Parallelizing Word-Search-2

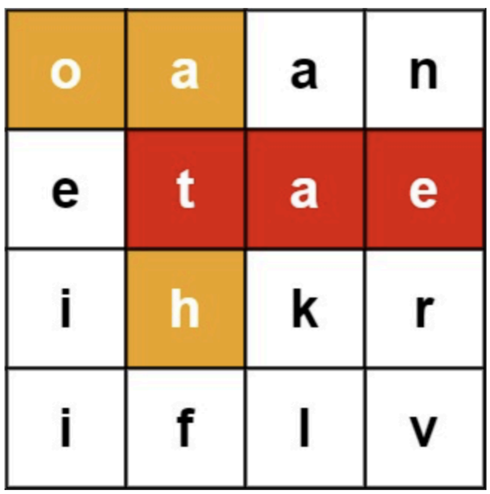
COMS W4995: Parallel Functional Programming

Ardrian Wong (UNI:aaw2179), Keith Lo (UNI:kl3695), Sean Zhang (UNI:srz2116)

Introduction:

For our project, we aim to parallelize the word search 2 problem on Leetcode.[1] In our problem, we are given a board of characters as well as a list of strings (words), and we must return all words that are found on the board. A word can be formed on the board by a series of adjacent characters, where a character is adjacent to another if it is vertically or horizontally neighboring. Furthermore, character cells may not be repeated within the same word. We give an example input, solution, and visualization:

Input: board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]], words = ["oath","pea","eat","rain"]
Output: ["eat","oath"]



We first solved this problem using a sequential algorithm, which we will present in the following section. Our next steps were to improve the speed of our solution by attempting three different methods of parallelizing portions of our sequential algorithm, to varying degrees of success. We will present all three methods along with their results and a comparison to the results of the sequential solution.

---

[1] Word Search II. *LeetCode*, from https://leetcode.com/problems/word-search-ii/description

Sequential Algorithm:

Our sequential algorithm utilizes a trie and DFS to solve the word search problem. We first store all target strings in a trie, which is an efficient data structure for prefix matching.

```haskell
-- Trie data structure
data Trie = Trie {
    children :: Map.Map Char Trie,
    isEnd :: Bool
} deriving (Show)

-- Create empty trie
emptyTrie :: Trie
emptyTrie = Trie Map.empty False

-- Insert a word into trie
insertWord :: String -> Trie -> Trie
insertWord "" trie = trie { isEnd = True }
insertWord (c:cs) trie =
    let childTrie = fromMaybe emptyTrie (Map.lookup c (children trie))
        newChild = insertWord cs childTrie
    in trie { children = Map.insert c newChild (children trie) }
```

Building a trie: we use foldr with insertWord, an emptyTrie, and our list of target strings.

Once we have our trie, we can begin searching the board, initiating DFS from each cell. We search in every valid direction from the current cell, comparing the characters of the neighboring cells to the children of the current trie node to see if our current path is the prefix of a target string, or if we can end the DFS branch early. We must also mark cells as visited so we don't revisit already used cells. We can also prune the trie once a word has been found to avoid searching for words we have already found on the board. Finally, we must unmark the visited cells once we have finished searching along a path, so they can be visited through searches from different paths/cells.

```haskell
-- Main search function
findWords :: [[Char]] -> [String] -> [String]
findWords board targetWords = nub $ concatMap (\(r,c) ->
    searchFromCell board trie r c []
    ) [(r,c) | r <- [0..rows-1], c <- [0..cols-1]]
  where
    rows = length board
    cols = length (head board)
    trie = foldr insertWord emptyTrie targetWords

-- Search from a specific cell
searchFromCell :: [[Char]] -> Trie -> Int -> Int -> String -> [String]
searchFromCell board trie row col currWord
    | row < 0 || row >= rows || col < 0 || col >= cols = []
    | board !! row !! col == '*' = []  -- Check for visited cell
    | not (Map.member curr (children trie)) = []
    | otherwise =
        let newTrie = fromMaybe emptyTrie (Map.lookup curr (children trie))
            newWord = currWord ++ [curr]
            foundWords = [newWord | isEnd newTrie]
            markedBoard = markCell board row col
            nextWords = concatMap (\(dr,dc) ->
                searchFromCell markedBoard newTrie (row+dr) (col+dc) newWord
                ) [(0,1), (1,0), (0,-1), (-1,0)]
        in foundWords ++ nextWords
  where
    rows = length board
    cols = length (head board)
    curr = board !! row !! col
```

findWords initiates searchFromCell from each cell in the board and concatenates the results.

Parallelism:

In order to speed up performance of our sequential algorithm, we attempt to introduce parallelism to our sequential algorithm via three different methods:

- Parallelize the search for each individual target word
- Parallelizing DFS branches set to a configurable depth
- Parallelize DFS by breaking up the word grid into a configurable number of subgrids

ParallelWords

The ParallelWords algorithm works by parallelizing the search of each target word. The algorithm uses similar logic to the sequential algorithm, except that within a single DFS call, it cannot find any word except the target word it is parallelized to search for. We modify the function searchFromCell to enable this behavior. Like in the sequential version, we store all the target words in a trie. Next, we use parMap to create a spark for each target word which will run the function searchSingleWord to initiate the DFS. rdeepseq is used to ensure that each spark is fully evaluated.

```haskell
findWords :: [[Char]] -> [String] -> [String]
findWords board targetWords =
    let trie = foldr insertWord emptyTrie targetWords
        -- Use parMap with rdeepseq to force full evaluation in parallel
        results = parMap rdeepseq (searchSingleWord board trie) targetWords
    in nub (concat results)


searchSingleWord :: [[Char]] -> Trie -> String -> [String]
searchSingleWord board trie word =
    searchUntilFound Set.empty [(r,c) | r <- [0..rows-1], c <- [0..cols-1]]
  where
    rows = length board
    cols = length (head board)

    searchUntilFound _ [] = []
    searchUntilFound visited ((r,c):rest) =
        case searchFromCell board trie r c [] word Set.empty of
            [] -> searchUntilFound visited rest
            found -> found

-- Modified to use Set for visited positions
searchFromCell :: [[Char]] -> Trie -> Int -> Int -> String -> String -> Set.Set Pos -> [String]
searchFromCell board trie row col currWord targetWord visited
    | row < 0 || row >= rows || col < 0 || col >= cols = []
    | Set.member (row, col) visited = []
    | not (Map.member curr (children trie)) = []
    | otherwise =
        let newTrie = fromMaybe emptyTrie (Map.lookup curr (children trie))
            newWord = currWord ++ [curr]
            foundWords = [newWord | isEnd newTrie && newWord == targetWord]
            newVisited = Set.insert (row, col) visited
            nextWords = concatMap (\(dr,dc) ->
                searchFromCell board newTrie (row+dr) (col+dc) newWord targetWord newVisited
                ) [(0,1), (1,0), (0,-1), (-1,0)]
        in foundWords ++ nextWords
  where
    rows = length board
    cols = length (head board)
    curr = board !! row !! col      You, 2 weeks ago • Initial work for project
```

ParallelDepth

The ParallelDepth algorithm works by using parallel processing while performing DFS that is limited by the depth. The depth control then limits how deep the DFS can go and explore before cutting it off from searching on that path. This prevents the search from becoming too computationally expensive for larger grids.

The algorithm uses a trie to store the target words and a visited set as in the sequential algorithm to facilitate DFS.

