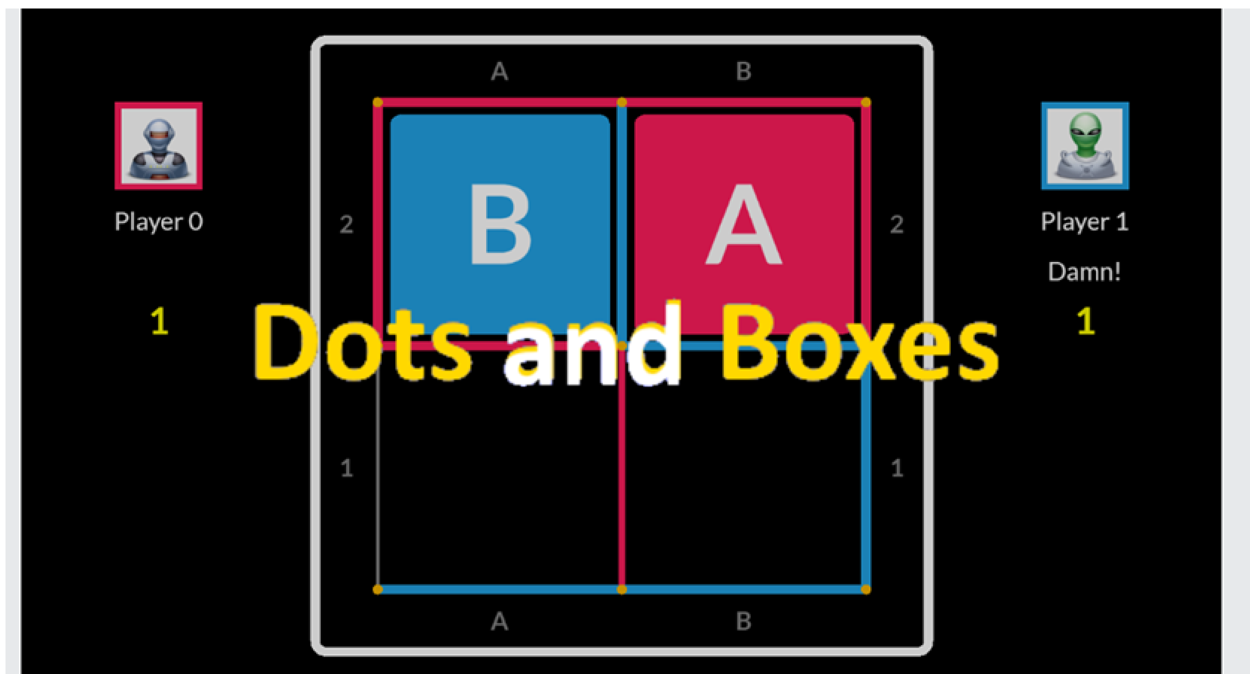
COMS 4995:
Parallel Functional Programming
2021 Fall

Dots and Boxes Parallelized Minimax

Sitong Feng (sf3049)
Yuyan Ke (yk2822)

Introduction

Dots and Boxes is a classic two-player adversarial game on a 2 x 2 square grid with empty boxes. (Note the size of the grid could change to increase difficulty.) Each player takes turns in drawing one side of a free box. The player that draws the last free side of the box owns that box, and if the last side is shared between two boxes, then that player takes both boxes, and his/her score will be incremented by 2. At the end of the game, the player with the most boxes/scores wins.



Problem Formulation

Our project aims to solve the game using the minimax algorithm, modeled as a tree that includes every possible state of the board. The game starts with AI making the move. For all possible edges available for AI to choose, it checks all possible moves that the human player can subsequently choose, and assume the human will choose one that minimizes the AI's chance's winning. Then minimax chooses AI's move based on the highest possible heuristic score.

Since the tree will be fully traversed to evaluate all possible moves, in a depth-first approach, it will generate a substantial search space which is very time-consuming on a large game board. Thus, we implemented a depth-limited sequential minimax algorithm to run on our 2x2 board. However, for a more optimal solution to the game, we decide to apply a parallel version of minimax to speed up the game. For our experiment, we will vary the number of cores, and depth of the parallel levels of the game trees, to achieve the most efficient and fastest performance of the game.

Implementation

Data Type Definition

For the representation of the game board, we created two additional data types of Edge and Box.

All edges in a game board are represented by the Edge data type, each with a unique integer identifier, eid, and a bool flag indicating if the edge has been taken for a given box.

```
data Edge = Edge {eid :: Int, flag :: Bool}
```

Similarly, all boxes in a given game configuration are each represented by the Box data type. Each box contains an unsorted list of 4 edges and a value representing the score earned for closing the given box.

```
data Box = Box
  { edges :: [Edge],
    val :: Int
  }
  deriving (Eq)
```

Since the game logic relies on an up-to-date, non-duplicate representation of all available moves remaining at a given point in time, the program maintains and moves around a set of available edges and a list of current boxes. In order to avoid duplication and define the equivalence of two Edge types derived from different bool flags for the same edge identification, we implemented instances of Ord and Eq for the Edge type. Additionally, in order to better display and print the data types, an instance of Show is written for Edge and Box data types.

```
instance Ord Edge where
  (Edge v1 _) `compare` (Edge v2 _) = v1 `compare` v2

instance Show Edge where
  show (Edge x f)="(Edge " ++ (show x) ++ " " ++ (show f) ++
")"

instance Eq Edge where
  (Edge v1 _) == (Edge v2 _) = v1 == v2

instance Show Box where
  show (Box l v) = "Box " ++ (printEdgeList l) ++ " " ++ (show
v)

printEdgeList :: [Edge] -> String
```

```
printEdgeList [] = []
printEdgeList (x : xs) = show x ++ " " ++ printEdgeList xs
```

Game Logic

Execution commands from the terminal:

1. `stack build`
2. `stack exec <executable-filename> – <configuration-file> <execution-mode>`

From the command line, two arguments are required for program execution – game configuration file and program execution mode. A set of 5 pre-configured game boards are provided in the game-config/ directory for execution; each row of the board configuration describes properties of a box with the first value representing the value, and the rest 4 integers identifying each edge of the box starting with the topmost edge counting clockwise. The 4 options of program modes are:

- 'algo-seq' for timing the sequential minimax algorithm on the entire game tree,
- 'algo-par' for timing the parallelized minimax algorithm for the entire game tree,
- 'game-seq' for game-playing with AI utilizing sequential minimax algorithm,
- 'game-par' for game-playing with AI utilizing parallel minimax algorithm.

The program loads the game board from the configuration file into a set of available edges and a list of boxes and passes this representation to the corresponding function for the given program mode.

For timing mode, the program will run the algorithm, output an optimal first move for the AI, and terminate. For both game-playing modes, the program will prompt the user to input: sequential tree depth - to limit the depth of minimax tree traversal and parallel tree depth to define the height of the top parallel tree.

Functions `gameStartSeq` and `gameStartPar` will call `gameLoopSeq` and `gameLoopPar` to start the game-playing interaction for each game-playing mode respectively. The two variations of the `gameLoop` are recursive calls to themselves with an alternation of a boolean flag to denote whether it's the AI (represented as False) or the human's (represented as True) turn. If it's the human's turn, the program will display the current board configuration by printing the states of all boxes and available edges to the console and prompt the user to select the next move based on edge identification. If it's AI's turn, the current edge set, box list, game score, and depth are passed into the corresponding minimax algorithm that would return the best move.

Once the next move is determined for either player, `nextGameState` will be called to update the game configuration and the score for the next iteration of the game loop.

The program follows the idea of zero-sum to track the scores and progress of both players with a single integer. The game starts with a neutral score of 0. Human scores are deducted from the score while the AI scores will be added to the score. The final game loop ends when there aren't any edges left and the result

of the game will be reported. Human wins if the score is negative, AI wins if positive, or a draw if the score is 0.

Sequential Solution

Minimax with depth limited approach

The base minimax starts with maximizing heuristics then alternates with minimizing the heuristics to traverse through the whole tree and return the best possible move for the computer-based on the opponent's actions.

The depth limited approach is added to improve the performance of minimax running on a large game board. Since most of the game states generated in the end are repetitive and take a huge amount of time, we have limited the levels (the default depth is 9 for optimal result with a speedy run) to return the optimal results evaluated so far for the tradeoff between time and optimal move.

rootnode -> on each first possible move, calls minimax to generate opponent's subsequent moves
terminal or depth = 0 -> end the minimax when there is no game action or depth reaches 0
not player -> computer's turn, which calls minimax on human's actions
player -> human's turn , which calls minimax on computer's actions

```
minimaxDepLim :: (Eq t, Num t) => Bool -> t -> (Set.Set Edge, [Box], Int) -> Edge -> (Int,
Edge)
minimaxDepLim player depth (edgeset, boxlist, aiScore) edge
 | rootnode      = bestMove [minimaxDepLim True newDepth x e | (x, e) <- initExpandedStates]
 | terminal || depth == 0 = (aiScore, edge)
 | not player    = bestMove [minimaxDepLim True newDepth x e | (x, e) <- subExpandedStates]
 | player        = worstMove [minimaxDepLim False newDepth x e | (x, e) <- subExpandedStates]
 | otherwise     = error "invalid game state"
 where
   newDepth = depth - 1
   initExpandedStates = [(getNextGameState e, e) | e <- edgelist]
   subExpandedStates = [(getNextGameState e, edge) | e <- edgelist]
   getNextGameState someEdge = nextGameState someEdge (edgeset, boxlist) aiScore player
   edgelist = Set.toList edgeset
   terminal = Set.null edgeset
   rootnode = edge == Edge 0 False
```

Parallel Solution

For the parallel optimization of minimax function, we analyzed the structure of the minimax function, which as mentioned, finds best action from all available move by calling itself recursively on each action to the near bottom of the search tree. This fits the pattern of the parallel strategy – map <function> <input-list>`using` parList rseq, with minimax being the function, taking in the input-list, which is the combination of the current game state with every remaining move (aka. edge) in the game. For this reason, we parallelize the execution of the minimax function for each possible input combination, and the collective results (aka. parResult...) get passed into bestMove/worstMove as the parameter. Additionally, rseq is chosen over rpar because the results from each minimax call need to be evaluated for the bestMove function so a best possible move could be determined by comparing scores that maximizes the game outcome.

The parallel solution also has a limited depth that dermines the height of the parallel tree. The top levels of the tree will run in parallel because more works are required of each nodes on the top levels near the root, and the work per sub-branch decreases as the tree grows closer to the leaf nodes. When reaching depth = 0, the minimax will evaluate rest of the tree with sequential approach to avoid overhead of creating too many sparks that do not accomplish much work, in which case it might not worth the overhead of generating the sparks.

```
minimaxParDep :: (Eq a, Eq t, Num t, Num a) => (Bool, a, t, (Set.Set Edge, [Box],
Int), Edge) -> (Int, Edge)
minimaxParDep (player, parDepth, seqDep, (edgeset, boxlist, aiScore), edge)
 | parDepth == 0     = minimaxDepLim player seqDep (edgeset, boxlist, aiScore) edge
 | rootnode          = bestMove parResultInitMax
 | terminal          = (aiScore, edge)
 | not player        = bestMove parResultSubMax
 | player            = worstMove parResultSubMin
 | otherwise         = error "invalid game state"
 where
   parResultInitMax = map minimaxParDep paramListInitMax `using` parList rseq
   parResultSubMax = map minimaxParDep paramListSubMax `using` parList rseq
   parResultSubMin = map minimaxParDep paramListSubMax `using` parList rseq

   newParDepth = parDepth - 1
   newSeqDepth = seqDep - 1
   paramListInitMax = [(True, newParDepth, newSeqDepth, x, e) | (x, e) <-
initExpandedStates]
   paramListSubMax = [(True, newParDepth, newSeqDepth, x, e) | (x, e) <-
subExpandedStates]
   paramListSubMin = [(False, newParDepth, newSeqDepth, x, e) | (x, e) <-
subExpandedStates]
```

```
   initExpandedStates = [(getNextGameState e, e) | e <- edgelist]
   subExpandedStates = [(getNextGameState e, edge) | e <- edgelist]
   getNextGameState someEdge = nextGameState someEdge (edgeset, boxlist) aiScore
player
   edgelist = Set.toList edgeset
   terminal = Set.null edgeset
   rootnode = edge == Edge 0 False
```

Best Move and Worst Move

The `bestMove` fuction takes in a list of resulting tuples `(score, edge)` from the minimax and returns the tuple with the best score, while `worstMove` returns the tuple with the worst score.

```
bestMove :: [(Int, Edge)] -> (Int, Edge)
bestMove [(score, edge)] = (score, edge)
bestMove ((score, edge) : (score', edge') : xs) = bestMove (if score >= score' then
(score, edge) : xs else (score', edge') : xs)
bestMove _ = error "BestMove: not a valid move"

worstMove :: [(Int, Edge)] -> (Int, Edge)
worstMove [(score, edge)] = (score, edge)
worstMove ((score, edge) : (score', edge') : xs) = worstMove (if score <= score' then
(score, edge) : xs else (score', edge') : xs)
worstMove _ = error "WorstMove: not a valid move"
```

Results / Evaluation

Evaluation Strategy

To derive a holistic analysis for the parallel and sequential version of minimax algorithm, we experimented with sequential version with depth limited approach, and parallel version with paralle depth limited approach.

Some parameters we varied include:

1. the number of cores to run on parallel version
2. the depth of the parallel tree (from 1 to 5). With the whole tree height remains the same, the higher the parallel top tree part, the lower in height for the remaining sequential minimax run.
3. For every combination, we run 3 times and takes the appropriate result to minimize the variance.

Moreoever, in order to analyze the effects of the parallel program and minimizing the effects of system performances, we conducted our analysis using a 2*2 game board. This configuration of the board has a reasonable sequential run-time (approximately 30 sec) and with parallelism applied, we expected the effects to be substantial, such that we could confidently conclude that the speed up is indeed from parallelism.

Analysis

In order to ensure that the sequential algorithm is truly running on 1 core, we applied the +RTS -N1 flag to the sequential execution. For parallel analysis, we experiemented with parallel depths ranging from 1 to 5 and number of cores from 2 to 10 in most cases. The numerical results associated with each run are recorded in a table form below and the optimal execution time for a given parallel depth is bolded and displayed in red.

For sequential algorithm limited to 1 core, the execution time was 33.972 sec. With parallism, the average execution time is approximately 8 sec with the optimal number of cores, nearly 4.25x speed up.

Experimentally, the most optimal execution time with parallel depth of 4 running on 8 cores was 7.553 sec with 4.5x speed up, but the second best execution time was 7.579 sec from parallel depth of 2 running on 8 cores. The subtle differences could be due to system performance at a given moment in time, but together, both runs suggest a range for an optimal execution, which is about 8 cores, running between parallel depth of 2 to 4.

