Partially Filled Magic Squares Solver

Zijian Zhang (zz2795), Cynthia Zhang (cz2696)

Introduction

A magic square is composed of a square array of numbers, usually positive integers, in which the sums of the numbers in each row, each column, and both main diagonals are the same. The order of the magic square n is the size of the sides, and the magic constant M of a magic square is the constant sum. If the square only contains the positive integer $1,2,\dots,n^2$, the magic square is called normal, and its magic constant is $M = n * \frac{n^2+1}{2}$. Generating magic squares with $n > 5$ is still an open challenge.

This magic square solver could solve normal partially filled magic squares with a size smaller than 5 within a reasonable time. It would produce all possible answers using a brute force algorithm. To measure the performance of our parallelization, we used a magic square of size 4 with only 1 number filled. We attempted several parallelization approaches, and the optimal performance of is 18.618s using 4 threads. For simplicity, we will only display the 4 threads' performance results of different approaches, and the rest of the data would be attached at the end.

Solving strategy

The solver uses a brute force strategy of filling in possible numbers one by one while checking for their validity. If the existing numbers don't sum to or exceed the magic constant, the solver would terminate the current branch early.

There are other algorithms for generating magic squares. We decided to use the brute force approach because the other algorithms either only apply to a specific type of magic square, or are difficult to be tailored to a partially filled magic square due to various constraints. For example, one useful magic square property we considered using is that any magic square can be rotated and reflected to produce 8 trivially distinct squares. Therefore, if we find one valid magic square, we find 8 trivially distinct squares. However, for a partially filled square, the existing numbers are in fixed positions and the other trivially distinct squares are not valid answers.

Testing

$$\begin{pmatrix} -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & 15 \end{pmatrix}$$

The above 4 by 4 matrix represents a magic square with 1 number filled. -1 represents an empty position.

We used the above matrix to measure the performance of our implementation. Because this matrix has many empty positions, it has many possible solutions and a high difficulty level.

This means that parallelization overhead will not dominate over the actual calculation time, and the performance differences between different approaches will be more obvious.

Sequential Implementation

1. Choose a blank position that appears first in the rows and columns.
2. Find all the numbers that can be placed in this position without making the magic square invalid.
    a. Since we are handling normal squares, all numbers between 1 and n^2 will only appear once. The numbers that can be filled in are simply the ones that have not existed in the square.
    b. For each number, check whether it will invalidate the magic square if it is put into the black position. If the sum of partially filled rows, cols, or diagonals exceeds the magic number, or if the sum of filled rows, cols, or diagonals does not equal the magic number, the square becomes invalid.
3. For each potential number, place it into the square, then repeat step 1 to find the next blank position.
    a. Check if the square is filled, add it to the result list.

This sequential implementation uses 36.614 seconds, which is the performance benchmark of our parallel implementations.

Parallel Implementation

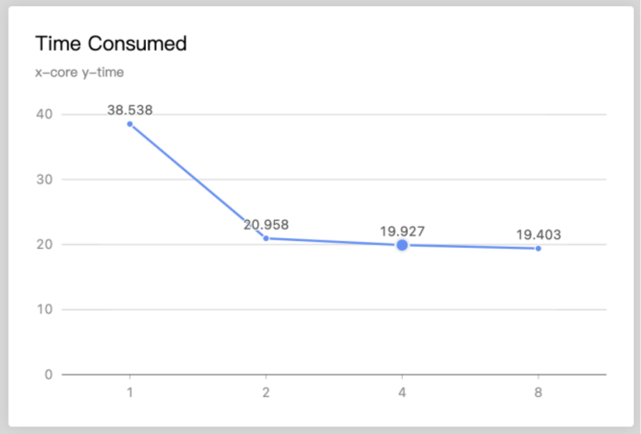
Attempt 1:

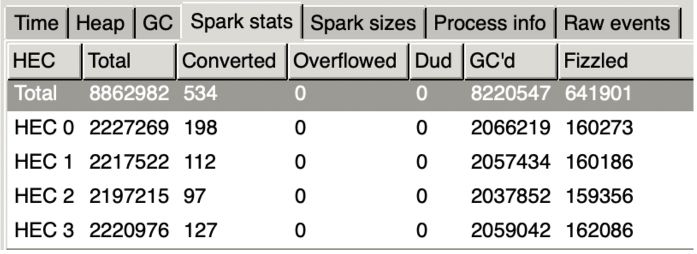
We first try using parMap to parallelize each potential number for each blank position, and use rdeepseq to ensure that each spark will calculate the final result. For example, given a blank position and a list of potential numbers [1,2,4,5], there will be four parallel sparks for four magics squares, each filled a different potential number in the blank position. Then, each spark will continue with step 1 to find the next blank position, and start parallel sparks for its own potential numbers to fill in the next blank position. Overall, the solver would create sparks following a tree structure, where for each blank position, a node would branch out and create sub sparks with one more position filled.

This approach creates too many unnecessary sparks because one spark is created for every possible number at every blank position with a light workload. After each spark is created, it will find the next potential numbers and create sparks for each of them, then simply wait until all of its sub sparks are finished. The experiment results aligned with our prediction that the solver creates 8862982 sparks.

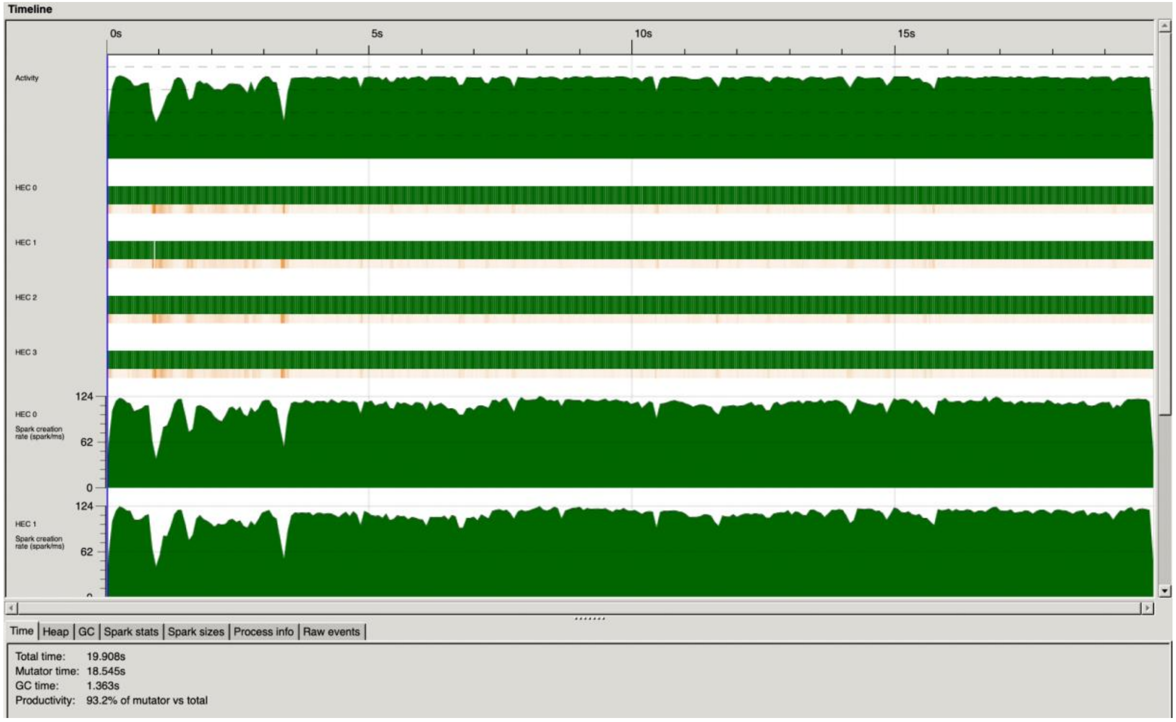
```
parMap :: NFData b => (a -> b) -> [a] -> [b]
parMap f xs = L.map f xs `using` parList rdeepseq

solverHelper :: Int -> (Square, S.Set Int) -> [Maybe Square]
solverHelper parLayer state
```

```
| squareFilled s = [Just s >>= (\x -> guard (validator x) >> return x)]
| otherwise = concat $ parMap (solverHelper (parLayer-1)) next
where (s, choiceS) = state
    (x, y) = findEmptyPos s 0
    choiceL = S.toList choiceS
    next = nextSteps state x y choiceL
```





HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	8862982	534	0	0	8220547	641901
HEC 0	2227269	198	0	0	2066219	160273
HEC 1	2217522	112	0	0	2057434	160186
HEC 2	2197215	97	0	0	2037852	159356
HEC 3	2220976	127	0	0	2059042	162086



```
Total time:     19.908s
Mutator time:   18.545s
GC time:        1.363s
Productivity:   93.2% of mutator vs total
```

