# Parallel File System Statistics

Matthew Nelson

## Preface

File systems can be described as large trees where directory nodes have a non-zero out-degree and files act as leaf nodes in the tree. Graph traversal algorithms are a popular space for parallelization, so when I started my project I figured that parallelizing the traversal of the file system tree and corresponding file information would be a cool thing to research.

In the following sections, I will describe the inspiration behind my project, the sequential implementation, the parallel implementations, benchmarks, the visualizer utility implementation, and areas my project could improve.

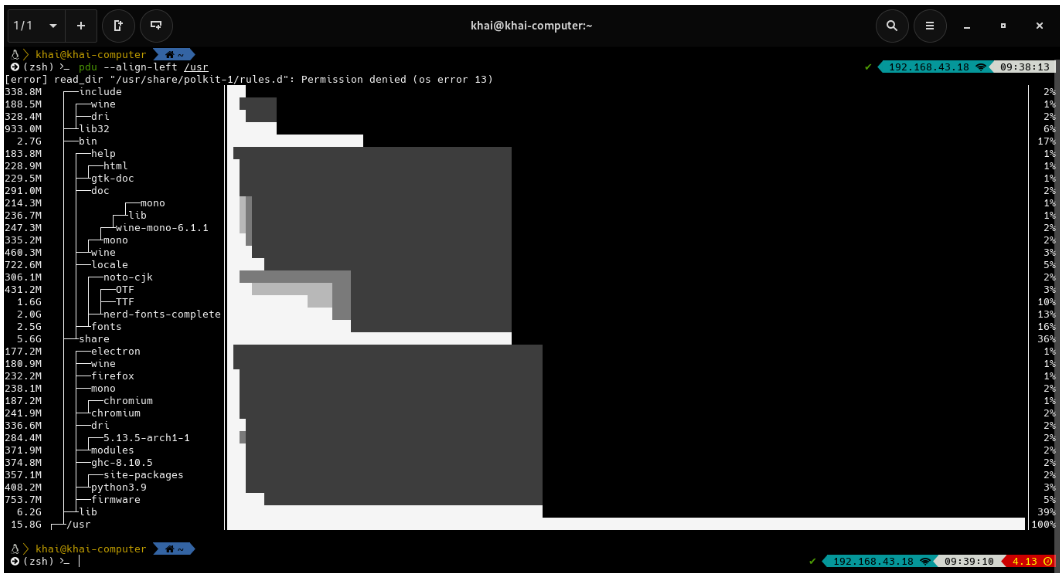
## Inspiration and Background

While researching a topic for my final project, I came across a discussion thread concerning the traversal of a large network file system. One of the users reported substantial speed up when parallelizing the traversal of the top-level directories in the file system, so I decided to explore relevant tooling in this area.

My project is largely inspired by the parallel-disk-usage utility, which is a “blazing fast directory tree analyzer” written in Rust. The utility traverses a given directory, collects file information, and outputs an aesthetically pleasing chart showing file sizes, path names, and tree structure.

The Rust implementation accomplishes this by using a pool of worker threads which take care of the traversal and file statistic collection for files as they are encountered. This somewhat goes against the Haskell-esk methodologies for parallelization, so I knew there would be challenges here.

