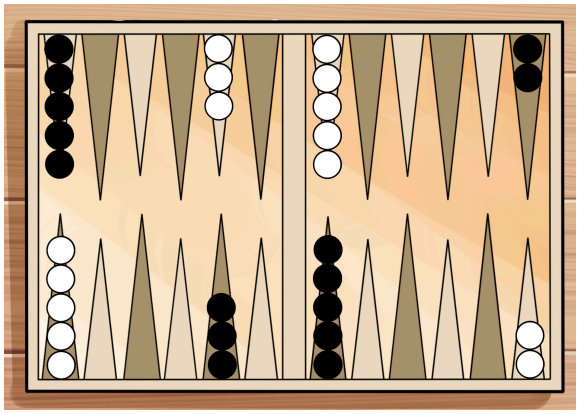



Backgammon AI in Parallel
By Zachary Singerman, zs2661

1. Introduction:

Backgammon, or Shesh-Besh in Middle Eastern cultures, is a game where two opposing players want to get each of their tiles off of the board before their opponent.

1.1. Rules

Moving:

The roll of the dice indicates how many points, or pips, the player is to move his pieces. The checkers are always moved forward, to a lower-numbered point. The following rules apply:

1. A checker may be moved only to an open point, one that is not occupied by two or more opposing checkers.
2. The numbers on the two dice constitute separate moves. For example, if a player rolls 5 and 3, he may move one checker five spaces to an open point and another checker three spaces to an open point, or he may move the one checker a total of eight spaces to an open point, but only if the intermediate point (either three or five spaces from the starting point) is also open.
3. A player who rolls doubles plays the numbers shown on the dice twice. A roll of 6 and 6 means that the player has four sixes to use, and he may

move any combination of checkers he feels appropriate to complete this requirement.

4. A player must use both numbers of a roll if this is legally possible (or all four numbers of a double). When only one number can be played, the player must play that number. Or if either number can be played but not both, the player must play the larger one. When neither number can be used, the player loses his turn. In the case of doubles, when all four numbers cannot be played, the player must play as many numbers as he can.

Capturing:

1. A point occupied by a single checker of either color is called a blot. If an opposing checker lands on a blot, the blot is hit and placed on the bar.
2. Any time a player has one or more checkers on the bar, his first obligation is to enter those checker(s) into the opposing home board. A checker is entered by moving it to an open point corresponding to one of the numbers on the rolled dice.
3. For example, if a player rolls 4 and 6, he may enter a checker onto either the opponent's four point or six point, so long as the prospective point is not occupied by two or more of the opponent's checkers.
4. If neither of the points is open, the player loses his turn. If a player is able to enter some but not all of his checkers, he must enter as many as he can and then forfeit the remainder of his turn.
5. After the last of a player's checkers has been entered, any unused numbers on the dice must be played, by moving either the checker that was entered or a different checker.

Bearing off:

1. Once a player has moved all of his fifteen checkers into his home board, he may commence bearing off. A player bears off a checker by rolling a number that corresponds to the point on which the checker resides, and then removing that checker from the board. Thus, rolling a 6 permits the player to remove a checker from the six point.
2. If there is no checker on the point indicated by the roll, the player must make a legal move using a checker on a higher-numbered point. If there are no checkers on higher-numbered points, the player is permitted (and required) to remove a checker from the highest point on which one of his

checkers resides. A player is under no obligation to bear off if he can make an otherwise legal move.

3. A player must have all of his active checkers in his home board in order to bear off. If a checker is hit during the bear-off process, the player must bring that checker back to his home board before continuing to bear off. The first player to bear off all fifteen checkers wins the game.

Thank you to https://www.bkgm.com/rules.html for providing clear rules of the game.

2. Project overview

Backgammon presents a computationally heavy challenge as every roll of the dice allows for many possible outcomes of the board, each factoring in the roll and positioning of your opponent.