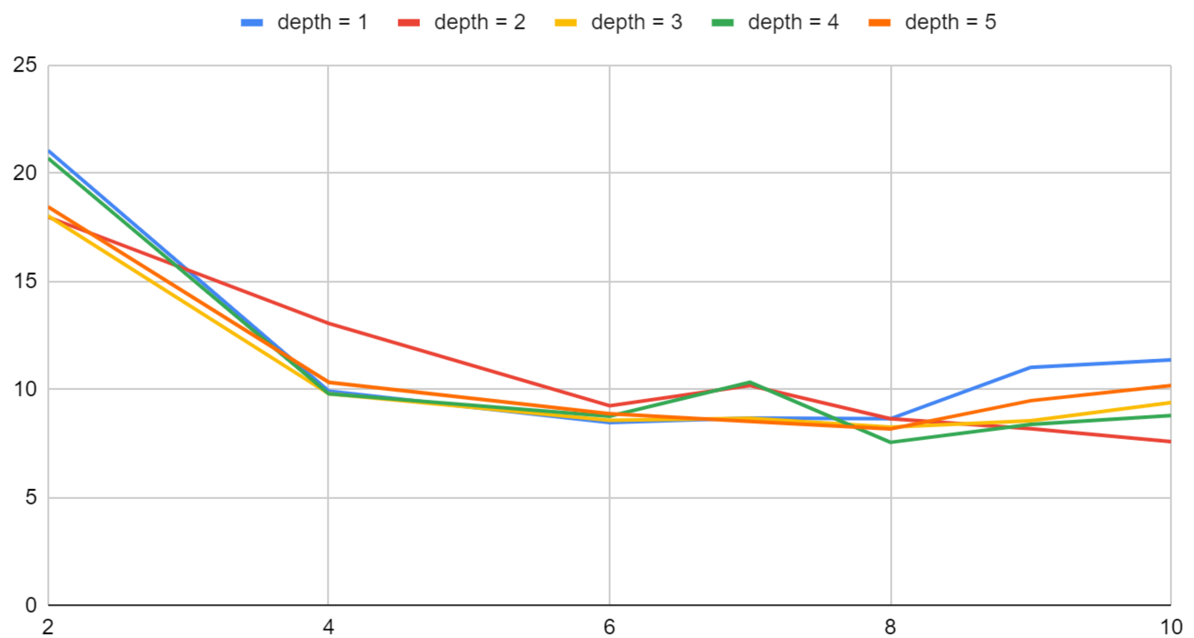
For the two optimal runs above, there is only 5% or less conversion rate of total sparks. This indicates that there is some optimization we can do in tuning parallel algorithm to bring down the overhead from spark generation, reducing the number of fizzled sparks to futher improve the run time.

Raw Results

			Sparks							
Parallel Depth	Cores	Time (s)	Conversion Rate	Total	Converted	Overflowed	Dud	GC'd	Fizzled	Memory (MiB)
Sequential		33.972	0	0	0	0	0	0	0	3
1	2	21.056	0.917	12	11	0	0	0	1	4
	4	9.914	0.917	12	11	0	0	0	1	6
	6	**8.465**	**0.917**	**12**	**11**	**0**	**0**	**0**	**1**	**8**
	7	8.669	0.917	12	11	0	0	0	1	9
	8	8.637	0.917	12	11	0	0	0	1	10
	9	11.017	0.917	12	11	0	0	0	1	11
	10	11.363	1	12	12	0	0	0	0	13
2	2	17.973	0.076	144	11	0	0	0	133	4
	3	13.046	0.347	144	50	0	0	0	94	5
	4	9.239	0.604	144	87	0	0	0	57	6
	5	10.172	0.201	144	29	0	0	0	115	7
	6	8.641	0.542	144	78	0	0	0	66	8
	7	8.18	0.306	144	44	0	0	0	100	9
	8	**7.579**	**0.438**	**144**	**63**	**0**	**0**	**0**	**81**	**10**
	9	8.443	0.257	144	37	0	0	0	107	11
	10	9.232	0.181	144	26	0	0	0	118	13
3	2	18.023	0.014	1464	20	0	0	0	1444	4
	4	9.792	0.139	1464	204	0	0	0	1260	6
	6	8.562	0.023	1464	33	0	0	0	1431	8
	7	8.658	0.15	1464	219	0	0	0	1245	9
	8	**8.256**	**0.408**	**1474**	**601**	**0**	**0**	**8**	**865**	**10**
	10	8.543	0.083	1464	122	0	0	0	1342	13
	12	9.385	0.223	1464	327	0	0	0	1137	15
4	2	20.693	0.002	13353	29	0	0	9	13315	4
	4	9.79	0.002	13344	31	0	0	0	13313	6
	6	8.754	0.035	13372	471	0	0	17	12884	8
	7	10.331	0.076	13363	1013	0	0	19	12331	9
	8	**7.553**	**0.033**	**13344**	**444**	**0**	**0**	**0**	**12900**	**10**
	9	8.38	0.01	13344	132	0	0	0	13212	11

	10	8.789	0.034	13363	451	0	0	19	12893	13
5	2	18.445	0	108384	28	0	0	36	108320	4
	4	10.323	0.004	108384	413	0	0	9	107962	6
	6	8.87	0.005	108384	505	0	0	12	107867	8
	7	8.5165	0.005	108384	505	0	0	12	107867	9
	8	**8.163**	**0.012**	**108441**	**1274**	**0**	**0**	**68**	**107099**	**10**
	9	9.477	0.016	108433	1784	0	0	58	106591	12
	10	10.179	0.005	108408	581	0	0	34	107793	13

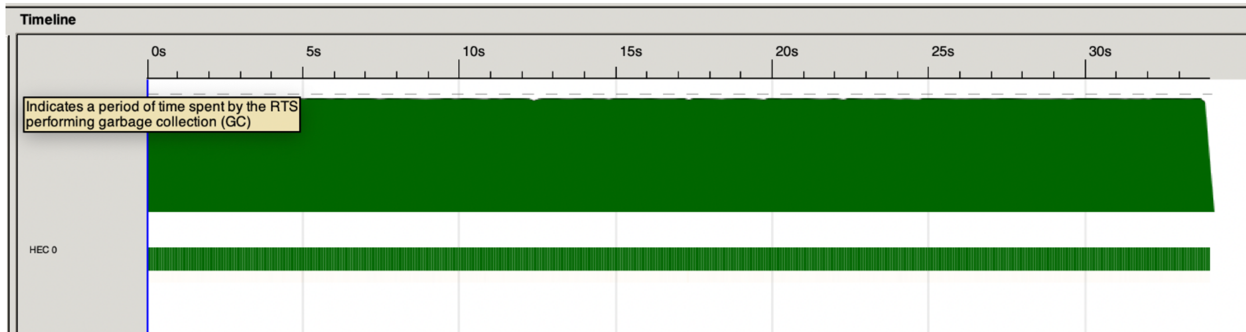
Result Graph



The graph shows that the most optimal results happened around 6 cores and 8 cores. This partly attributes the hardware limitations of our computer which has 8 cores. However, the least run time happened with parallel depth of 4 on 8 cores, which is close to the depth = 5 above. Beyond 8 cores, most graphs appear to rise in execution time again, suggesting that parallel overheads are starting to reduce the parallel performance. Furthermore, from the threadscope captures below, it’s appearant that for certain parallel depths, more cores cause load imbalance which explains the increase in runtime.