As the DFS starts exploring, the algorithm uses parMap and rseq to explore neighboring cells at the same time at each level of the recursion. parMap splits the recursive search into parallel tasks and allows for the multiple neighbors to be explored simultaneously. Rseq ensures that the results from these parallel evaluations are done immediately so that lazy evaluation does not occur.

```haskell
findWords :: Int -> [[Char]] -> [String] -> [String]
findWords depth board targetWords =
    nub $ concat $ parMap rseq (\(r, c) -> searchFromCell board trie Set.empty r c [] depth 0 rows cols)
    [(r, c) | r <- [0..rows-1], c <- [0..cols-1]]
  where
    rows = length board
    cols = length (head board)
    trie = foldr insertWord emptyTrie targetWords

searchFromCell :: [[Char]] -> Trie -> Set (Int, Int) -> Int -> Int -> String -> Int -> Int -> Int -> Int -> [String]
searchFromCell board trie visited row col currWord depth level rows cols
    | row < 0 || row >= rows || col < 0 || col >= cols = []
    | Set.member (row, col) visited = []
    | not (Map.member curr (children trie)) = []
    | otherwise =
        let newTrie = fromMaybe emptyTrie (Map.lookup curr (children trie))
            newWord = currWord ++ [curr]
            foundWords = [newWord | isEnd newTrie]
            newVisited = Set.insert (row, col) visited

            nextWords = if level < depth
                        then concat $ parMap rseq (\(r, c) -> searchFromCell board newTrie newVisited r c newWord depth (level + 1) rows cols)
                                    [(row+1, col), (row, col+1), (row-1, col), (row, col-1)]
                        else concatMap (\(r, c) -> searchFromCell board newTrie newVisited r c newWord depth (level + 1) rows cols)
                                    [(row+1, col), (row, col+1), (row-1, col), (row, col-1)]
        in foundWords ++ nextWords
  where
    curr = board !! row !! col
```

findWords has an extra parameter to control the recursion depth during the DFS. searchFromCell searches neighboring cells at the same time recursively, but stops early if the level of recursion exceeds the depth that is set.

ParallelSubgrids

The ParallelSubgrids algorithm works by first splitting the original board into a configurable number of subgrids (but always a square number. Ie: an input of 1 splits the original board into 1 subgrid, so it remains the same. An input of 2 splits the original board evenly into 4 subgrids, 3 splits into 9 subgrids, and so on), and sparks a search to be carried out in each of those subgrids. Any target strings found in each of the subgrids are then concatenated into a single list, producing the same result as the sequential search algorithm.

It is key to understand that “searching a subgrid” means searching for all strings that start in that subgrid, not just searching for strings that exist entirely within that subgrid. This means that a DFS path that starts in one subgrid can end in another. An initial misunderstanding of this concept in the code was the cause of a bug which resulted in extremely fast search times but also incomplete solutions, because the DFS paths terminated prematurely whenever they hit the boundaries of a subgrid.

ParallelSubgrids borrows the same trie data structure and DFS algorithm from our sequential solution. It also introduces a new function to split the board into subgrids, which returns the coordinate bounds of each subgrid.

```haskell
splitBoard :: Int -> [[Char]] -> [(Int, Int, Int, Int)]
splitBoard n board
    | n <= 1 = [(0, rows, 0, cols)]
    | n >= rows = [(r, r + 1, c, c + 1) | r <- [0..rows-1], c <- [0..min rows cols-1]]
    | n >= cols = [(r, r + 1, c, c + 1) | r <- [0..min rows cols-1], c <- [0..cols-1]]
    | otherwise =
        [(r * subRows, (r + 1) * subRows, c * subCols, (c + 1) * subCols)
         | r <- [0..n-1], c <- [0..n-1]]
  where
    rows = length board
    cols = length (head board)
    subRows = rows `div` n
    subCols = cols `div` n
```

splitBoard expects an Int representing the square root of the total number of subgrids to produce and the original board, and returns a list of 4-Int tuples, each representing a corner boundary of a subgrid.

These bounds are then used to define the individual coordinates of each cell within the subgrid that we need to initiate DFS from. Finally, we have a wrapper function that utilizes parMap with deepseq as the strategy to spark and force evaluation of the searches of the subgrids in parallel. We also create a trie in this wrapper function to avoid constructing copies of the trie in each subgrid search, as is the case with the findWords function in our sequential algorithm.

```haskell
findWordsSubgrids :: Int -> [[Char]] -> [String] -> [String]
findWordsSubgrids splits board wordsList =
    let subBoards = splitBoard splits board
        trie = foldr SequentialSearch.insertWord SequentialSearch.emptyTrie wordsList
        results = parMap rdeepseq (\subBoard -> findWordsTrie board trie subBoard) subBoards
    in nub (concat results)

findWordsTrie :: [[Char]] -> SequentialSearch.Trie -> (Int, Int, Int, Int)-> [String]
findWordsTrie board trie (rStart, rEnd, cStart, cEnd) =
    nub $ concatMap (\(r,c) ->
        SequentialSearch.searchFromCell board trie r c []
    ) [(r,c) | r <- [rStart..min rEnd (rows-1)], c <- [cStart..min cEnd (cols-1)]]
  where
    rows = length board
    cols = length (head board)
```

findWordsSubgrids generates the subgrid bounds, constructs the trie to be used in all subsequent parallel subgrid searches, and sparks parallel evaluation of the subgrid searches, before concatenating and returning results from each subgrid search. findWordsTrie initiates DFS from each cell to search for strings within the given bounds, similarly to findWords from the sequential solution.

Challenges:

One challenge we encountered was generating suitable data for testing. Leetcode's hardest test cases proved too small to adequately test our parallelized algorithms, and word search generators couldn't create grids with target words that snake around. To solve this, we wrote a custom test case generation script. It generates grids of random letters and produces a configurable number of target words using a randomized depth-first search, resulting in more varied and challenging test cases. We set word sizes to be between eight to fifteen characters long.

We also encountered a challenge with parallelization itself. Initially we tried to utilize par and pseq from Control.Parallel, but encountered issues with our results array not being fully evaluated in parallel and filled with thunks, so the evaluation would still just occur sequentially when the found target words were eventually printed out. Therefore we utilized parMap, rpar, rseq, and rdeepseq from Control.Parallel.Strategies in order to be able to force evaluation.

Hardware:

All testing was conducted on a 2022 Macbook Air with an Apple M2 chip, 8 cores and 8 hardware threads.

Algorithm Evaluation:

We benchmark performance on the following the following three test cases:

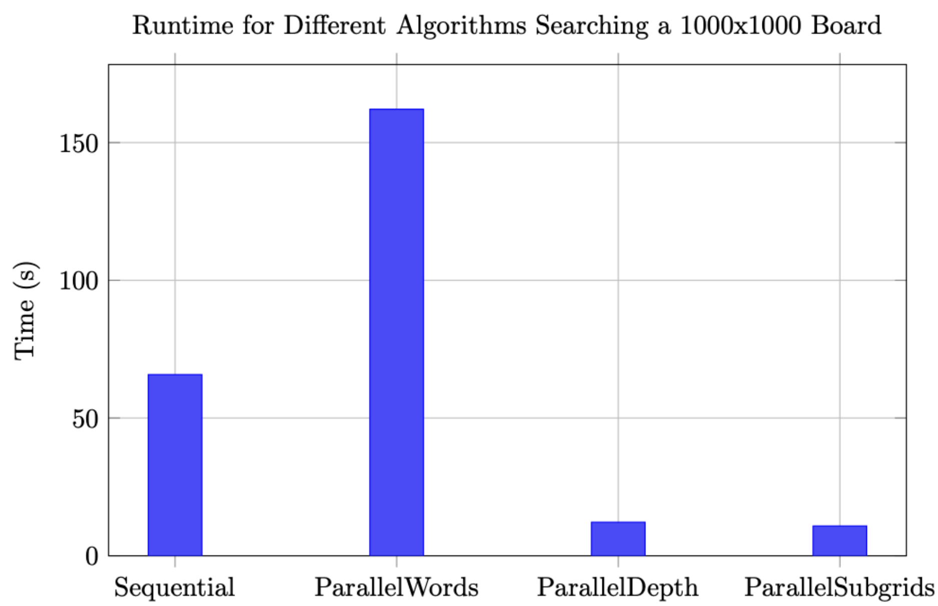
- 100x100 grid with 10 target words
- 500x500 grid with 20 target words
- 1000x1000 grid with 30 target words

We first parse the input from disk and then time the execution of the algorithm itself. This approach ensures that we exclude I/O time from our benchmarks.

Note: Target word length ranges from 8-15 characters.