Shown below is an example visual output from the Rust implementation that my project aspired to emulate:



```
khai@khai-computer
(zsh) >_ pdu --align-left /usr                               192.168.43.18  09:38:13
[error] read_dir "/usr/share/polkit-1/rules.d": Permission denied (os error 13)
338.8M    ┌──include                                                          2%
188.5M    │ ┌──wine                                                           1%
328.4M    │ ├──dri                                                            2%
933.0M    ├─┴lib32                                                            6%
  2.7G    ├──bin                                                             17%
183.8M    │ ┌──help                                                           1%
228.9M    │ │ ┌──html                                                         1%
229.5M    │ ├─┴gtk-doc                                                        1%
291.0M    │ ├──doc                                                            2%
214.3M    │ │       ┌──mono                                                   1%
236.7M    │ │     ┌─┴lib                                                      1%
247.3M    │ │   ┌─┴wine-mono-6.1.1                                            2%
335.2M    │ │ ┌─┴mono                                                         2%
460.3M    │ ├─┴wine                                                           3%
722.6M    │ ├──locale                                                         5%
306.1M    │ │ ┌──noto-cjk                                                     2%
431.2M    │ │ │ ┌──OTF                                                        3%
  1.6G    │ │ │ ├──TTF                                                       10%
  2.0G    │ │ ├─┴nerd-fonts-complete                                         13%
  2.5G    │ ├─┴fonts                                                         16%
  5.6G    ├─┴share                                                         36%
177.2M    │ ┌──electron                                                       1%
180.9M    │ ├──wine                                                           1%
232.2M    │ ├──firefox                                                        1%
238.1M    │ ├──mono                                                           2%
187.2M    │ │ ┌──chromium                                                     1%
241.9M    │ ├─┴chromium                                                       2%
336.6M    │ ├──dri                                                            2%
284.4M    │ │ ┌──5.13.5-arch1-1                                               2%
371.9M    │ ├─┴modules                                                        2%
374.8M    │ ├──ghc-8.10.5                                                     2%
357.1M    │ │ ┌──site-packages                                                2%
408.2M    │ ├─┴python3.9                                                      3%
753.7M    │ ├──firmware                                                       5%
  6.2G    ├─┴lib                                                           39%
 15.8G  ┌─┴/usr                                                           100%

khai@khai-computer
(zsh) >_                                              192.168.43.18  09:39:10  4.13
```

**Sequential Traversal**
I first wrote a sequential directory traversal utility in order to have a control to benchmark my parallelization efforts against. The sequential traversal is rather straightforward: given a starting root directory, all files including subdirectories are parsed. Stat information for all files is collected and appended to a results list. Next, all subdirectories are isolated and prepared for traversal. The recursive traversal function is then called on these subdirectories, and the file statistic information for all recursive calls is appended to the final list upon return from each traversal function call. The final list of file information is then propagated back up to the original traversal call on the root directory. The implementation follows a fairly straightforward pre-order DFS traversal of the file system tree. It's my intention in the parallelization stage to support parallel traversal of each subtree after a certain depth.

```
traverseFS :: Int -> FilePath -> IO [FileInfo]
traverseFS depthVal path = do
 exists <- doesDirectoryExist path
 case exists of
   True -> do
     -- Get the contents of the directory
     dirContents <- listDirectory path

     -- Convert paths to file information, and get seperate list for sub-directories
     let filePaths = map (\fileName -> path ++ "/" ++ fileName) dirContents
     fileInfos <- getFileInfos filePaths depthVal
     let subDirsInfos = [subDir | subDir <- fileInfos, (isDir (stats subDir))]
     let subDirPaths = [(fPath subDir) | subDir <- subDirsInfos]

     -- Recursively explore the sub-directories and grab their [FileInfo]s
     overallResult <- concatMapM (traverseFS (depthVal + 1)) subDirPaths

     return (fileInfos ++ overallResult)
   False -> do
     -- FileInfo for this file already collected via parent directory, so ignore it
     return []
```

In order to hold the file statistics I foresaw as being useful, I defined my own data constructors for FileInfo and FileStats. These collectively contain file information such as the file path, the device the file is stored on, the depth from the root directory the file is located at, the size of the file, and a flag indicating if it is a directory or not.

