Parallel Functional Programming Final Project Report - Crossword Solver

Project Team Members: Rose Huang (rh2805) and Biqing Qiu (bq2134)

Description: We created a crossword solver for a crossword board with no hints. Given a board with blanks and blacked out boxes, we searched for the right words to place into the blanks using brute force, and then introduced parallelism. We selected words of the right length from a dictionary text file and verified that the solution is right by checking for collisions. We return all possible solutions.

Data: We are using three test crossword puzzles found online (references below). The crossword puzzles are 6x9, 7x7 and 9x9 respectively. The word pool used for the search is a dictionary of 60 words containing all words used in the solutions of the three crossword puzzles (so each returns at least one solution) plus 20 most common medium-length and short English words found online. We also created a test with no solution of a very small crossword with a very small dictionary of 3 words to verify that our crossword solver still works when there are no solutions.

Strategy: Fitting words of the right length to board: From the given blanks, which we represent as the data type **Sites** with data constructors **squares** ((x,y) coordinates) and **len**, we fit words from the dictionary of the right length into the blank using recursion. Each time we recurse, we place a word of the right length into the blank and then check against the already filled sites to verify that each square has only one letter. We do this verification by taking the returned solution, a list of tuples of Strings and Sites, and check that at each square there is no collision of letters; if so, we filter out the solution. We recurse until our base case, which is when there are no blanks left to fill.

Verifying Solution: We verify our solution as we fill in the blanks, pruning out solutions that have collisions of different letters in the same blank, as described in our strategy. If there isn't a solution with the given dictionary, our crossword solver returns nothing. If there are multiple solutions, we return all of the unique solutions.

Parallelizing the Solver: After obtaining a list of candidate words of the right length for a blank, we use **parPair** to parallelize the solver to continue the search with half the list per thread. Our parallelization essentially breaks a tree search into two different branches at each level and solves the branches in parallel. We use rpar to evaluate to WHNF, which is adequate for our application.

Report on Parallelization:

Test 1:

```
Biqings-MacBook-Pro:crossword biqing$ time ./crosswordSolver words76.txt test_site1.txt +RTS -ls -N8
original board:
┌                         ┐
  'X' 'X' 'X' 'X' ' ' ' '
  'X' ' ' ' ' ' ' ' ' 'X'
  'X' 'X' 'X' 'X' 'X' 'X'
  'X' ' ' 'X' ' ' ' ' 'X'
  'X' ' ' 'X' 'X' 'X' ' '
  'X' 'X' 'X' ' ' 'X' ' '
  ' ' ' ' 'X' 'X' 'X' ' '
  ' ' ' ' ' ' 'X' ' ' ' '
  'X' 'X' 'X' 'X' 'X' ' '
└                         ┘
solutions:
┌                         ┐
  'p' 'e' 'r' 'l' ' ' ' '
  'r' ' ' ' ' ' ' ' ' 'w'
  'o' 'n' 'l' 'i' 'n' 'e'
  'l' ' ' 'i' ' ' ' ' 'b'
  'o' ' ' 'n' 'f' 's' ' '
  'g' 'n' 'u' ' ' 'q' ' '
  ' ' ' ' 'x' 'm' 'l' ' '
  ' ' ' ' ' ' 'a' ' ' ' '
  'w' 'h' 'i' 'c' 'h' ' '
└                         ┘

┌                         ┐
  'p' 'e' 'r' 'l' ' ' ' '
  'r' ' ' ' ' ' ' ' ' 'w'
  'o' 'n' 'l' 'i' 'n' 'e'
  'l' ' ' 'i' ' ' ' ' 'b'
  'o' ' ' 'n' 'f' 's' ' '
  'g' 'n' 'u' ' ' 'q' ' '
  ' ' ' ' 'x' 'm' 'l' ' '
  ' ' ' ' ' ' 'a' ' ' ' '
  'e' 'm' 'a' 'c' 's' ' '
└                         ┘
```

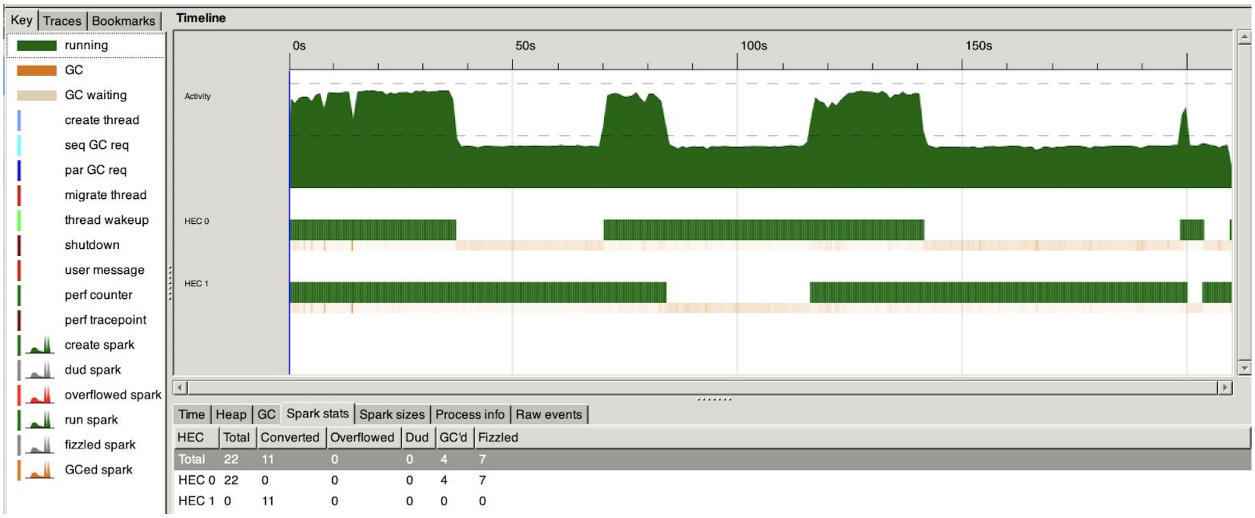
There are limited Threadscope graphs for this test because some time data is too large and Threadscope is killed trying to display. However, we can look at the time performance to see that parallelization achieves a speed-up.