Benchmark Results and Observations:

Benchmark Performance Overview



All parallel algorithms were run with 8 threads. ParallelDepth has depth 8 and ParallelSubgrids has 196 subgrids.

We experienced varying levels of success reducing runtime with our three parallel algorithms. ParallelWords took about 2.5x as long to run as the sequential algorithm on the largest board, even when using the maximum number of available threads. ParallelDepth and ParallelSubgrids were both much more successful attempts at parallelizing our word search algorithm, with both of them yielding runtimes about 6x faster than the sequential algorithm when used with the maximum number of available threads and optimal settings (that were encountered in our testing)

Sequential Algorithm

	Board Size		
	100x100	**500x500**	**1000x1000**
Time (s)	0.02597	5.491277	65.772943

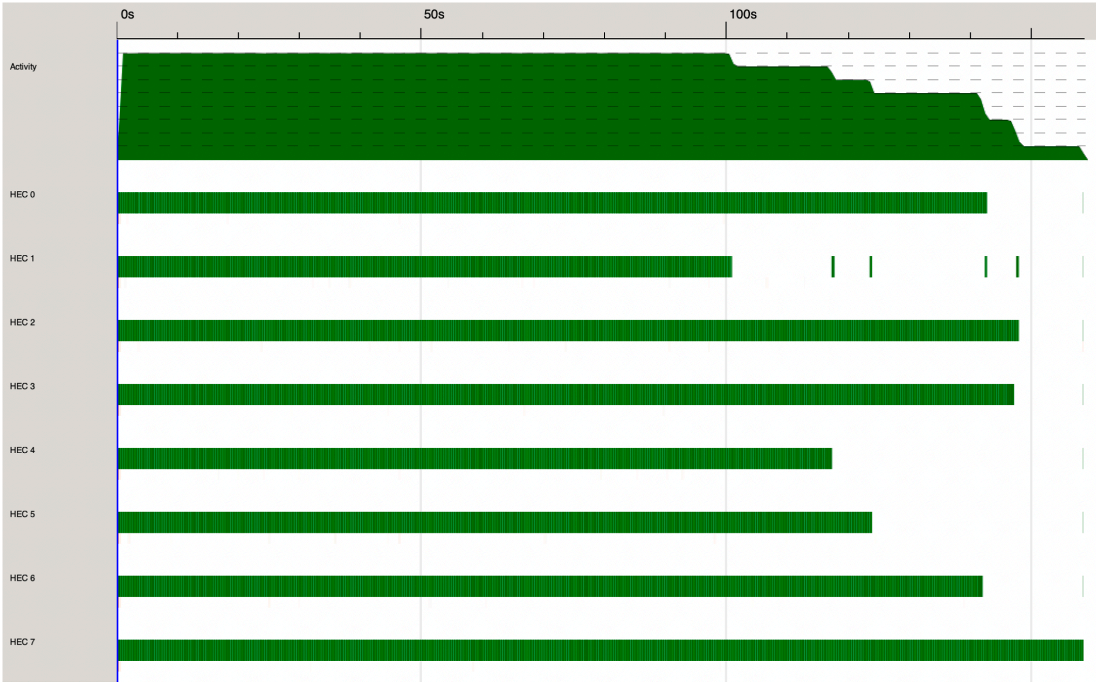
Table 1: Sequential algorithm runtimes.

Performance for the sequential algorithm significantly increases across test cases as the grid size becomes larger and there are more target words to find.

ParallelWords

Threads	**Board Size**		
	100x100	**500x500**	**1000x1000**
1	0.068571	25.964812	842.595844
2	0.042393	14.382055	460.157849
3	0.032634	11.135458	305.842003
4	0.025253	8.376221	257.679208
5	0.025845	7.921799	205.091986
6	0.020172	7.079879	192.660891
7	0.019657	6.866078	163.534461
8	0.021034	6.735405	162.122001

Table 2: ParallelWords runtimes (in seconds).



HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	30	29	0	0	0	1
HEC 0	0	4	0	0	0	0
HEC 1	30	1	0	0	0	0
HEC 2	0	3	0	0	0	1
HEC 3	0	4	0	0	0	0
HEC 4	0	4	0	0	0	0
HEC 5	0	4	0	0	0	0
HEC 6	0	4	0	0	0	0
HEC 7	0	5	0	0	0	0

ParallelWords threadscope graph and spark stats for 1000x1000 board, -N8.

The ParallelWords algorithm does not create that many sparks relative to our other methods of parallelization, so almost all sparks are converted.

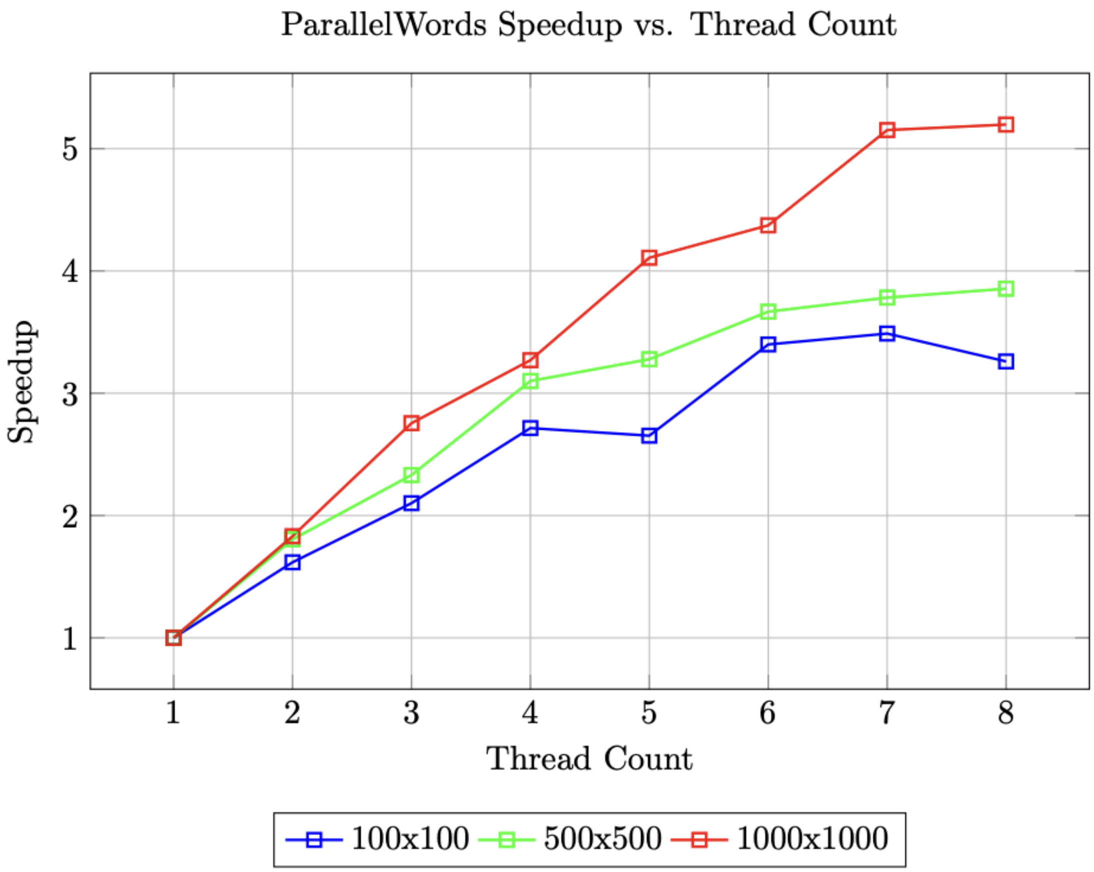


Figure 1: Speedup for varying thread counts across different board sizes.

The ParallelWords relative to itself does see a speed up as we add more cores up to our machine's level of parallelism and as the grid size increases.

