Project Report - Playing Blokus with MaxN

GitHub Repository

Team Member: Pablo Ordorica Wiener (UNI: po2311)

Course: COMS W4995 - Parallel Functional Programming

- **Semester:** Fall 2025
- **Instructor:**
    - Maxwell Levatich
    - Stephen Edwards
- **TA:** Kyle Edwards

Motivation

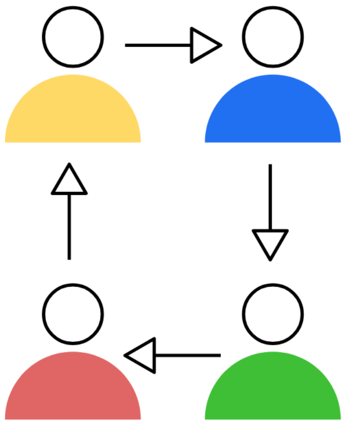
This project has two main ingredients: (1) the MaxN search algorithm and (2) the board game Blokus. The pairing was deliberate from personal motivation, a bit of a happy accident that connects to today's tech world.

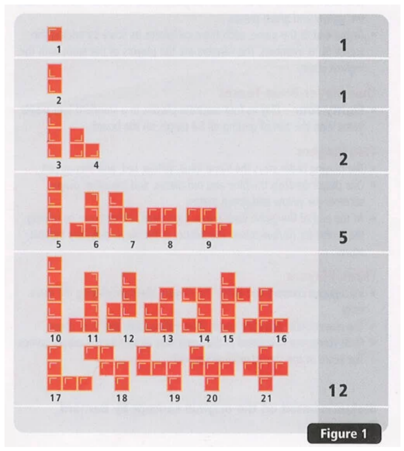
I personally enjoy multiplayer board games with hidden information, bluffing, and backstabbing—like Coup or Secret Hitler. Unfortunately, those are hard to model precisely. Blokus, in contrast, is a four-player, perfect-information game with a rich state space and large branching factor. It preserves the “multiplayer” aspect of the games I like while remaining fully observable, which makes it a good testbed for studying the parallelization of multi-agent game search algorithms.

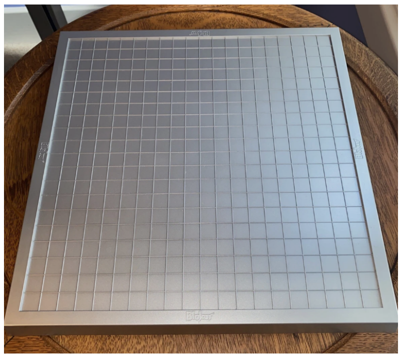
MaxN is an extension of the classical minimax algorithm from two-player games to n-player settings, where each player optimizes their own component of a joint score vector rather than a single scalar value. This kind of multi-agent search is increasingly relevant to modern “agentic” AI workflows, where several autonomous agents interact, collaborate, or compete while planning under tight computational budgets.

Blokus

Blokus is a four-player, perfect-information, turn-based strategy game played on a 20×20 grid. Each of the four players controls a set of 21 uniquely shaped polyomino pieces.







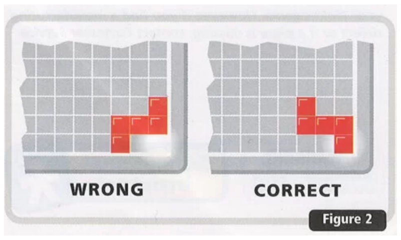
4 Players	Pieces	Board 20x20

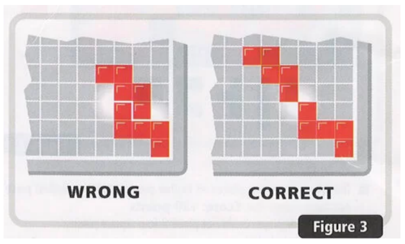
Each player's goal is to place as many of their pieces on the board as possible; the player who covers the most squares at the end of the game wins.

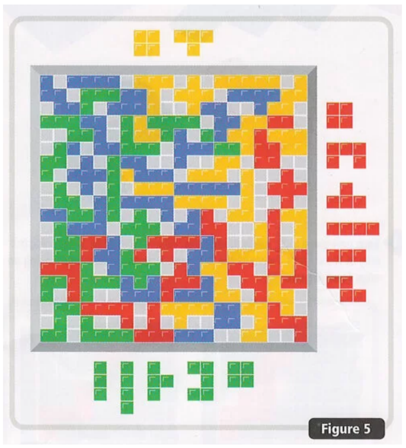
Players take turns clockwise placing one piece per turn on the board:

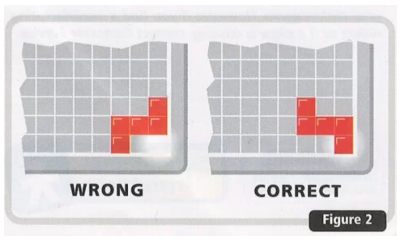
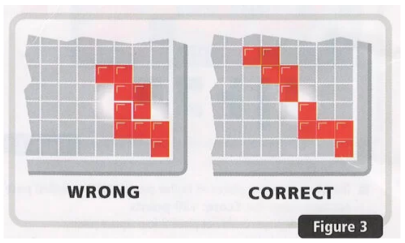
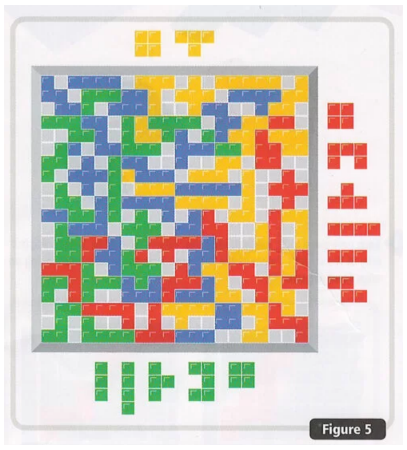
- On a player's first turn, they must place one of their pieces such that it covers the board's corner square corresponding to them.
- On all subsequent turns, a player must place a piece such that at least one corner (diagonal adjacency) of the new piece touches a corner of one of their previously placed pieces; however, no edges of the new piece may touch the edges of any previously placed piece (corner-only contact is required).
- A player must place a piece if a legal placement exists; if no legal placements are available, the player passes turn to the next player.
- Once placed, a piece cannot be moved or removed.