This project parallelizes the expectiminimax alpha-beta pruning algorithm to track which possible moves are the best to take based on the best future board positions. Because a singular game of backgammon is generally 50 total dice rolls, it would not make sense to go until the end of the game as that would be far too deep of a tree – especially with the dice roll aspect of the expectiminimax tree branching to 21 new chance branches after every move.

The algorithm only goes at max 3 plies deep, a ply being a max and min node with chance nodes in between. This is due to the fact that going 4 plies deep is far too computationally heavy and not worth waiting for. Because expectiminimax has chance nodes, that gives every single Max and Min node an additional 21 nodes to cover based on what the roll of the dice will be for the opponent.

The expectiminimax algorithm will allow for two computer players to play against each other, each trying to win and each analyzing all of their possible moves. By using alpha-beta pruning, we will be able to eliminate the moves that lead to worse board positions. What a worse board position is is described later in the report.

To run the code, the user must either input their own setup of the board or use one of the pre-loaded examples in the code.

The goal of this project is to create a fast working Backgammon AI that knows the best possible move in a short amount of time.

2.1. Heuristics based on backgammon

My alpha-beta pruning algorithm values different nodes based on the following heuristics:

A. Borne Off Score
   a. For each piece that a player has borne off, that node is awarded 2 extra points
   b. This is weighted so heavily because bearing off pieces is the actual move that allows a player to win

B. Blots
   a. A blot is when there is a piece alone on a stack
   b. These are weighted negatively because they risk being captured and sent to the bar
   c. When a piece gets sent to the bar, they lose all of their progress and weigh down on that player
   d. For every blot, the heuristic subtracts 2 points

C. Protected Stacks
   a. When there are more protected stacks for one player, it limits the other player's ability to actually have freedom in their moves
   b. This may force them to leave certain pieces open as blots for the taking in future turns

In this algorithm, White is the Min player and Black is the Max. Therefore, Black values higher heuristics and white values lower.

2.2. The Board representation

```
data Point --Where a piece is on the board
   = P Int --On the board itself, from 1-24
   | Bar --Captured piece which has to come in from off of the board
   | BearOff --Player that can get into the winning spot
   deriving (Eq, Ord, Show)

data Player = White | Black
   deriving (Eq, Ord, Show)
```

```
type Stack = (Player, Int) --Represents all pieces on a point
                           --Player is which player owns the point
                           --Int is how many pieces are there

type BoardPoints = Map Point Stack --Maps board points to stacks

data Board = Board
   { points  :: BoardPoints    --The map of points on stacks
   , bar     :: Map Player Int --How many pieces each person has captured
   , bearOff :: Map Player Int --How many players each one has borne off
   }
   deriving (Eq, Show)
```

The board itself is represented as a data type that tracks the points on the theoretical board (spaces 1-24), the bar, and the borne off pieces. The points are of type BoardPoints, which map each point to a specific stack. Each stack holds which player controls the stack and how many pieces they have on said stack. The data type point is responsible for defining the position (P) of the board, 1-24, the bar, and the BearOff section.

2.3. The MoveGen Monad

```
type MoveGen a = StateT Board [] a
```

To support my algorithm, I made the MoveGen monad. This monad generates new moves. It holds a StateT, transformer of the State monad, and a List, [], monad. StateT allows for the board to change as moves are made and the List allows for multiple outcomes to be explored. Because of the monad, illegal moves are not explored because they fail to produce results and are therefore not stored.

3. Sequential solution

The Sequential Solution is very straightforward. The Expectiminimax algorithm is run on a particular board and it returns the moves that it finds best to make. This entails which pieces are moving from where to where and when, as order is important. It runs faster on simpler boards with pieces in higher stacks or when there are pieces on the bar, as these lead to less variance in moves.