Overall from the graphs above, the ParallelWords algorithm performs the most poorly, being more than 2x slower than the sequential algorithm.
This is due to the following factors:

- Poor method of parallelization: A spark is generated for each target word, and each spark is only able to search for its target word, unlike in the sequential algorithm where any target word can be found during a DFS call. We may have to traverse redundant paths for similar target words. Further, performance is greatly affected by the number of target words relative to the number of hardware threads. If all threads are busy, sparks created for other target words must wait to be run.
- Imbalanced workloads: Each target word can vary in length as well as in difficulty to find, hence in the threadscope plot we see that some cores have to do significantly more work than others.

ParallelDepth

Threads	Depth							
	1	2	3	4	5	6	7	8
1	0.011322	0.008303	0.008536	0.008033	0.007537	0.008134	0.008045	0.008056
2	0.005653	0.006023	0.005975	0.005881	0.005636	0.005456	0.005758	0.005735
3	0.005386	0.004665	0.005009	0.004220	0.004296	0.005503	0.004584	0.004349
4	0.005150	0.004027	0.005055	0.003764	0.004427	0.003619	0.003592	0.003517
5	0.005276	0.004489	0.004071	0.003365	0.003793	0.004183	0.003475	0.003655
6	0.003883	0.004574	0.004072	0.003841	0.003716	0.003658	0.003434	0.003916
7	0.003335	0.004316	0.004705	0.002976	0.003273	0.003236	0.003327	0.003888
8	0.003247	0.003345	0.003188	0.002966	0.002966	0.003227	0.003143	0.003054

Table 3: ParallelDepth runtimes (in seconds) for a 100x100 board.

Threads	Depth							
	1	2	3	4	5	6	7	8
1	0.991254	1.029463	1.054374	1.025064	1.016329	1.026675	1.011514	1.027848
2	0.994696	0.969331	1.029407	0.981233	0.979413	1.006443	1.009403	0.989883
3	0.973599	1.053631	1.005478	0.990769	1.111675	1.031081	1.027112	1.044589
4	1.017879	0.983769	1.032793	1.025403	1.005702	1.011490	0.983962	1.000132
5	1.042388	1.052232	1.004388	1.021215	1.004454	1.073756	1.003388	1.053224
6	1.059077	1.050075	1.023931	1.042896	1.041506	1.019629	1.017971	0.999466
7	1.055391	1.081520	1.013213	1.038237	1.064888	1.088402	1.044566	1.090042
8	1.042492	1.033371	1.043631	1.050237	1.036543	1.054486	1.052909	1.054817

Table 4: ParallelDepth runtimes (in seconds) for a 500x500 board.

Threads	Depth							
	1	2	3	4	5	6	7	8
1	11.70577	12.113667	13.801015	12.34687	11.565189	11.279337	11.866786	11.241193
2	11.28168	12.60596	11.886404	12.59457	11.249191	12.183987	11.441741	11.525345
3	11.907646	12.11129	12.81159	11.322576	12.4556	11.543598	12.298782	11.504465
4	11.485883	11.072132	11.509928	11.482786	11.998397	11.600105	11.861788	11.005588
5	11.684131	11.83232	11.658778	11.768344	12.078482	11.875231	12.134167	11.808079
6	11.37276	12.003249	11.371989	12.191675	12.297596	11.051718	11.889646	11.54167
7	11.954613	11.941797	12.601917	12.127493	11.678604	11.495271	11.818165	11.987016
8	11.671538	11.944176	11.662381	11.772531	11.695728	11.620753	13.652866	12.189594

Table 5: ParallelDepth runtimes (in seconds) for a 1000x1000 board.

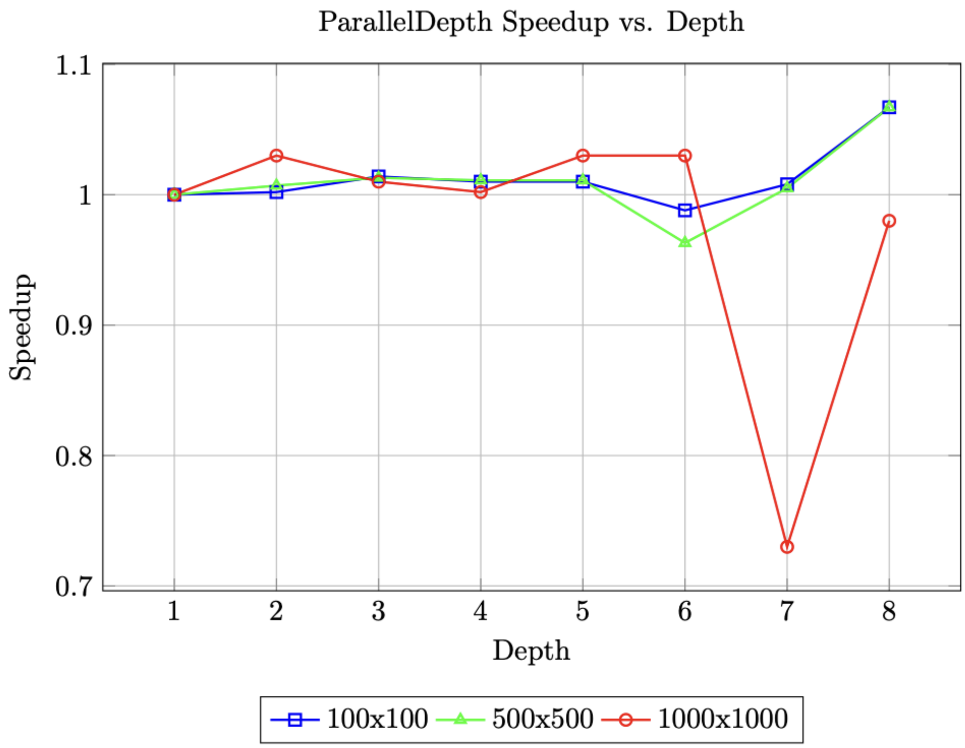


Figure 2: Speedup for varying depth across different board sizes, -N8.

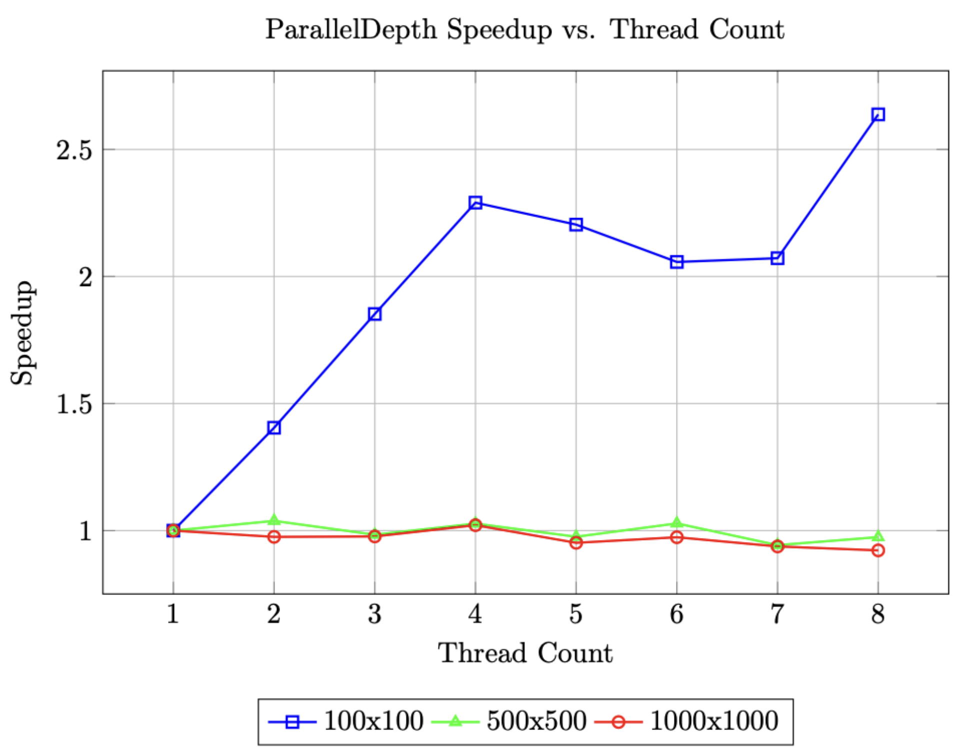


Figure 3: Speedup for varying thread count across different board sizes, depth 8.