The game ends when all four players have passed consecutively (i.e., no legal moves remain for any player). Players count the number of unit squares in their remaining pieces, each unit square is -1 point. A player earns +15 points if they placed all their pieces and an additional 5 bonus points if the last piece they placed was the one unit square piece.







		
First turn	Rest of turns	End of game

The Blokus Game Engine

The Blokus game engine encapsulates the board, pieces, players, legal move generation, and a heuristic evaluation function in a Haskell module, `/blokus/src/Blokus.hs`.

Players and Board

The game engine has 4 players and a 20x20 board. Each board cell is either empty or occupied by exactly one player. The board is represented as an `Array` indexed by `Int` coordinates (x,y).

```haskell
data Player = P1 | P2 | P3 | P4
    deriving (Eq, Ord, Show, Enum, Bounded)

-- | Board coordinates, with (0,0) in one corner and (19,19) in the opposite
type Coord = (Int, Int)

-- | Contents of a single board cell: empty or occupied by a player.
data Cell = Empty | Occupied Player
    deriving (Eq, Show)

-- | The 20x20 Blokus board indexed by coordinates.
type Board = A.Array Coord Cell

-- | The initial empty 20x20 board with all cells set to 'Empty'.
initBoard :: Board
initBoard =
    A.array ((0,0), (19,19))
        [ ((i,j), Empty) | i <- [0..19], j <- [0..19] ]
```

Pieces

Each piece is identified by a (personally given) `PieceId`. A `Piece` stores up to 8 possible distinct orientations (rotations and mirrors) of the piece's shape.

```haskell
-- | Identifier for each of the 21 Blokus polyomino pieces.
data PieceId
    = Mono
    | Domino
    | TrominoI | TrominoV
    | TetrominoI | TetrominoL | TetrominoT | TetrominoS | TetrominoSquare
    | PentominoI | PentominoL | PentominoS | PentominoP | PentominoU
    | PentominoF | PentominoT | PentominoV | PentominoM | PentominoZ
    | PentominoY | PentominoX
    deriving (Eq, Ord, Show, Enum, Bounded)

-- | A polyomino shape as a list of squares in local coordinates.
type Shape = [Coord]

-- | Index into a piece's vector of distinct orientations.
newtype Orientation = Orientation Int
    deriving (Eq, Ord, Show)

-- | A Blokus piece: its identity and all distinct oriented shapes.
data Piece = Piece
    { pieceId      :: PieceId
    , orientations :: V.Vector Shape
    }
```

Each orientation's shape is normalized, i.e. translated on the x,y-coordinate plane so that its minimum (x, y) moves to (0, 0). This makes it easy to detect and remove duplicate shapes that arise from symmetries.

For example, consider the vertically standing domino with base shape [(0,0),(0,1)]:

```
    y
    ↑
  2 |
  1 |  ■
  0 |  ■
 -1 |
    +-|-|-|----------→ x
       0 1 2
```

Mirroring this shape across the x-axis gives [(0,0),(0,-1)]:

```
    y
    ↑
  2 |
  1 |
  0 |  ■
 -1 |  ■
    +-|-|-|----------→ x
       0 1 2
```

You and I know that this new orientation is the same shape. If we normalize the mirrored shape by “sliding it up” so that its minimum y becomes 0, we get [(0,0),(0,1)] again. After normalization and sorting, both orientations share the same shape.

Up to eight distinct orientation shapes are stored for each piece.

```
-- | Translate a shape so that its minimum x and y coordinates become (0,0),
-- and sort the coordinate list to obtain a canonical representation.
-- This is used to normalize rotated/reflected shapes so duplicates
-- can be removed.
translateToOrigin :: Shape -> Shape
translateToOrigin [] = []
translateToOrigin ps =
    let (minx, miny) = Pre.foldr
            (\(x,y) (mx,my) -> (min x mx, min y my))
            (head ps)
            ps
    in L.sort [ (x - minx, y - miny) | (x,y) <- ps ]

-- | Compute all distinct orientations (rotations and reflections) of a base
-- normalized to a common origin. The resulting vector is indexed by 'Orient
```

```haskell
orientationsOf :: Shape -> V.Vector Shape
orientationsOf base = V.fromList $ L.nub
    [ base
    , translateToOrigin $ mirrorShapeY base
    , translateToOrigin $ mirrorShapeX base
    , translateToOrigin $ mirrorShapeX $ mirrorShapeY base
    , translateToOrigin $ rotateShape90 base
    , translateToOrigin $ rotateShape90 $ mirrorShapeY base
    , translateToOrigin $ rotateShape90 $ mirrorShapeX base
    , translateToOrigin $ rotateShape90 $ mirrorShapeX $ mirrorShapeY base
    ]

-- | All pieces in the game keyed by 'PieceId', with their precomputed orien
allPieces :: M.Map PieceId Piece
allPieces =
    M.fromList
        [ (pid, Piece pid (orientationsOf shape))
        | (pid, shape) <- M.toList baseShapes
        ]
```

Game State and Moves

A `GameState` corresponds to a node in the MaxN tree that knows:

- the current board,
- whose turn it is,
- which pieces each player still has available to play,
- the cells on the board that are touching on the corner a player's pieces played where they could look to play a new piece, and
- which players have passed.