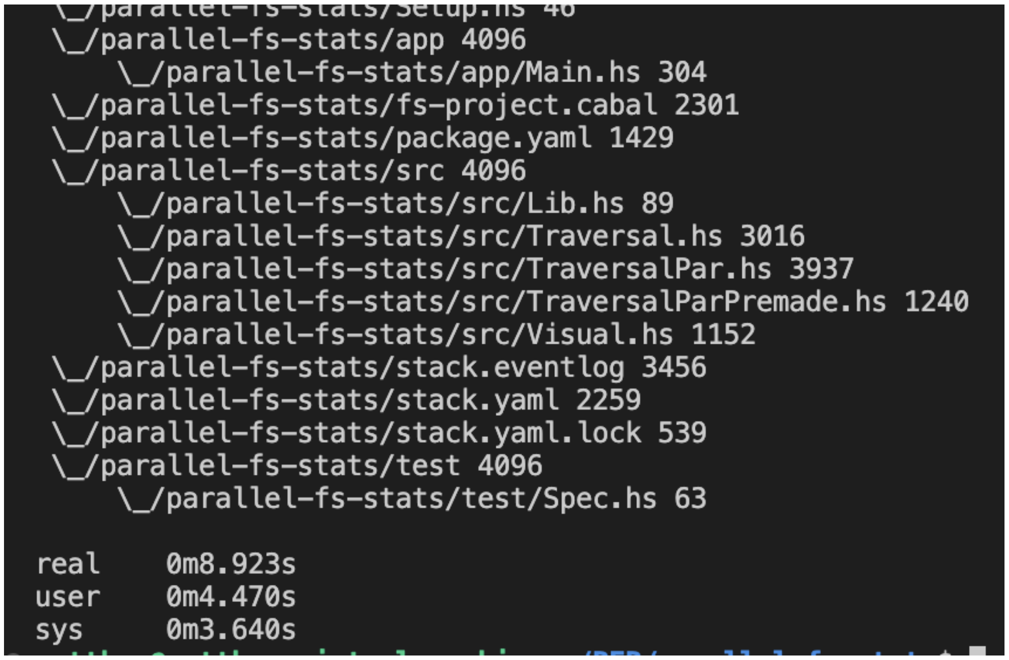
```
data FileInfo = FileInfo { fPath :: FilePath
                         , stats :: FileStats
                         , depth :: Int }

data FileStats = FileStats { devID :: DeviceID
                           , fSize :: FileOffset
                           , isDir :: Bool }
printFields :: [a -> String] -> a -> String
printFields functs object = intercalate " " $ map ($ object) functs
```

```haskell
instance Show FileStats where
 show = printFields [show . devID, show . fSize, show . isDir]

instance Show FileInfo where
 show = printFields [show . fPath, show . stats, show . depth]
```

The show type class for FileInfo and FileStats make it very convenient to adjust how each line for each file should display. These are directly used in my visualizer utility rather than defining separate visual representations.

The results from the sequential traversal function are then passed to the visualizer utility. The visualizer utility outputs one line for each encountered file based on its depth in the tree, file size, and file name. Although I did not spend as much time as I would have liked on beautifying the output, it serves as a solid foundation to build more functional visualizer functions such as statistic aggregation for each directory, a prettier tree output, and supporting barcharts for relative size percentages of the files contained in each subdirectory. For now, I’ve retained information about the entire path leading to a file as the lexicographical ordering of the absolute paths results in a proper tree output representing the file system. The removal of all except the file name from the actual output would likely improve the usefulness of the utility.