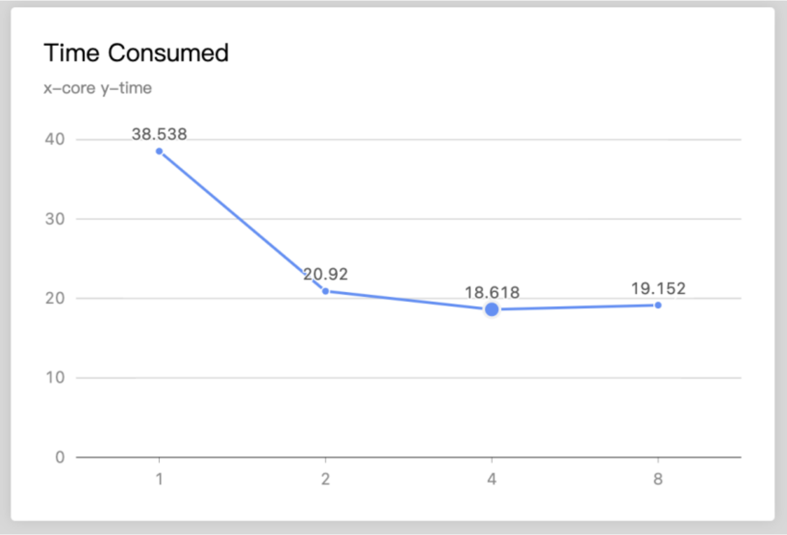
Attempt 2:

We improved upon attempt 1 by limiting the number of sparks the solver will create. Let n be the size of the magic square. The solver now will only produce parallel sparks for the first n blank positions. The code uses an integer parLayer to keep track of the positions by decrementing for every blank position processed. After that, when parLayer becomes 0, each spark will run sequentially to fill in the rest of the blank positions. Going back to the tree structure analogy, this approach would only branch out n times, creating a tree with a max height of n.

One issue with this attempt is that when there are fewer correct answers, it's possible for the majority of the branches to get invalidated and terminated early, resulting in very few ending sparks running parallelly. To resolve this problem, attempt 3 try to parallelize the internal calculation of each spark. Nevertheless, we do not think this is a huge issue for performance even if the situation occurs. For smaller magic squares, the workload of the internal calculations is light, so a few sparks can still do the task quickly. This issue is more obvious for larger magic squares. However, one reason that generating larger magic squares remains an open challenge is because of the enormous amount of valid magic squares. Thus, it's very unlikely to have only a few correct answers.

As the data showed, the solver now only creates 21752 sparks, which is significantly less than the sparks created in attempt 1. Its runtime is slightly better compared to attempt 1.

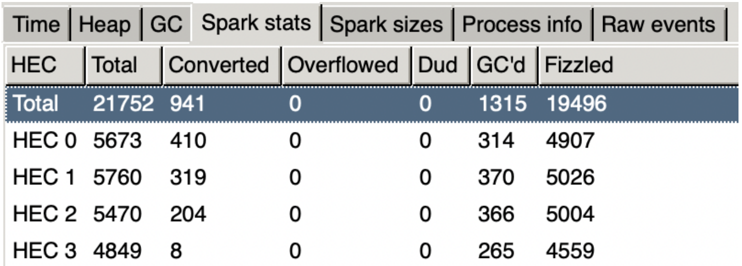
```
solverHelper :: Int -> (Square, S.Set Int) -> [Maybe Square]
solverHelper parLayer state
  | squareFilled s = [Just s >>= (\x -> guard (validator x) >> return x)]
  | parLayer > 0 = concat $ parMap (solverHelper (parLayer-1)) next
  | otherwise = concat $ L.map (solverHelper parLayer) next
  where (s, choiceS) = state
    (x, y) = findEmptyPos s 0
    choiceL = S.toList choiceS
    next = nextSteps state x y choiceL
```



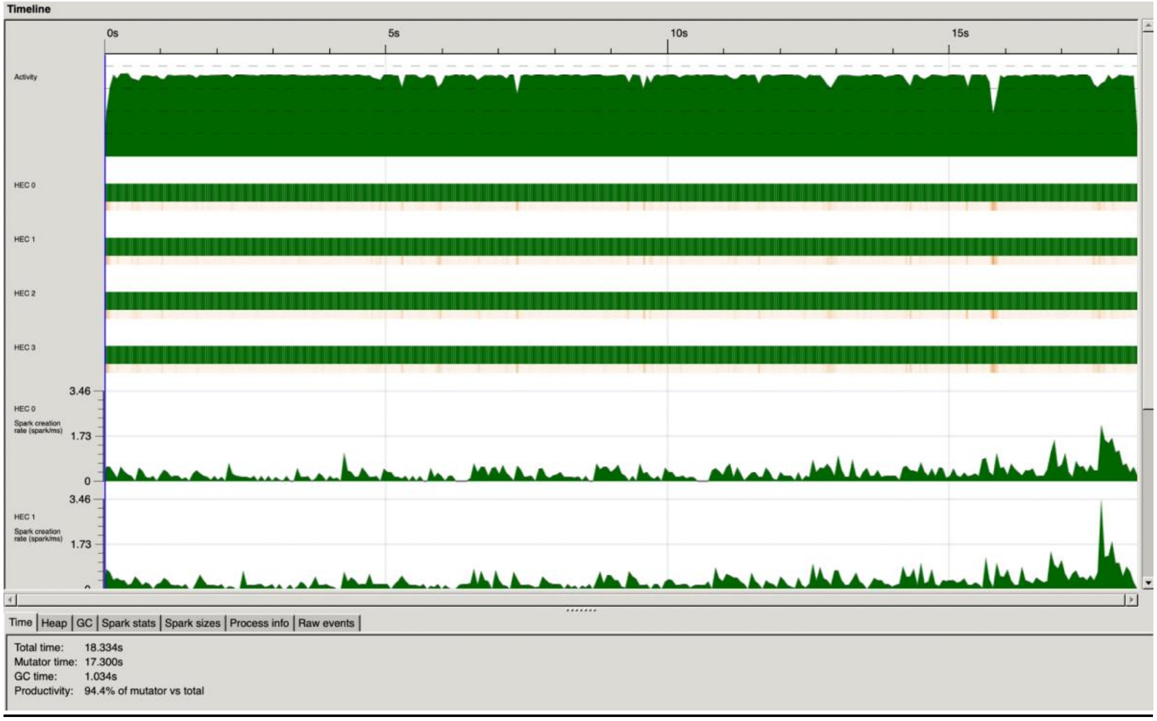
Even though we used rdeepset, there's a fair amount of sparks ended up fizzled, which means that they evaluated to WHNF after creation. One possible cause is that the tasks needed for a spark are not finished when the spark is created, so it evaluates to WHNF temporarily to wait for the previous tasks.

This may happen when a spark in the solver creates parallel sparks for its potential numbers in the next blank position, but has not finished finding the numbers. Thus, the newly created sparks will become fizzled.

This also may happen after a spark finished finding all its potential numbers and has created sub-sparks to calculate them. Since we used rdeepseq to require an NF result, the spark could become fizzled when it is waiting for the sub sparks to finish their calculations. Attempt 4 tries to explore this possibility.



HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	21752	941	0	0	1315	19496
HEC 0	5673	410	0	0	314	4907
HEC 1	5760	319	0	0	370	5026
HEC 2	5470	204	0	0	366	5004
HEC 3	4849	8	0	0	265	4559



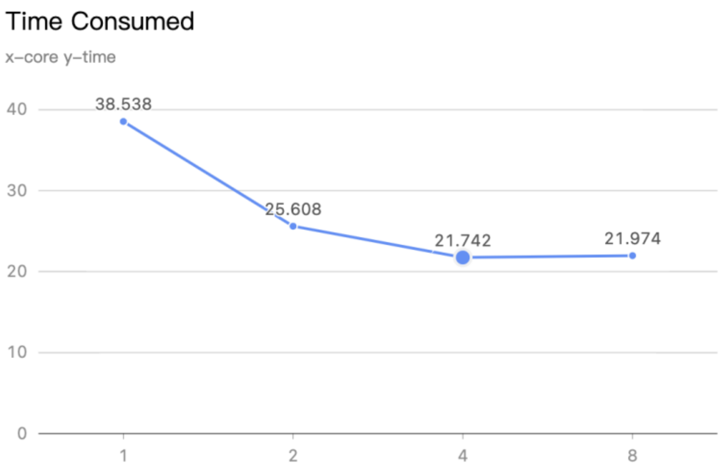
```
Total time:     18.334s
Mutator time:   17.300s
GC time:        1.034s
Productivity:   94.4% of mutator vs total
```

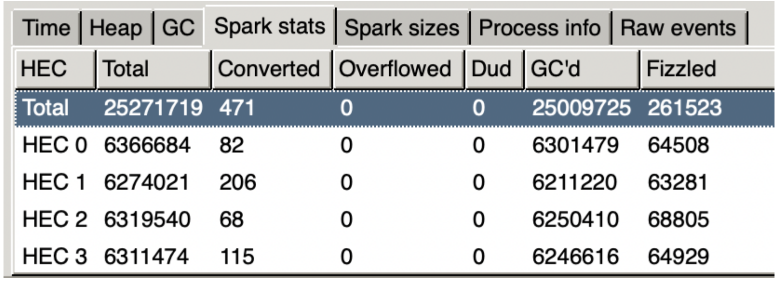
Attempt 3:

For a parallel spark in attempt 2, its main task is to find the next blank position and the potential numbers to fill in. Then, it will create parallel sparks for each potential number and wait for them to calculate the rest.