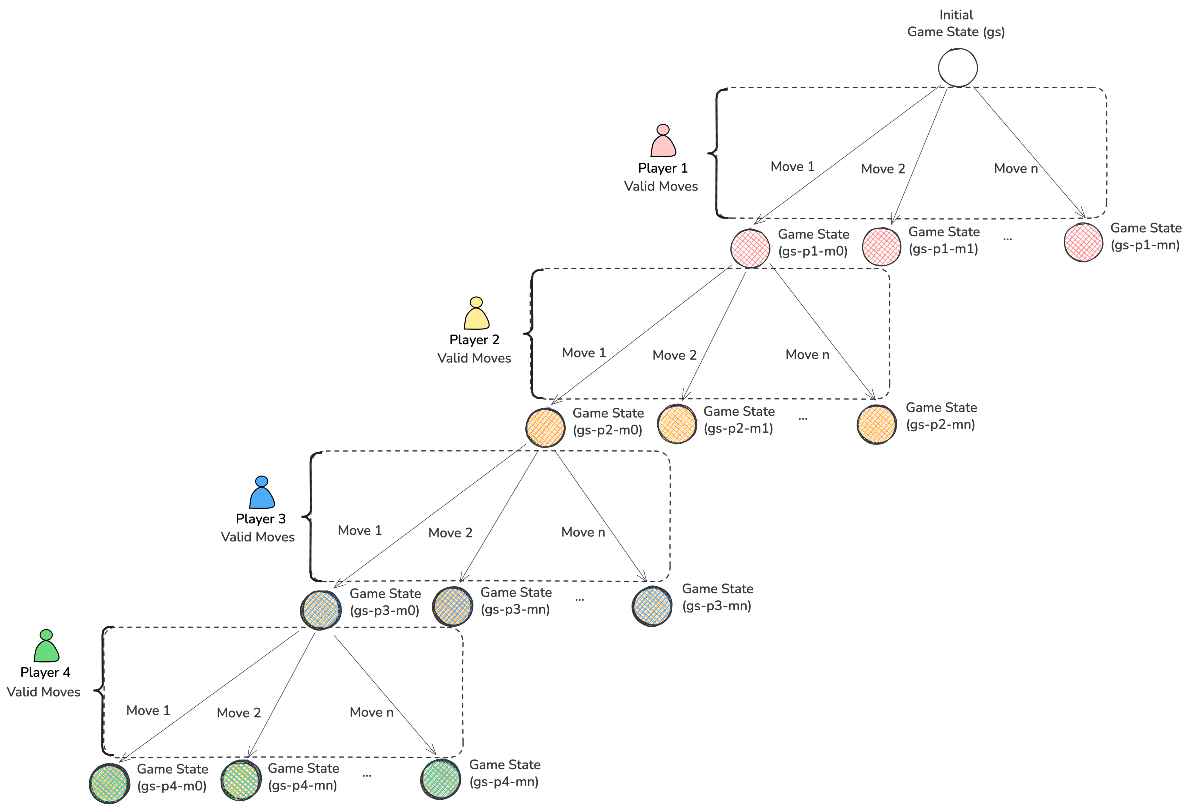
```haskell
-- | Full game state at a node in the MaxN search tree.
data GameState = GameState
    { gsBoard           :: Board
    , gsPlayerToMove    :: Player
    , gsRemainingPieces :: M.Map Player (S.Set PieceId)
    , gsCorners         :: M.Map Player (S.Set Coord)
    , gsPassed          :: S.Set Player
    }
    deriving (Eq, Show)
```

And a `Move` corresponds to the edges in the MaxN tree that take one game state to another. It corresponds to:

- which player the move belongs to,
- which piece the player is playing,
- how that piece is oriented,
- where the piece's local origin is anchored on the board.

```haskell
-- | A concrete move in the game: a player places one of their pieces on the
data Move = Move
    { mvPlayer      :: Player
    , mvPieceId     :: PieceId
    , mvOrientation :: Orientation
    , mvAnchorCoord :: Coord
    }
    deriving (Eq, Show)
```

The following diagram illustrates the game states as nodes and the moves as edges on a 4-depth MaxN tree for 4-player Blokus.



Generating, Validating and Applying Moves at a Game State

Given a `GameState`, the engine first enumerates all *candidate* moves for the player to move. `validMoves` dispatches to either the first-move generator or the normal generator depending on whether the player has placed any pieces yet:

```haskell
validMoves :: GameState -> [Move]
validMoves gs
    | isFirstMove gs = validFirstMoves gs
    | otherwise      = validNonFirstMoves gs

isFirstMove :: GameState -> Bool
isFirstMove gs =
    case M.lookup p (gsRemainingPieces gs) of
        Just ps -> S.size ps == M.size baseShapes
        Nothing -> False
    where
        p = gsPlayerToMove gs
```

Both `validFirstMoves` and `validNonFirstMoves` are implemented with a common helper, `validMovesWith`, which systematically builds `Move` values by combining:

- the current player `p = gsPlayerToMove gs`,
- each remaining `PieceId` for that player,
- each distinct orientation of that piece, and
- each candidate corner in `gsCorners` as an anchor point.