ParallelDepth performed the second best of the algorithms that we tested. From Figure 2 we saw that the depth does not really affect the performance of the DFS across the 3 boards. In Figure 3, changing the thread count for the smallest board increased the speedup sharply and then steadily decreased. The thread count does not really affect the performance of the larger boards. This may be due to the fact that as we added in the initial extra threads, the overhead needed to manage the threads is relatively small as compared to when more threads were added later on. The lower amount of threads means that it can also handle memory access more efficiently as well. This may also be the reason why the thread count does not really affect the performance of the mid and large sized boards since they are 25 and 100 times larger (respectively) than the smallest board.

HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	4547676	24752	1053762	0	698930	2303288
HEC 0	6272	3558	0	0	3	133695
HEC 1	7736	3878	0	0	4	136728
HEC 2	4499552	15	1053762	0	698683	118
HEC 3	7296	3901	0	0	34	578039
HEC 4	7232	3641	0	0	44	506654
HEC 5	8184	4056	0	0	55	605699
HEC 6	5104	2490	0	0	51	171990
HEC 7	6300	3213	0	0	56	170365

ParallelDepth threadscope graph and spark stats for 1000x1000 board, depth 8, -N8.

ParallelDepth creates a lot of sparks as the board increases in size. This can overwhelm the number of threads that we are running with. In previous iterations, we also tried to sequentially traverse the grid without the use of parMap, but it still created many sparks that fizzled.

ParallelSubgrids

Threads	Subgrids									
	1	**4**	**16**	**36**	**64**	**100**	**144**	**196**	**256**	**10000**
1	0.024479	0.025394	0.026255	0.025326	0.028618	0.029674	0.028398	0.031161	0.031018	0.100272
2	0.025297	0.017083	0.015871	0.015709	0.015880	0.018148	0.017909	0.018792	0.018752	0.066717
3	0.025024	0.013823	0.012235	0.012123	0.011854	0.013303	0.012689	0.013752	0.013382	0.054323
4	0.025199	0.012065	0.010920	0.009399	0.010206	0.010670	0.010137	0.011543	0.010811	0.046213
5	0.025344	0.012343	0.009682	0.008915	0.009125	0.009867	0.009661	0.011878	0.009865	0.043245
6	0.025693	0.011762	0.009317	0.007954	0.008712	0.009308	0.008532	0.009205	0.009066	0.042134
7	0.025376	0.011391	0.008821	0.008109	0.008055	0.007972	0.007843	0.008088	0.007939	0.039518
8	0.026029	0.012216	0.008822	0.007067	0.007051	0.007374	0.007796	0.007573	0.007233	0.037288

Table 6: ParallelSubgrids runtimes (in seconds) for a 100x100 board.

Threads	Subgrids									
	1	**4**	**16**	**36**	**64**	**100**	**144**	**196**	**256**	**250000**
1	5.287914	5.274163	5.532518	5.365108	5.368514	5.556007	5.346388	5.330809	5.663053	18.613967
2	5.157888	3.463677	3.163879	2.809433	3.101689	3.057095	2.944736	2.877457	2.663584	17.433022
3	5.270334	2.762326	2.099948	2.145811	1.980316	2.001097	1.959838	1.966364	2.014911	17.597134
4	5.403939	1.908469	1.718607	1.574695	1.636697	1.578234	1.492481	1.456364	1.526854	17.840008
5	5.319036	1.925707	1.485665	1.406676	1.410876	1.390942	1.307516	1.310828	1.340352	18.974303
6	5.262596	1.918358	1.395577	1.265433	1.224612	1.217723	1.21675	1.172789	1.235272	21.056172
7	5.358119	1.922741	1.333493	1.188606	1.118327	1.118669	1.091671	1.084218	1.192155	17.953054
8	5.385506	1.926112	1.173612	1.102894	1.061973	1.044060	0.990113	0.992282	1.032097	18.358832

Table 7: ParallelSubgrids runtimes (in seconds) for a 500x500 board.

Threads	Subgrids									
	1	**4**	**16**	**36**	**64**	**100**	**144**	**196**	**256**	**1000000**
1	65.801883	65.945074	66.799887	66.388034	67.485956	67.299456	67.051436	67.883795	67.056246	310.392841
2	64.714528	37.785980	36.057934	35.417008	35.813678	35.300202	35.437499	35.487171	35.627685	257.808374
3	65.622570	34.679536	27.001564	25.326437	25.149757	24.586220	24.508003	24.854055	25.193567	240.548454
4	67.589009	21.303343	20.879299	19.191283	18.958903	18.670745	18.48981	18.772108	18.509092	237.812547
5	66.191874	21.598264	18.539671	16.277188	16.005820	16.318601	16.025915	15.863053	15.930072	231.578546
6	66.546687	21.820648	15.904608	14.798572	14.075356	14.538098	14.106605	13.942253	13.983451	242.619650
7	66.201424	22.373584	15.098906	13.849142	12.851644	13.162786	12.713559	12.491494	12.521412	245.427277
8	67.099902	22.044958	12.315757	12.435074	11.946892	11.677746	11.468487	10.809685	11.501718	244.793032

Table 8: ParallelSubgrids runtimes (in seconds) for a 1000x1000 board.

ParallelSubgrids yields improved results compared to our sequential algorithm across all board sizes. As a sanity check, we test ParallelSubgrids with just 1 subgrid, as that should be equivalent to the sequential algorithm. We also include results for the maximum number of subgrids for each board, which represents running DFS from every single cell in parallel. However, at those settings, runtime is actually slower than the sequential algorithm. At all other settings in between, ParallelSubgrids shows an improvement in runtime compared to the sequential algorithm, with significant improvements being seen as the number of subgrids initially increases, and diminishing returns as the number of subgrids increases further.

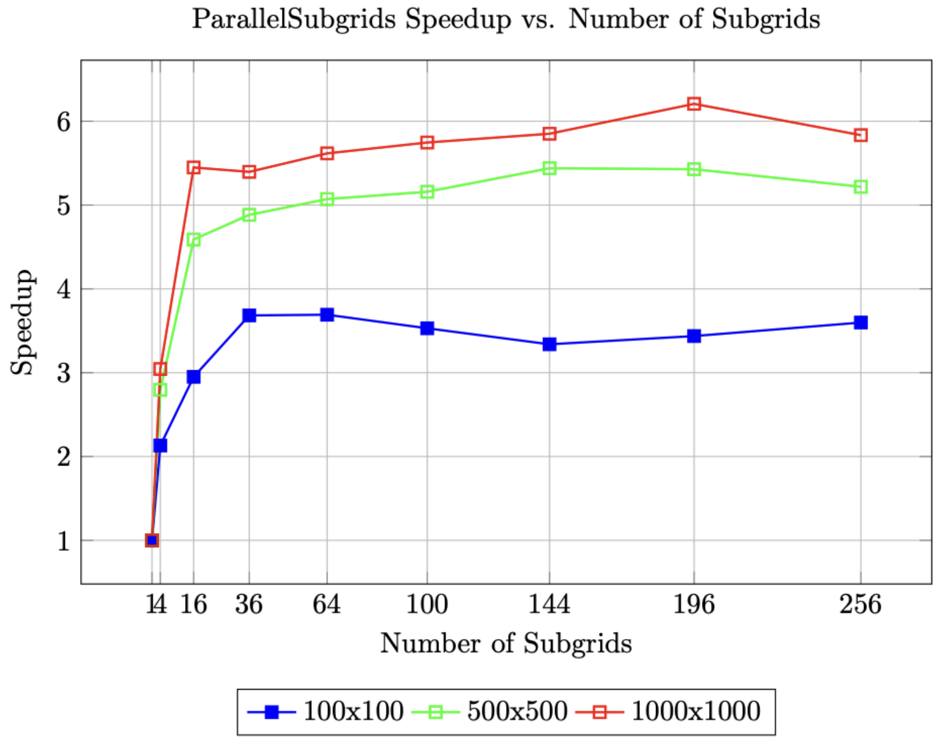


Figure 4: Speedup for varying numbers of subgrids for different board sizes, -N8.