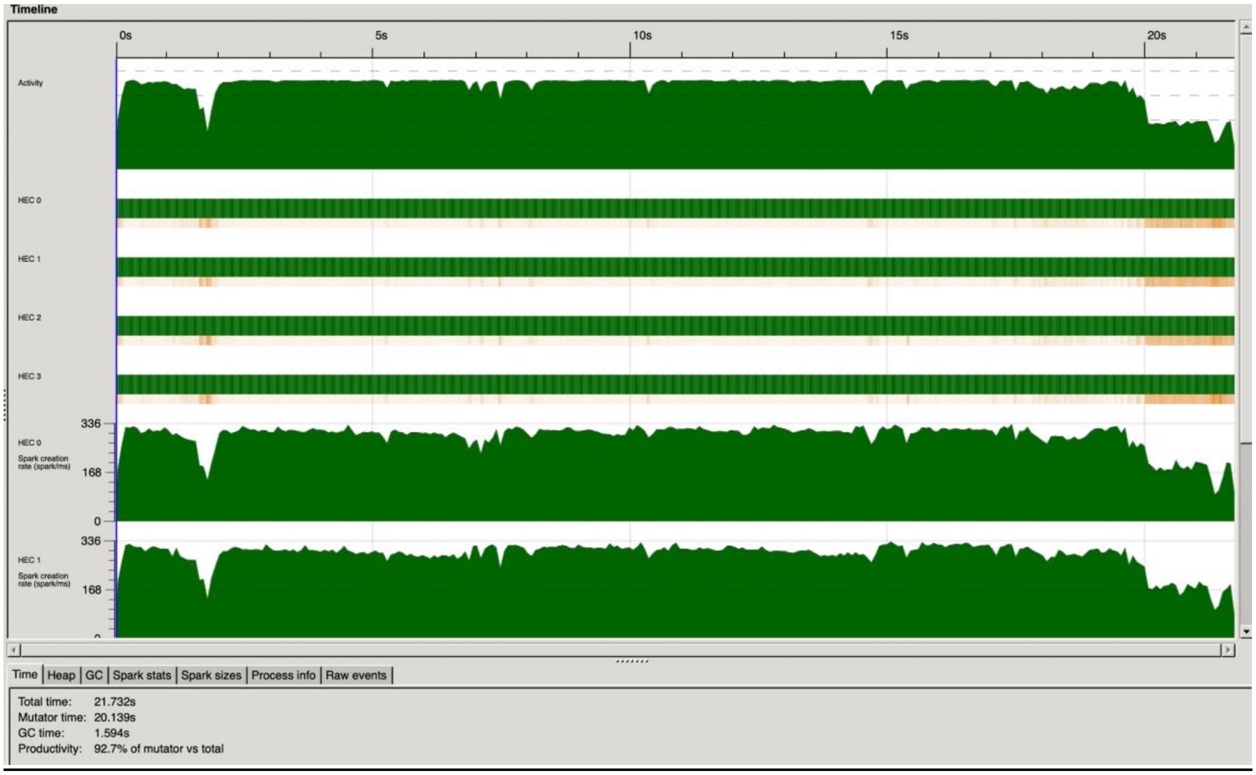
We added parallelization to the function that finds the potential next numbers, attempting to further reduce runtime. However, it runs slower compared to our previous implementations and produced much more sparks. We think it is possible that the parallelization overhead dominates over the performance benefits. Since the solver only handles squares with sizes smaller than or equal to 4, there will be at most 16 potential numbers at each step, which is not big enough for it to benefit from parallelization. If the solver handles bigger magic squares, this parallelization could potentially be helpful since there will be more potential numbers.

```
splitHelper :: [Int] -> Int -> [[Int]]
splitHelper l s
    | n > 0 = cur : splitHelper remain s
    | otherwise = []
    where n = length cur
        cur = take s l
        remain = drop s l

splitList :: [Int] -> [[Int]]
splitList l
    | n < 4 = [l]
    | otherwise = splitHelper l size
    where n = length l
        x = round (sqrt (fromIntegral n))
        size = n `div` x

solverHelper :: Int -> (Square, S.Set Int) -> [Maybe Square]
solverHelper parLayer state
   ... (same as before)
   choiceLL = splitList choiceL
   next = concat $ parMap (nextSteps state x y) choiceLL
```





HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	25271719	471	0	0	25009725	261523
HEC 0	6366684	82	0	0	6301479	64508
HEC 1	6274021	206	0	0	6211220	63281
HEC 2	6319540	68	0	0	6250410	68805
HEC 3	6311474	115	0	0	6246616	64929

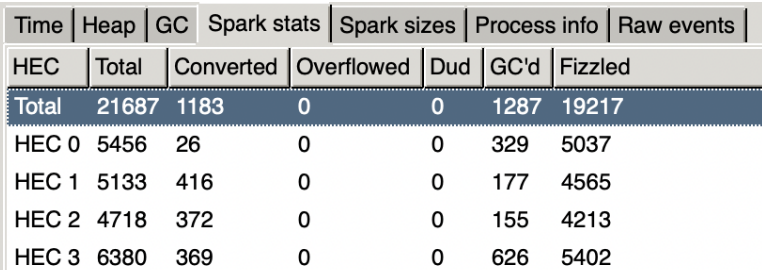


```
Total time:     21.732s
Mutator time:   20.139s
GC time:        1.594s
Productivity:   92.7% of mutator vs total
```

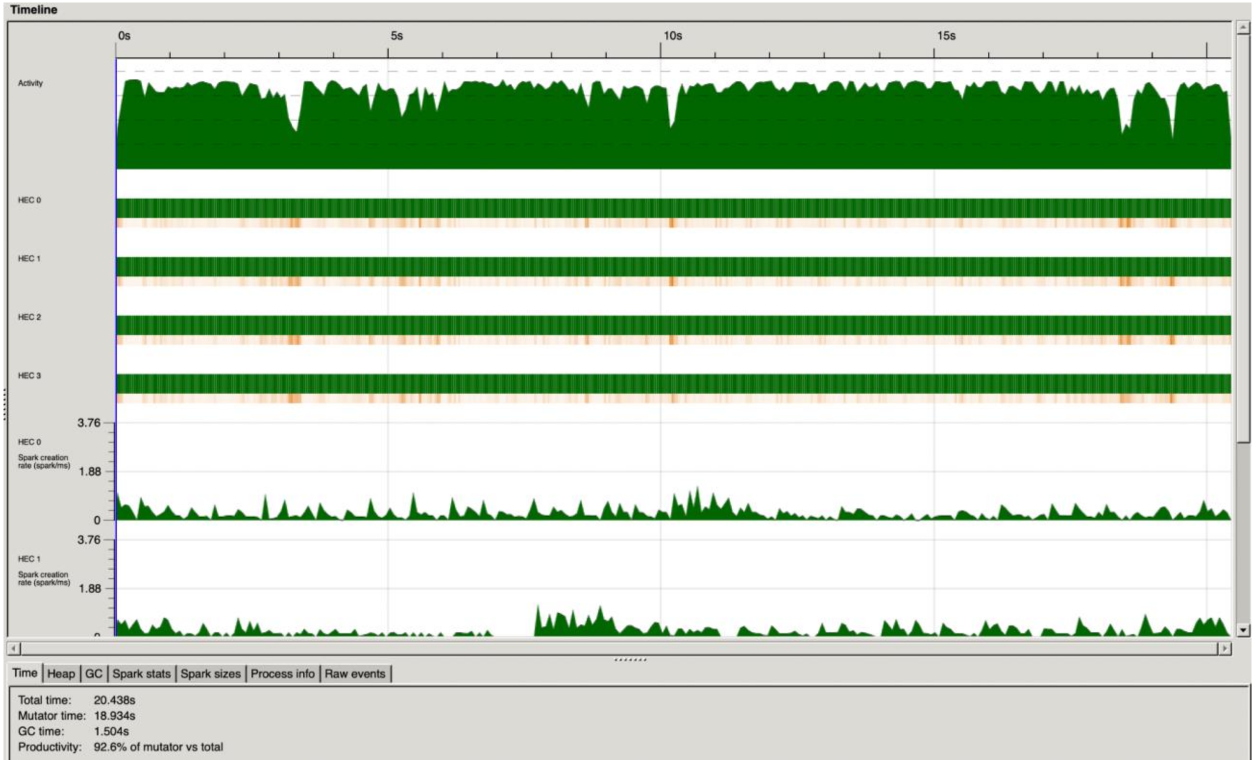
Attempt 4:

Attempting to decrease the number of sparks fizzled, instead of using rdeepseq for all the sparks, only the sparks that compute sequentially will use rdeepseq, in another word, the sparks with parLayer = 1. The sparks with parLayer > 1 will create sub-sparks for each of its potential next numbers. In this attempt, we use rseq to evaluate the sparks with parLayer > 1 to WHNF, so they won't need to wait until their sub-sparks are done.