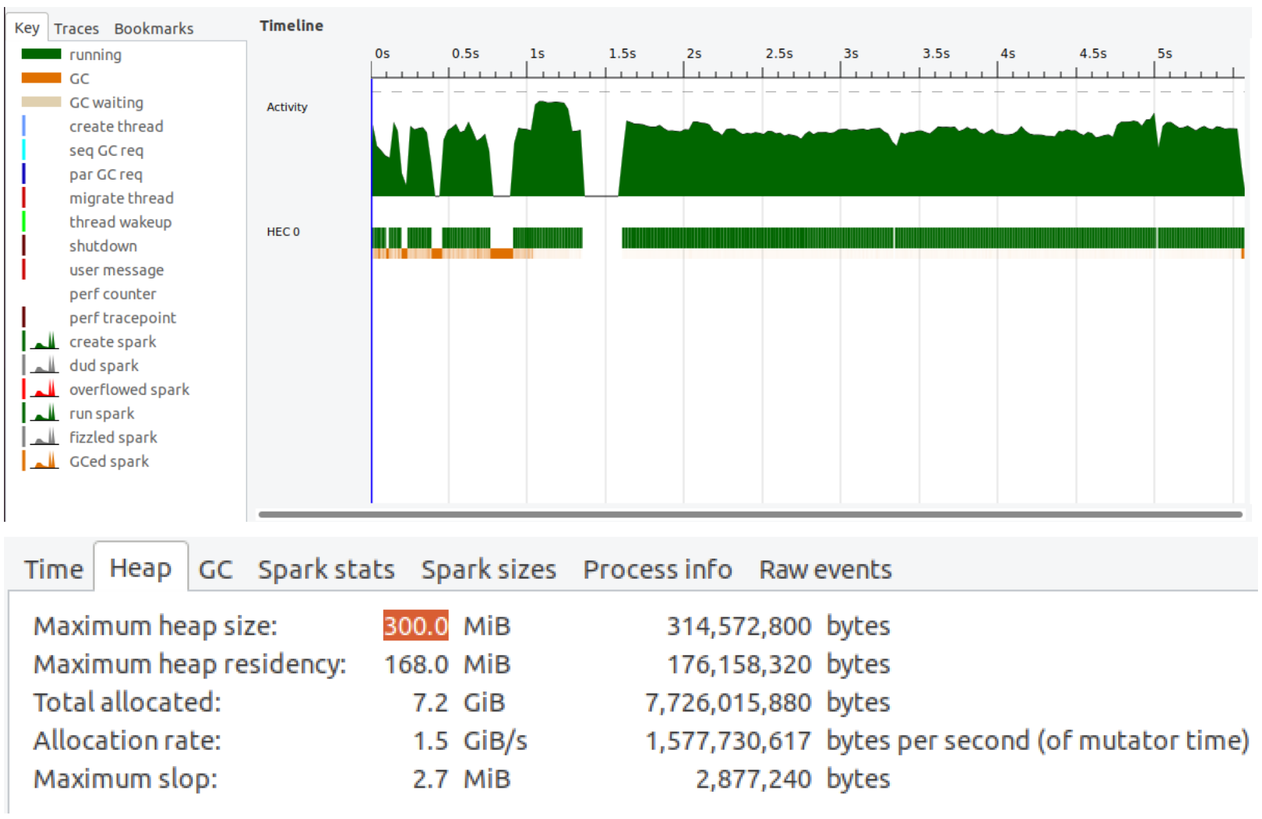
Shown below is an example from the tail end of the output after running a sequential traversal on the root directory of my final project:



```
  \_/parallel-fs-stats/Setup.hs 46
  \_/parallel-fs-stats/app 4096
      \_/parallel-fs-stats/app/Main.hs 304
  \_/parallel-fs-stats/fs-project.cabal 2301
  \_/parallel-fs-stats/package.yaml 1429
  \_/parallel-fs-stats/src 4096
      \_/parallel-fs-stats/src/Lib.hs 89
      \_/parallel-fs-stats/src/Traversal.hs 3016
      \_/parallel-fs-stats/src/TraversalPar.hs 3937
      \_/parallel-fs-stats/src/TraversalParPremade.hs 1240
      \_/parallel-fs-stats/src/Visual.hs 1152
  \_/parallel-fs-stats/stack.eventlog 3456
  \_/parallel-fs-stats/stack.yaml 2259
  \_/parallel-fs-stats/stack.yaml.lock 539
  \_/parallel-fs-stats/test 4096
      \_/parallel-fs-stats/test/Spec.hs 63

 real    0m8.923s
 user    0m4.470s
 sys     0m3.640s
```