4. Parallelization

The Parallelization of this project takes place at the root node. It parallelizes every legal move from the root. Sparks are created at rdeepseq. The algorithm only parallelizes from the root because every expectiminimax with Alpha Beta pruning is sequential. By parallelizing somewhere else, the overhead would outweigh the benefits at a deeper depth. We also maximize the utilization of our cores by parallelizing all of the legal moves of the root node.

```
scoredMoves :: [(Move, Double)]
scoredMoves =
  withStrategy (parList rdeepseq) $
    map scoreMove (moves :: [Move])
```

This parallelization generates all legal moves. After that, each move is mapped to (Move, Double). The Double is computed by the expectiminimax search. By Using withStrategy, we create the sparks. ParList goes over the list of moves, creating a new spark for a new core at each move. Rdeepseq requires a full evaluation of each move. In this case, we sacrifice speed to prevent laziness. In theory, every use of scoredMoves runs on a different core.

5. Performance Evaluations

To test the performance of the algorithms, I ran the serial and parallel algorithms at search depths 2 and 3 on five different board setups.

The five board setups are as follows:

Starting board:

```
startingBoard =
  Board
    { points =
        Map.fromList
          [ --White Pieces
            (P 24, (White, 2))
          , (P 13, (White, 5))
          , (P 8,  (White, 3))
          , (P 6,  (White, 5))

            --Black Pieces
          , (P 1,  (Black, 2))
          , (P 12, (Black, 5))
          , (P 17, (Black, 3))
          , (P 19, (Black, 5))
          ]
    , bar =
        Map.fromList
          [(White, 0), (Black, 0)]
    , bearOff =
        Map.fromList
          [ (White, 0), (Black, 0)]
    }
```

Middle Game (an average board):

```
midgameBoard :: Board
midgameBoard = Board
  { points = Map.fromList
      [ (P 3,  (White, 2))
      , (P 8,  (White, 3))
      , (P 10, (White, 3))
      , (P 16, (White, 3))
      , (P 21, (White, 2))
      , (P 5,  (Black, 3))
      , (P 12, (Black, 3))
      , (P 19, (Black, 2))
      , (P 23, (Black, 2))
      , (P 24, (Black, 3))
      ]
  , bar = Map.fromList [(White,1), (Black,2)]
  , bearOff = Map.fromList [(White,0), (Black,0)]
  }
```

Critical hit (a deciding moment in the game):

```
criticalHitBoard :: Board
criticalHitBoard = Board
  { points = Map.fromList
      [ (P 5,  (White, 4))
      , (P 7,  (White, 1))
      , (P 11, (White, 3))
      , (P 16, (White, 3))
      , (P 22, (White, 1))
      , (P 3,  (Black, 1))
      , (P 6,  (Black, 4))
      , (P 12, (Black, 1))
      , (P 19, (Black, 5))
      , (P 24, (Black, 3))
      ]
  , bar = Map.fromList [(White,1), (Black,1)]
  , bearOff = Map.fromList [(White,2), (Black,0)]
  }
```

Endgame (for White):

```
endgameWhite :: Board
endgameWhite = Board
  { points = Map.fromList
      [ (P 1, (White, 3))
      , (P 2, (White, 3))
      , (P 4, (White, 3))
      , (P 6, (Black, 3))
      , (P 12, (Black, 4))
      , (P 18, (Black, 2))
      , (P 19, (Black, 2))
      , (P 22, (Black, 3))
      ]
  , bar = Map.fromList [(White,0), (Black,1)]
  , bearOff = Map.fromList [(White,6), (Black,0)]
  }
```