However, this attempt does not show improvement in performance time or the number of fizzled sparks. It does decrease the number of fizzled sparks by 279, which could mean that our earlier analysis is correct. Nevertheless, we concluded that most fizzled sparks are the result of program synchronization.

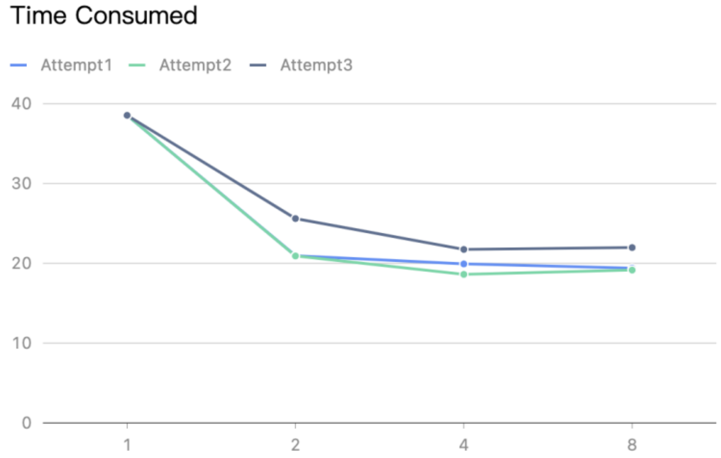


HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	21687	1183	0	0	1287	19217
HEC 0	5456	26	0	0	329	5037
HEC 1	5133	416	0	0	177	4565
HEC 2	4718	372	0	0	155	4213
HEC 3	6380	369	0	0	626	5402



```
Total time:     20.438s
Mutator time:   18.934s
GC time:        1.504s
Productivity:   92.6% of mutator vs total
```

Conclusion



Attempt 2 with 4 threads gives the optimal performance. The parallelization is able to decrease the runtime of the sequential solver by half, with a 2.2x speedup, completing the testing matrix using 18.618s, as opposed to 36.614s originally.

Speedup will varie for different matrices, because it depends on how many blank position and potential numbers the solver need to try, and also how many branches get terminated early.

For every attempts, the workload is distributed evenly among the threads, and all threads are untilized almost fully. This behavior is as expected because the tasks of each sparks are relatively independent and uniformly distributed. Since our computer only have four threads, it's expected that the thread untilization of each attempts with 8 threads becomes more sporadic, and we only considered the performance result up to 4 threads.

Future Improvements

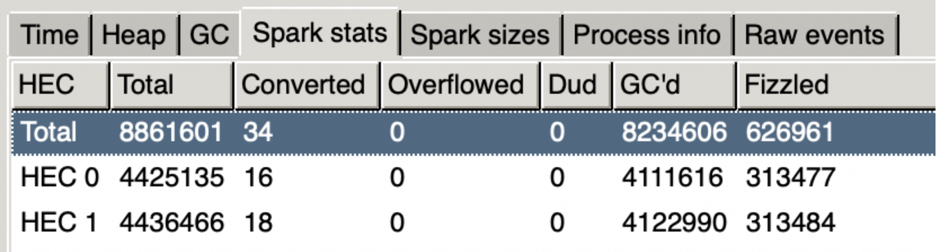
There exist faster sequential algorithms that potentially could lead to a better parallelization speedup. For example, instead of finding potential numbers for each blank position, the solver could generate and test each potential rows that satisfy the magic square requirements, which would eliminate false rows and reduce duplicates that get processed. The parallelization steps would be the same as the brute force algorithm. The number of potential rows are significantly less than potential numbers in each blank positions, so the solver will create less sparks. However, the workload for each spark to generate all potential rows are heavier. The overall performance may or may not improve.

In terms of parallelization, one potential improvement would be having two processes, one starts with the empty position from the beginning, and the other starts from the end. Nevertheless, the complication is how to merge two processes when they reach the middle point. Also, the Repa module is another potential tool to try.

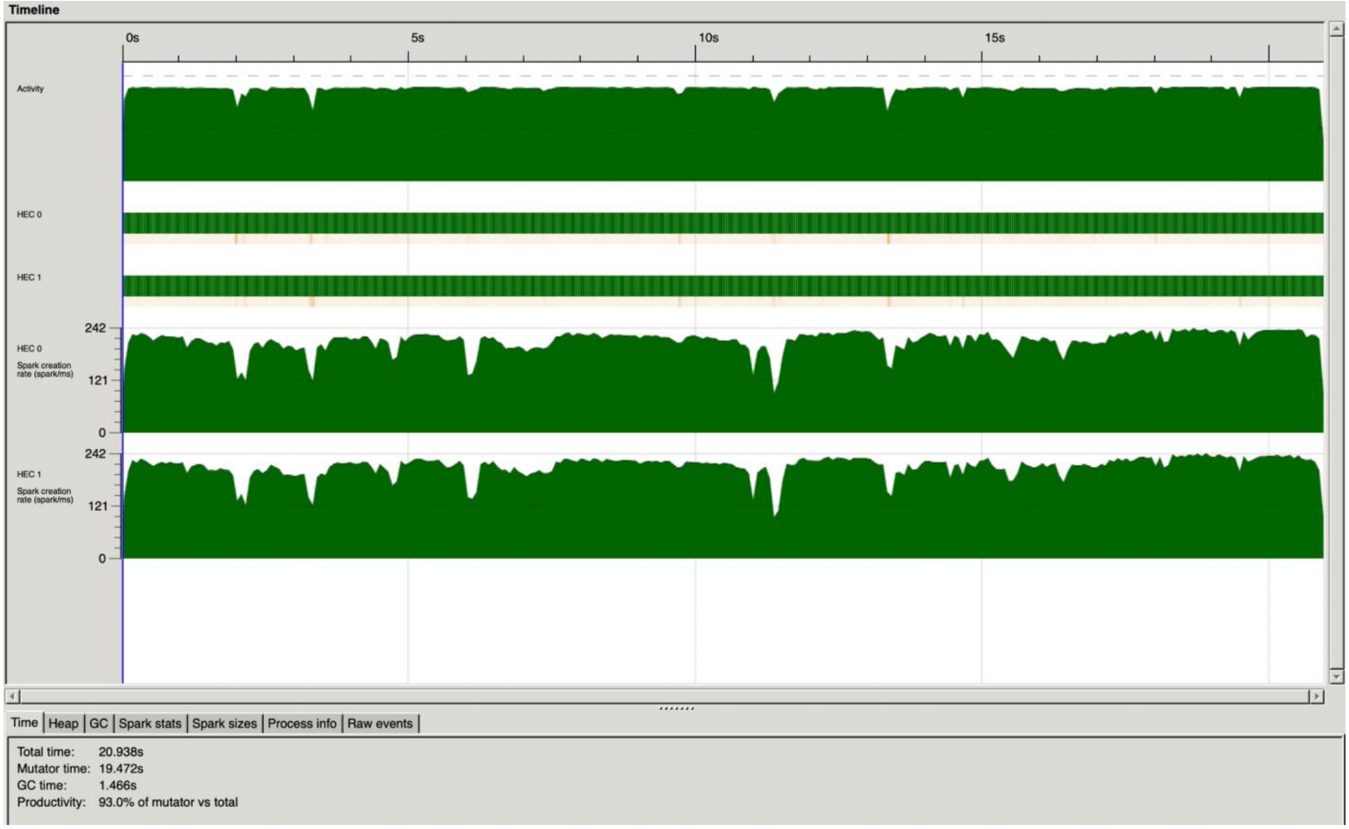
Having more physical threads is another way to improve the performance. Since the tasks of each thread are relatively independent, in theory, the parallel solver would speedup as the number of physical threads increase.