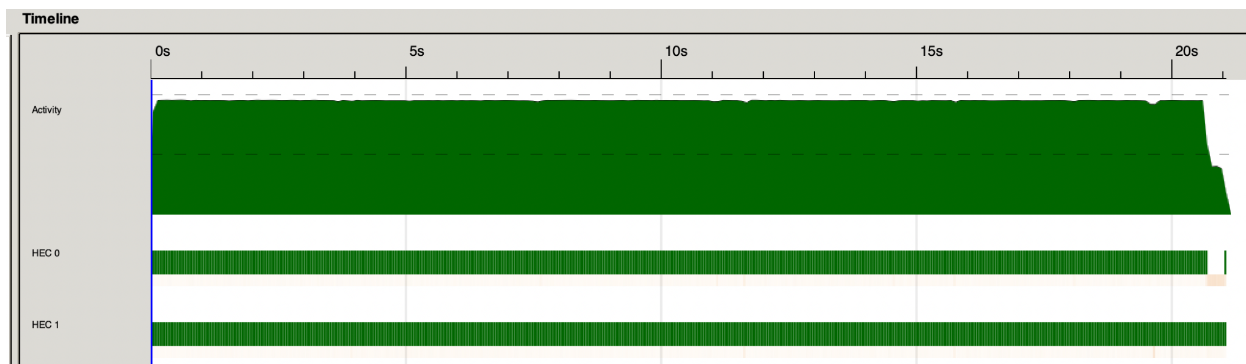
Threadscope

Sequential

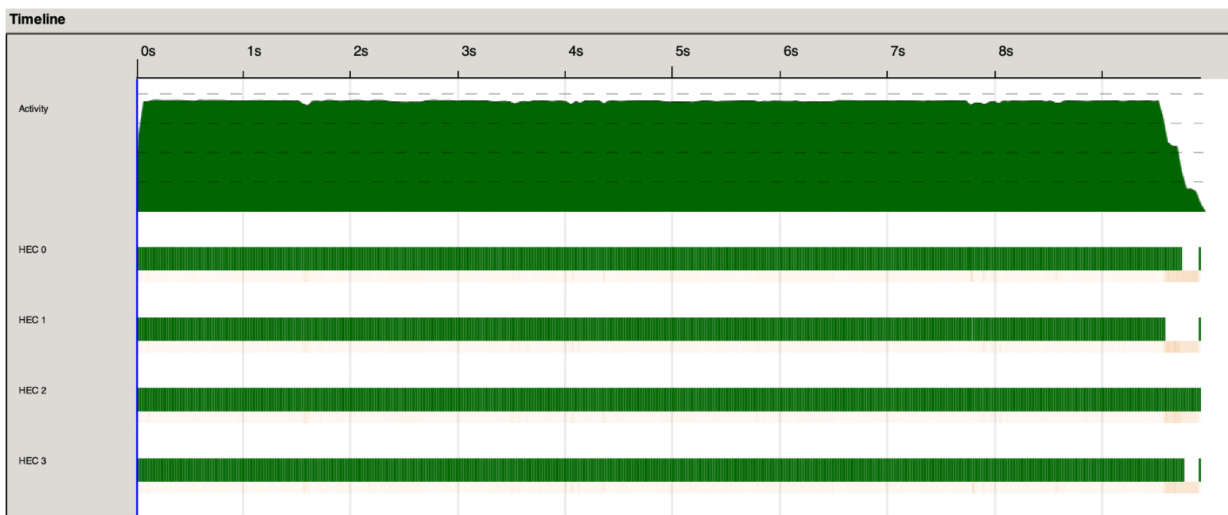


Parallel Depth = 1

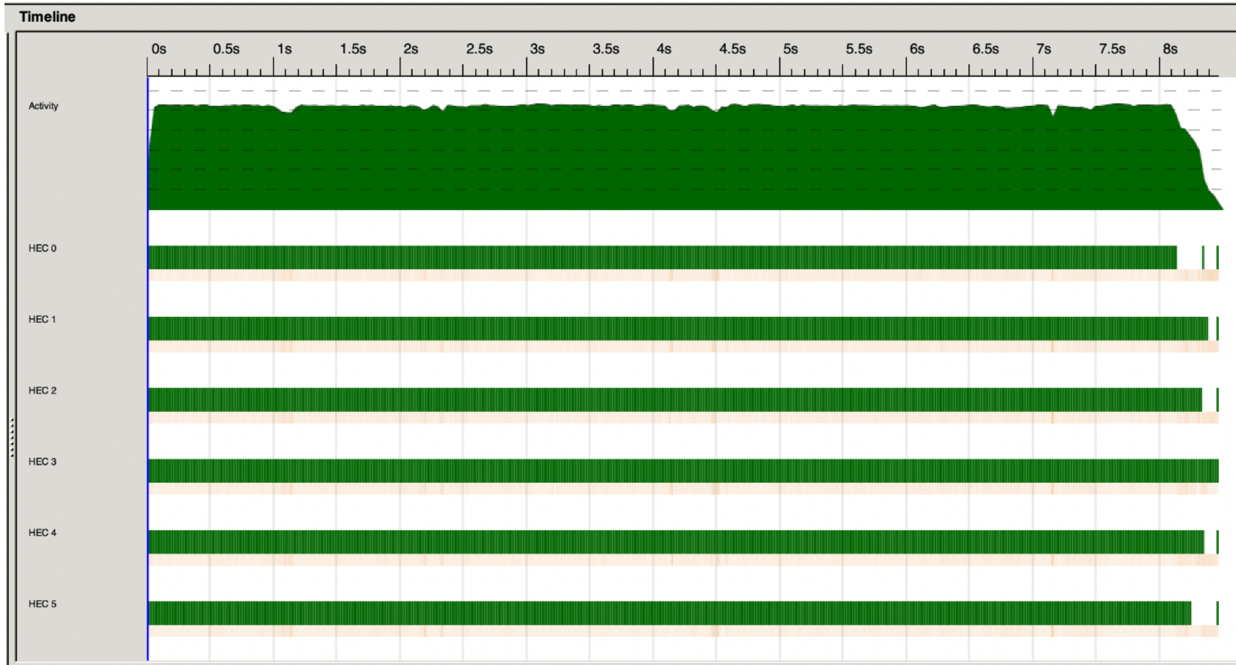
-N2



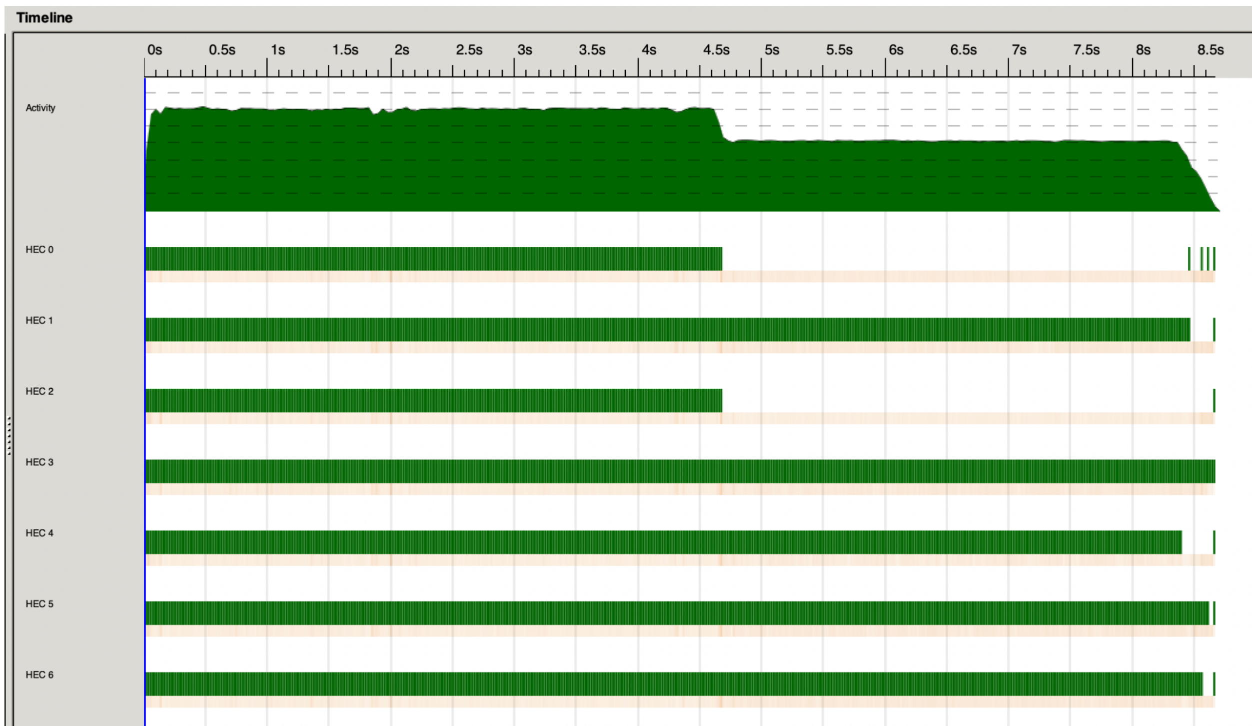
-N4



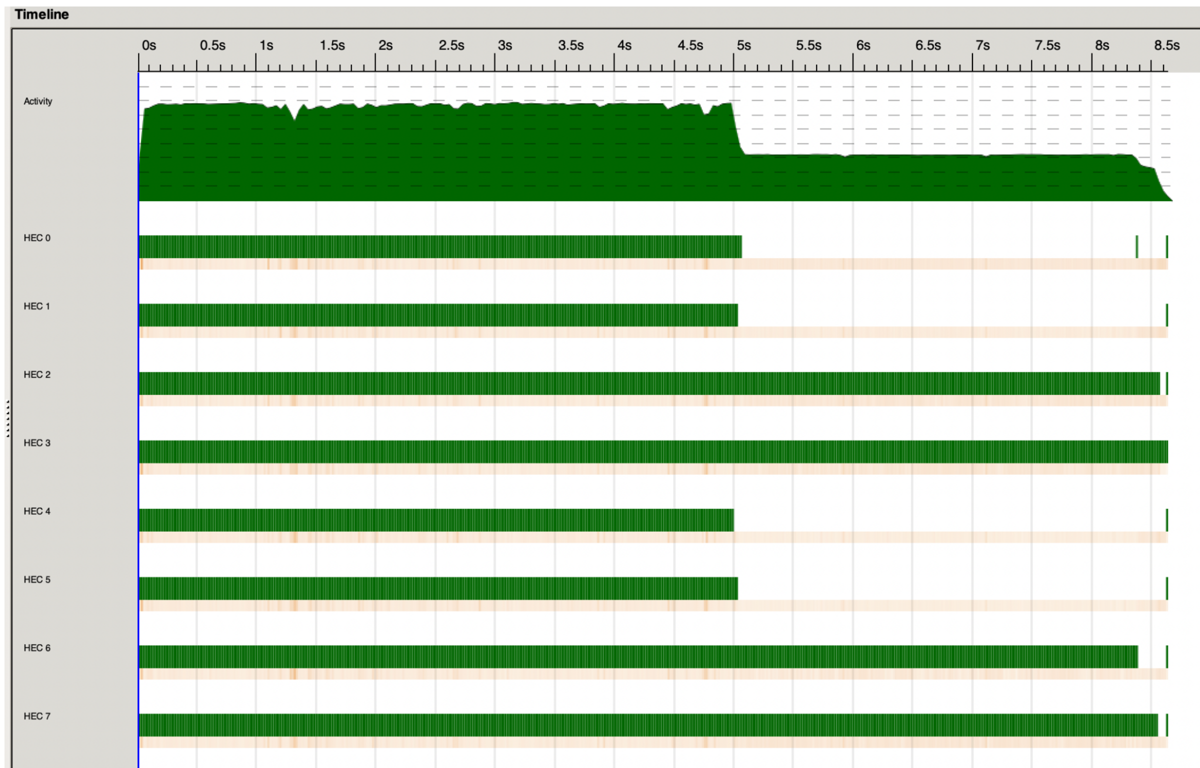
-N6



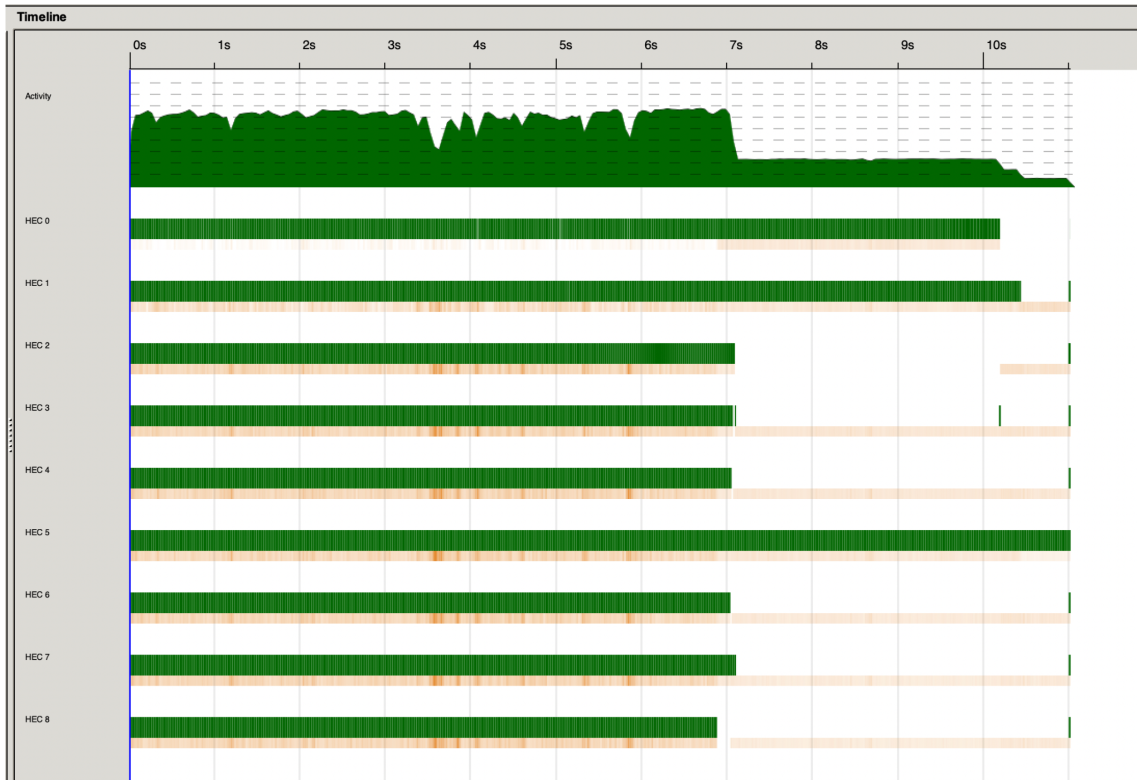
-N7



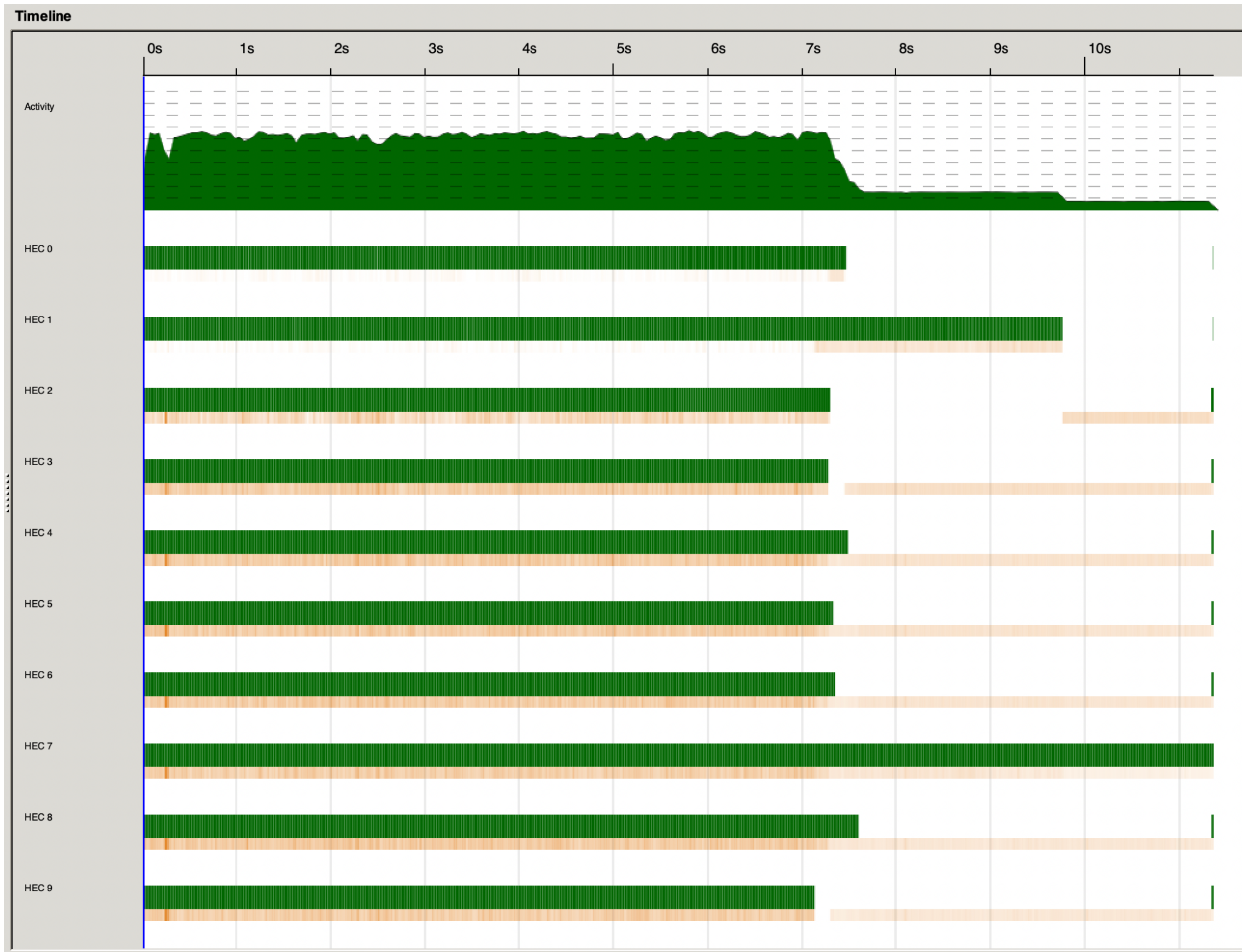
-N8



-N9

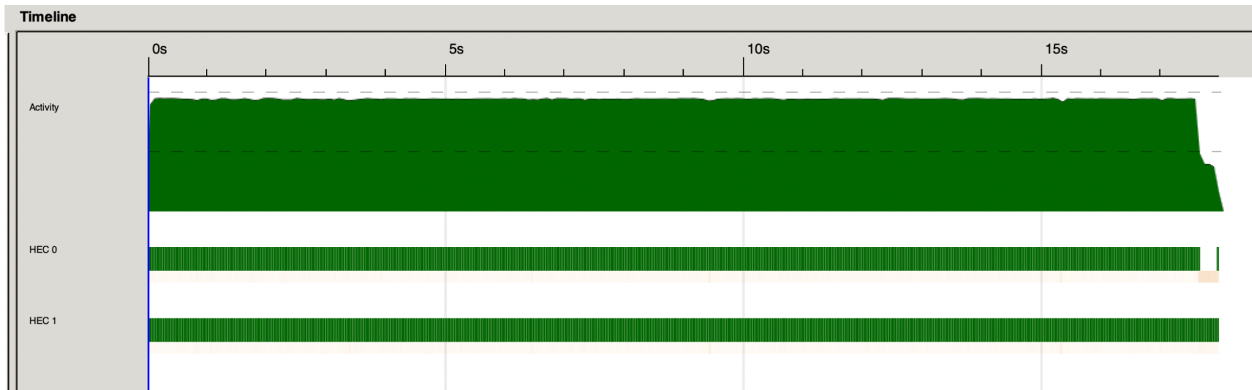


-N10

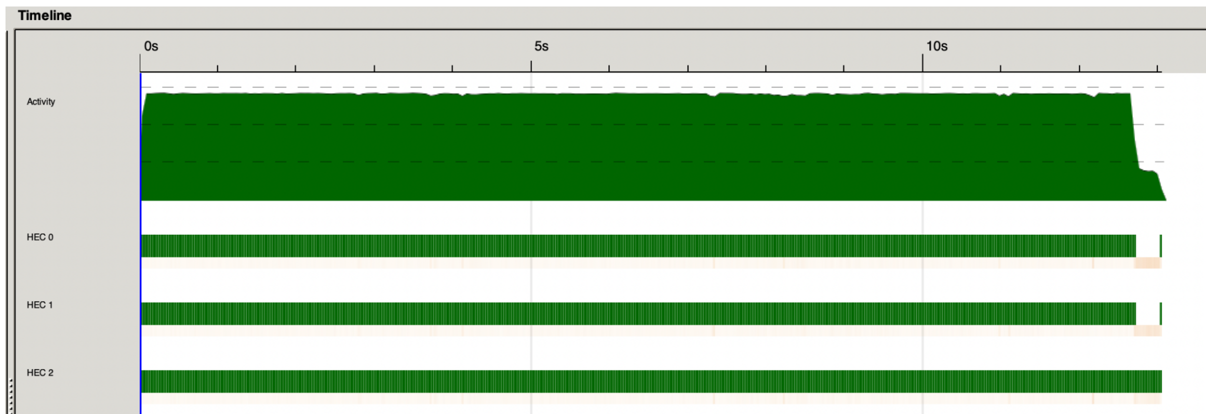


Parallel Depth = 2

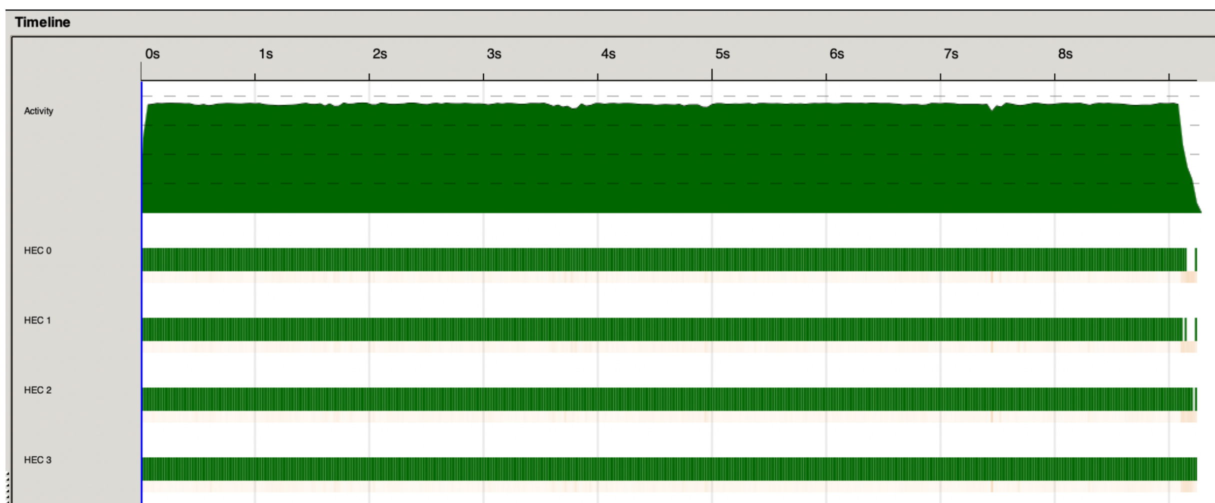
-N2



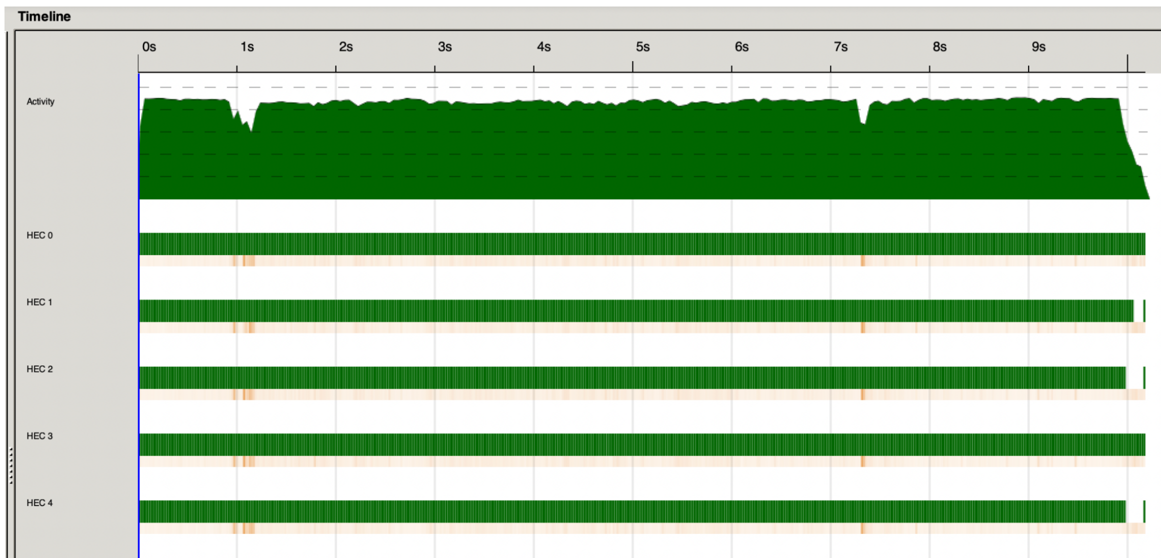