1 Core:

real 4m0.394s
user 3m51.535s
sys 0m3.357s

2 Cores:

real 3m29.848s
user 4m39.351s
sys 0m10.091s



From 1 core to 2 cores, there is a ~12% speedup from 4min to 3.5min. This shows that parallelization does achieve better performance. A total of 22 sparks were created in core 1 and 11 converted in core 2, showing good parallelization.

4 Cores:

real 4m31.123s
user 7m20.470s
sys 0m31.556s

8 Cores:

real 4m32.557s
user 7m29.513s
sys 0m32.104s

For 4 cores and 8 cores, there is no significant speedup, because of overhead. As we are using parPair, we expect that only 2 cores are used and 2 cores give the best performance.

Test 2:

Solution:

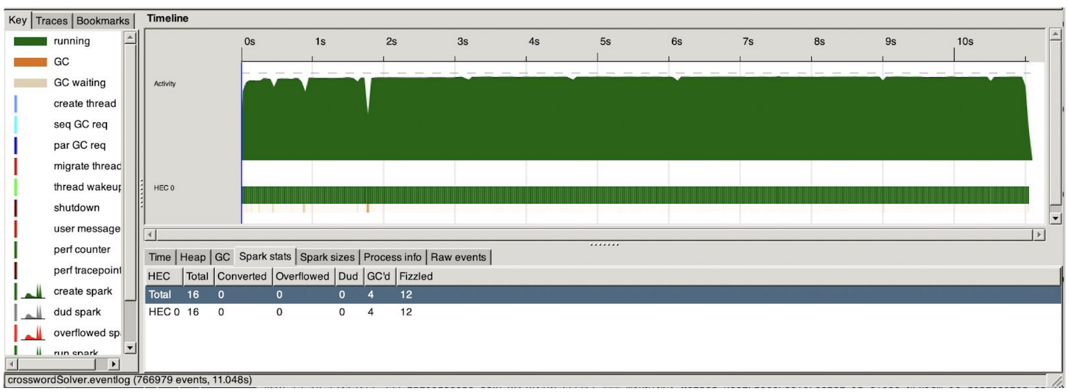
```
dyn-160-39-128-152:crossword rosehuang$ time ./crosswordSolver words76.txt test_site2.txt +RTS -ls -N4
original board:
[ ' ' 'X' ' ' 'X' ' ' ' ' 'X'
  'X' 'X' 'X' 'X' 'X' 'X' 'X'
  ' ' 'X' ' ' 'X' ' ' ' ' 'X'
  'X' 'X' 'X' 'X' 'X' 'X' 'X'
  'X' ' ' ' ' 'X' ' ' 'X' ' '
  'X' 'X' 'X' 'X' 'X' 'X' 'X'
  'X' ' ' ' ' 'X' ' ' 'X' ' ' ]
solutions:
[ ' ' 'b' ' ' 's' ' ' ' ' 'f'
  's' 'l' 'i' 'p' 'p' 'e' 'r'
  ' ' 'u' ' ' 'i' ' ' ' ' 'o'
  'w' 'e' 'd' 'd' 'i' 'n' 'g'
  'e' ' ' ' ' 'e' ' ' 'e' ' '
  'a' 'd' 'd' 'r' 'e' 's' 's'
  'k' ' ' ' ' 's' ' ' 't' ' ' ]
```