For each such combination, the engine computes the piece's absolute board coordinates and passes them to a legality predicate:

```haskell
-- | First-move variant that uses 'isValidFirstMove'.
validFirstMoves :: GameState -> [Move]
validFirstMoves = validMovesWith isValidFirstMove

-- | Normal (non-first) move generation using full Blokus rules.
validNonFirstMoves :: GameState -> [Move]
validNonFirstMoves = validMovesWith isValidMove

validMovesWith
    :: (Board -> Player -> [Coord] -> Bool)
    -> GameState
    -> [Move]
validMovesWith isValid gs =
    [ Move p pid (Orientation oi) anchor
    | let p         = gsPlayerToMove gs
    , let board     = gsBoard gs
    , let remPieces = gsRemainingPieces gs M.! p
    , pid <- S.toList remPieces
    , let piece     = getPieceById pid
    , (oi, shape) <- zip [0..] (V.toList (orientations piece))
    , corner <- S.toList (gsCorners gs M.! p)
    , (anchor, coords) <- candidatePlacements shape corner
    , isValid board p coords
    ]
```

Validation enforces the Blokus rules at the level of absolute board coordinates.

- On the first move, `isValidFirstMove` only checks that all covered cells are in-bounds, empty, and that the player's starting corner is covered.
- For all later moves, `isValidMove` additionally enforces "no edge contact with own pieces" and "at least one corner contact with own pieces":

```
-- | First-move legality: must stay in bounds, land on empty cells,
-- and cover the required corner for the current player.
isValidFirstMove :: Board -> Player -> [Coord] -> Bool
isValidFirstMove board p coords =
       all isInBounds coords
    && all (isCoordEmpty board) coords
    && coversCorner
  where
    requiredCorner :: Player -> Coord
    requiredCorner P1 = (0,0)
    requiredCorner P2 = (0,19)
    requiredCorner P3 = (19,19)
    requiredCorner P4 = (19,0)

    coversCorner = requiredCorner p `elem` coords

-- | Check whether a candidate placement of a piece is a legal Blokus move
-- for the given player on the given board.
--
-- The rules enforced are:
--
-- * All piece cells lie within the 20x20 board.
-- * All piece cells are empty on the board.
-- * No piece cell is edge-adjacent to another piece of the same player.
-- * At least one piece cell is corner-adjacent to another piece of the same
--   (or to the encoded off-board starting corner for the first move).
-- | Check whether a candidate placement of a piece is a legal Blokus move
-- for the given player on the given board.
```

```haskell
isValidMove :: Board -> Player -> [Coord] -> Bool
isValidMove board p coords =
       all isInBounds coords
    && all (isCoordEmpty board) coords
    && noEdgeTouchSameColor
    && hasCornerTouchSameColor
  where
    noEdgeTouchSameColor :: Bool
    noEdgeTouchSameColor =
        all (\c -> not (isInBounds c && cellAt board c == Occupied p))
            (findShapeAdjacent coords)

    hasCornerTouchSameColor :: Bool
    hasCornerTouchSameColor =
        any (\c -> isInBounds c && cellAt board c == Occupied p)
            (findShapeCorners coords)
```

Once a `Move` has been deemed legal, applying it produces the child `GameState` corresponding to an outgoing edge in the MaxN tree.

`applyValidMove`:

- fills the corresponding cells on the board,
- advances `gsPlayerToMove` in clockwise order,
- deletes the used piece from `gsRemainingPieces`,
- updates that player's corner frontier, and
- clears their "passed" flag:

```haskell
applyValidMove :: GameState -> Move -> GameState
applyValidMove gs m =
    GameState
        { gsBoard           = updateBoard (gsBoard gs) m
        , gsPlayerToMove    = nextPlayer (mvPlayer m)
        , gsRemainingPieces = updateRemaining (gsRemainingPieces gs) m
        , gsCorners         = updateCorners gs m
        , gsPassed          = S.delete (mvPlayer m) (gsPassed gs)
        }
    where
        updateRemaining
            :: M.Map Player (S.Set PieceId)
            -> Move
            -> M.Map Player (S.Set PieceId)
        updateRemaining remain mv =
            M.adjust (S.delete (mvPieceId mv)) (mvPlayer mv) remain
```

If `validMoves` returns an empty list for the current player, it's important to use `passTurn`, marking that player as having passed and moving the turn to the next player.

Evaluating Game States

Lastly, we need a way to compare the game states reached after different move sequences, so the Blokus module defines a Scores type that stores a 4-tuple of heuristic scores, one component per player. The heuristic follows the “Blokus Game Solver” senior project by C. Chao et al. at Cal Poly, which combines three ingredients: tiles already placed on the board (material), the number of frontier cells that touch a corner of the player’s pieces (mobility), and a penalty for the total area of remaining pieces in hand.

```
-- | 4-tuple of heuristic scores, one component per player.
-- Higher is better for the corresponding player.
data Scores = Scores Int Int Int Int
    deriving (Generic, NFData)

-- | Project the score for a given player from a 'Scores' vector.
scoreOfPlayer :: Player -> Scores -> Int
scoreOfPlayer P1 (Scores s _ _ _) = s
scoreOfPlayer P2 (Scores _ s _ _) = s
scoreOfPlayer P3 (Scores _ _ s _) = s
scoreOfPlayer P4 (Scores _ _ _ s) = s

tilesPlaced :: Player -> GameState -> Int
tilesPlaced p gs =
    length [ ()
           | ((_,_), Occupied q) <- A.assocs (gsBoard gs)
           , q == p
           ]

cornersCount :: Player -> GameState -> Int
cornersCount p gs = S.size (gsCorners gs M.! p)

remainingPieces :: Player -> GameState -> Int
remainingPieces p gs =
    sum [ pieceSize pid
        | pid <- S.toList (gsRemainingPieces gs M.! p)
        ]
```

```haskell
evaluate :: GameState -> Scores
evaluate gs = Scores (scoreFor P1) (scoreFor P2) (scoreFor P3) (scoreFor P4)
  where
    scoreFor p =
          10 * tilesPlaced p gs
        +  3 * cornersCount p gs
        -  1 * remainingPieces p gs
```

MaxN

Sequential MaxN

Note

Platform

I ran all the experiments on my computer that is a 2021 MacBook Pro with the M1 Max processor that has 10 CPU cores (8 high-performance and 2 high-efficiency cores) and 64GB of main memory.