-N3



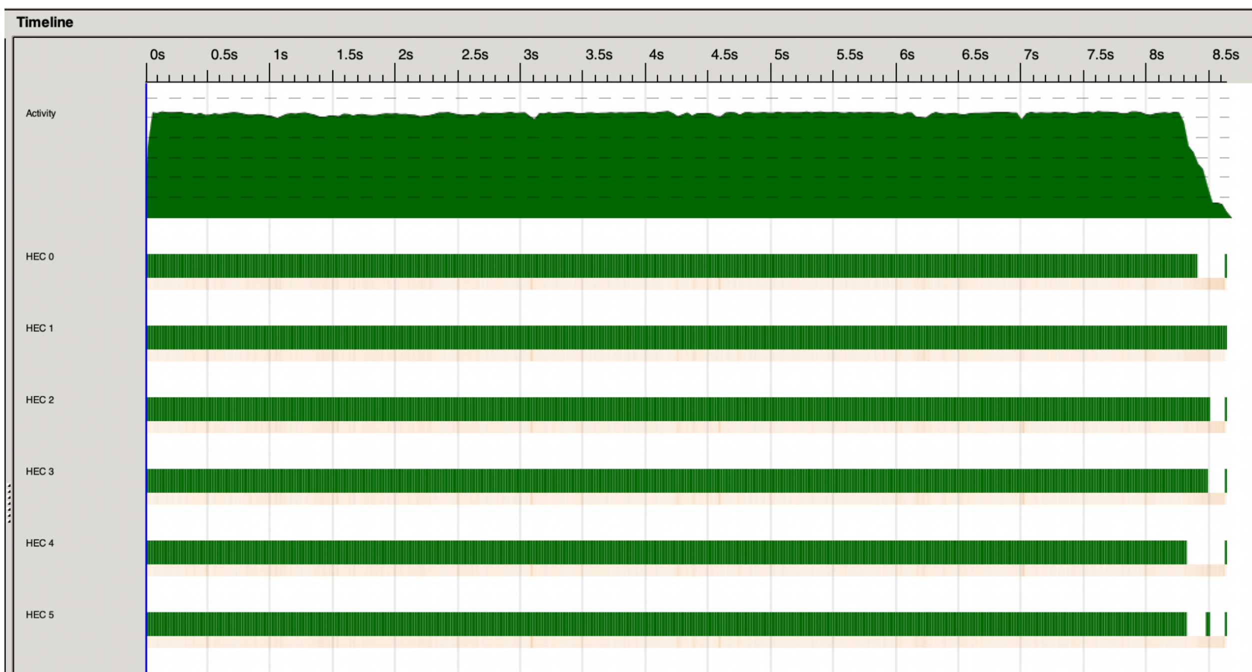
-N4



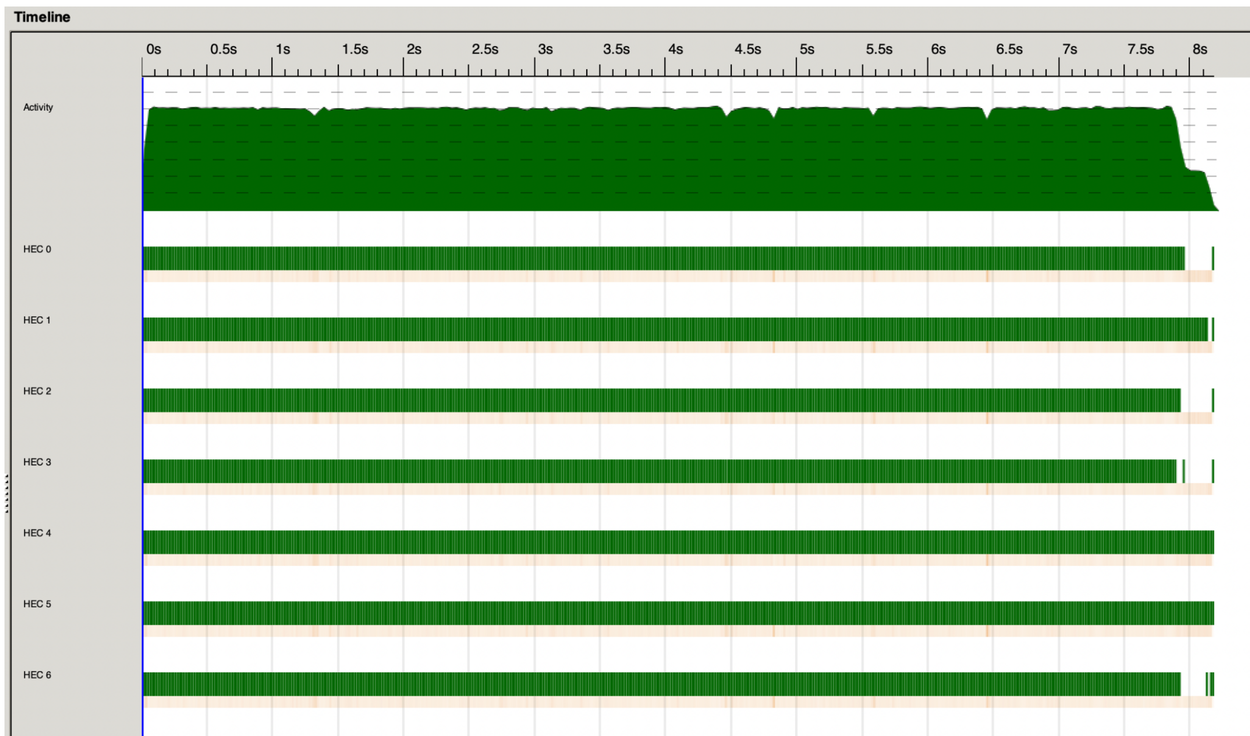
-N5



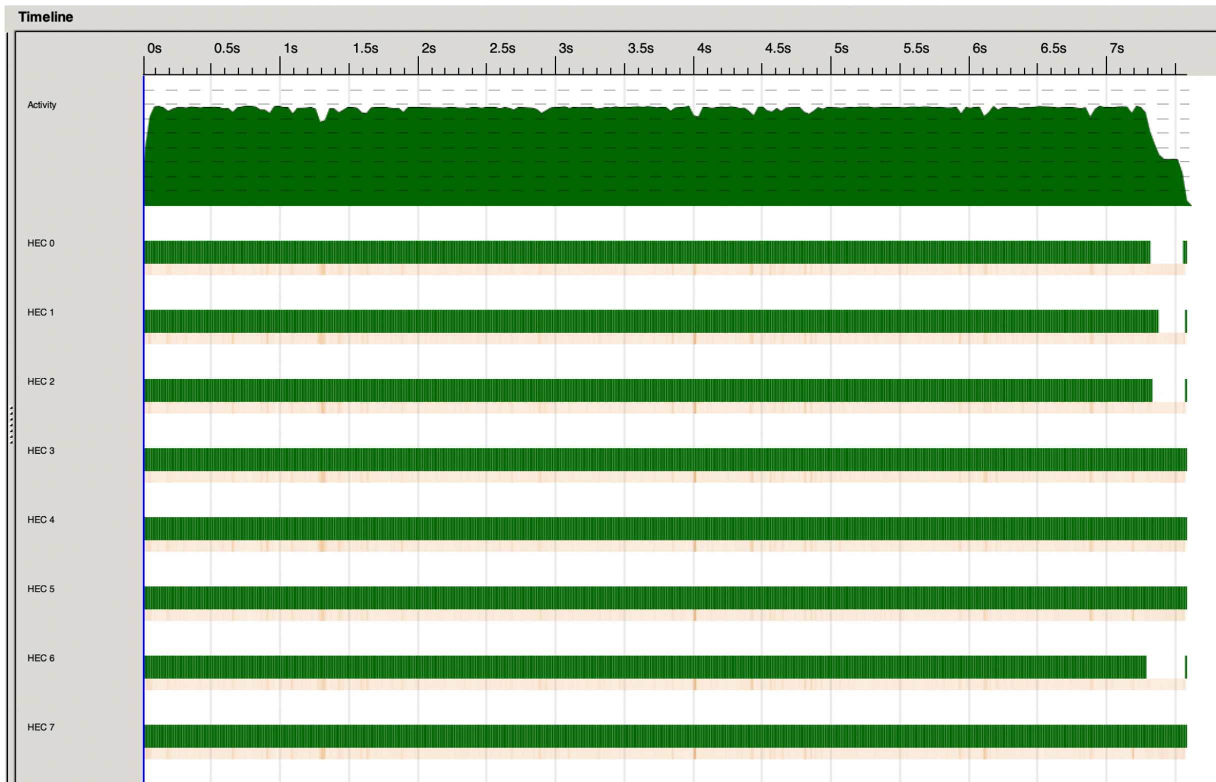
-N6



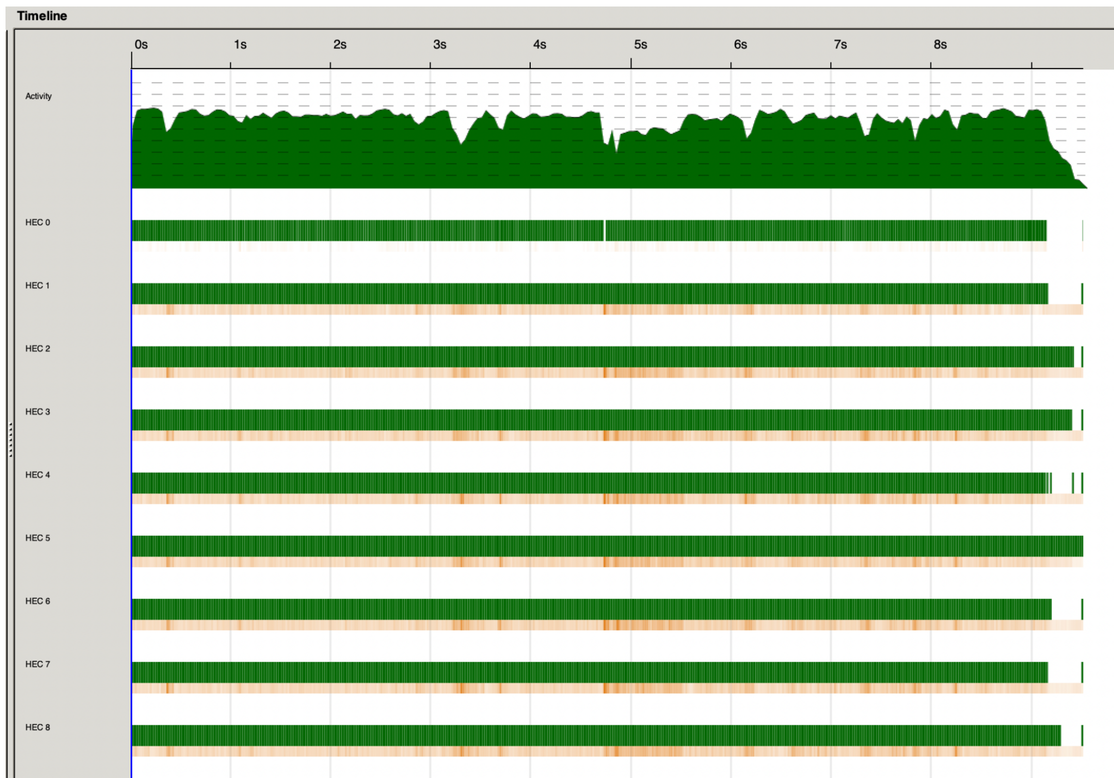
-N7



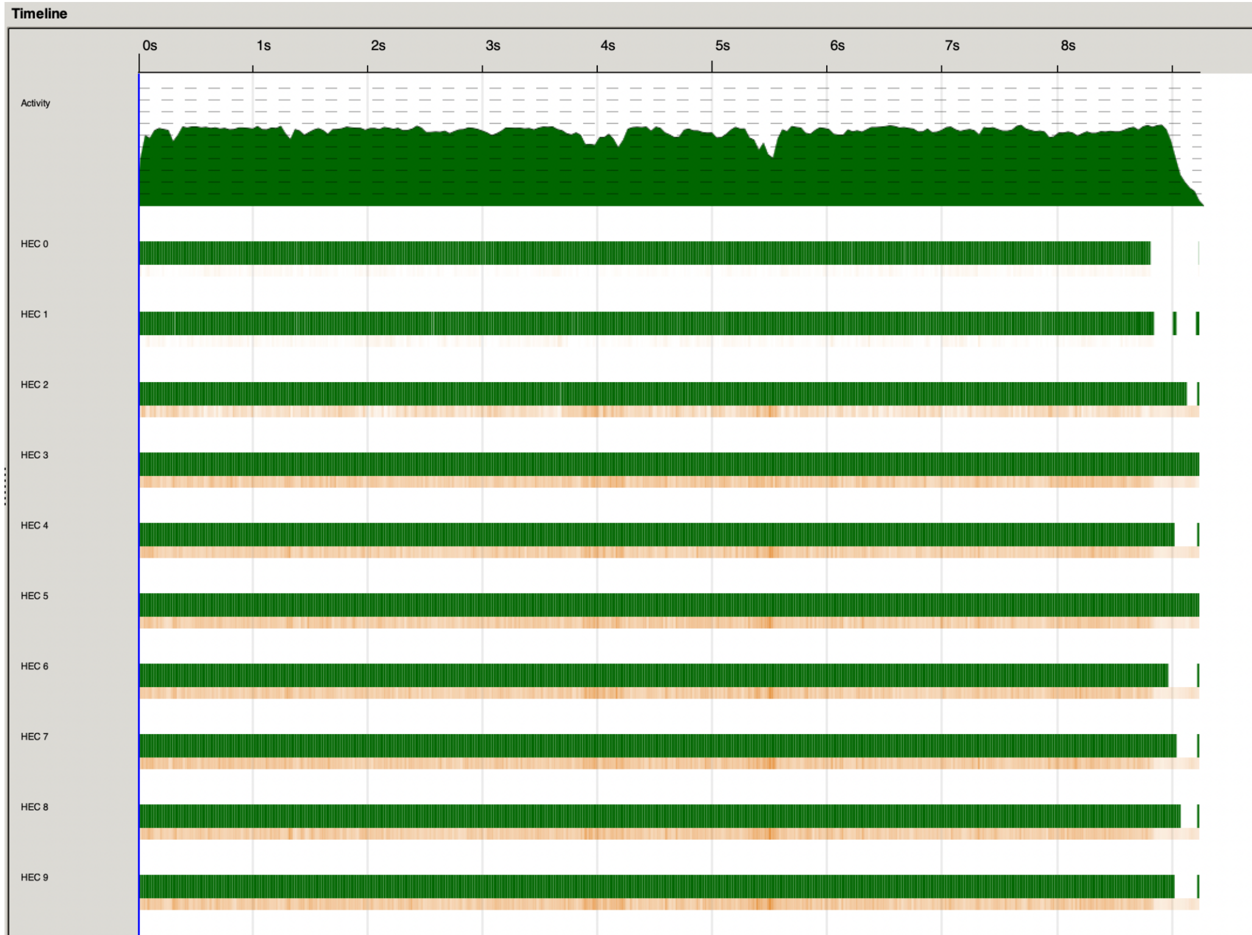
-N8



-N9

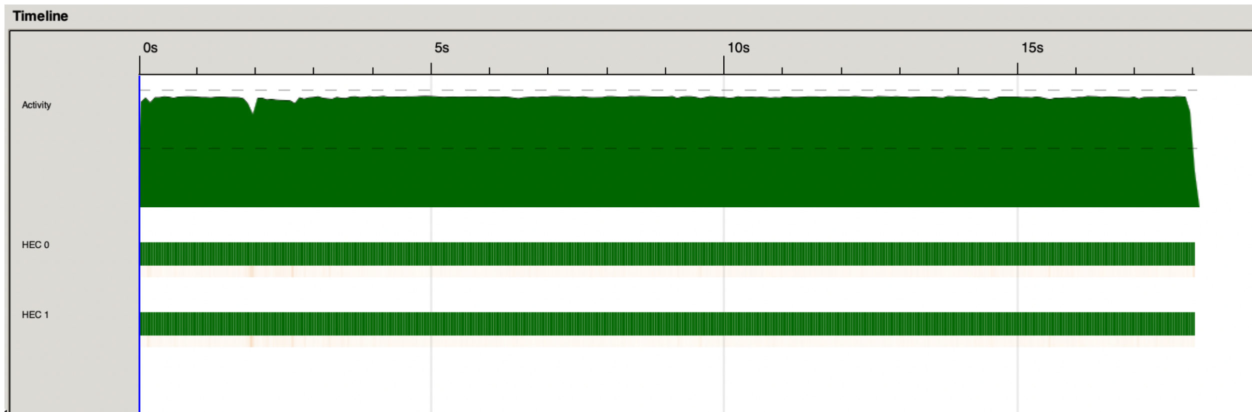


-N10

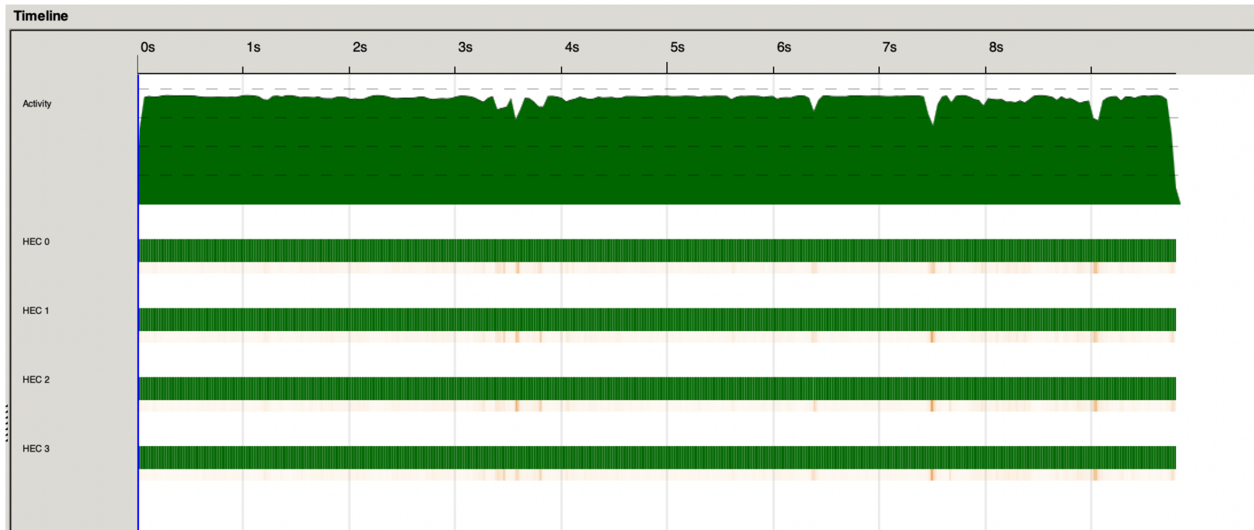


Parallel Depth = 3

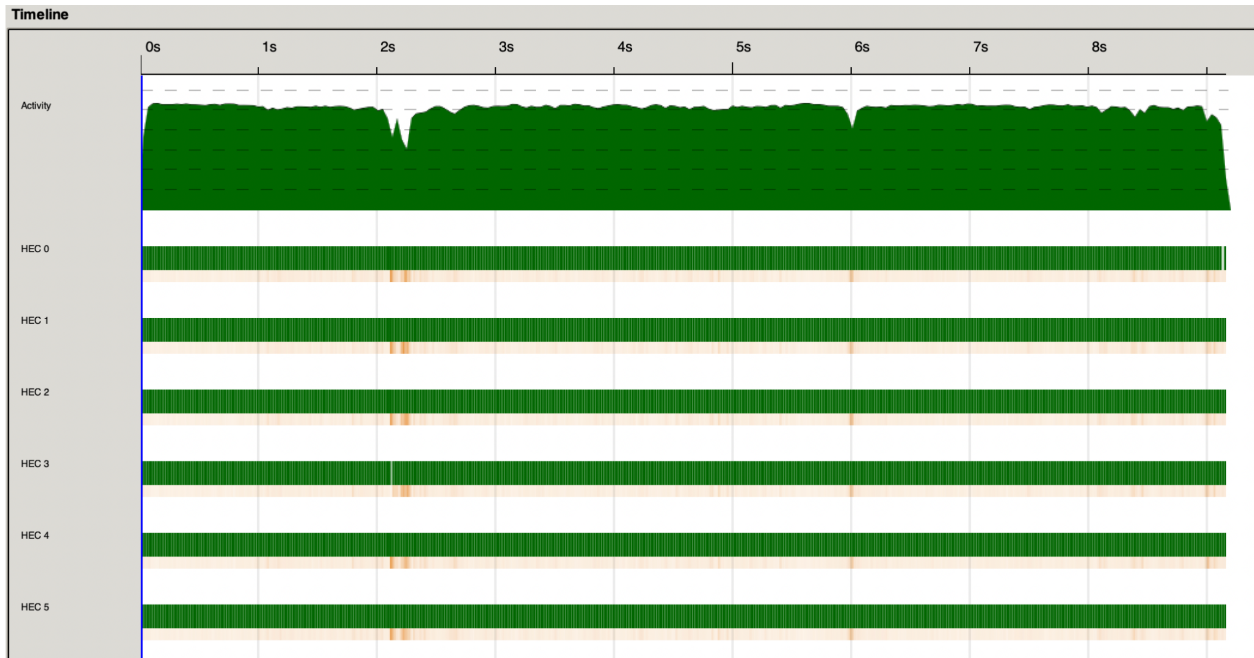
-N2



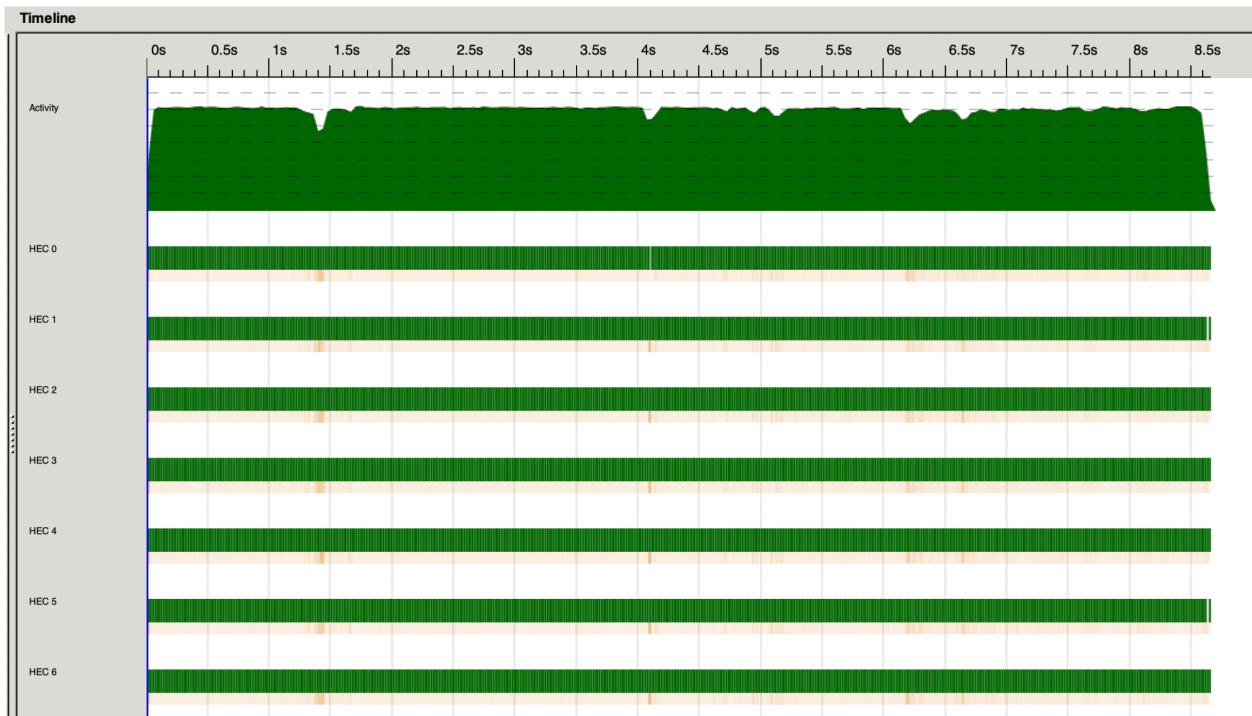
-N4



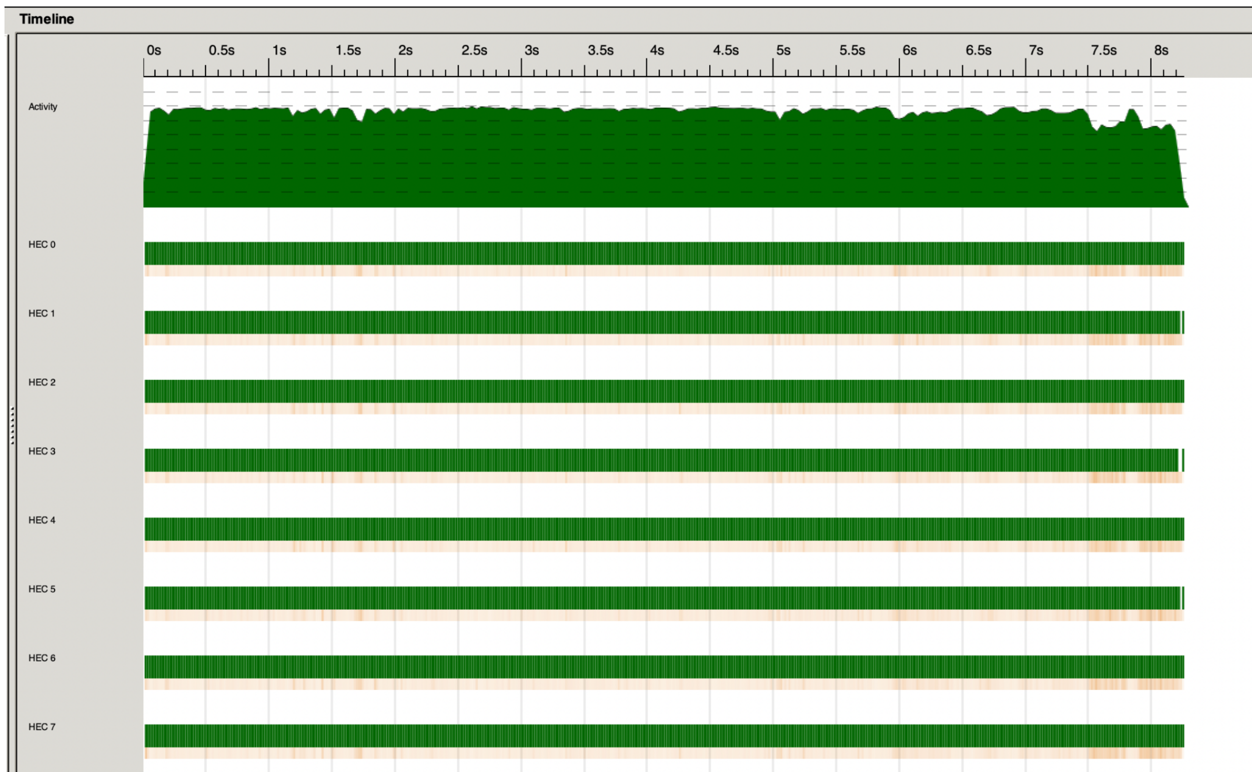
-N6



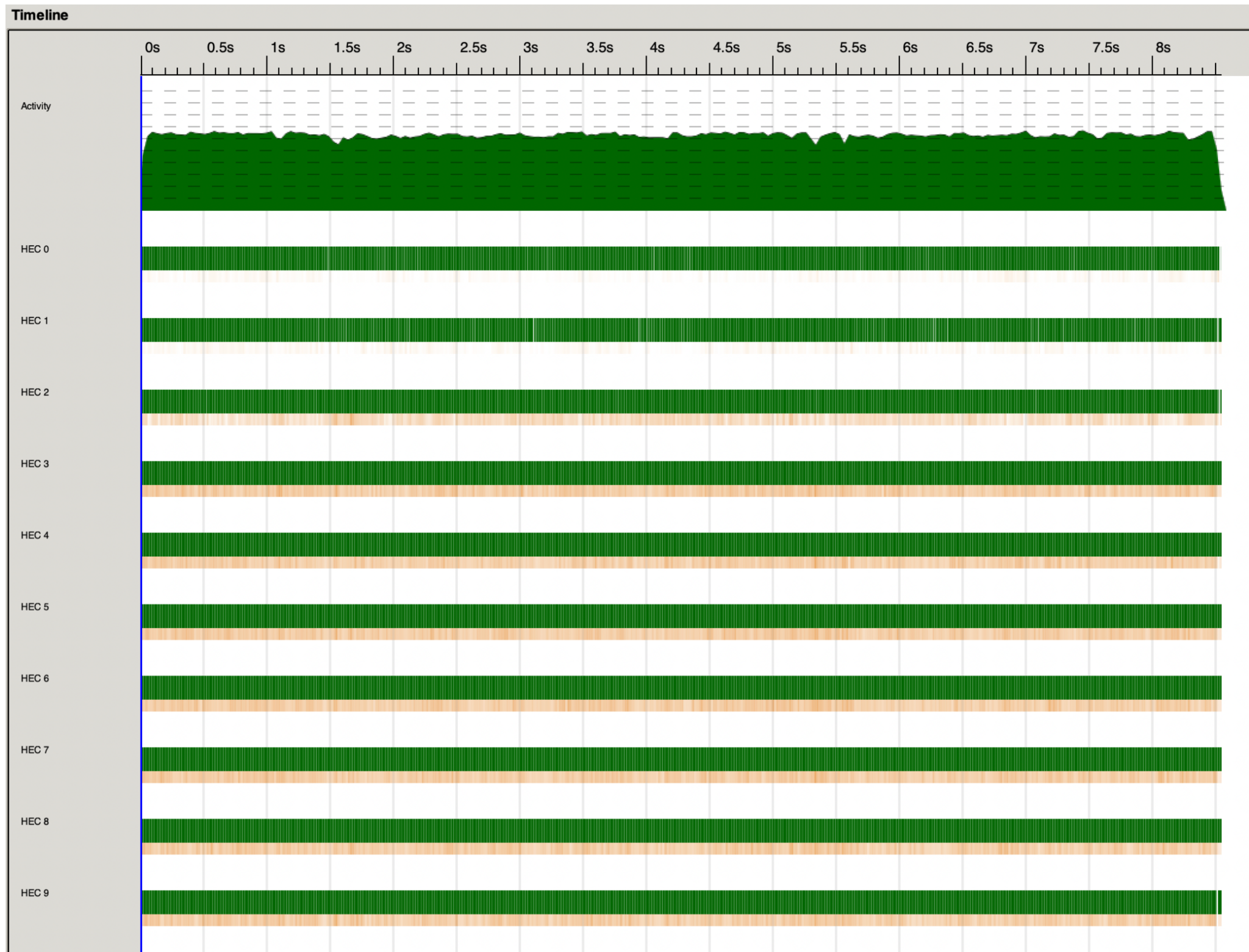
-N7