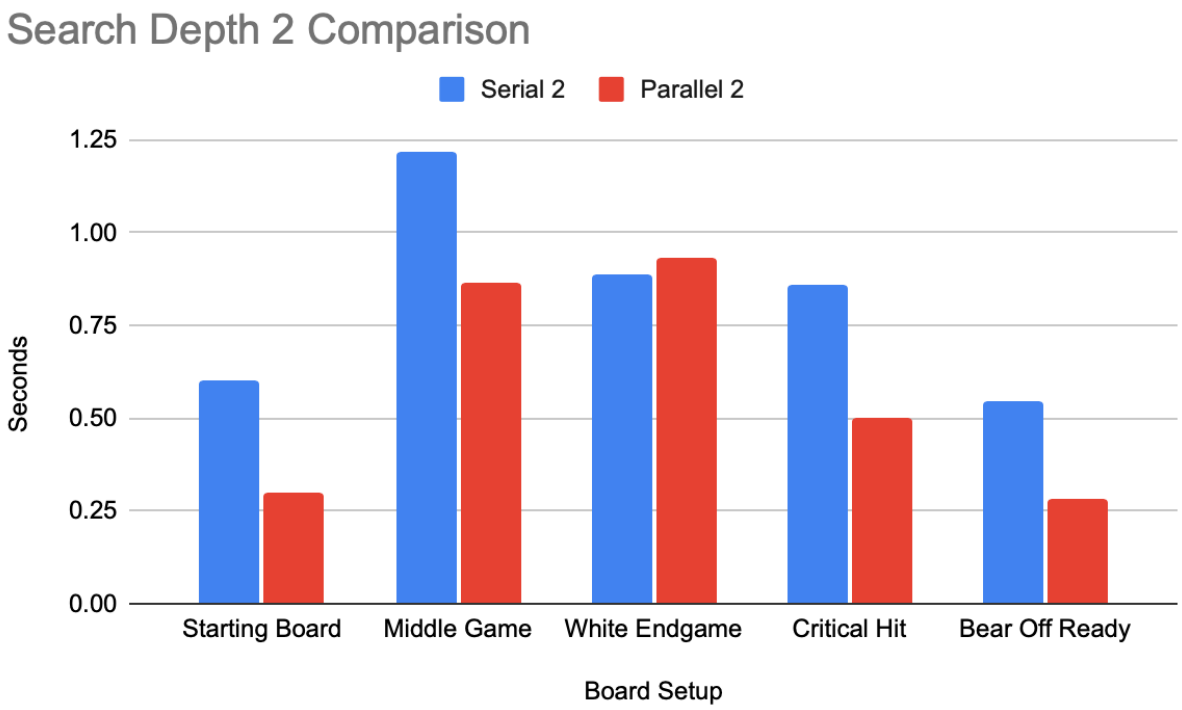
Bear Off Scenario (for both players):

```
bearOffReadyBoard :: Board
bearOffReadyBoard = Board
  { points = Map.fromList
      [ (P 1,  (White, 3))
      , (P 2,  (White, 1))
      , (P 4,  (White, 4))
      , (P 6,  (White, 3))
      , (P 19, (Black, 3))
      , (P 20, (Black, 3))
      , (P 24, (Black, 2))
      ]
  , bar = Map.fromList [(White,0), (Black,0)]
  , bearOff = Map.fromList [(White,4), (Black,7)]
  }
```