Additional Data:

Attempt 1 - 2 threads:

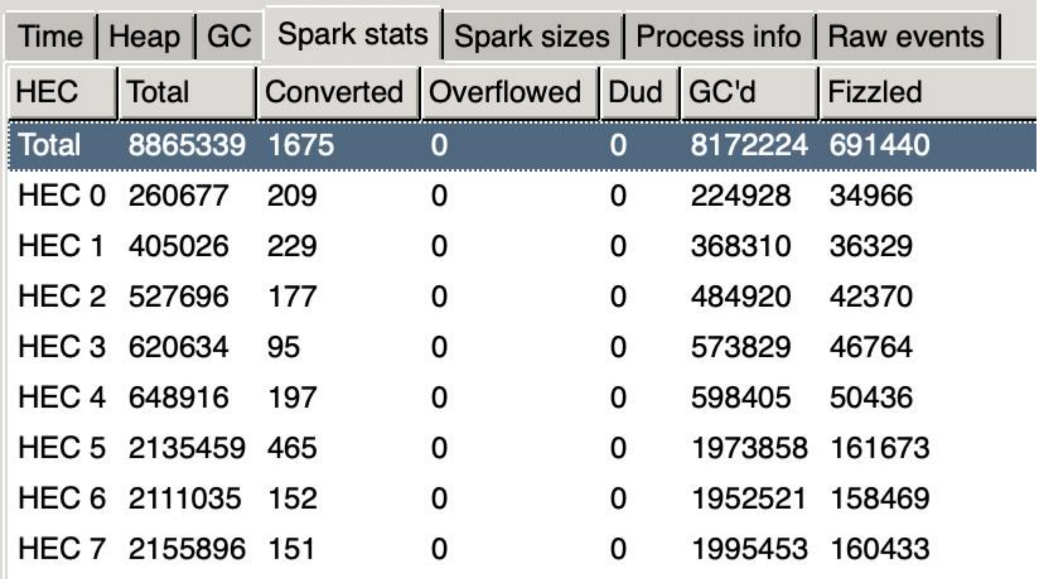


HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	8861601	34	0	0	8234606	626961
HEC 0	4425135	16	0	0	4111616	313477
HEC 1	4436466	18	0	0	4122990	313484

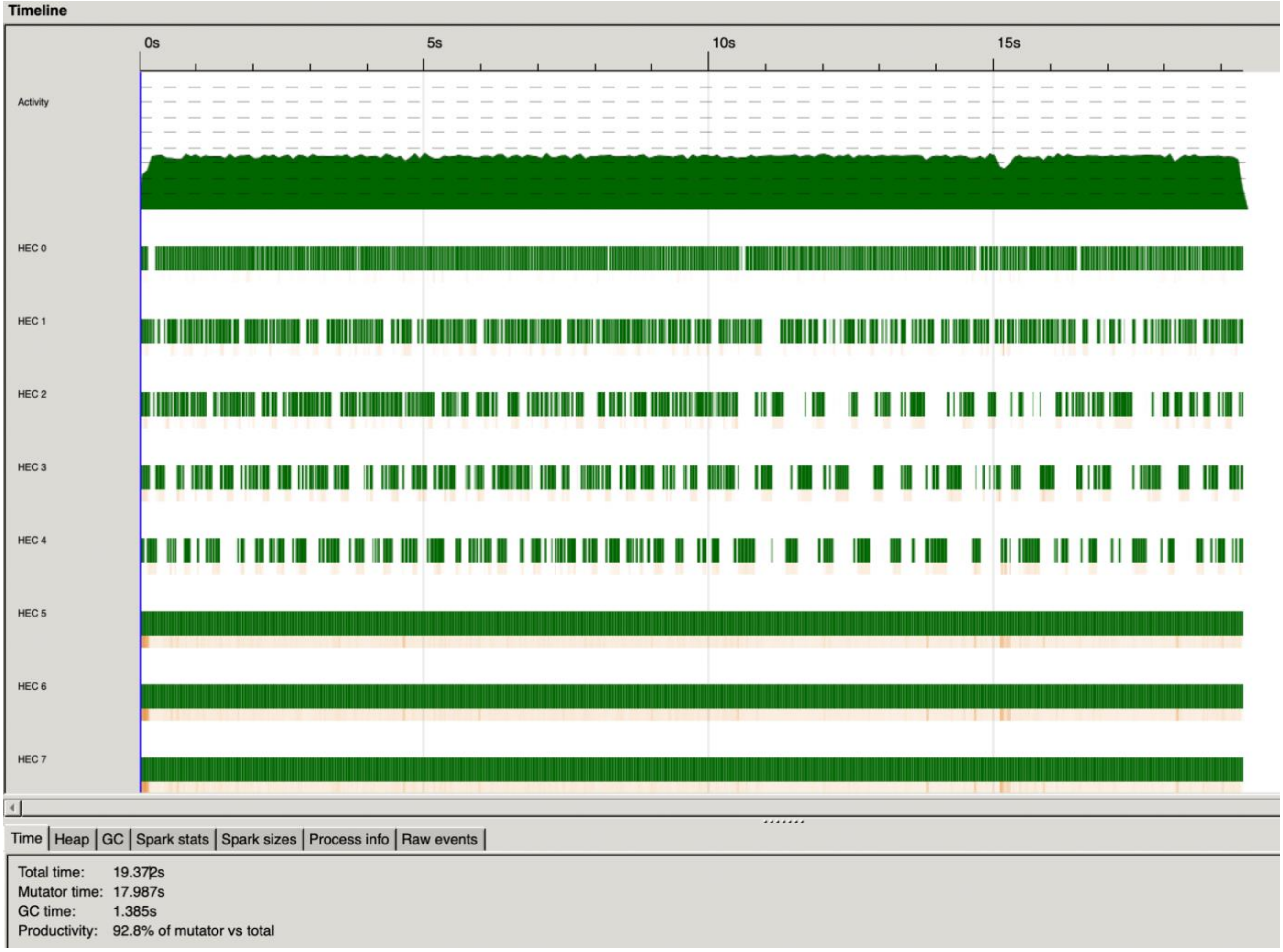


```
Total time:     20.938s
Mutator time:   19.472s
GC time:        1.466s
Productivity:   93.0% of mutator vs total
```

Attempt 1 - 8 threads:

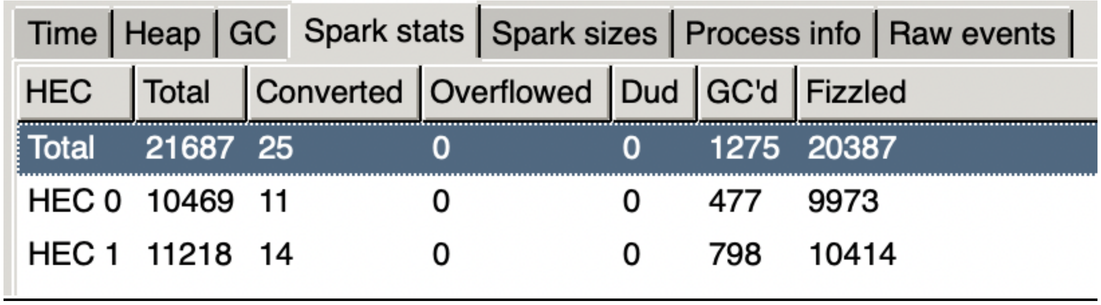


HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	8865339	1675	0	0	8172224	691440
HEC 0	260677	209	0	0	224928	34966
HEC 1	405026	229	0	0	368310	36329
HEC 2	527696	177	0	0	484920	42370
HEC 3	620634	95	0	0	573829	46764
HEC 4	648916	197	0	0	598405	50436
HEC 5	2135459	465	0	0	1973858	161673
HEC 6	2111035	152	0	0	1952521	158469
HEC 7	2155896	151	0	0	1995453	160433