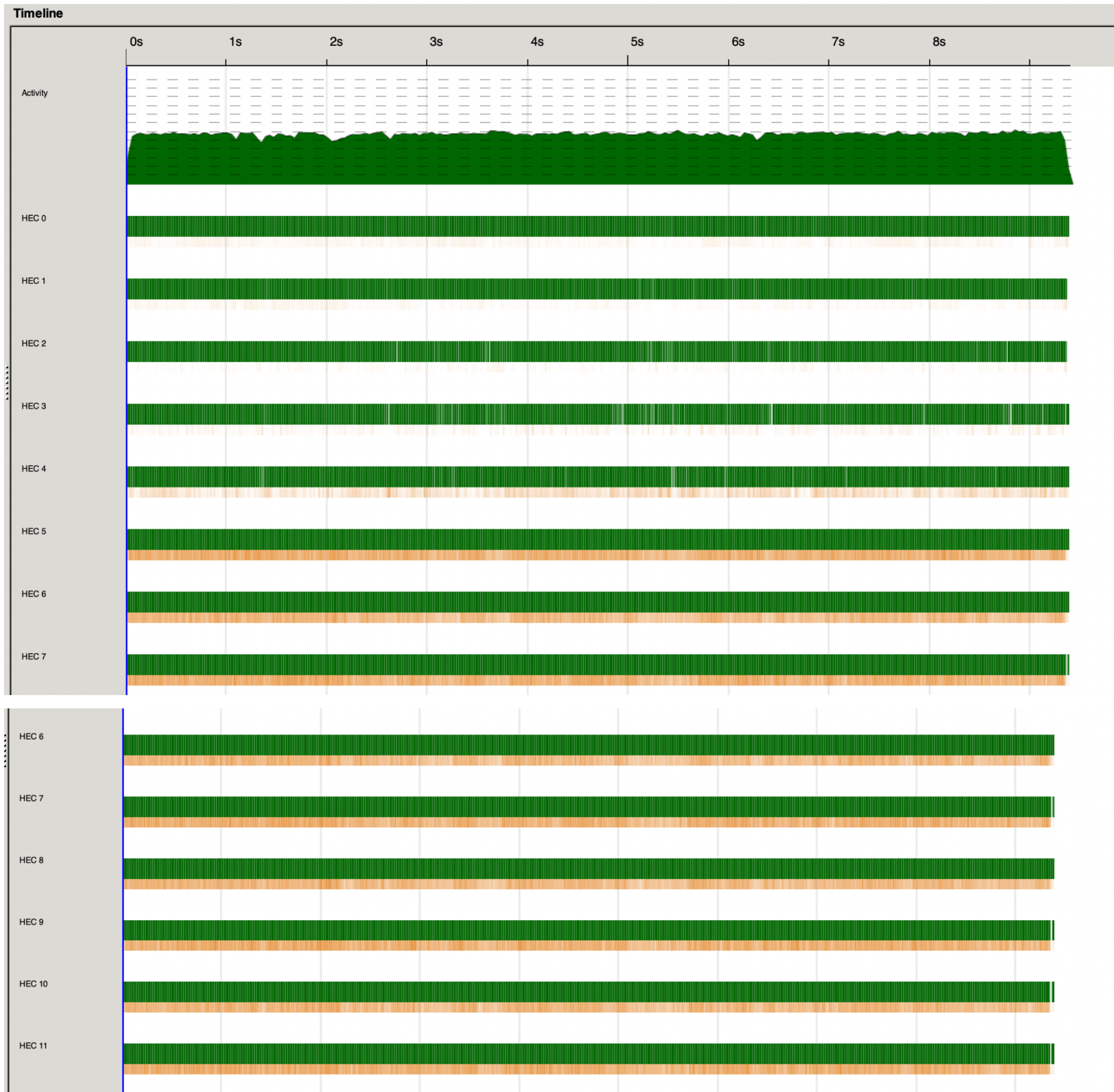
-N8



-N10

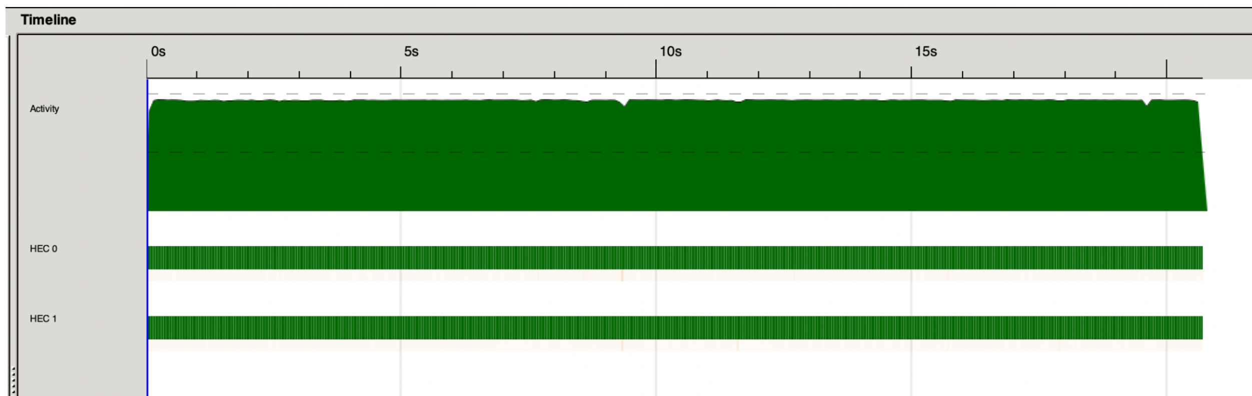


-N12

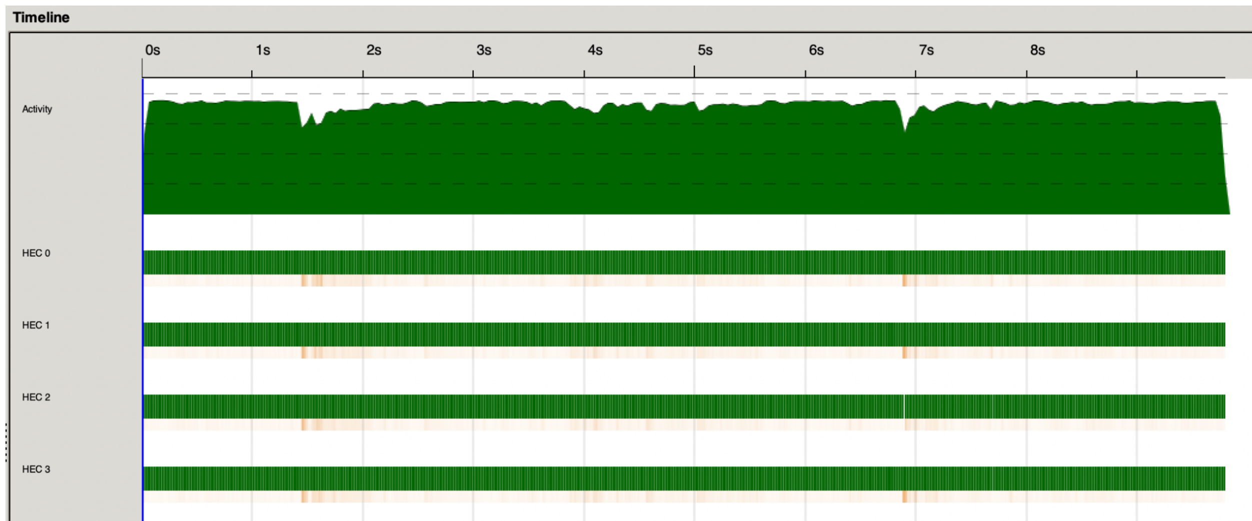


Parallel Depth = 4

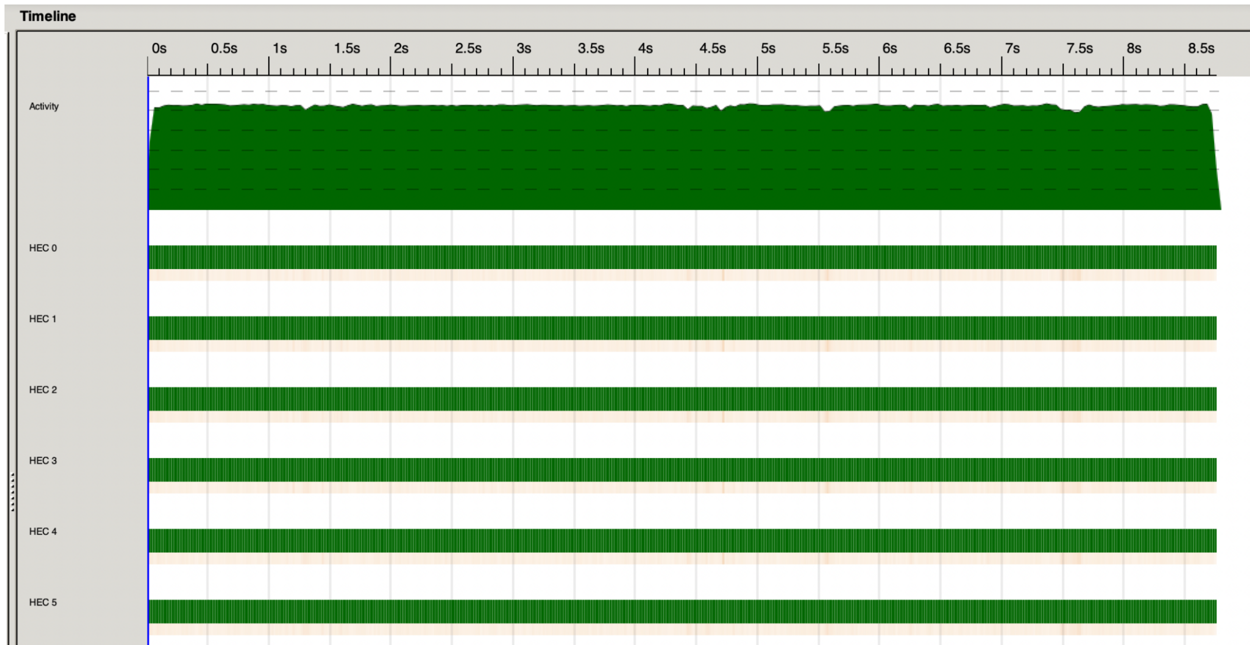
-N2



-N4

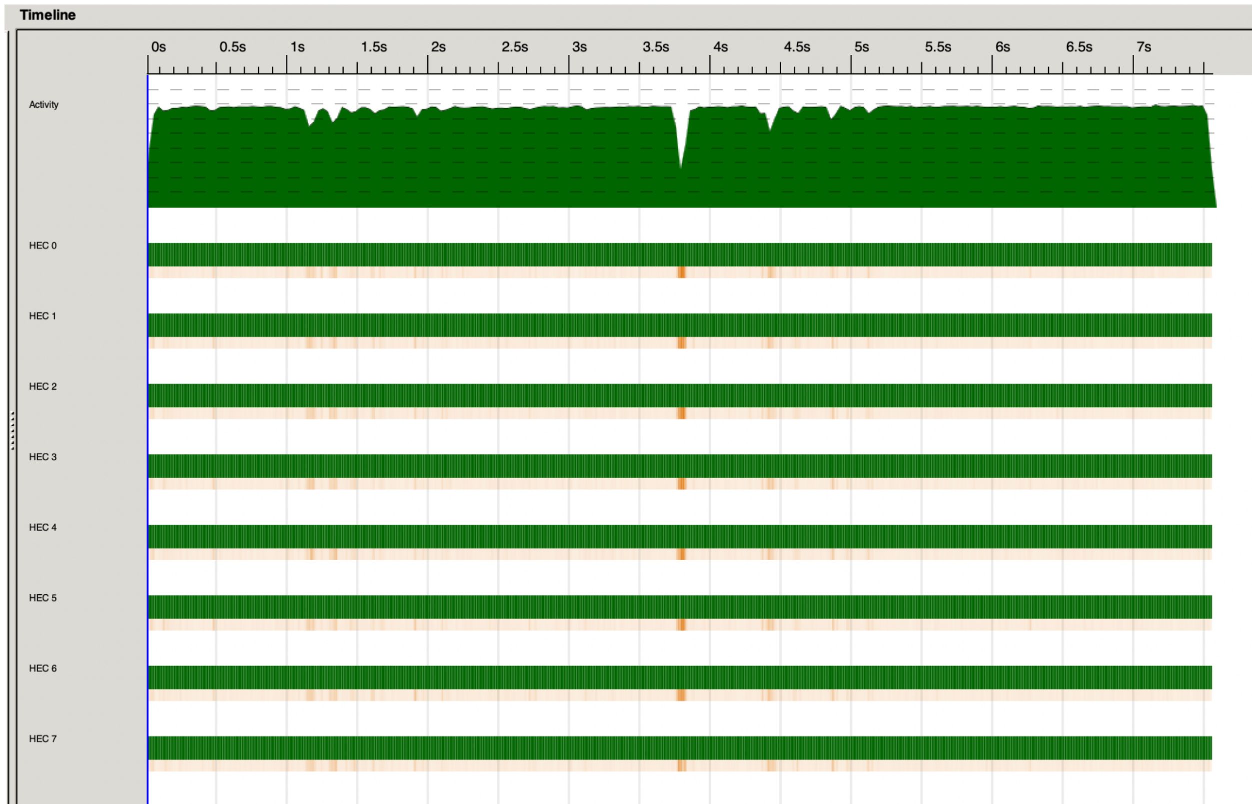


-N6

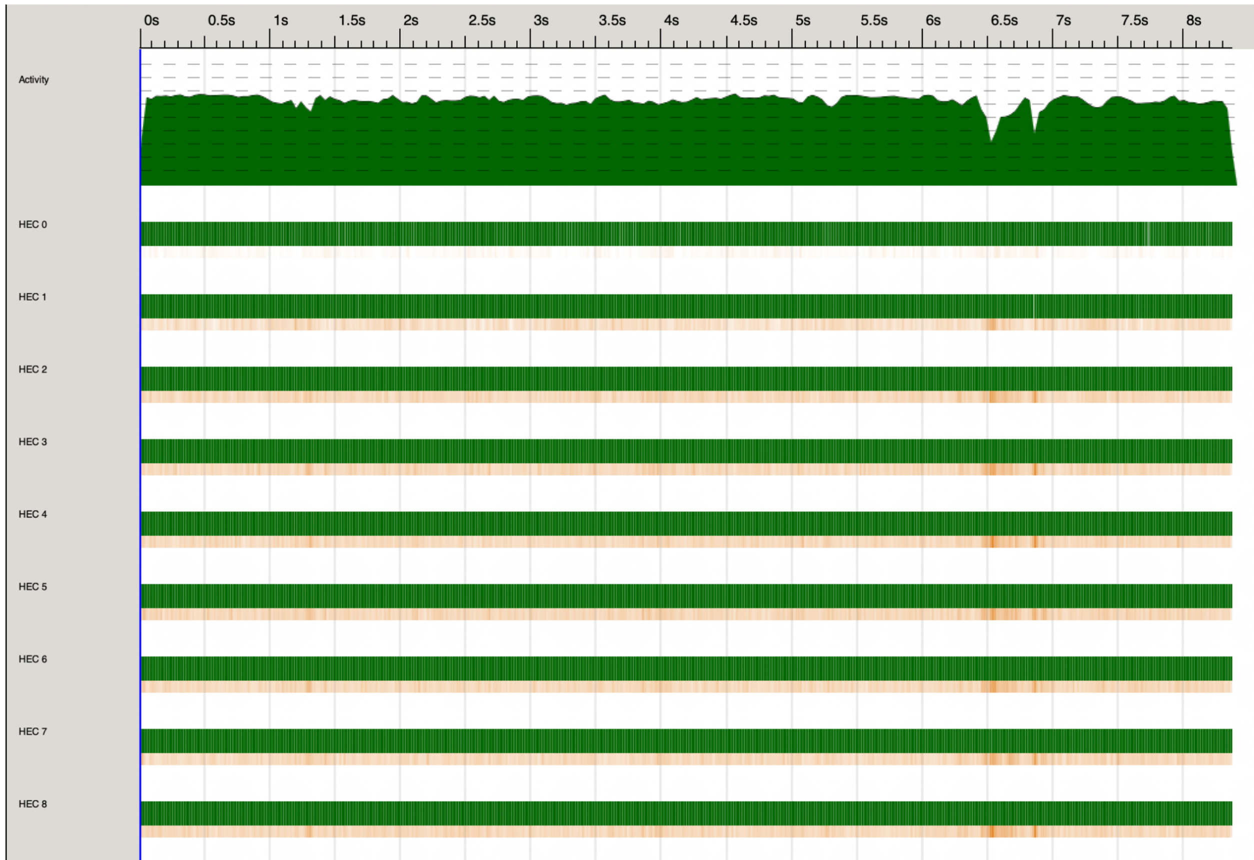


-N7

-N8



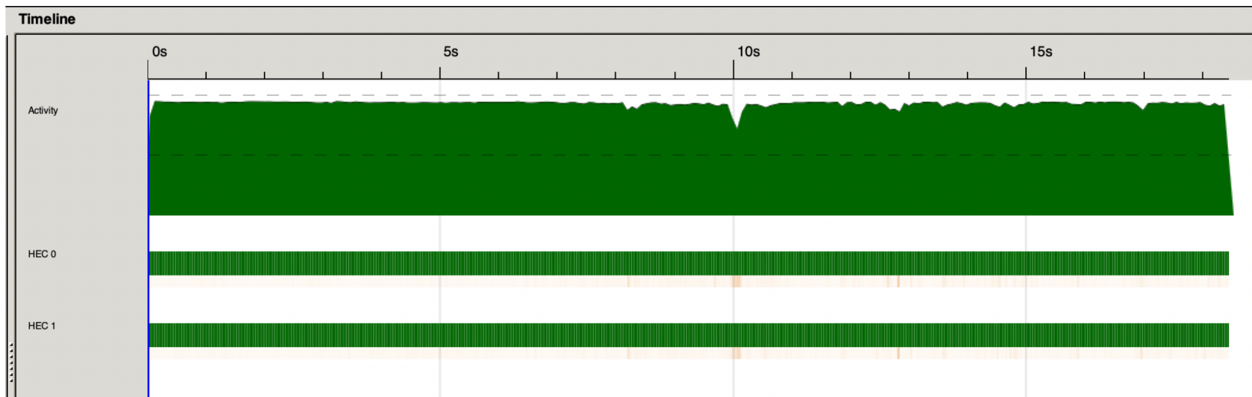
-N9



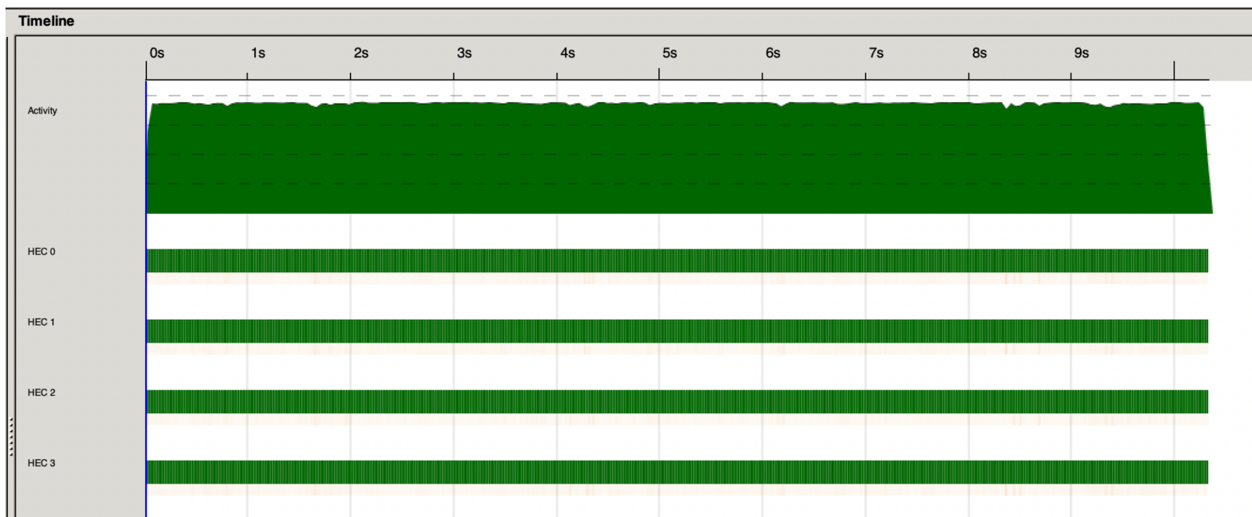
-N10

Parallel Depth = 5

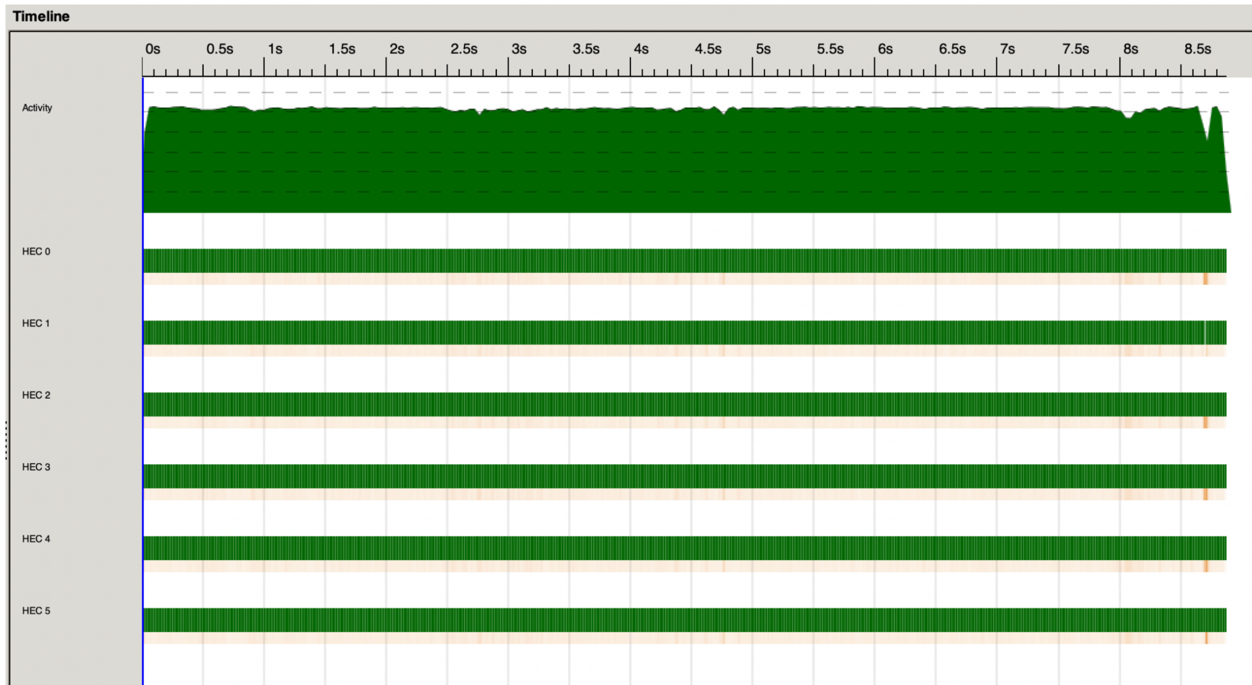
-N2



-N4

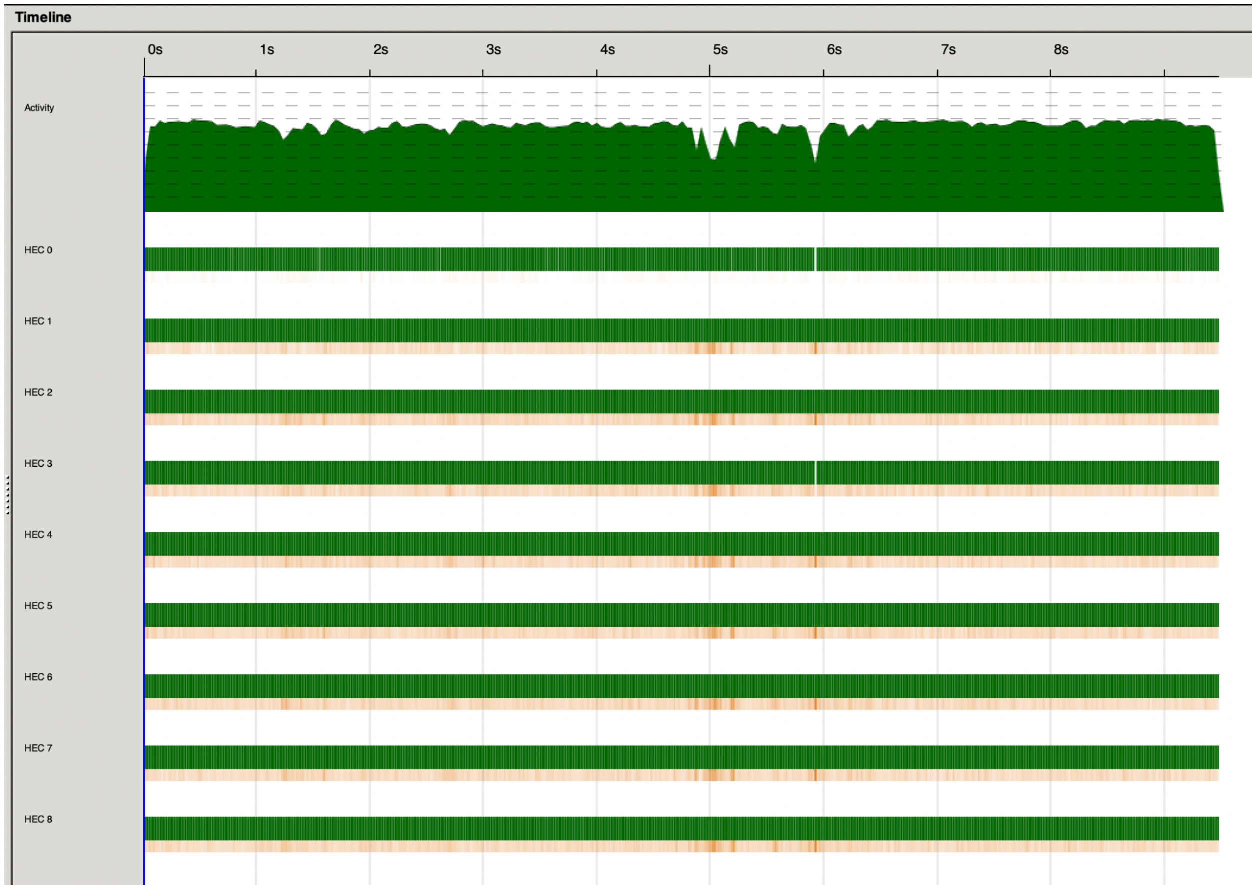


-N6

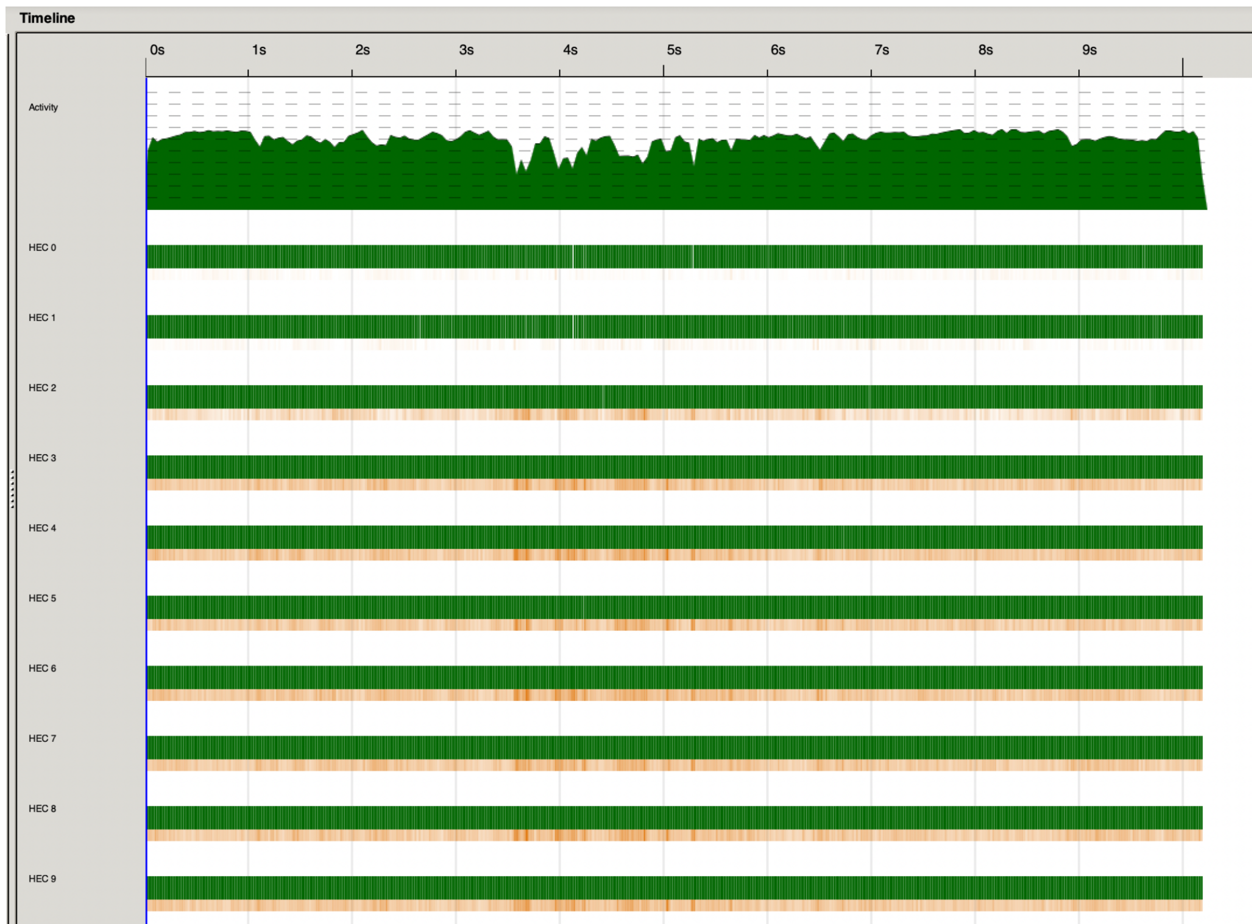


-N8

-N9



-N10



Future Work

Based on the current progress and experimental results of the project, the following could be considered for future work:

- Implement alpha-beta pruning
- Utilize a cache table in the minimax algorithm to avoid duplicated sub-branch calculation
- Fine-tuning spark parameters to reduce overheads from high fizzle rate
- Improve game board display on the console

Code Listing

Main.hs

- Main function that gets the user input command and starts the game

```
module Main where
import Lib
import System.Environment (getArgs, getProgName)
import System.Exit (die)

main :: IO ()
main = do
 args <- getArgs
 case args of
   [config_file, mode] -> do gameFunc config_file mode
   _ -> do
     prog <- getProgName
     putStrLn $ "Usage: " ++ prog ++ " <configuration_file> <program-mode>"
     putStrLn "'algo-seq': running sequential minimax algorithm ONLY"
     putStrLn "'algo-par': running parallel minimax algorithm ONLY"
     putStrLn "'game-seq': game-mode using sequential minimax algorithm"
     putStrLn "'game-par': game-mode using parallel minimax algorithm"
     die ""
```