Speedup relative to the number of subgrids shows a sharp increase up until around 16 subgrids and diminishing returns past that. We hypothesize that the speedup doesn't show much significant improvement past 16 subgrids as parallel execution becomes more and more bottlenecked by the number of threads we have available on our benchmark machine (8).

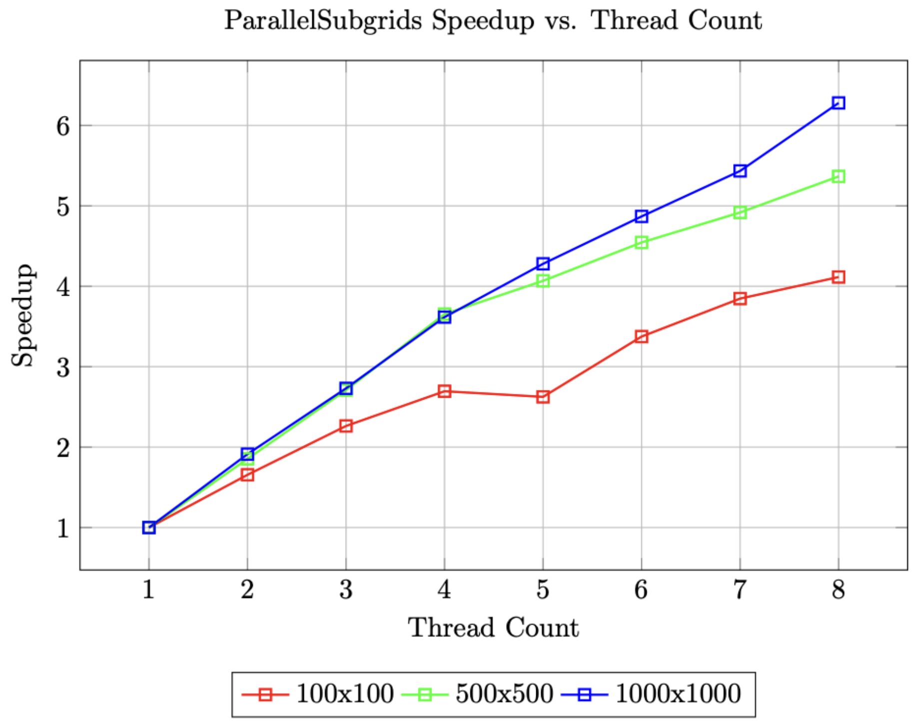
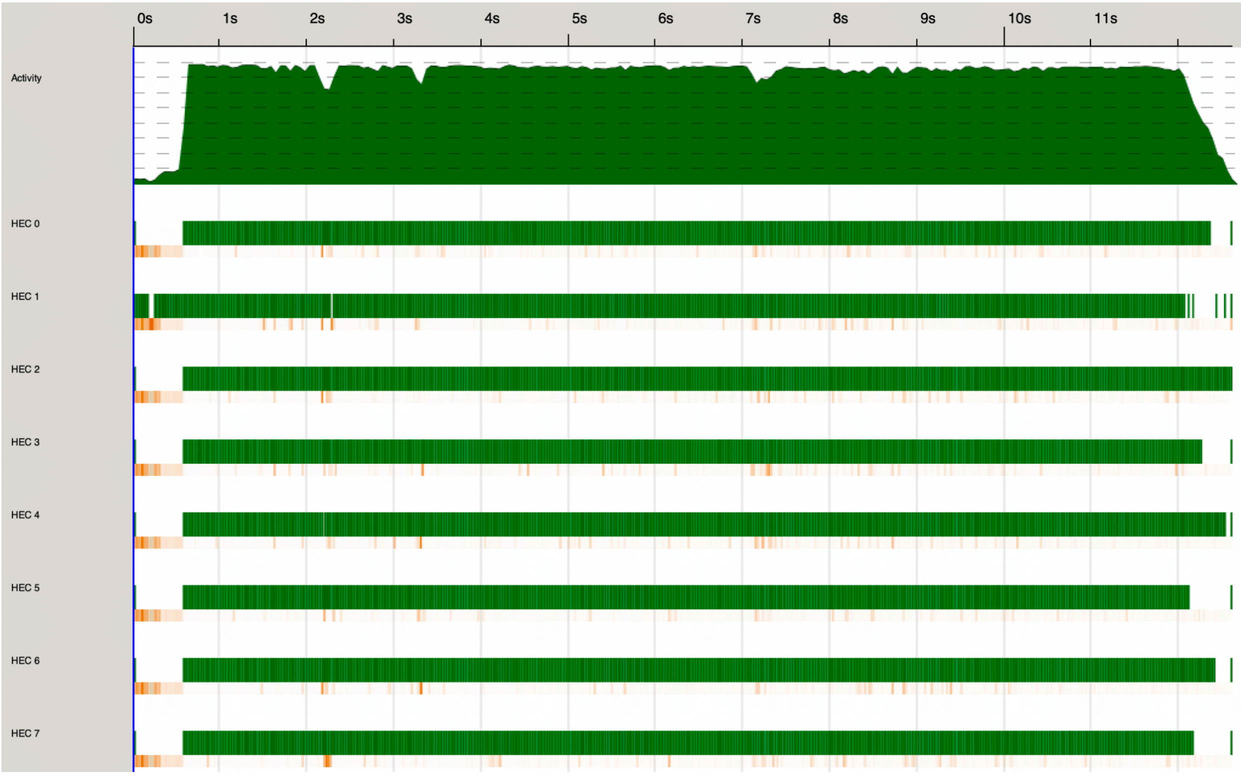


Figure 5: Speedup for varying thread counts for different board sizes, each split into 196 subgrids.

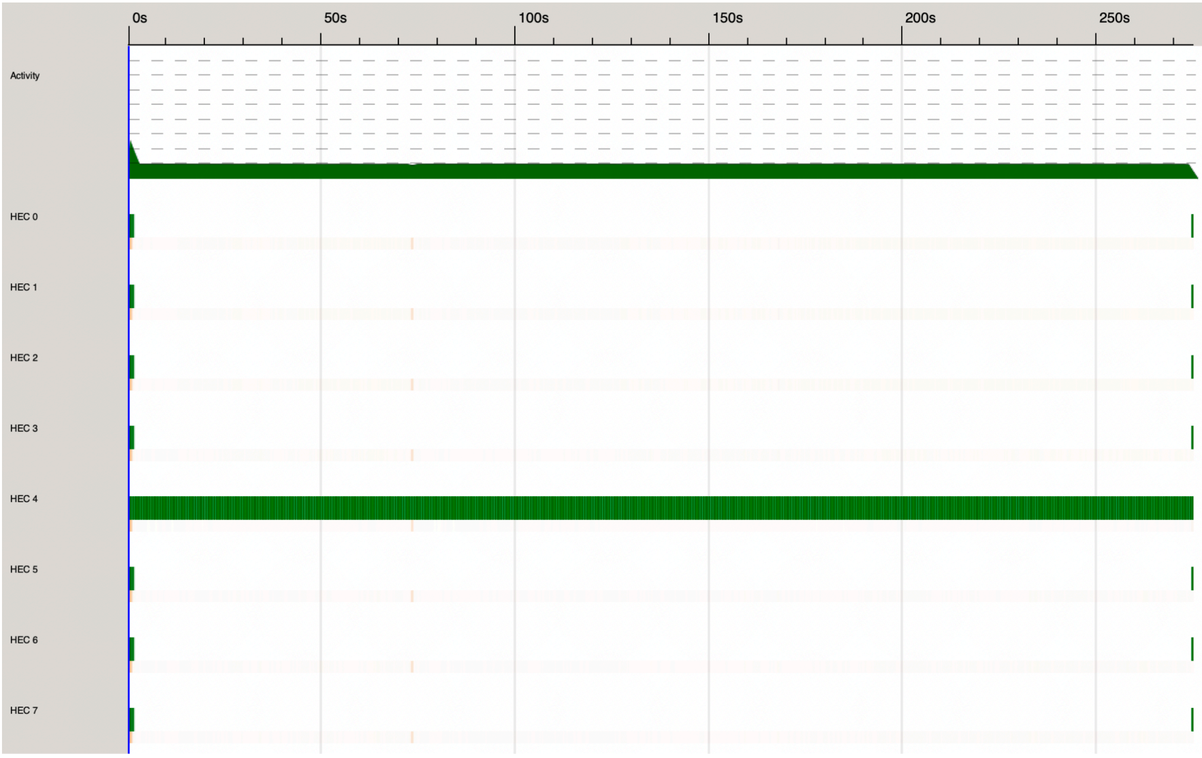
Speedup relative to the thread count increases relatively linearly. It also seems with both subgrid and thread counts, that the ParallelSubgrids method actually scales better with larger board sizes. Smaller boards may have too little work in each subgrid, making the creation of new sparks less efficient due to the overhead costs outweighing the benefits of parallelism.



HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	196	195	0	0	0	1
HEC 0	0	25	0	0	0	0
HEC 1	196	22	0	0	0	0
HEC 2	0	25	0	0	0	1
HEC 3	0	24	0	0	0	0
HEC 4	0	25	0	0	0	0
HEC 5	0	25	0	0	0	0
HEC 6	0	25	0	0	0	0
HEC 7	0	24	0	0	0	0

ParallelSubgrids threadscope graph and spark stats for 1000x1000 board, 196 subgrids, -N8.

Parallel load balancing is quite even up to 196 subgrids, with all but 1 of those 196 sparks being converted.



HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	1000000	8467	704813	0	0	0
HEC 0	0	1206	0	0	0	0
HEC 1	0	1239	0	0	0	0
HEC 2	0	1166	0	0	0	0
HEC 3	0	1169	0	0	0	0
HEC 4	1000000	0	704813	0	0	0
HEC 5	0	1271	0	0	0	0
HEC 6	0	1229	0	0	0	0
HEC 7	0	1187	0	0	0	0

ParallelSubgrids threadscope graph and spark stats for 1000x1000 board, 1 million subgrids, -N8.

The spark stats for 1 million subgrids are much less promising. Far too many sparks are created at once, leading to over 70% of them overflowing.

Conclusion & Future Work:

Overall, the word search 2 problem was a decent candidate for parallelism.
While ParallelWords performed poorly, ParallelDepth and ParallelSubgrids showed significant performance increases over our sequential algorithm.

A few possible directions to explore in future work:

- Test performance on machine with high hardware thread count
- Tune test cases to get more granular performance results of our algorithms given our current hardware setup
- Investigate if there are other algorithms that could be used for more efficient parallelism

Code Listing:

Main.hs:

```
module Main (main) where

import qualified SequentialSearch
import qualified ParallelDepthSearch
import qualified ParallelWordsSearch
import qualified ParallelSubgridSearch
import InputParser ( parseBoard, parseWords )
import InputParser
import System.Environment (getArgs)
import Data.Time.Clock (getCurrentTime, diffUTCTime)
import Control.DeepSeq ( deepseq )
import Control.DeepSeq

main :: IO ()
main = do
    args <- getArgs
    case args of
        [filename, solution] -> processFile filename solution Nothing
        [filename, solution, paramStr] ->
            case reads paramStr :: [(Int, String)] of
                [(n, "")] | n > 0 -> processFile filename solution (Just n)
                _ -> error "Invalid input for number of subgrid / depth: please provide a positive integer."
        _ -> putStrLn "Usage: ./wordsearch <filename> <solution> <optional: number of subgrid / depth>"

processFile :: FilePath -> String -> Maybe Int -> IO ()
processFile filename solution param = do
    contents <- readFile filename
    case lines contents of
        [boardStr, wordsStr] -> do
            let board = parseBoard boardStr
            let wordsList = parseWords wordsStr

            -- Debug output
            putStrLn "Parsed Board:"
            mapM_ print board
            putStrLn "Parsed Words:"
            print wordsList

            if null board || any null board
                then putStrLn "Error: Invalid board format"
                else do
                    -- Time the findWords operation
                    start <- getCurrentTime
                    let results = runSolution solution board wordsList param
                    results `deepseq` return () -- Force evaluation
                    mapM_ putStrLn results
                    end <- getCurrentTime
                    putStrLn $ "Time taken: " ++ show (diffUTCTime end start)
        _ -> putStrLn "Error: Input file must contain exactly two lines"

runSolution :: String -> [[Char]] -> [String] -> Maybe Int -> [String]
runSolution solution board wordsList param =
    case solution of
        "sequential" -> SequentialSearch.findWords board wordsList
        "parallelwords" -> ParallelWordsSearch.findWords board wordsList
        "paralleldepth" ->
            case param of
                Just n -> ParallelDepthSearch.findWords n board wordsList
                Nothing -> error "Missing depth argument for 'paralleldepth' solution."
        "parallelsubgrids" ->
            case param of
                Just n -> ParallelSubgridSearch.findWordsSubgrids n board wordsList
                Nothing -> error "Missing subgrids argument for 'parallelsubgrids' solution."
        _ -> error "Invalid solution argument."
```

InputParser.hs

```haskell
module InputParser (parseBoard, parseWords) where

import Data.Char (isAlpha)

-- Parse a single cell content (e.g., "a" -> 'a')
parseCell :: String -> Char
parseCell s =
    case filter isAlpha s of
        (c:_) -> c
        [] -> error $ "Invalid cell content: " ++ s

-- Parse input string to get board
parseBoard :: String -> [[Char]]
parseBoard input =
    let content = init $ tail input  -- Remove outer brackets
        rows = splitRows content
        parsedRows = map parseRow rows
    in parsedRows
  where
    splitRows :: String -> [String]
    splitRows [] = []
    splitRows s =
        let (row, rest) = break (==']') (dropWhile (/='[') s)
        in if null rest
           then []
           else (tail row) : splitRows (tail rest)

    parseRow :: String -> [Char]
    parseRow s = map parseCell (splitCells s)

    splitCells :: String -> [String]
    splitCells [] = []
    splitCells s =
        let (cell, rest) = break (==',') (dropWhile (not . isAlpha) s)
        in if null cell
           then splitCells rest
           else cell : splitCells rest

-- Parse input string to get words
parseWords :: String -> [String]
parseWords input =
    let content = init $ tail input  -- Remove outer brackets
        wordsList = splitWords content
    in map (filter isAlpha) wordsList
  where
    splitWords :: String -> [String]
    splitWords [] = []
    splitWords s =
        let (word, rest) = break (==',') (dropWhile (not . isAlpha) s)
        in if null word
           then splitWords rest
           else word : splitWords rest
```

SequentialSearch.hs

```
module SequentialSearch (findWords, insertWord, Trie(..), emptyTrie, searchFromCell) where

import qualified Data.Map as Map
import Data.Maybe (fromMaybe)
import Data.List (nub)

-- Trie data structure
data Trie = Trie {
    children :: Map.Map Char Trie,
    isEnd :: Bool
} deriving (Show)

-- Create empty trie
emptyTrie :: Trie
emptyTrie = Trie Map.empty False

-- Insert a word into trie
insertWord :: String -> Trie -> Trie
insertWord "" trie = trie { isEnd = True }
insertWord (c:cs) trie =
    let childTrie = fromMaybe emptyTrie (Map.lookup c (children trie))
        newChild = insertWord cs childTrie
    in trie { children = Map.insert c newChild (children trie) }

-- Main search function
findWords :: [[Char]] -> [String] -> [String]
findWords board targetWords = nub $ concatMap (\(r,c) ->
    searchFromCell board trie r c []
    ) [(r,c) | r <- [0..rows-1], c <- [0..cols-1]]
  where
    rows = length board
    cols = length (head board)
    trie = foldr insertWord emptyTrie targetWords

-- Search from a specific cell
searchFromCell :: [[Char]] -> Trie -> Int -> Int -> String -> [String]
searchFromCell board trie row col currWord
    | row < 0 || row >= rows || col < 0 || col >= cols = []
    | board !! row !! col == '*' = []  -- Check for visited cell
    | not (Map.member curr (children trie)) = []
    | otherwise =
        let newTrie = fromMaybe emptyTrie (Map.lookup curr (children trie))
            newWord = currWord ++ [curr]
            foundWords = [newWord | isEnd newTrie]
            markedBoard = markCell board row col
            nextWords = concatMap (\(dr,dc) ->
                searchFromCell markedBoard newTrie (row+dr) (col+dc) newWord
                ) [(0,1), (1,0), (0,-1), (-1,0)]
        in foundWords ++ nextWords
  where
    rows = length board
    cols = length (head board)
    curr = board !! row !! col

-- Mark/unmark cell (replace with temporary character)
markCell :: [[Char]] -> Int -> Int -> [[Char]]
markCell board row col =
    take row board ++
    [take col (board !! row) ++ ['*'] ++ drop (col+1) (board !! row)] ++
    drop (row+1) board
```