**Sequential Benchmarks**

For all benchmarks across each implementation, I tested the traversal and output speed on a directory containing a fresh clone of the linux kernel and the root directory for this project's source code. The sequential implementation benchmarks serve as a baseline to judge the parallelized implementations against.



| | | | | |
|---|---|---|---|---|
| Maximum heap size: | 300.0 | MiB | 314,572,800 | bytes |
| Maximum heap residency: | 168.0 | MiB | 176,158,320 | bytes |
| Total allocated: | 7.2 | GiB | 7,726,015,880 | bytes |
| Allocation rate: | 1.5 | GiB/s | 1,577,730,617 | bytes per second (of mutator time) |
| Maximum slop: | 2.7 | MiB | 2,877,240 | bytes |

*Sequential traversal and visual output*

**Parallel Implementation**

Since the file system is organized in the structure of a tree, it stands to reason that we could parallelize the traversal of various subtrees since the contents of a directory are independent from the contents of another directory contained at the same depth and subsequent depths. In order to avoid excessive parallel traversals, it makes sense to parallelize only up to a certain depth, and then revert back to a sequential traversal. This approach places an upper bound on the number of parallel tasks that the implementation can generate in an attempt to better utilize the available resources of the machine

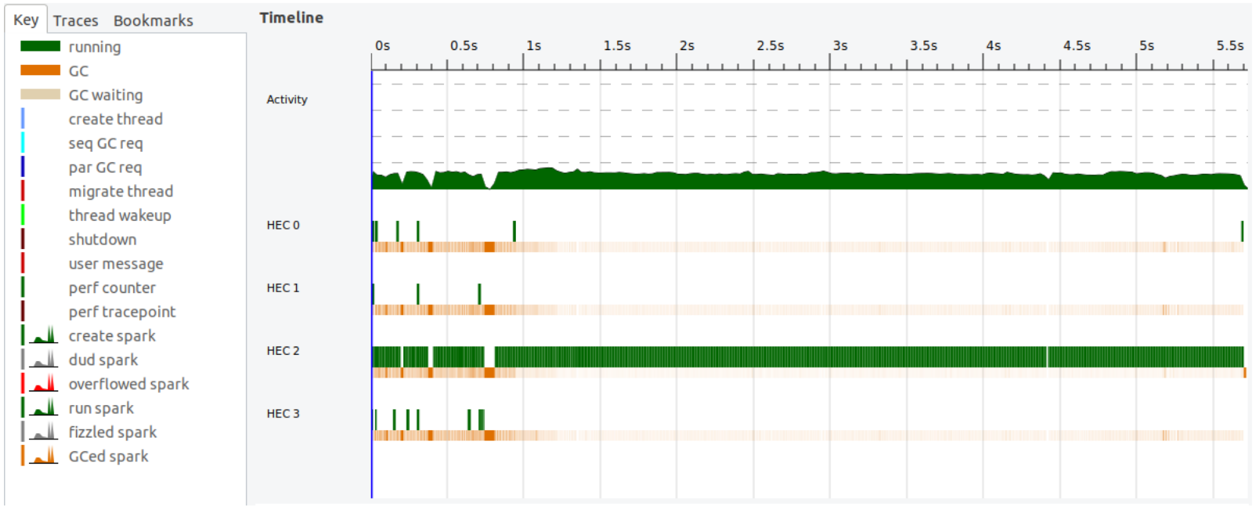
```
traverseFSPar :: Int -> FilePath -> IO [FileInfo]
traverseFSPar depthVal path = do
 exists <- doesDirectoryExist path
 case exists of
   True -> do
     -- Get the contents of the directory
     dirContents <- listDirectory path
     -- Convert paths to file information, and get seperate list for
sub-directories
     let filePaths = map (\fileName -> path ++ "/" ++ fileName) dirContents
     fileInfos <- getFileInfos filePaths depthVal
     let subDirsInfos = [subDir | subDir <- fileInfos, (isDir (stats subDir))]
     let subDirPaths = [(fPath subDir) | subDir <- subDirsInfos]

     -- Parallel traverse if above depth of 3 (0,1,2), sequential otherwise
     case (depthVal < 3) of
       True -> do
         let overallResultLists = parMap rpar (traverseFSPar (depthVal + 1))
subDirPaths
         overallResult <- consolIOLists overallResultLists
         return (fileInfos ++ overallResult)
       False -> do
         overallResult <- concatMapM (traverseFSPar (depthVal + 1)) subDirPaths
         return (fileInfos ++ overallResult)

   False -> do
     -- FileInfo for this file already collected via parent directory, so
ignore it
     return []
```

