
–Boards: 3x3, 4x4, 5x5
–Hardware: Multi-core CPU with support for 64 threads
Results Graphs:

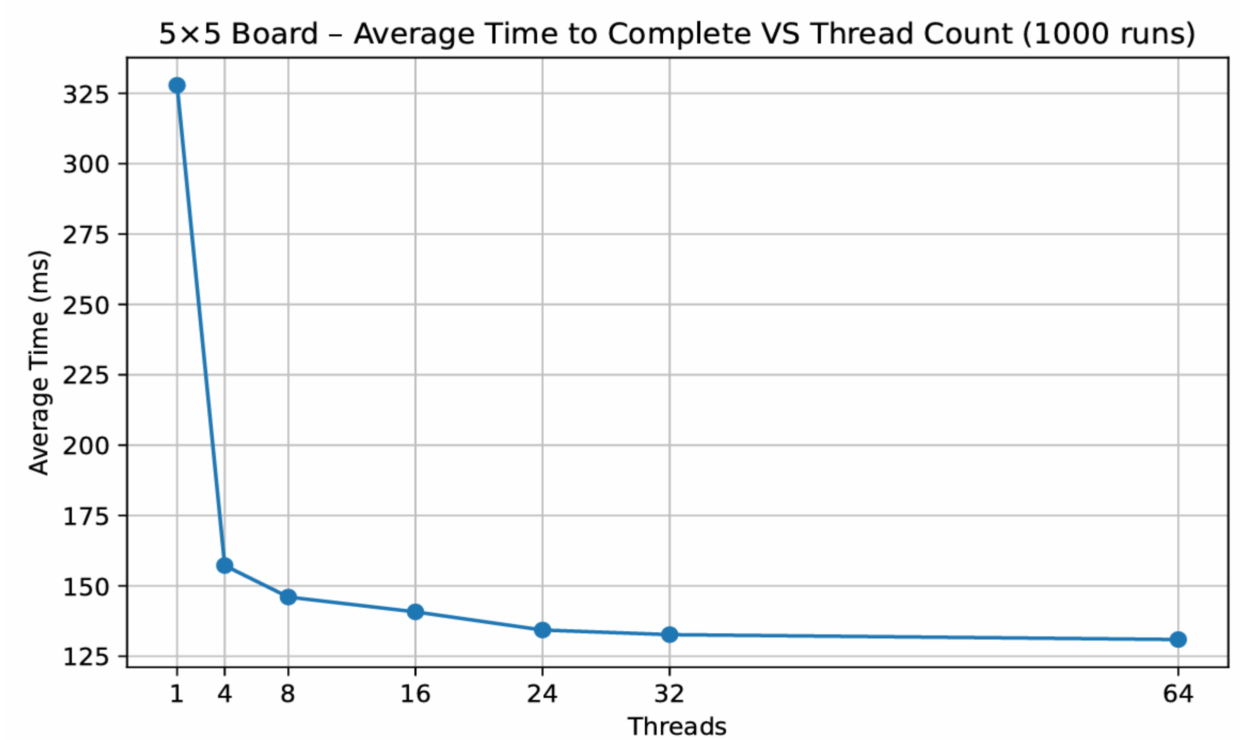
3x3 Board:



4x4 Board:



5x5 Board:



Observations:

–Significant speedup for low-to-mid thread counts.

–3x3 experienced significant speed up with 4 threads but got worse with more threads, probably because it is a small game tree.

–Diminishing returns after ~24 threads due to overhead and finite branching factor at depth 4.

–Larger boards see greater absolute improvements.

6. Conclusion

We implemented a parallel Minimax for Dots and Boxes using Java's ExecutorService with a fixed-size thread pool.

–The coarse-grained decomposition - one task per root move - minimized communication overhead.

–- Our parallel implementation achieved up to ~2.8× speedup on small boards and ~2.5× on larger boards before plateauing.

– The bottleneck at high thread counts comes from thread scheduling overhead and limited parallelizable work at shallow depths.
– Future work could include alpha-beta pruning and dynamic work-stealing to improve scalability.