ParallelWordsSearch.hs

```haskell
module ParallelWordsSearch (findWords, searchSingleWord, searchFromCell) where

import qualified Data.Map as Map
import qualified Data.Set as Set
import Data.Maybe (fromMaybe)
import Data.List (nub)
import Control.Parallel.Strategies ( parMap, rdeepseq )
import Control.Parallel.Strategies

data Trie = Trie {
    children :: Map.Map Char Trie,
    isEnd :: Bool
} deriving (Show)

emptyTrie :: Trie
emptyTrie = Trie Map.empty False

insertWord :: String -> Trie -> Trie
insertWord "" trie = trie { isEnd = True }
insertWord (c:cs) trie =
    let childTrie = fromMaybe emptyTrie (Map.lookup c (children trie))
        newChild = insertWord cs childTrie
    in trie { children = Map.insert c newChild (children trie) }

-- Type alias for position
type Pos = (Int, Int)

findWords :: [[Char]] -> [String] -> [String]
findWords board targetWords =
    let trie = foldr insertWord emptyTrie targetWords
        -- Use parMap with rdeepseq to force full evaluation in parallel
        results = parMap rdeepseq (searchSingleWord board trie) targetWords
    in nub (concat results)

searchSingleWord :: [[Char]] -> Trie -> String -> [String]
searchSingleWord board trie word =
    searchUntilFound Set.empty [(r,c) | r <- [0..rows-1], c <- [0..cols-1]]
  where
    rows = length board
    cols = length (head board)

    searchUntilFound _ [] = []
    searchUntilFound visited ((r,c):rest) =
        case searchFromCell board trie r c [] word Set.empty of
            [] -> searchUntilFound visited rest
            found -> found

-- Modified to use Set for visited positions
searchFromCell :: [[Char]] -> Trie -> Int -> Int -> String -> String -> Set.Set Pos -> [String]
searchFromCell board trie row col currWord targetWord visited
    | row < 0 || row >= rows || col < 0 || col >= cols = []
    | Set.member (row, col) visited = []
    | not (Map.member curr (children trie)) = []
    | otherwise =
        let newTrie = fromMaybe emptyTrie (Map.lookup curr (children trie))
            newWord = currWord ++ [curr]
            foundWords = [newWord | isEnd newTrie && newWord == targetWord]
            newVisited = Set.insert (row, col) visited
            nextWords = concatMap (\(dr,dc) ->
                searchFromCell board newTrie (row+dr) (col+dc) newWord targetWord newVisited
                ) [(0,1), (1,0), (0,-1), (-1,0)]
        in foundWords ++ nextWords
  where
    rows = length board
    cols = length (head board)
    curr = board !! row !! col
```

ParallelDepthSearch.hs

```haskell
module ParallelDepthSearch (findWords, insertWord, Trie(..), emptyTrie, searchFromCell) where

import qualified Data.Map as Map
import Data.Maybe (fromMaybe)
import Data.List (nub)
import Data.Set (Set)
import qualified Data.Set as Set
import Control.Parallel.Strategies (parMap, rseq)


data Trie = Trie {
    children :: Map.Map Char Trie,
    isEnd :: Bool
} deriving (Show)


emptyTrie :: Trie
emptyTrie = Trie Map.empty False

insertWord :: String -> Trie -> Trie
insertWord "" trie = trie { isEnd = True }
insertWord (c:cs) trie =
    let childTrie = fromMaybe emptyTrie (Map.lookup c (children trie))
        newChild = insertWord cs childTrie
    in trie { children = Map.insert c newChild (children trie) }

findWords :: Int -> [[Char]] -> [String] -> [String]
findWords depth board targetWords =
    nub $ concat $ parMap rseq (\(r, c) -> searchFromCell board trie Set.empty r c [] depth 0 rows cols)
    [(r, c) | r <- [0..rows-1], c <- [0..cols-1]]
  where
    rows = length board
    cols = length (head board)
    trie = foldr insertWord emptyTrie targetWords

searchFromCell :: [[Char]] -> Trie -> Set (Int, Int) -> Int -> Int -> String -> Int -> Int -> Int -> Int -> [String]
searchFromCell board trie visited row col currWord depth level rows cols
    | row < 0 || row >= rows || col < 0 || col >= cols = []
    | Set.member (row, col) visited = []
    | not (Map.member curr (children trie)) = []
    | otherwise =
        let newTrie = fromMaybe emptyTrie (Map.lookup curr (children trie))
            newWord = currWord ++ [curr]
            foundWords = [newWord | isEnd newTrie]
            newVisited = Set.insert (row, col) visited

            nextWords = if level < depth
                        then concat $ parMap rseq (\(r, c) -> searchFromCell board newTrie newVisited r c newWord depth (level + 1) rows cols)
                                    [(row+1, col), (row, col+1), (row-1, col), (row, col-1)]
                        else concatMap (\(r, c) -> searchFromCell board newTrie newVisited r c newWord depth (level + 1) rows cols)
                                    [(row+1, col), (row, col+1), (row-1, col), (row, col-1)]
        in foundWords ++ nextWords
  where
    curr = board !! row !! col
```

ParallelSubgridSearch.hs

```haskell
module ParallelSubgridSearch (findWordsSubgrids) where

import Data.List (nub)
import Control.Parallel.Strategies ( parMap, rdeepseq )
import Control.Parallel.Strategies
import qualified SequentialSearch

findWordsSubgrids :: Int -> [[Char]] -> [String] -> [String]
findWordsSubgrids splits board wordsList =
    let subBoards = splitBoard splits board
        trie = foldr SequentialSearch.insertWord SequentialSearch.emptyTrie wordsList
        results = parMap rdeepseq (\subBoard -> findWordsTrie board trie subBoard) subBoards
    in nub (concat results)

findWordsTrie :: [[Char]] -> SequentialSearch.Trie -> (Int, Int, Int, Int)-> [String]
findWordsTrie board trie (rStart, rEnd, cStart, cEnd) =
    nub $ concatMap (\(r,c) ->
        SequentialSearch.searchFromCell board trie r c []
    ) [(r,c) | r <- [rStart..min rEnd (rows-1)], c <- [cStart..min cEnd (cols-1)]]
  where
    rows = length board
    cols = length (head board)

splitBoard :: Int -> [[Char]] -> [(Int, Int, Int, Int)]
splitBoard n board
    | n <= 1 = [(0, rows, 0, cols)]
    | n >= rows = [(r, r + 1, c, c + 1) | r <- [0..rows-1], c <- [0..min rows cols-1]]
    | n >= cols = [(r, r + 1, c, c + 1) | r <- [0..min rows cols-1], c <- [0..cols-1]]
    | otherwise =
        [(r * subRows, (r + 1) * subRows, c * subCols, (c + 1) * subCols)
         | r <- [0..n-1], c <- [0..n-1]]
  where
    rows = length board
    cols = length (head board)
    subRows = rows `div` n
    subCols = cols `div` n
```