Brand	Apple
Model	M1 Max
Cores	10 (8 high-performance + 2 high-efficiency)
Memory	64GB

In the sequential version, the search runs on a single CPU core of the M1 Max without any explicit parallelism or concurrency. At each node, the algorithm either evaluates the position directly or recursively explores all legal moves to a fixed depth or terminal state.

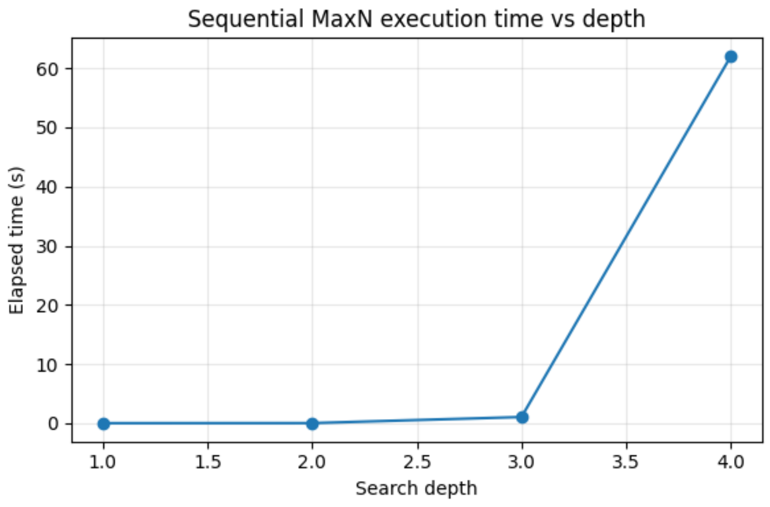
The maxN function takes a search depth and a GameState, and returns the Scores for that node:

```haskell
-- | Pure sequential MaxN (no parallelism).
maxN :: Int -> Bl.GameState -> Bl.Scores
maxN depth gs
    | depth == 0 || Bl.isTerminal gs = Bl.evaluate gs
    | otherwise =
        case Bl.validMoves gs of
            [] ->
                let gsPass = Bl.passTurn gs
                in maxN depth gsPass
            moves ->
                let p      = Bl.gsPlayerToMove gs
                    scores = [ maxN (depth - 1) (Bl.applyValidMove gs m)
                             | m <- moves
                             ]
                in maximumBy (comparing (Bl.scoreOfPlayer p)) scores
```

At each node, maxN checks a standard two-part base case: if the search depth has reached zero, or if the game is in a terminal state, it calls `evaluate` to compute the heuristic `Scores` for that position. Otherwise, it queries the Blokus rules engine for all legal `Move` values from the current `GameState`, using `validMoves`. If there are no valid moves, the current player is forced to pass, and the search continues from the resulting `GameState` with the same depth. When there are valid moves, maxN recursively evaluates every child position by applying each move with `applyValidMove`, decreasing the depth by one on each recursive call.

Because MaxN is a multi-player generalization of minimax, the selection step is player-relative: the function finds the child that maximizes the score component for the player whose turn it is at the current node. It does this by mapping the recursive scores into a list and then calling `maximumBy (comparing (scoreOfPlayer p))`, where `p = gsPlayerToMove gs`. This means every node uses the same ordering (maximize the current player's component of the `Scores` tuple) and never explicitly minimizes, which matches the MaxN definition.

The following graph shows the execution time taken for each depth on the sequential implementation. In my experiments, a search depth greater than 4 was terribly slow, and as such, I use a depth of 4 as the "working" depth for parallelization experiments.