1 Core:

real 0m11.066s

user 0m10.837s

sys 0m0.169s

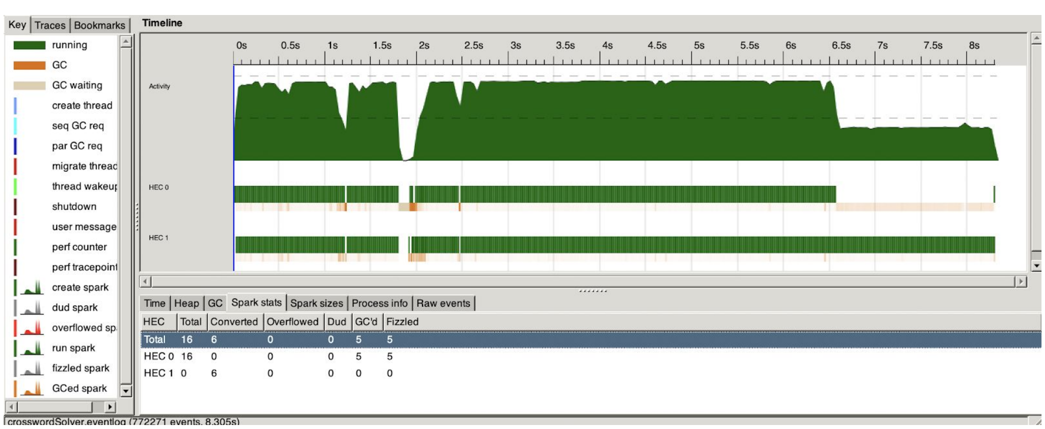


2 Cores:

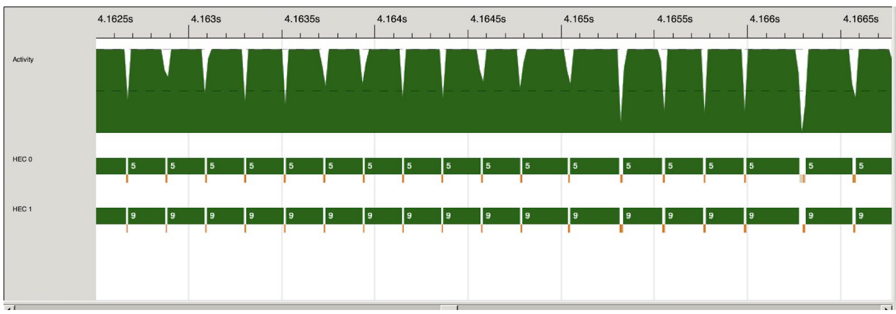
real 0m8.341s

user 0m14.086s

sys 0m0.263s

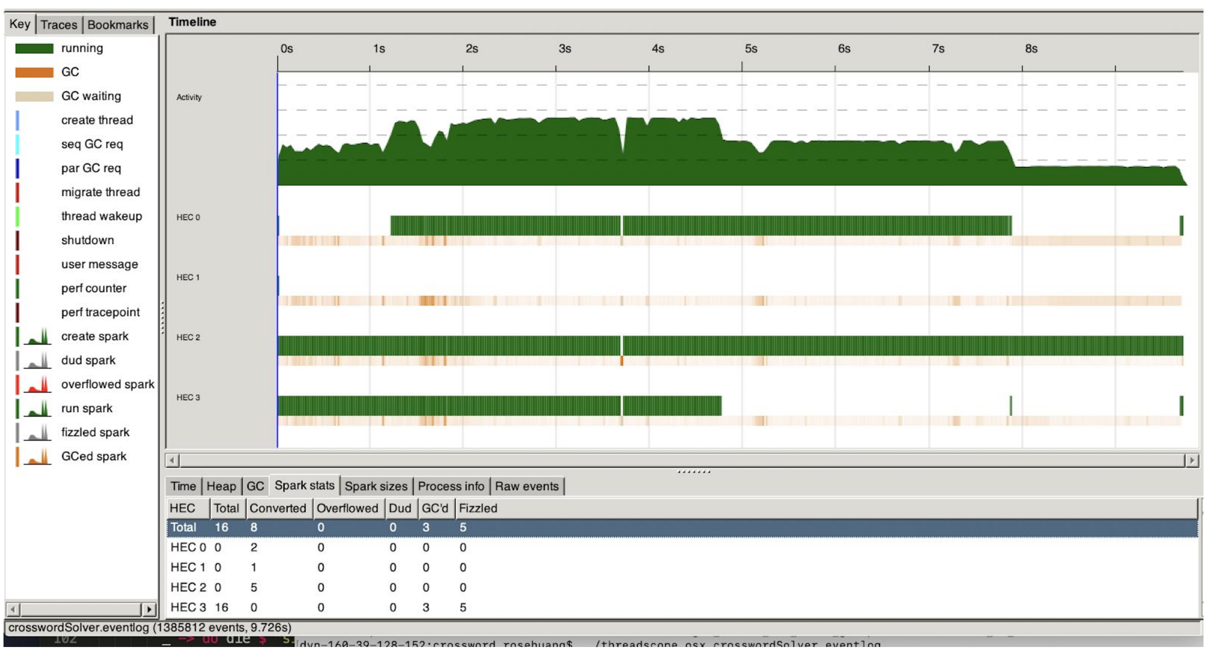


From 2.5 s to 6.5 s, we see great parallelism with 2 cores (zoomed in pic below). 16 sparks were created in total, all in the first core and 6 sparks were converted in the second core. This shows an efficient use of sparks. There is also a speed up from 11.066s to 8.341s, a 24.6% speedup. The first core is not used for the last few seconds, most likely because of the non-parallel conversion of our crossword to a printable string form.



4 Cores

real 0m9.764s
user 0m22.015s
sys 0m0.878s



For 4 cores, the second core is not utilized at all. The other 3 cores run in parallel from approximately 1.25 s to 4.5 s. Using 4 cores is also slower than using 2 cores due to extra overhead.