Lib.hs

- contains all game modeling and different minimax functions

```
module Lib
 ( gameFunc,
 )
where

import Control.Parallel.Strategies (parList, rseq, using)
import qualified Data.Map.Strict as Map
import qualified Data.Set as Set
import Debug.Trace ()
import Foreign.C.String (castCharToCSChar)
import System.Exit (die)
import System.IO (Handle, IOMode (ReadMode), hGetLine, hIsEOF, withFile)

gameFunc :: String -> String -> IO ()
gameFunc config_file mode = do
 (eSet, bList) <- readConfig config_file
 case mode of
   "algo-seq" -> do
     putStrLn "Running SEQUENTIAL minimax algorithm with tree depth = 9"
     let (score, edge) = minimaxDepLim False 9 (eSet, bList, 0) (Edge 0 False)
     putStrLn $ "algorithm suggested: " ++ show edge
     return ()
   "algo-par" -> do
     let pDepth = 1
     putStrLn $ "Running PARALLEL minimax algorithm with tree depth = 9 and parallel
depth = " ++ show pDepth
     let (score, edge) = minimaxParDep (False, pDepth, 9, (eSet, bList, 0), Edge 0
False)
     putStrLn $ "algorithm suggested: " ++ show edge
     return ()
   "game-seq" -> do
     putStrLn "Playing game with SEQUENTIAL minimax algorithm"
     depth <- getAlgoDepth
     putStrLn "Game Board: "
     mapM_ print bList
     putStrLn "Available edges :"
     putStrLn $ "" ++ printEdgeList (Set.toList eSet)
     putStrLn "SEQUENTIAL Game is starting..."
     gameStartSeq eSet bList depth
```

```
   "game-par" -> do
     putStrLn "Playing game with PARALLEL minimax algorithm"
     putStrLn "Game Board: "
     mapM_ print bList
     putStrLn "Available edges :"
     putStrLn $ "" ++ printEdgeList (Set.toList eSet)
     depth <- getAlgoDepth
     pDepth <- getParDepth
     putStrLn (printEdgeList (Set.toList eSet))
     putStrLn "PARALLEL Game is starting..."
     gameStartPar eSet bList depth pDepth
   _ -> die "Invalid game mode: options are 'algo-seq', 'algo-par', 'game-seq',
'game-par'"

-- Int = unique identification per edge
-- Bool = whether the edge is taken
data Edge = Edge {eid :: Int, flag :: Bool}

data Box = Box
 { edges :: [Edge],
   val :: Int
 }
 deriving (Eq)

instance Ord Edge where
 (Edge v1 _) `compare` (Edge v2 _) = v1 `compare` v2

instance Show Edge where
 show (Edge x f) = "(Edge " ++ show x ++ " " ++ show f ++ ")"

instance Eq Edge where
 (Edge v1 _) == (Edge v2 _) = v1 == v2

instance Show Box where
 show (Box l v) = "Box " ++ printEdgeList l ++ " " ++ show v

printEdgeList :: [Edge] -> String
printEdgeList [] = []
printEdgeList (x : xs) = show x ++ " " ++ printEdgeList xs

readConfig :: String -> IO (Set.Set Edge, [Box])
readConfig fname = withFile fname ReadMode initiateGameBoard
```

```haskell
initiateGameBoard :: Handle -> IO (Set.Set Edge, [Box])
initiateGameBoard h = do
 res <- hIsEOF h
 if res
   then return (Set.empty, [])
   else do
     (b, eList) <- initBox h
     (eSet, bList) <- initiateGameBoard h
     let newESet = foldl (flip Set.insert) eSet eList
     return (newESet, b : bList)

initBox :: Handle -> IO (Box, [Edge])
initBox h = do
 line <- hGetLine h
 case words line of
   l@[v, e1, e2, e3, e4] -> do
     let edgeL = foldl (\acc x -> Edge (read x) False : acc) [] (tail l)
     let box = Box edgeL (read v)
     return (box, edgeL)
   _ -> die "Board Configuration Read Error."

getAlgoDepth :: IO Int
getAlgoDepth = do
 putStrLn "Enter a depth for the AI search tree: "
 putStrLn "(if running on 2x2 board, enter depth < 9 for speed up): "
 read <$> getLine

getParDepth :: IO Int
getParDepth = do
 putStrLn "Enter a depth for parallelism: "
 read <$> getLine


-- note: computer makes the first move
gameStartSeq :: Set.Set Edge -> [Box] -> Int -> IO ()
gameStartSeq edgeSet boxList depth =
 if Set.null edgeSet || null boxList
   then die "Error: starting the game because edge set or box list is empty."
   else do
     -- putStrLn "inside game starts, before game loop"
     res <- gameLoopSeq edgeSet boxList False 0 depth
```

```haskell
     case res `compare` 0 of
       LT -> putStrLn "Human WIN!"
       EQ -> putStrLn "DRAW!"
       GT -> putStrLn "Computer WIN!"

-- note: computer makes the first move
gameStartPar :: Set.Set Edge -> [Box] -> Int -> Int -> IO ()
gameStartPar edgeSet boxList depth pDepth =
 if Set.null edgeSet || null boxList
   then die "Error: starting the game because edge set or box list is empty."
   else do
     -- putStrLn "inside game starts, before game loop"
     res <- gameLoopPar edgeSet boxList False 0 depth pDepth
     case res `compare` 0 of
       LT -> putStrLn "Human WIN!"
       EQ -> putStrLn "DRAW!"
       GT -> putStrLn "Computer WIN!"

{-
gameLoop: send game control between player and computer

param 1: set of remaining edges
param 2: list of remaining boxes
param 3: flag denoting turn (computer = False, human = True)
param 4: integer cumulative score of computer
param 5: integer cumulative score of human

Return: -1 = COMPUTER win
         0 = Tie
         1 = HUMAN win

-}
gameLoopSeq :: Set.Set Edge -> [Box] -> Bool -> Int -> Int -> IO Int
gameLoopSeq eSet bList t aiScore depth
 | Set.null eSet = return aiScore
 | t = do
   putStrLn "HUMAN move:"
   putStrLn "Available box list: "
   mapM_ print bList
   eId <- getHumanMove eSet
   let nextEdgeH = Edge eId False
```

```
   let (newEdgeH, newBoxH, newScoreH) = nextGameState nextEdgeH (eSet, bList) aiScore
t
   putStrLn $ "Score after human move: " ++ show newScoreH
   putStrLn "-----------------------------------------------"
   gameLoopSeq newEdgeH newBoxH False newScoreH depth
 | otherwise = do
   putStrLn "AI move:"
   let (_, nextEdgeC) = minimaxDepLim False depth (eSet, bList, aiScore) (Edge 0
False)
   putStrLn $ "AI chose:" ++ show nextEdgeC
   let (newEdgeC, newBoxC, newScoreC) = nextGameState nextEdgeC (eSet, bList) aiScore
t
   putStrLn $ "Score after AI move: " ++ show newScoreC
   putStrLn "-----------------------------------------------"
   gameLoopSeq newEdgeC newBoxC True newScoreC depth

gameLoopPar :: Set.Set Edge -> [Box] -> Bool -> Int -> Int -> Int -> IO Int
gameLoopPar eSet bList t aiScore depth pDepth
 | Set.null eSet = return aiScore
 | t = do
   putStrLn "HUMAN move:"
   putStrLn "Available box list: "
   mapM_ print bList
   eId <- getHumanMove eSet
   let nextEdgeH = Edge eId False
   let (newEdgeH, newBoxH, newScoreH) = nextGameState nextEdgeH (eSet, bList) aiScore
t
   putStrLn $ "Score after human move: " ++ show newScoreH
   putStrLn "-----------------------------------------------"
   gameLoopSeq newEdgeH newBoxH False newScoreH depth
 | otherwise = do
   putStrLn "AI move:"
   let (_, nextEdgeC) = minimaxParDep (False, pDepth, depth, (eSet, bList, aiScore),
Edge 0 False)
   putStrLn $ "AI chose:" ++ show nextEdgeC
   let (newEdgeC, newBoxC, newScoreC) = nextGameState nextEdgeC (eSet, bList) aiScore
t
   putStrLn $ "Score after AI move: " ++ show newScoreC
   putStrLn "-----------------------------------------------"
   gameLoopSeq newEdgeC newBoxC True newScoreC depth

{-
```

```
cmpScore:

param1: computer score
param2: human score

Return: -1 = COMPUTER win
         0 = Tie
         1 = HUMAN win

cmpScore :: Int a => a -> a -> a
cmpScore cScore hScore | cScore > hScore = -1
                       | hScore > cScore =  1
                       | otherwise       =  0
-}

nextGameState :: Edge -> (Set.Set Edge, [Box]) -> Int -> Bool -> (Set.Set Edge, [Box],
Int)
nextGameState e (eSet, bList) score player =
 if player
   then (newS, newL, score - sUpdate)
   else (newS, newL, score + sUpdate)
 where
   (newS, newL, sUpdate) = gameAction e (eSet, bList)

{-
gameAction:

param1: player selected edge (type of Edge) to remove
param2: (x1, x2, x3), s.t. x1 is the current set of available edges,
                           x2 is the current list of available boxes,
                           x3 is the current score of the player.

Return: (e1, e2, e3), s.t. e1 is new set of remaining edges,
                           e2 is new list of remaining boxes,
                           e3 is the updated score
-}

gameAction :: Edge -> (Set.Set Edge, [Box]) -> (Set.Set Edge, [Box], Int)
gameAction targetEdge (eSet, bList) = (Set.delete targetEdge eSet, newBoxList,
scoreChanged)
 where
   applyAction (accl, newList) b
```

```
      | not (containEdge targetEdge (edges b)) = (accl, b : newList)
      | containEdge targetEdge (edges b) && boxFilled b = (accl + val b, newList)
      | otherwise = (accl, newBox b : newList)
    newBox box = Box (newEdgeList (edges box)) (val box)
    newEdgeList oldList = Edge (eid targetEdge) True : filter (/= targetEdge) oldList
    containEdge tar eList = targetEdge `elem` eList
    boxFilled box = all (\(Edge _ f) -> f) (filter (/= targetEdge) (edges box))
    (scoreChanged, newBoxList) = foldl applyAction (0, []) bList

getHumanMove :: Set.Set Edge -> IO Int
getHumanMove eSet = do
 putStrLn "Please make the next move."
 putStrLn $ "Available edges: " ++ show (printEdgeList $ Set.toList eSet)
 read <$> getLine



-- base minimax
-- minimax :: Bool -> (Set.Set Edge, [Box], Int) -> Edge -> (Int, Edge)
-- minimax player (edgeset, boxlist, aiScore) edge
--    | rootnode        = bestMove [minimax True x e | (x, e) <- initExpandedStates]
--    | terminal        = (aiScore, edge)
--    | not player      = bestMove [minimax True x e | (x, e) <- subExpandedStates]
--    | player          = worstMove [minimax False x e | (x, e) <- subExpandedStates]
--    | otherwise       = error "invalid game state"
--    where
--      initExpandedStates = [(getNextGameState e, e) | e <- edgelist]
--      subExpandedStates = [(getNextGameState e, edge) | e <- edgelist]
--      getNextGameState someEdge = nextGameState someEdge (edgeset, boxlist) aiScore
player
--      edgelist = Set.toList edgeset
--      terminal = Set.null edgeset
--      rootnode = edge == Edge 0 False



{-
minimaxDepLim:

param1: player -> current player (True for human, False for AI)
param2: depth -> levels remained to traverse from current node
param3: (edgeset, boxlist, aiScore) ->  edgeset is a set of available edges,
                                       boxlist is a list of available boxes,
                                       aiScore is the current score of AI/board.
```

```haskell
param4: edge -> the initially chosen edge for this branch of the tree
return: (aiScore, Edge) -> best move for AI
-}

minimaxDepLim :: (Eq t, Num t) => Bool -> t -> (Set.Set Edge, [Box], Int) -> Edge ->
(Int, Edge)
minimaxDepLim player depth (edgeset, boxlist, aiScore) edge
 | rootnode     = bestMove [minimaxDepLim True newDepth x e | (x, e) <-
initExpandedStates]
 | terminal || depth == 0 = (aiScore, edge)
 | not player   = bestMove [minimaxDepLim True newDepth x e | (x, e) <-
subExpandedStates]
 | player       = worstMove [minimaxDepLim False newDepth x e | (x, e) <-
subExpandedStates]
 | otherwise    = error "invalid game state"
 where
   newDepth = depth - 1
   initExpandedStates = [(getNextGameState e, e) | e <- edgelist]
   subExpandedStates = [(getNextGameState e, edge) | e <- edgelist]
   getNextGameState someEdge = nextGameState someEdge (edgeset, boxlist) aiScore
player
   edgelist = Set.toList edgeset
   terminal = Set.null edgeset
   rootnode = edge == Edge 0 False

{-
minimaxParDep:

param1: player -> current player (True for human, False for AI)
param2: parDepth -> parallel levels remained to traverse from current node
param2: seqDepth -> sequential levels remained to traverse from current node
param4: (edgeset, boxlist, aiScore) ->  edgeset is a set of available edges,
                                       boxlist is a list of available boxes,
                                       aiScore is the current score of AI/board.
param5: edge -> the initially chosen edge for this branch of the tree
return: (aiScore, Edge) -> best move for AI
-}
minimaxParDep :: (Eq a, Eq t, Num t, Num a) => (Bool, a, t, (Set.Set Edge, [Box],
Int), Edge) -> (Int, Edge)
minimaxParDep (player, parDepth, seqDepth, (edgeset, boxlist, aiScore), edge)
 | parDepth == 0            = minimaxDepLim player seqDepth (edgeset, boxlist,
aiScore) edge
```

```haskell
 | rootnode                 = bestMove parResultInitMax
 | terminal                 = (aiScore, edge)
 | not player               = bestMove parResultSubMax
 | player                   = worstMove parResultSubMin
 | otherwise                = error "invalid game state"
 where
   parResultInitMax = map minimaxParDep paramListInitMax `using` parList rseq
   parResultSubMax = map minimaxParDep paramListSubMax `using` parList rseq
   parResultSubMin = map minimaxParDep paramListSubMax `using` parList rseq

   newParDepth = parDepth - 1
   newSeqDepth = seqDepth - 1
   paramListInitMax = [(True, newParDepth, newSeqDepth, x, e) | (x, e) <-
initExpandedStates]
   paramListSubMax = [(True, newParDepth, newSeqDepth, x, e) | (x, e) <-
subExpandedStates]
   paramListSubMin = [(False, newParDepth, newSeqDepth, x, e) | (x, e) <-
subExpandedStates]

   initExpandedStates = [(getNextGameState e, e) | e <- edgelist]
   subExpandedStates = [(getNextGameState e, edge) | e <- edgelist]
   getNextGameState someEdge = nextGameState someEdge (edgeset, boxlist) aiScore
player
   edgelist = Set.toList edgeset
   terminal = Set.null edgeset
   rootnode = edge == Edge 0 False

{-
bestMove:

param1: [(score, edge)] -> a list of available move and the associated heuristic

return: (aiScore, Edge) -> best move for AI(with highest heuristic)
-}
bestMove :: [(Int, Edge)] -> (Int, Edge)
bestMove [(score, edge)] = (score, edge)
bestMove ((score, edge) : (score', edge') : xs) = bestMove (if score >= score' then
(score, edge) : xs else (score', edge') : xs)
bestMove _ = error "BestMove: not a valid move"

{-
worstMove:
```

```haskell
param1: [(score, edge)] -> a list of available move and the associated heuristic

return: (aiScore, Edge) -> worst move for AI(with lowerst heuristic)
-}
worstMove :: [(Int, Edge)] -> (Int, Edge)
worstMove [(score, edge)] = (score, edge)
worstMove ((score, edge) : (score', edge') : xs) = worstMove (if score <= score' then
(score, edge) : xs else (score', edge') : xs)
worstMove _ = error "WorstMove: not a valid move"
```