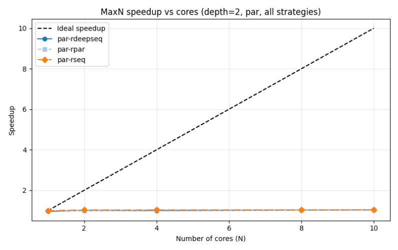


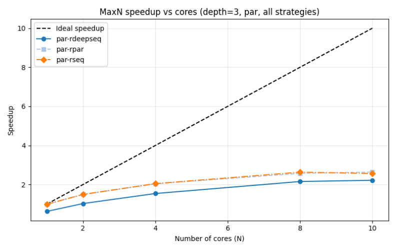
Parallel Approach 1: Spark Every Node

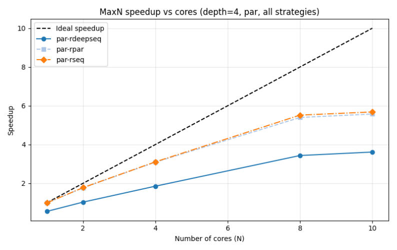
This first parallel approach keeps the MaxN logic identical to the sequential version but evaluates all children of a node in parallel using `parList strat` from `Control.Parallel.Strategies`, where strat can be parList rpar, parList rseq, or parList rdeepseq. The idea is to spark a separate parallel computation for each child score, so the runtime can distribute them across the available cores.

```haskell
-- | Parallel MaxN parameterized by a 'Strategy' for the list of child score
--
-- Example strategies:
--   * @parList rseq@
--   * @parList rpar@
--   * @parList rdeepseq@
maxNStrat
    :: Strategy [Bl.Scores]  -- ^ Strategy applied to the list of child scor
    -> Int                   -- ^ Search depth.
    -> Bl.GameState
    -> Bl.Scores
maxNStrat strat depth gs
    | depth == 0 || Bl.isTerminal gs = Bl.evaluate gs
    | otherwise =
        case Bl.validMoves gs of
            [] ->
                let gsPass = Bl.passTurn gs
                in maxNStrat strat depth gsPass
            moves ->
                let p      = Bl.gsPlayerToMove gs
                    scores = [ maxNStrat strat (depth - 1) (Bl.applyValidMov
                             | m <- moves
                             ] `using` strat
                in maximumBy (comparing (Bl.scoreOfPlayer p)) scores
```

The following speedup plots for depths 2, 3, and 4 show, as discussed in class, that “sparking at every node” only pays off once the search tree is large enough. For depth 2, the overhead of creating and managing sparks dominates, so speedup stays essentially flat at 1.0 even as the number of cores increases. At depth 3, parList rseq and parList rpar reach only about 2–2.5× on 8–10 cores, while parList rdeepseq lags slightly due to extra forcing cost. By depth 4, there is enough work per node that spark overhead is partially amortized: parList rseq and parList rpar climb to around 5.5–6× speedup on 8–10 cores, whereas parList rdeepseq tops out near 3.5×.







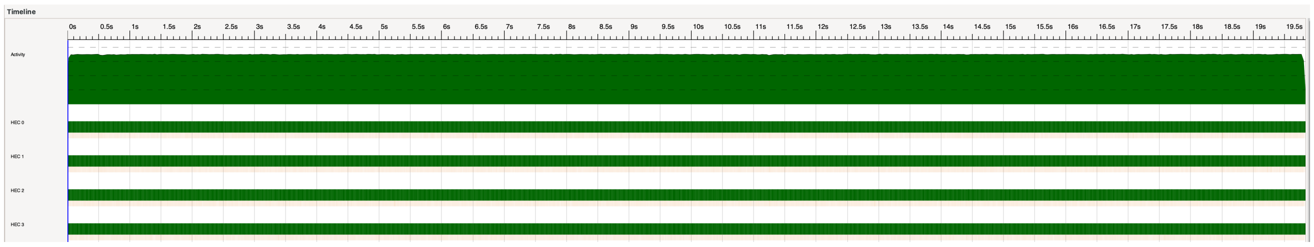
For depth 4 with 4 cores, all three "spark every node" strategies show very high spark counts, but almost all sparks are wasted (GC'd or fizzled). The main differences are in how many sparks each strategy creates, how many get converted, and how much total time they take.

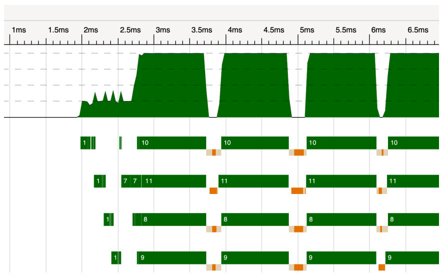
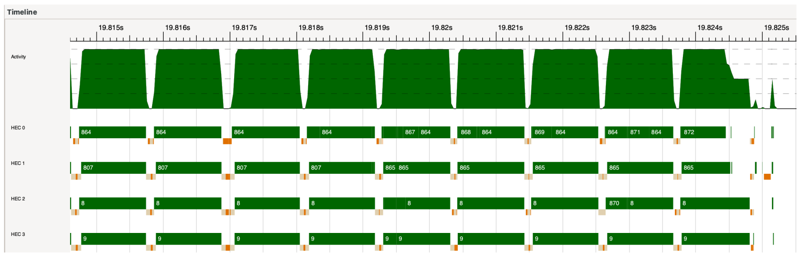
Spark statistics (depth 4, N = 4)

Strategy	Total sparks	Converted	GC'd	Fizzled	Converted %	Elapsed time (s)
par-rpar	25,489,516	5,529	24,856,744	627,243	~0.022%	19.83
par-rseq	12,740,460	754	12,547,283	192,423	~0.0059%	19.74
par-rdeepseq	12,740,460	1,360	12,455,236	283,864	~0.0107%	33.34

- The three strategies produce tens of millions of sparks at depth 4, but only a tiny fraction are converted into parallel work. Converted percentages are on the order of 0.006–0.02%; the rest are either GC'd before use or fizzle when forced by another thread.
- `parList rpar` roughly doubles the spark count relative to `parList rseq`/`parList rdeepseq`, but that does not translate into better performance; the extra sparks just add GC work and bookkeeping with only marginally more conversions.
- `parList rdeepseq` uses the same spark count as `parList rseq` but converts more sparks (1,360 vs 754) and does less GC copying. However, forcing every `Scores` to normal form inside the spark roughly doubles the mutator time compared to `rseq` (126s vs 71s user time), leading to a much worse wall-clock time (~33s).

Looking at ThreadScope, the timelines for depth 4 and 4 cores look fairly similar across strategies; below are screenshots for the start, full run, and end of `par-rpar`:



Start End

Productivity is high (>96%), so the runtime is not drowning in GC. The main issue is that the extra parallel work scheduling and deep forcing (especially with `rdeepseq`) add CPU time without enough additional overlap to compensate. The spark statistics and ThreadScope traces reinforce the takeaway from the speedup plots: fine-grained "spark every node" parallelism generates far more sparks than the runtime can use effectively, so oversparking becomes the primary bottleneck rather than raw compute.