Test 3:

Solution:

```
original board:
'X' 'X' 'X' 'X' ' ' 'X' 'X' 'X' 'X'
'X' ' ' 'X' ' ' ' ' ' ' 'X' ' ' 'X'
'X' 'X' 'X' 'X' ' ' 'X' 'X' 'X' 'X'
' ' 'X' ' ' 'X' 'X' 'X' ' ' 'X' ' '
'X' 'X' 'X' 'X' ' ' 'X' 'X' 'X' 'X'
' ' 'X' ' ' 'X' 'X' 'X' ' ' 'X' ' '
'X' 'X' 'X' 'X' ' ' 'X' 'X' 'X' 'X'
'X' ' ' 'X' ' ' ' ' ' ' 'X' ' ' 'X'
'X' 'X' 'X' 'X' ' ' 'X' 'X' 'X' 'X'
```

```
'd' 'r' 'i' 'p' ' ' 'd' 'i' 'r' 't'
'o' ' ' 'c' ' ' ' ' ' ' 'o' ' ' 'o'
'a' 'g' 'e' 'd' ' ' 'q' 'u' 'i' 't'
' ' 'r' ' ' 'i' 'o' 'u' ' ' 'g' ' '
'd' 'i' 'r' 't' ' ' 'i' 'd' 'l' 'e'
' ' 'p' ' ' 't' 'o' 'e' ' ' 'o' ' '
'h' 'e' 'r' 'o' ' ' 't' 'r' 'o' 't'
'a' ' ' 'a' ' ' ' ' ' ' 'u' ' ' 'o'
't' 'a' 'p' 'e' ' ' 'd' 'e' 'b' 't'
```

```
'd' 'r' 'i' 'p' ' ' 'e' 'y' 'e' 's'
'o' ' ' 'c' ' ' ' ' ' ' 'o' ' ' 'a'
'a' 'g' 'e' 'd' ' ' 'q' 'u' 'i' 't'
' ' 'r' ' ' 'i' 'o' 'u' ' ' 'g' ' '
'd' 'i' 'r' 't' ' ' 'i' 'd' 'l' 'e'
' ' 'p' ' ' 't' 'o' 'e' ' ' 'o' ' '
'h' 'e' 'r' 'o' ' ' 't' 'r' 'o' 't'
'a' ' ' 'a' ' ' ' ' ' ' 'u' ' ' 'o'
't' 'a' 'p' 'e' ' ' 'd' 'e' 'b' 't'
```

```
'd' 'r' 'i' 'p' ' ' 'd' 'i' 'r' 't'
'o' ' ' 'c' ' ' ' ' ' ' 'o' ' ' 'o'
'a' 'g' 'e' 'd' ' ' 'q' 'u' 'i' 't'
' ' 'r' ' ' 'i' 'o' 'u' ' ' 'g' ' '
'd' 'i' 'r' 't' ' ' 'i' 'd' 'l' 'e'
' ' 'p' ' ' 't' 'o' 'e' ' ' 'o' ' '
'h' 'e' 'r' 'o' ' ' 't' 'r' 'o' 't'
'a' ' ' 'a' ' ' ' ' ' ' 'u' ' ' 'o'
't' 'a' 'p' 'e' ' ' 'n' 'e' 's' 't'
```

```
solutions:
'd' 'i' 'r' 't' ' ' 'd' 'i' 'r' 't'
'o' ' ' 'u' ' ' ' ' ' ' 'o' ' ' 'o'
'a' 'g' 'e' 'd' ' ' 'q' 'u' 'i' 't'
' ' 'r' ' ' 'i' 'o' 'u' ' ' 'g' ' '
'd' 'i' 'r' 't' ' ' 'i' 'd' 'l' 'e'
' ' 'p' ' ' 't' 'h' 'e' ' ' 'o' ' '
'h' 'e' 'r' 'o' ' ' 't' 'r' 'o' 't'
'a' ' ' 'a' ' ' ' ' ' ' 'u' ' ' 'o'
't' 'a' 'p' 'e' ' ' 'd' 'e' 'b' 't'
```

```
'd' 'i' 'r' 't' ' ' 'e' 'y' 'e' 's'
'o' ' ' 'u' ' ' ' ' ' ' 'o' ' ' 'a'
'a' 'g' 'e' 'd' ' ' 'q' 'u' 'i' 't'
' ' 'r' ' ' 'i' 'o' 'u' ' ' 'g' ' '
'd' 'i' 'r' 't' ' ' 'i' 'd' 'l' 'e'
' ' 'p' ' ' 't' 'h' 'e' ' ' 'o' ' '
'h' 'e' 'r' 'o' ' ' 't' 'r' 'o' 't'
'a' ' ' 'a' ' ' ' ' ' ' 'u' ' ' 'o'
't' 'a' 'p' 'e' ' ' 'd' 'e' 'b' 't'
```

```
'd' 'i' 'r' 't' ' ' 'd' 'i' 'r' 't'
'o' ' ' 'u' ' ' ' ' ' ' 'o' ' ' 'o'
'a' 'g' 'e' 'd' ' ' 'q' 'u' 'i' 't'
' ' 'r' ' ' 'i' 'o' 'u' ' ' 'g' ' '
'd' 'i' 'r' 't' ' ' 'i' 'd' 'l' 'e'
' ' 'p' ' ' 't' 'h' 'e' ' ' 'o' ' '
'h' 'e' 'r' 'o' ' ' 't' 'r' 'o' 't'
'a' ' ' 'a' ' ' ' ' ' ' 'u' ' ' 'o'
't' 'a' 'p' 'e' ' ' 'n' 'e' 's' 't'
```

```
'd' 'i' 'r' 't' ' ' 'e' 'y' 'e' 's'
'o' ' ' 'u' ' ' ' ' ' ' 'o' ' ' 'a'
'a' 'g' 'e' 'd' ' ' 'q' 'u' 'i' 't'
' ' 'r' ' ' 'i' 'o' 'u' ' ' 'g' ' '
'd' 'i' 'r' 't' ' ' 'i' 'd' 'l' 'e'
' ' 'p' ' ' 't' 'h' 'e' ' ' 'o' ' '
'h' 'e' 'r' 'o' ' ' 't' 'r' 'o' 't'
'a' ' ' 'a' ' ' ' ' ' ' 'u' ' ' 'o'