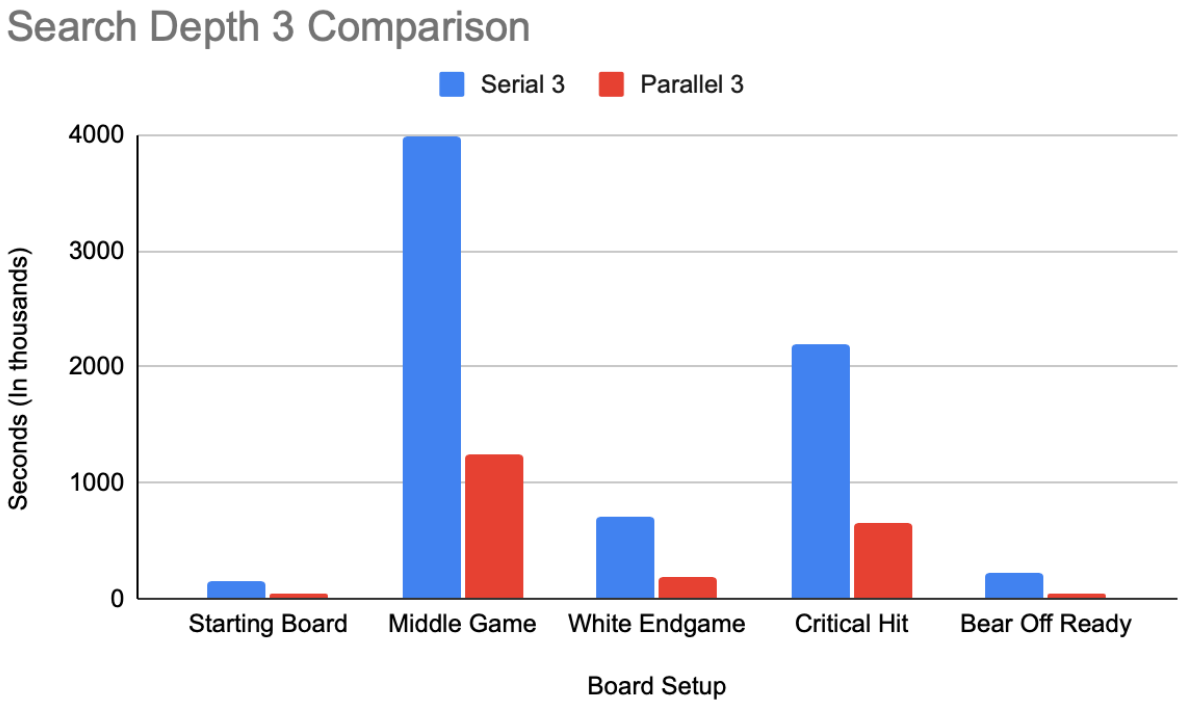
5.1. Search Depth Speed Comparison

When comparing the serial vs parallel approach, we are analyzing the parallel approach with all 8 cores running. The Blue is the Serial implementation and the Red is the Parallel. (Comparisons on following page)

For a search depth of 2:

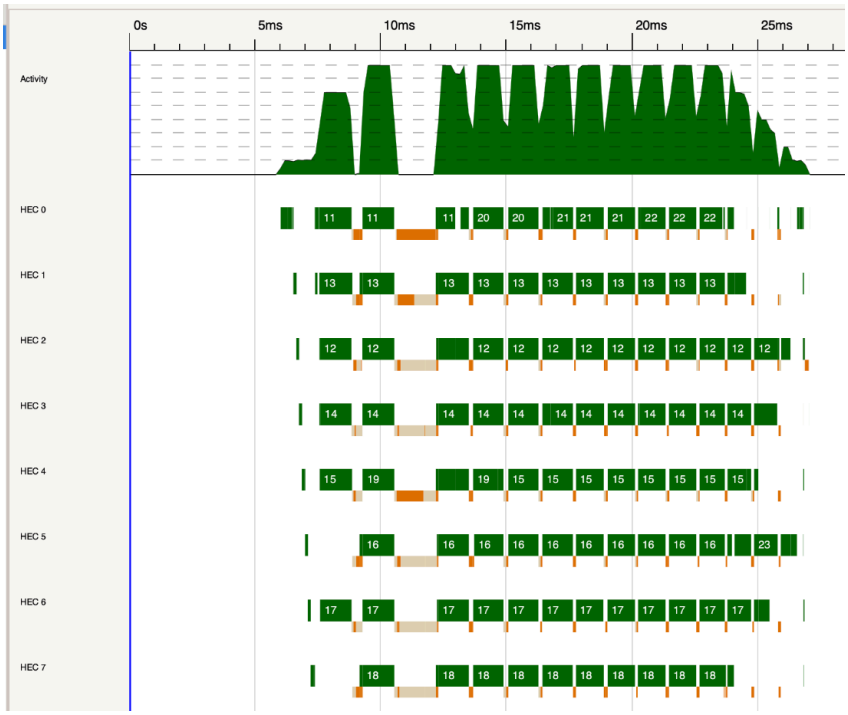


For a search depth of 3:

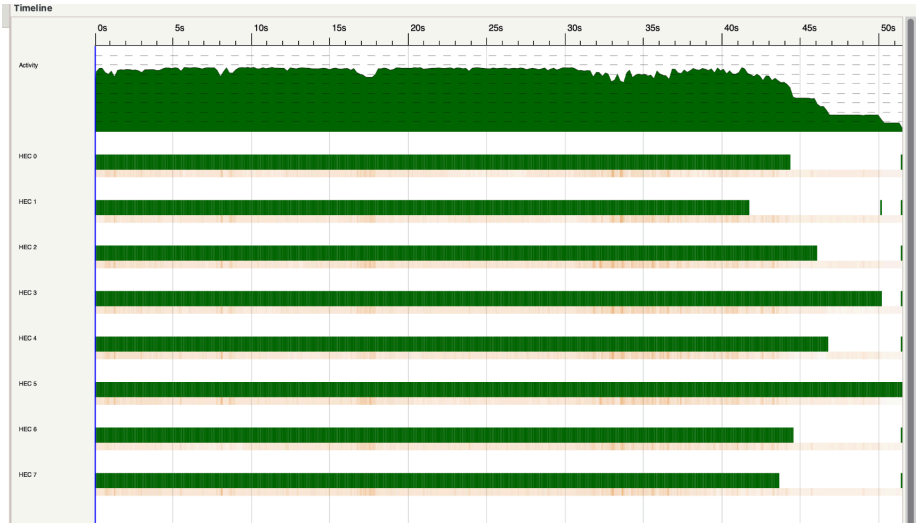


5.2. Threadscope/Spark Creation analysis

The threadscope analysis for each parallel test at depth 2 looked like this:



While the threadscope analysis for each parallel test at depth 3 looked like this:



Spark Table:

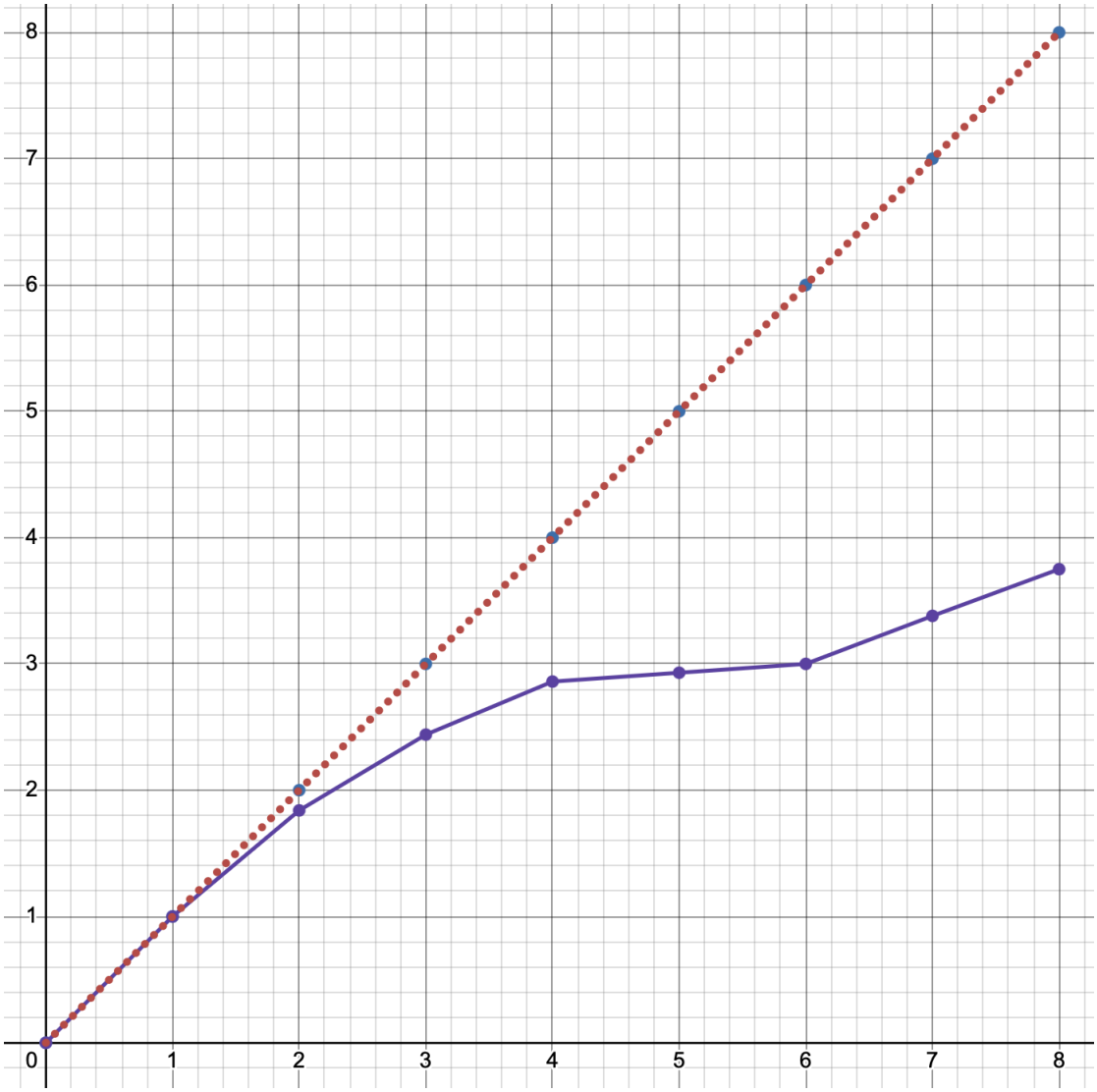
	Search Depth 2	Search Depth 3
Starting Board	21 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled	21 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled
Middle Game	56 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled	56 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled
Critical Hit	54 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled	54 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled
White Endgame	23 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled	23 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled
Bear Off Ready	36 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled	36 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled

This makes sense in terms of how we parallelized the algorithm, as each board would only produce a spark for each legal move at the root. Because of how expectiminimax worked with alpha beta pruning and the parallelization, it checks out that boards like Middle Game and Critical Hit would take the longest as they had the most legal moves, leading to the largest trees.

The only outlier is how the Bear Off Ready board took less time than the White Endgame board even though it created more sparks. The reasoning for this is that the Bear Off Ready board would have less nodes to cover as it went deeper as players may have won and there would be far less legal moves in subsequent turns following the original parallelization.

5.3. Speedup Analysis

For the purposes of analyzing a speedup analysis, I decided to analyze the White Endgame board as analyzing the Critical Hit of Middle Game boards would have taken too long, even though they may give more accurate speedup results. That chart ends up looking like this: (Pictured next page)



Y-Axis - How much the program sped up
X-Axis - Number of Cores used
Red Dotted Line - Ideal Speedup
Purple line - True Speedup

As we can see, the speedup was much less than the 1-to-1 speedup that was hoped for. One reason for why certain numbers of cores worked better than others is because of the number of moves available and the mathematical split of the parallelization. For this board setup, there were 23 possible moves. 23/4 is equal to 5 with 3 left over. 23/5 is equal to 4 with 3 left over, 23/6 = 3 with 5 left over. Each one of these times when a core was added yet the speedup was not great could be assigned to the fact that each additional core only required one more division of the cores' work, making the benefits of the core very minimal

6. Conclusion

All in all, this AI to predict the best move in backgammon is exceedingly slow, but parallelization helps immensely. Even though the speedup is far less than ideal, it is still immensely helpful, especially in tight endgames where each move could determine a win or a loss. This system could be worked to have better heuristics to find a more accurate better move, but the current heuristics suffice for this backgammon solver which is more focused on the benefits of parallelism.