Parallel Approach 2: Spark Up to Depth Budget

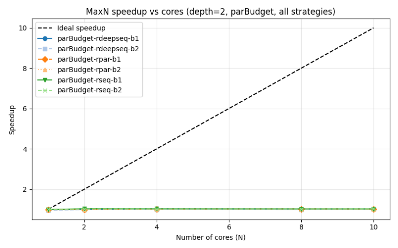
This second approach keeps the same MaxN recurrence but limits where parallelism is applied. Instead of sparking at every node, it only uses `parList strat` while a parallelization budget is positive; below that depth the search is purely sequential. This creates a small number of coarse-grained sparks near the top of the tree.

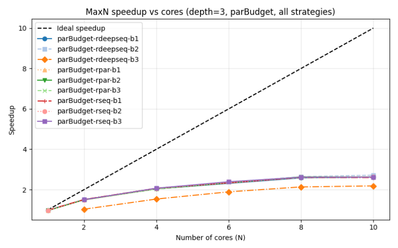
This depth-limited parallelism avoids the exponential spark explosion seen in the "spark every node" approach, while still exposing enough parallel work at large branching-factor nodes.

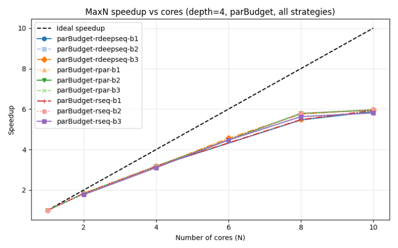
```haskell
-- | Parallel MaxN with a 'Strategy' and a parallelization budget.
--
-- 'parBudget' controls how many levels from the current node down will
-- use the strategy; below that the search is sequential.
--
-- Typical usage:
--   * @maxNStratParBudget (parList rseq)     depth 1 gs@
--   * @maxNStratParBudget (parList rdeepseq) depth 2 gs@
maxNStratParBudget
    :: Strategy [Bl.Scores]  -- ^ Strategy for the list of child scores.
    -> Int                   -- ^ Search depth.
    -> Int                   -- ^ Parallelization budget (levels).
    -> Bl.GameState
    -> Bl.Scores
maxNStratParBudget strat depth parBudget gs
    | depth == 0 || Bl.isTerminal gs = Bl.evaluate gs
    | otherwise =
        case Bl.validMoves gs of
            [] ->
                let gsPass = Bl.passTurn gs
                in maxNStratParBudget strat depth parBudget gsPass
            moves ->
                let p         = Bl.gsPlayerToMove gs
                    evalChild m =
                        maxNStratParBudget strat (depth - 1) (parBudget - 1)
                                                (Bl.applyValidMove gs m)
                    scoresBase = [ evalChild m | m <- moves ]
                    scores     =
                        if parBudget > 0
                           then scoresBase `using` strat
                           else scoresBase
                in maximumBy (comparing (Bl.scoreOfPlayer p)) scores
```

The following graphs for depths 2, 3, and 4 with `parBudget` 1-3 show a very different pattern from "spark every node". At depth 2, the tree is still too small for parallelism to help: speedup remains essentially flat at 1.0 for all strategies and budgets. At depth 3, `parBudget` with budgets 1 and 2 yields speedups around 2–2.7× on 8–10 cores for `rpar` and `rseq`, while `rdeepseq` is slightly lower but still close. At depth 4, all strategies and budgets 1–3 achieve roughly 4.5–6× speedup at 8–10 cores, with the curves for `rpar`, `rseq`, and `rdeepseq` nearly on top of each other. Limiting parallelism to the top of the tree removes most of the oversparking penalty, so the exact evaluation strategy matters much less.

Compared to "spark every node," the parBudget approach delivers similar or better speedups with much more predictable behavior and far fewer sparks.







With parBudget = 2 at depth 4 and 4 cores, depth-limited parallelism produces only a few thousand sparks instead of tens of millions, and almost all GC overhead disappears.

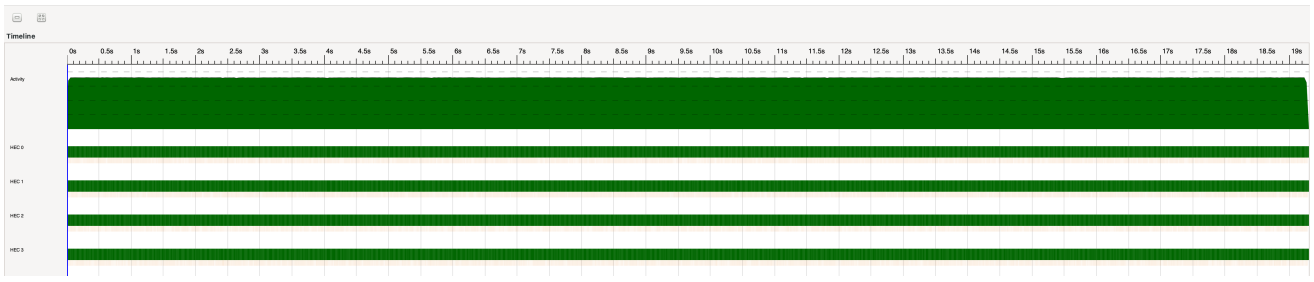
Spark statistics (depth 4, parBudget=2, N = 4)