't' 'a' 'p' 'e' ' ' 'n' 'e' 's' 't'
```

```
'd' 'r' 'i' 'p' ' ' 'd' 'i' 'r' 't'
'o' ' ' 'c' ' ' ' ' ' ' 'o' ' ' 'o'
'a' 'g' 'e' 'd' ' ' 'q' 'u' 'i' 't'
' ' 'r' ' ' 'i' 'o' 'u' ' ' 'g' ' '
'd' 'i' 'r' 't' ' ' 'i' 'd' 'l' 'e'
' ' 'p' ' ' 't' 'h' 'e' ' ' 'o' ' '
'h' 'e' 'r' 'o' ' ' 't' 'r' 'o' 't'
'a' ' ' 'a' ' ' ' ' ' ' 'u' ' ' 'o'
't' 'a' 'p' 'e' ' ' 'd' 'e' 'b' 't'
```

```
'd' 'r' 'i' 'p' ' ' 'e' 'y' 'e' 's'
'o' ' ' 'c' ' ' ' ' ' ' 'o' ' ' 'a'
'a' 'g' 'e' 'd' ' ' 'q' 'u' 'i' 't'
' ' 'r' ' ' 'i' 'o' 'u' ' ' 'g' ' '
'd' 'i' 'r' 't' ' ' 'i' 'd' 'l' 'e'
' ' 'p' ' ' 't' 'h' 'e' ' ' 'o' ' '
'h' 'e' 'r' 'o' ' ' 't' 'r' 'o' 't'
'a' ' ' 'a' ' ' ' ' ' ' 'u' ' ' 'o'
't' 'a' 'p' 'e' ' ' 'd' 'e' 'b' 't'
```

```
'd' 'r' 'i' 'p' ' ' 'd' 'i' 'r' 't'
'o' ' ' 'c' ' ' ' ' ' ' 'o' ' ' 'o'
'a' 'g' 'e' 'd' ' ' 'q' 'u' 'i' 't'
' ' 'r' ' ' 'i' 'o' 'u' ' ' 'g' ' '
'd' 'i' 'r' 't' ' ' 'i' 'd' 'l' 'e'
' ' 'p' ' ' 't' 'h' 'e' ' ' 'o' ' '
'h' 'e' 'r' 'o' ' ' 't' 'r' 'o' 't'
'a' ' ' 'a' ' ' ' ' ' ' 'u' ' ' 'o'
't' 'a' 'p' 'e' ' ' 'n' 'e' 's' 't'
```

```
'd' 'r' 'i' 'p' ' ' 'e' 'y' 'e' 's'
'o' ' ' 'c' ' ' ' ' ' ' 'o' ' ' 'a'
'a' 'g' 'e' 'd' ' ' 'q' 'u' 'i' 't'
' ' 'r' ' ' 'i' 'o' 'u' ' ' 'g' ' '
'd' 'i' 'r' 't' ' ' 'i' 'd' 'l' 'e'
' ' 'p' ' ' 't' 'h' 'e' ' ' 'o' ' '
'h' 'e' 'r' 'o' ' ' 't' 'r' 'o' 't'
'a' ' ' 'a' ' ' ' ' ' ' 'u' ' ' 'o'
't' 'a' 'p' 'e' ' ' 'n' 'e' 's' 't'
```

```
'd' 'i' 'r' 't' ' ' 'd' 'i' 'r' 't'
'o' ' ' 'u' ' ' ' ' ' ' 'o' ' ' 'o'
'a' 'g' 'e' 'd' ' ' 'q' 'u' 'i' 't'
' ' 'r' ' ' 'i' 'o' 'u' ' ' 'g' ' '
'd' 'i' 'r' 't' ' ' 'i' 'd' 'l' 'e'
' ' 'p' ' ' 't' 'o' 'e' ' ' 'o' ' '
'h' 'e' 'r' 'o' ' ' 't' 'r' 'o' 't'
'a' ' ' 'a' ' ' ' ' ' ' 'u' ' ' 'o'
't' 'a' 'p' 'e' ' ' 'd' 'e' 'b' 't'
```

```
'd' 'i' 'r' 't' ' ' 'e' 'y' 'e' 's'
'o' ' ' 'u' ' ' ' ' ' ' 'o' ' ' 'a'
'a' 'g' 'e' 'd' ' ' 'q' 'u' 'i' 't'
' ' 'r' ' ' 'i' 'o' 'u' ' ' 'g' ' '
'd' 'i' 'r' 't' ' ' 'i' 'd' 'l' 'e'
' ' 'p' ' ' 't' 'o' 'e' ' ' 'o' ' '
'h' 'e' 'r' 'o' ' ' 't' 'r' 'o' 't'
'a' ' ' 'a' ' ' ' ' ' ' 'u' ' ' 'o'
't' 'a' 'p' 'e' ' ' 'd' 'e' 'b' 't'
```

```
'd' 'i' 'r' 't' ' ' 'd' 'i' 'r' 't'
'o' ' ' 'u' ' ' ' ' ' ' 'o' ' ' 'o'
'a' 'g' 'e' 'd' ' ' 'q' 'u' 'i' 't'
' ' 'r' ' ' 'i' 'o' 'u' ' ' 'g' ' '
'd' 'i' 'r' 't' ' ' 'i' 'd' 'l' 'e'
' ' 'p' ' ' 't' 'o' 'e' ' ' 'o' ' '
'h' 'e' 'r' 'o' ' ' 't' 'r' 'o' 't'
'a' ' ' 'a' ' ' ' ' ' ' 'u' ' ' 'o'
't' 'a' 'p' 'e' ' ' 'n' 'e' 's' 't'
```

```
'd' 'i' 'r' 't' ' ' 'e' 'y' 'e' 's'
'o' ' ' 'u' ' ' ' ' ' ' 'o' ' ' 'a'
'a' 'g' 'e' 'd' ' ' 'q' 'u' 'i' 't'
' ' 'r' ' ' 'i' 'o' 'u' ' ' 'g' ' '
'd' 'i' 'r' 't' ' ' 'i' 'd' 'l' 'e'
' ' 'p' ' ' 't' 'o' 'e' ' ' 'o' ' '
'h' 'e' 'r' 'o' ' ' 't' 'r' 'o' 't'
'a' ' ' 'a' ' ' ' ' ' ' 'u' ' ' 'o'
't' 'a' 'p' 'e' ' ' 'n' 'e' 's' 't'
```