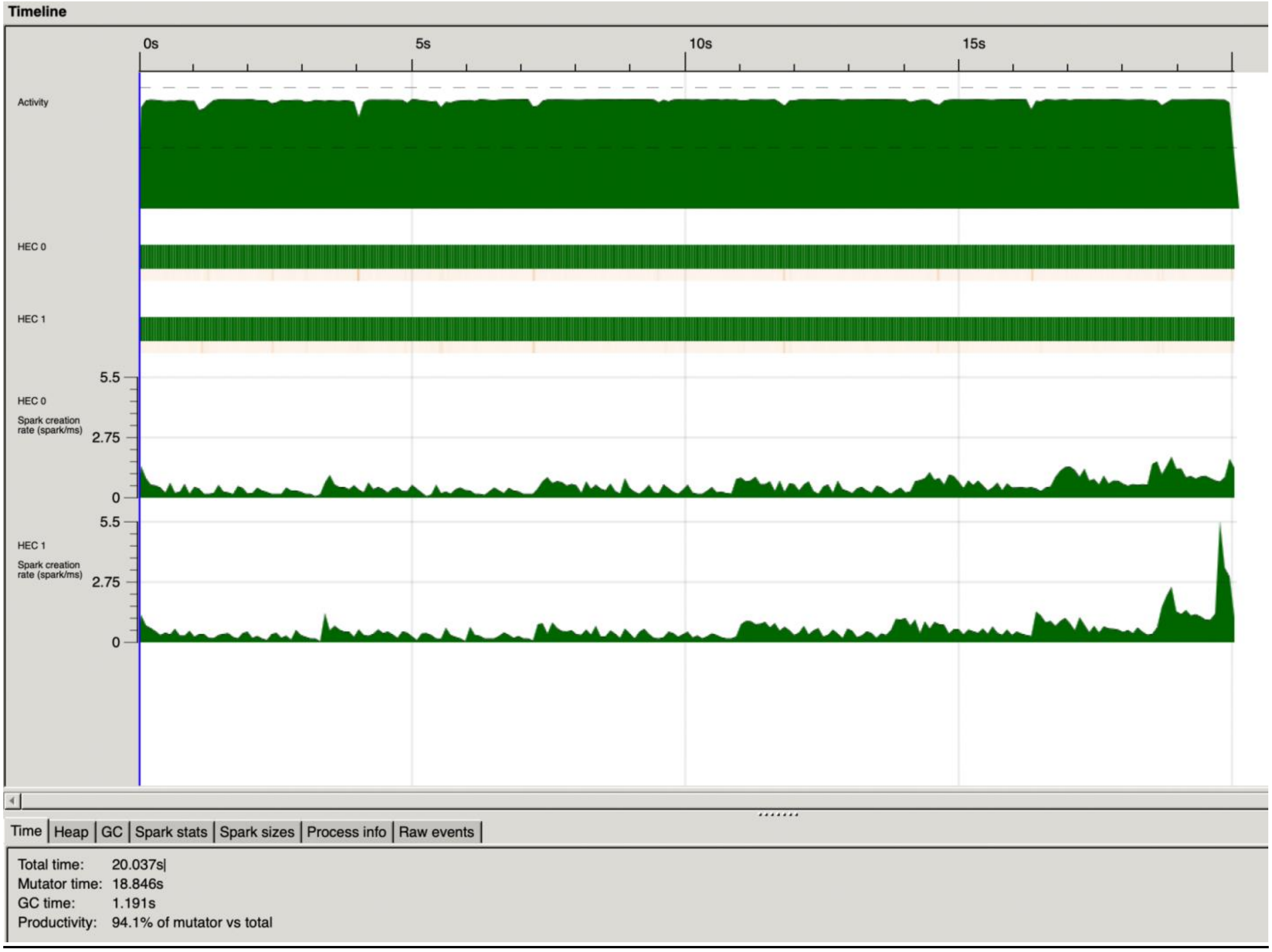


```
Total time:     19.372s
Mutator time:   17.987s
GC time:        1.385s
Productivity:   92.8% of mutator vs total
```

Attempt 2 - 2 threads:

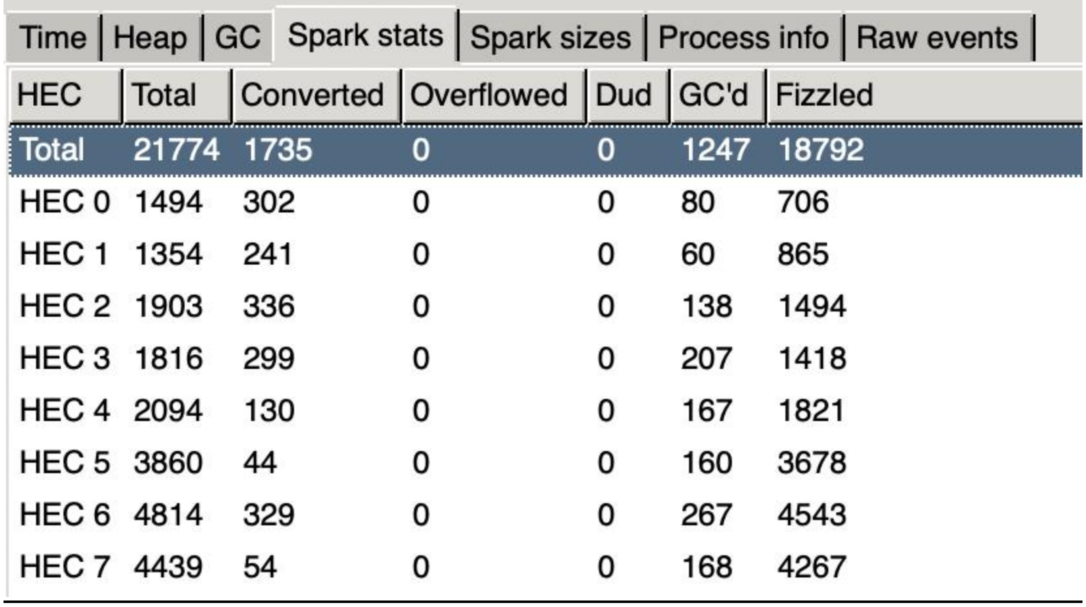


HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	21687	25	0	0	1275	20387
HEC 0	10469	11	0	0	477	9973
HEC 1	11218	14	0	0	798	10414

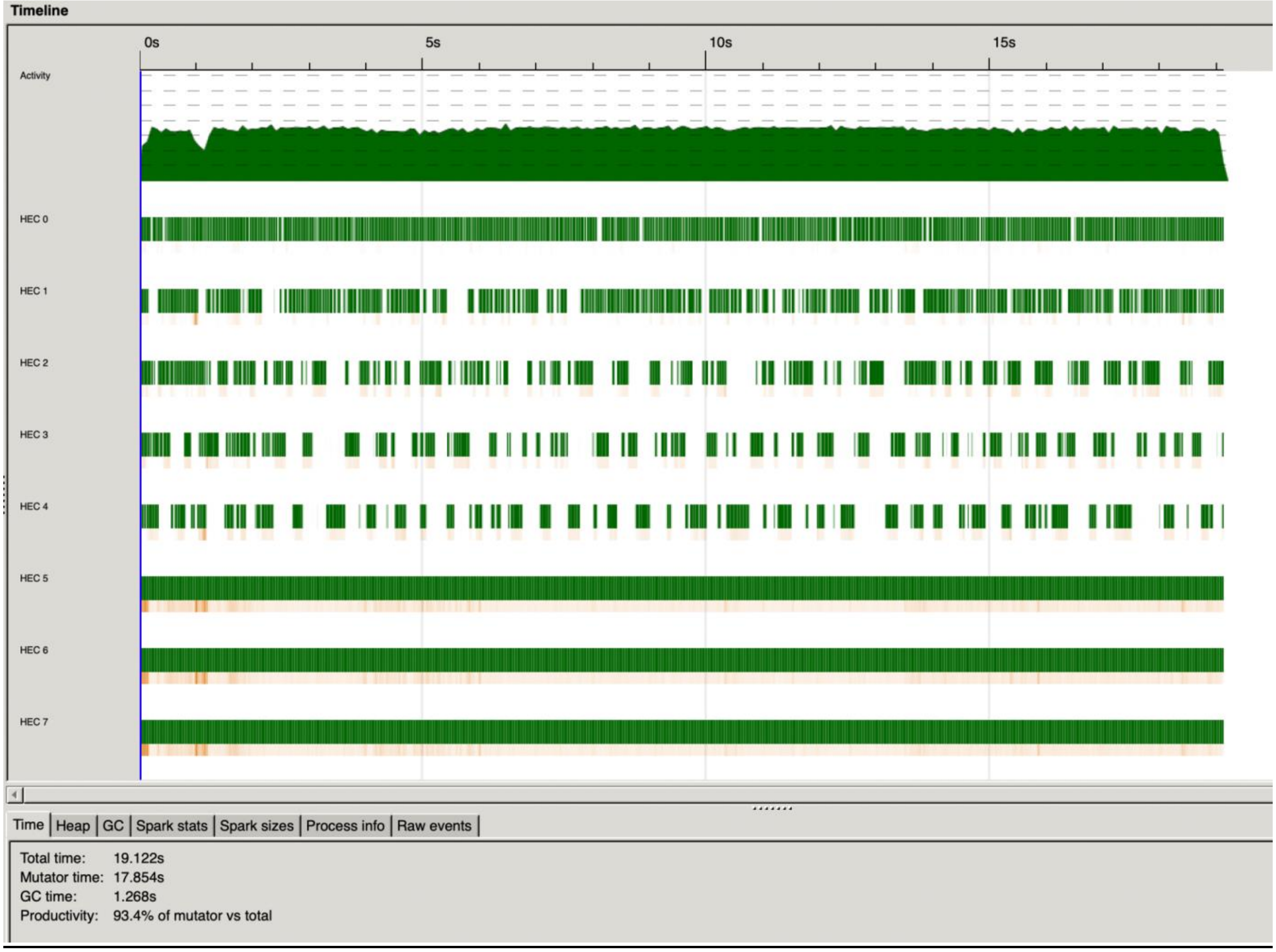


Total time: 20.037s
Mutator time: 18.846s
GC time: 1.191s
Productivity: 94.1% of mutator vs total

Attempt 2 - 8 threads:

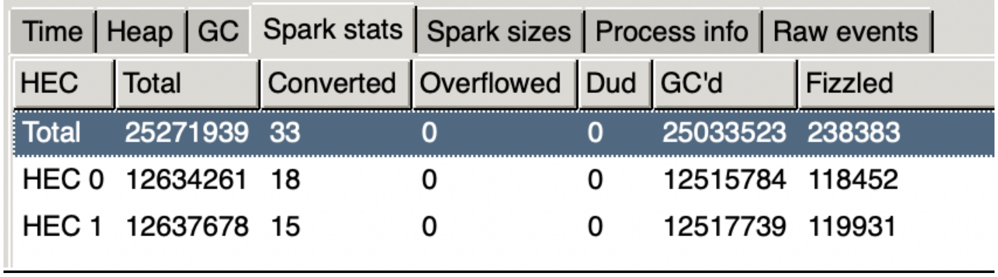


HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	21774	1735	0	0	1247	18792
HEC 0	1494	302	0	0	80	706
HEC 1	1354	241	0	0	60	865
HEC 2	1903	336	0	0	138	1494
HEC 3	1816	299	0	0	207	1418
HEC 4	2094	130	0	0	167	1821
HEC 5	3860	44	0	0	160	3678
HEC 6	4814	329	0	0	267	4543
HEC 7	4439	54	0	0	168	4267

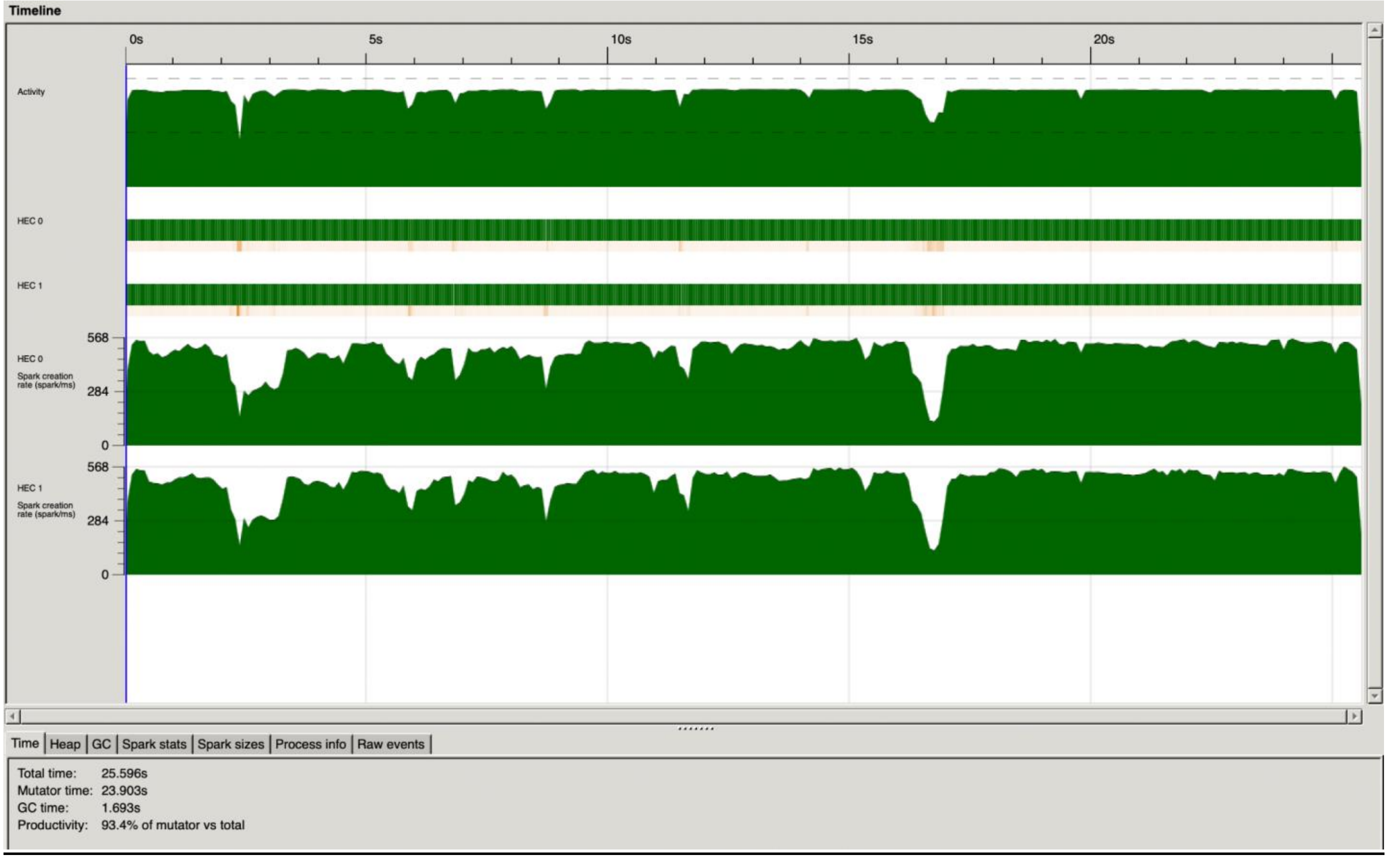


Total time: 19.122s
Mutator time: 17.854s
GC time: 1.268s
Productivity: 93.4% of mutator vs total

Attempt 3 - 2 threads:

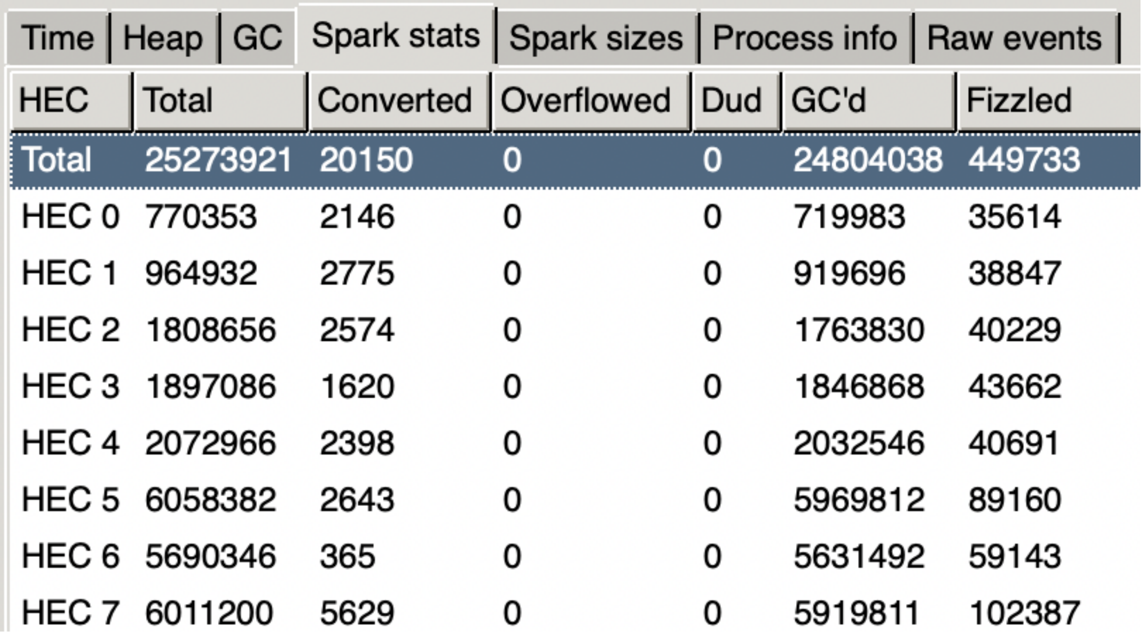


HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	25271939	33	0	0	25033523	238383
HEC 0	12634261	18	0	0	12515784	118452
HEC 1	12637678	15	0	0	12517739	119931