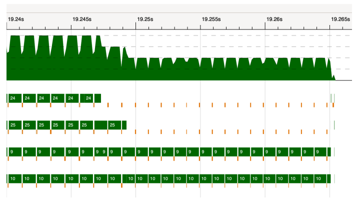
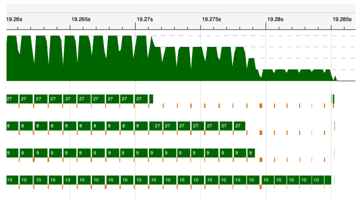
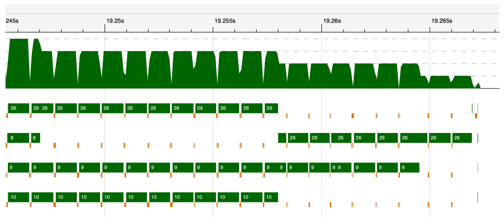
Strategy	**Total sparks**	**Converted**	**GC'd**	**Fizzled**	**Converted %**	**Elapsed time (s)**
parBudget-rpar-b2	7,320	132	87	7,101	~1.8%	19.30

Strategy	Total sparks	Converted	GC'd	Fizzled	Converted %	Elapsed time (s)
parBudget-rseq-b2	3,660	64	0	3,596	~1.7%	19.27
parBudget-rdeepseq-b2	3,660	63	0	3,597	~1.7%	19.27

- Spark counts drop from tens of millions in the full-tree approach to just a few thousand at this depth and budget.
- Conversion rates improve to around 1.7–1.8%, and there are effectively no GC'd sparks for `rseq`/`rdeepseq`; all sparks are either used or fizzle.
- All three strategies have nearly identical elapsed times (~19.3 seconds) and very similar allocation and GC profiles, which matches the overlapping curves in the parBudget speedup plots.

The ThreadScope timelines for parBudget show a clean block of parallel work, a solid band of green (mutator work) with short GC pauses and no obvious load imbalance across cores. Together with the spark stats, these traces confirm that the parBudget design achieves coarse-grained, well-utilized parallelism at the top of the tree without the massive oversparking and GC overhead of the full-tree strategy.



End rpar	End rseq	End rdeepseq

Productivity is higher (≈98% user, ≈94% elapsed) and GC time is small; most time now goes into useful MaxN search rather than spark management.

Summary of Results

The sequential MaxN implementation provides the baseline: on my M1 Max MacBook Pro, depth 4 is the deepest practical "working depth" before runtime becomes prohibitively slow.

The first parallel approach (“spark every node” with `parList` strategies) improves performance at depth 3 and especially depth 4, reaching around 5.5–6× speedup on 8–10 cores for the best case. However, it suffers from massive oversparking: tens of millions of sparks are created, and only a tiny fraction ever get converted into useful parallel work.

The second approach (`parBudget`) limits parallelism to the top levels of the tree and achieves comparable or better speedups with far fewer sparks and much cleaner runtime behavior.

Conclusion

The experiments show that parallelizing MaxN for Blokus is possible and does yield meaningful speedup. However, the large branching factor of the MaxN algorithm and the amount of work per node dominate over the specific parallelization strategy. Future work could explore parallelizing the work *inside* each node (move generation, validation, and scoring), not just the tree structure, and improving the Blokus engine itself with more efficient data types and optimized operations. Both directions would increase the amount of useful computation per spark and should have a positive impact on parallel performance.

Reproducing/Running Experiments

All experiments are driven by a Bash script which automates running the `blokus` Haskell project via a small `Main` module. The Haskell project can also be built and executed directly with Stack, independent of the script.

First change directory into the `blokus/` directory:

```
cd blokus
```

Then build the project:

```
stack build
```

You can now run the executable in its three modes.

Sequential MaxN:

For depth 4 sequential search

```
stack run -- seq 4
```

Parallel MaxN (spark every node) on 8 cores and generate spark stats and Threadscope trace:

- depth 4, parList rseq, using all available cores

```
stack run -- par 4 rseq +RTS -N8 -s -l -RTS
```

- depth 4, parList rpar

  ```
  stack run -- par 4 rpar +RTS -N8 -s -l -RTS
  ```

- depth 4, parList rdeepseq

  ```
  stack run -- par 4 rdeepseq +RTS -N8 -s -l -RTS
  ```

Parallel MaxN with depth budget (parBudget) on 8 cores and generate spark stats and Threadscope trace:

- depth 4, parList rseq, parallel budget 2

  ```
  stack run -- parBudget 4 rseq 2 +RTS -N8 -s -l -RTS
  ```

- depth 4, parList rpar, parallel budget 3

  ```
  stack run -- parBudget 4 rpar 3 +RTS -N8 -s -l -RTS
  ```

- depth 4, parList rdeepseq, parallel budget 2

  ```
  stack run -- parBudget 4 rdeepseq 2 +RTS -N8 -s -l -RTS
  ```

The Bash script, `run.sh`, lives at the root of the `blokus` project and automates whole batches of runs. It takes a couple of command line arguments to configure which experiments to run and whether or not to generate ThreadScope traces:

> Tip
>
> You can make `run.sh` executable with `chmod +x run.sh`.

- `./run.sh` – run all three implementations (`seq`, `par`, and `parBudget`), generate timing and spark stats log files in `blokus/logs/`, and generate ThreadScope `.eventlog` files.

- `./run.sh par noevents` – run only the `par` (spark-every-node) approach, generating timing and spark stats, but no ThreadScope traces.
- `./run.sh par events` – run only the `par` approach and also generate ThreadScope traces.
- `./run.sh parBudget events` – run only the `parBudget` approach with eventlogs; `./run.sh parBudget noevents` skips them.

> Important
>
> Depths, core counts, strategies, and budgets are controlled by the arrays at the top of the script (`DEPTHS`, `CORES`, `STRATS`, `PAR_BUDGETS`). Adjust those to sweep different parameter sets.