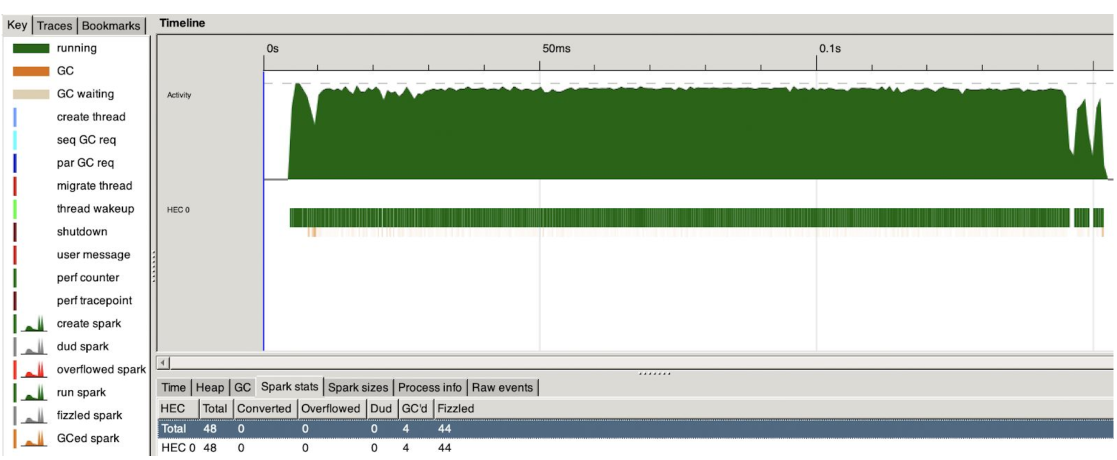
```
'd' 'r' 'i' 'p' ' ' 'e' 'y' 'e' 's'
'o' ' ' 'c' ' ' ' ' ' ' 'o' ' ' 'a'
'a' 'g' 'e' 'd' ' ' 'q' 'u' 'i' 't'
' ' 'r' ' ' 'i' 'o' 'u' ' ' 'g' ' '
'd' 'i' 'r' 't' ' ' 'i' 'd' 'l' 'e'
' ' 'p' ' ' 't' 'o' 'e' ' ' 'o' ' '
'h' 'e' 'r' 'o' ' ' 't' 'r' 'o' 't'
'a' ' ' 'a' ' ' ' ' ' ' 'u' ' ' 'o'
't' 'a' 'p' 'e' ' ' 'n' 'e' 's' 't'
```