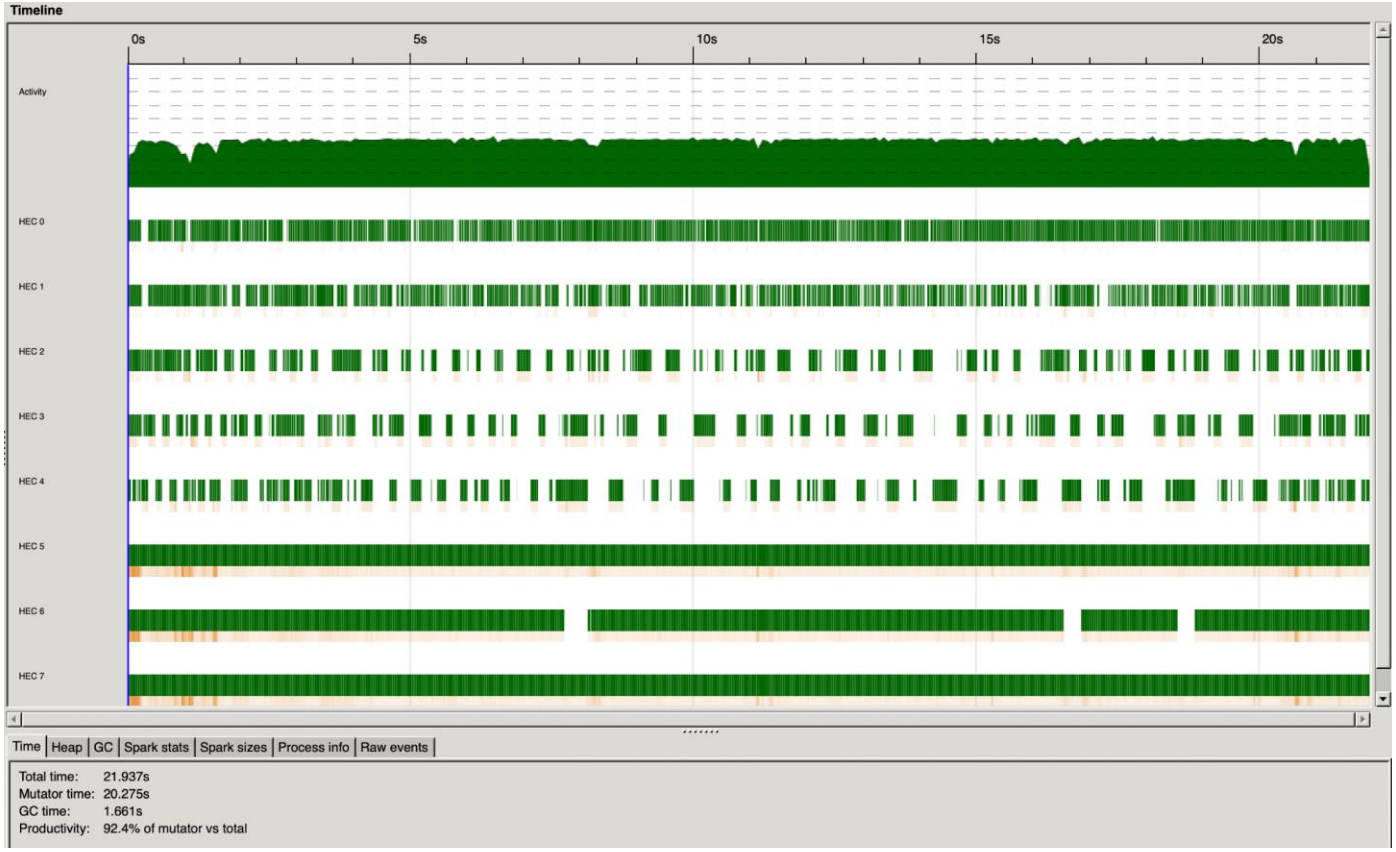


```
Total time:     25.596s
Mutator time:   23.903s
GC time:        1.693s
Productivity:   93.4% of mutator vs total
```

Attempt 3 - 8 threads:



HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	25273921	20150	0	0	24804038	449733
HEC 0	770353	2146	0	0	719983	35614
HEC 1	964932	2775	0	0	919696	38847
HEC 2	1808656	2574	0	0	1763830	40229
HEC 3	1897086	1620	0	0	1846868	43662
HEC 4	2072966	2398	0	0	2032546	40691
HEC 5	6058382	2643	0	0	5969812	89160
HEC 6	5690346	365	0	0	5631492	59143
HEC 7	6011200	5629	0	0	5919811	102387



Total time: 21.937s
Mutator time: 20.275s
GC time: 1.661s
Productivity: 92.4% of mutator vs total

Code:

Main.hs

```
--------------------------------------------------------------------------
import Solver (solver, Square)
import qualified Data.List as L
import System.Environment(getArgs, getProgName)
import System.Exit(die)

prettyPrint :: Maybe Square -> IO()
prettyPrint (Just s) = putStrLn $ unlines $ L.map (unwords . L.map show) s
prettyPrint Nothing = putStrLn ""

parseHelper :: [String] -> [Int]
parseHelper (x:xs) = (read x :: Int) : parseHelper xs
parseHelper _ = []

parseInput :: [String] -> Square
parseInput (x:xs) = (parseHelper $ words x) : parseInput xs
parseInput _ = []

main :: IO ()
main = do
        args <- getArgs
        (filename) <- case args of
            [filename] -> return (filename)
            _ -> do pn <- getProgName
                    die $ "Usage: " ++ pn ++ " <filename> +RTS -N? -ls"
        file <- readFile filename
        let inputMagic = parseInput $ lines file
        mapM_ prettyPrint (solver inputMagic)
```

MagicSquareSolver.hs

```
--------------------------------------------------------------------------
module Solver
(solver, Square)
where

import qualified Data.List as L
import qualified Data.Set as S
import Control.Monad -- guard
import Control.Parallel.Strategies hiding (parMap)

type Square = [[Int]]
```

```
magic :: Square
magic = [[2,16,13,3],
         [11,5,8,10],
         [7,9,12,6],
         [14,4,1,15]]

-- prettyPrint :: Maybe Square -> IO()
-- prettyPrint (Just s) = putStrLn $ unlines $ L.map (unwords . L.map
show) s
-- prettyPrint Nothing = putStrLn ""

fDiagonal :: Square -> Int -> [Int]
fDiagonal (x:xs) i = (x !! i) : fDiagonal xs (i+1)
fDiagonal _ _ = []

aDiagonal :: Square -> Int -> Int -> [Int]
aDiagonal (x:xs) i n = (x !! (n-1-i)) : aDiagonal xs (i+1) n
aDiagonal _ _ _ = []

validator :: Square -> Bool
validator s = (all (== magicNum) rowSumf) && (all (< magicNum) rowSumu)
      where n = length s
            magicNum = (1 + (n*n)) * n `div` 2
            fDiagonalR = filter (/=(-1)) $ fDiagonal s 0
            aDiagonalR = filter (/=(-1)) $ aDiagonal s 0 n
            rows_ = L.map (filter (/=(-1))) s
            rows = rows_ ++ [fDiagonalR] ++ [aDiagonalR]
            filledRow = filter (\x -> length x == n) rows
            unfilledRow = filter (\x -> length x < n) rows
            rowSumf = L.map sum filledRow
            rowSumu = L.map sum unfilledRow


squareFilled :: Square -> Bool
squareFilled = and . L.map (all (/= -1))

findEmptyPos :: Square -> Int -> (Int, Int)
findEmptyPos (x:xs) curRow
      | emptyPos == length x = findEmptyPos xs $ curRow + 1
      | otherwise = (curRow, emptyPos)
      where emptyPos = length $ takeWhile (/= -1) x
findEmptyPos _ _ = (-1, -1)


updateAtPos :: Square -> Int -> Int -> Int -> Square
updateAtPos s x y val = prevRows ++ updatedRow:(tail otherRows)
      where (prevRows, otherRows) = splitAt x s
```

```
            curRow = head otherRows
            (prevElems, otherElems) = splitAt y curRow
            updatedRow = prevElems ++ val:(tail otherElems)


nextSteps :: (Square, S.Set Int) -> Int -> Int -> [Int] -> [(Square, S.Set
Int)]
nextSteps state x y (v:vs)
      | validator newS && validator newST = curPair:nextSteps state x y vs
      | otherwise = nextSteps state x y vs
      where (s, choiceS) = state
            newS = updateAtPos s x y v
            newST = L.transpose newS
            curPair = (newS, S.delete v choiceS)
nextSteps _ _ _ _ = []

parMapDeep :: NFData b => (a -> b) -> [a] -> [b]
parMapDeep f xs = L.map f xs `using` parList rdeepseq

parMap :: NFData b => (a -> b) -> [a] -> [b]
parMap f xs = L.map f xs `using` parList rseq

splitHelper :: [Int] -> Int -> [[Int]]
splitHelper l s
      | n > 0 = cur : splitHelper remain s
      | otherwise = []
      where n = length cur
            cur = take s l
            remain = drop s l

splitList :: [Int] -> [[Int]]
splitList l
      | n < 4 = [l]
      | otherwise = splitHelper l size
      where n = length l
            x = round (sqrt (fromIntegral n))
            size = n `div` x

solverHelper :: Int -> (Square, S.Set Int) -> [Maybe Square]
solverHelper parLayer state
      | squareFilled s = [Just s >>= (\x -> guard (validator x) >> return
x)]
      | parLayer > 1  = concat $ parMap (solverHelper (parLayer-1)) next
      | parLayer == 1 = concat $ parMapDeep (solverHelper (parLayer-1))
next
      | otherwise = concat $ L.map (solverHelper parLayer) next
      where (s, choiceS) = state
```

```
            (x, y) = findEmptyPos s 0
            choiceL = S.toList choiceS
            next = nextSteps state x y choiceL

solver :: Square -> [Maybe Square]
solver s = solverHelper (n+1) (s, S.difference allState curState)
      where n = length s
            allState = S.fromList [1..n*n]
            curState = foldl (S.union) S.empty $ L.map S.fromList s
```