1 Core:

real 0m0.166s

user 0m0.142s

sys 0m0.015s



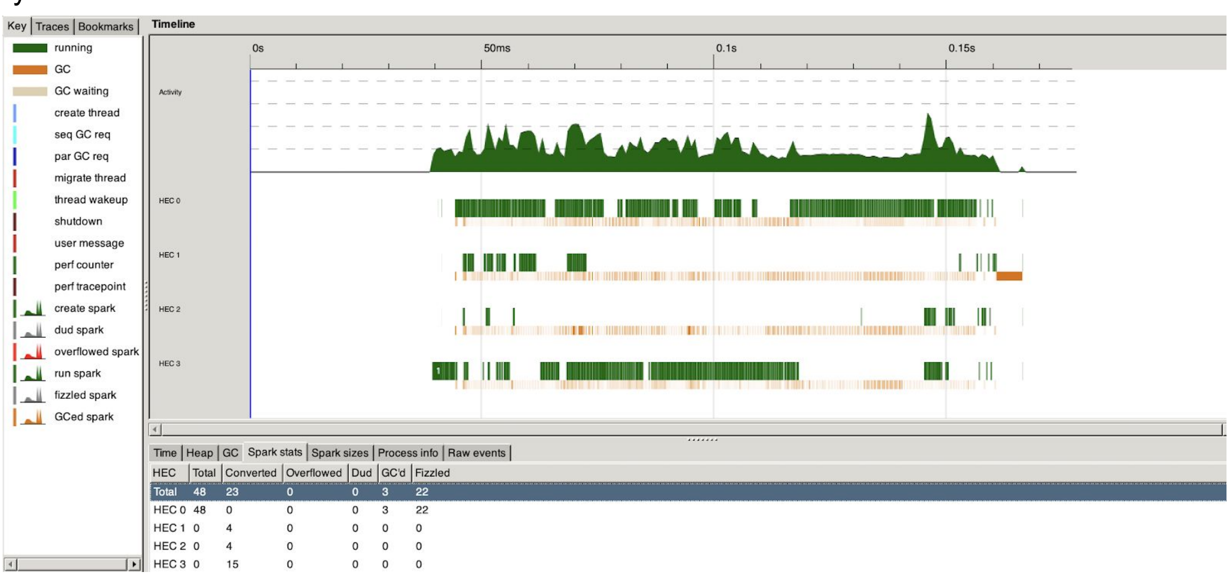
2 Cores:

real 0m0.137s
user 0m0.150s
sys 0m0.015s

As with the first two tests, there is a small performance speedup from 1 core to 2 cores from 0.166s to 0.137s (17.5% faster). A total of 48 sparks were created in the second core, 22 of which converted in the first core. However, the graph indicates large chunks of time in which the two cores are not being utilized in parallel.

4 Cores:

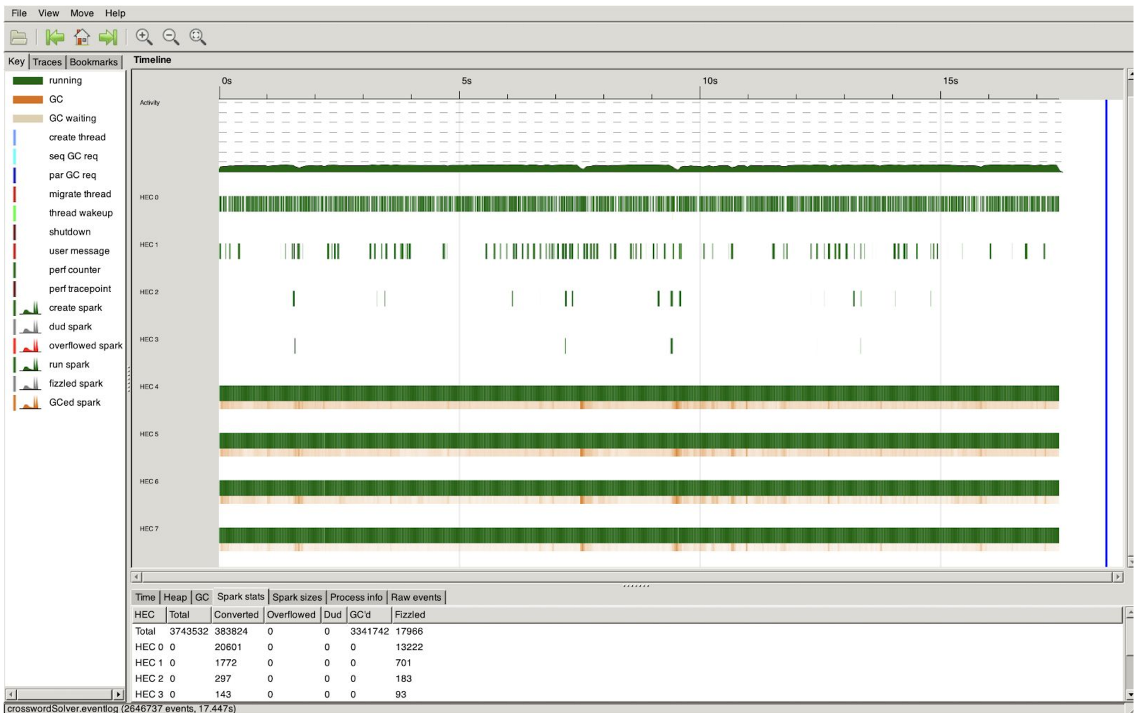
real 0m0.195s
user 0m0.194s
sys 0m0.033s



As with the first two tests, performance with 4 cores is worse than with 2 cores, showing 2 cores may be optimal for parPair.

Comparison between parPair and parList

We tried two versions of parallelism, using parPair and parList respectively (see the Appendix for parList implementation). With parList, we originally thought we achieved parallelization.



This is the graph plotted with 8 cores. At first glance it seems there is parallelization across 4 cores. However, a large number of sparks created were garbage collected or fizzled. There was also no performance improvement going from 1 core to 4 cores.

Running again on 4 cores. When we zoomed in, we noticed that actually only 1 core was utilized! This is despite the illusion from the earlier graphs that 4 cores were occupied. Hence there is actually no parallelization and explains the lack of performance improvement.

Results on parList parallelization on Test 2:

1 Core:

real 0m12.523s
user 0m11.783s
sys 0m0.294s

4 Cores:

real 0m19.524s
user 0m26.610s
sys 0m2.607s

8 Cores:

real 0m17.550s
user 0m24.377s
sys 0m2.924s

We believe that parList could be used to achieve parallelization, however, our implementation was probably wrong. parPair achieves a much more reasonable outcome with true parallelization.

APPENDIX:

Below is the code and results for when we tried to implement parList.

```
solve' :: Map.Map Int [String] -> [Site] -> [[(String, Site)]]
solve' _ []      = [[]]
solve' dict (s:ss) =
    if possWords == []
    then error ("No words of length " ++ show (len s))
    else do
        solveAgain <- solve' dict ss
        filter verifySquares
          (map (\x -> trySolve x  ++ solveAgain) possWords `using` parList rseq)
    where possWords = Map.findWithDefault [] (len s) dict
          trySolve :: String -> [(String, Site)]
          trySolve thisword = do
                return (thisword, s)
```

The next 2 pages include the final code we submitted with the parPair implementation.

```haskell
{-
PFP Final Project
Names: Rose Huang (rh2805) and Biqing Qiu (bq2134)
-}

import qualified Data.Map.Strict as Map
import qualified Data.List as List
import qualified Data.Matrix as Matrix
import System.IO(readFile)
import System.Environment(getArgs)
import System.Exit(die)
import Data.Ord (comparing)
import Data.Function (on)
import Data.Char(isAlpha, toLower)
import Control.Parallel.Strategies hiding (parPair)
import Control.Monad

type Square    = (Int, Int)
data Site      = Site {squares :: [Square], len :: Int} deriving (Show,Eq)
data Crossword =
  Crossword {wdict :: Map.Map Int [String], sites :: [Site]}
  deriving (Show,Eq)

-- convert list of strings from site file to list of sites
toSites :: [String] -> [Site]
toSites s = map (\x -> Site {squares = map (\y -> read y::(Int, Int))
            $ words x, len = length $ words x}) s

-- convert list of strings from dict file to map with length as key and list
-- of words as value
toDict :: [String] -> Map.Map Int [String]
toDict dictWords = Map.fromListWithKey (\_ x y -> x++y)
                   $ map (\w -> (length w, [w])) dictWords

-- test to ensure there are no two different letters on the same squares
verifySquares :: [(String, Site)] -> Bool
verifySquares xs = all allEqual $ groupBySquare xs
    where allEqual []      = True
          allEqual (x:xss) = all (x==) xss

-- make into list of lists of chars, grouped by squares
groupBySquare :: [(String, Site)] -> [[Char]]
groupBySquare xs = map (map snd)
                   $ List.groupBy ((==) `on` fst)
                   $ List.sortBy (comparing fst)
                   $ concatMap makeSqChar $ xs

-- assign each character to a square
makeSqChar :: (String, Site) -> [(Square, Char)]
makeSqChar (str,s) = zip (squares s) str

-- parallel evaluation in pairs
parPair :: Strategy (a, b)
parPair (a, b) = do
    a' <- rpar a
    b' <- rpar b
```

```haskell
    return (a', b')

-- return solution of crossword as a list of squares and letters
solve :: Crossword -> [Map.Map Square Char]
solve cw = map (Map.fromList . (concatMap makeSqChar)) solutions
    where solutions = List.nub $ solve' (wdict cw) (sites cw)

solve' :: Map.Map Int [String] -> [Site] -> [[(String, Site)]]
solve' _ []      = [[]]
solve' dict (s:ss) = if possWords == []
                        then error ("No words of length " ++ show (len s))
                        else do
                            let (a, b) = splitAt (length possWords `div` 2) possWords
                                (aa, bb) = (trySolve a, trySolve b) `using` parPair
                            aa ++ bb
    where possWords = Map.findWithDefault [] (len s) dict
          trySolve thiswords = do
                try <- thiswords
                solveAgain <- solve' dict ss
                let attempt = (try, s) : solveAgain
                Control.Monad.guard $ verifySquares attempt
                return attempt

-- return solution as prettyMatrix String
toMatrix :: Int -> Int -> Map.Map Square Char -> String
toMatrix rows cols solution = Matrix.prettyMatrix
    $ Matrix.matrix rows cols getLetter where
    getLetter (i,j) = case Map.lookup (i,j) solution of
        Nothing -> ' '
        Just c -> c

-- reads dict and sites file, construct Crossword, solve
main :: IO ()
main = do
  args <- getArgs
  case args of
    [dictFile, siteFile] -> do
      dictContents <- readFile dictFile
      siteContents <- readFile siteFile
      let dimensions:siteStrings = lines siteContents
          processedWords =
            map (map toLower . filter isAlpha) (lines dictContents)
          solutions = solve
            $ Crossword (toDict processedWords) (toSites (siteStrings))
          originalBoard = Map.fromList
            $ zip (concatMap squares (toSites siteStrings)) (repeat 'X')
      case (map (\x -> read x :: Int) $ words dimensions) of
        [rows, cols] -> do
            putStrLn "original board:"
            putStrLn $ toMatrix rows cols originalBoard
            putStrLn "solutions:"
            mapM_ putStrLn $ map (toMatrix rows cols) solutions
        _ -> do die $ "siteFile doesn't include dimensions"
    _ -> do die $ "Usage: ./crosswordSolver <dict file> <site file>"
```