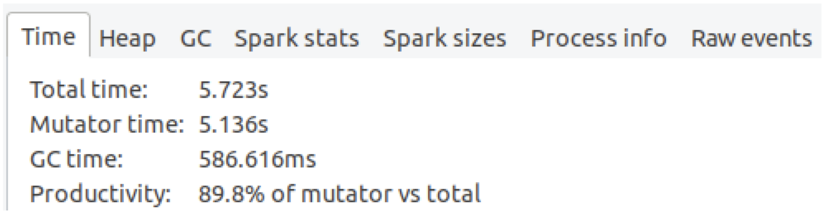
The main changes in the parallel traversal compared to the sequential traversal are contained in the “case (depthVal < 3) of” block and beyond. For the first 3 levels of directories after the root directory, the traversals of the subtrees are parallelized. Beyond this depth, the original sequential traversal methodology is used.

The intended power of this implementation comes from the parMap and rpar calls used when generating the overallResultLists which contain the FileInfo for all files located in the subdirectories. The rpar function creates a spark for the given argument, which is the traverseFSPar call, and parMap applies this to every single subdirectory in the list. The intention here is to traverse each subdirectory and its corresponding subtree in parallel until the depth limit is reached before reverting to the sequential traversal.

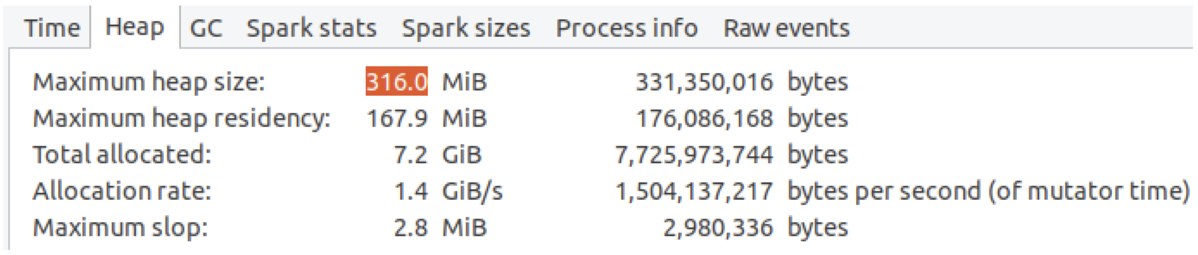
**Parallel Benchmarks**
After finally figuring out how to parallelize my tree traversal implementation, or so I thought, I was excited to run the benchmark and review the results.

Shown below are times and threadscope outputs for running the parallel traversal and visualizer utility on the same directory containing a fresh clone of the linux kernel and my project’s root directory.





```
Total time:      5.723s
Mutator time:    5.136s
GC time:         586.616ms
Productivity:    89.8% of mutator vs total
```



```
Maximum heap size:        316.0  MiB          331,350,016  bytes
Maximum heap residency:   167.9  MiB          176,086,168  bytes
Total allocated:            7.2  GiB        7,725,973,744  bytes
Allocation rate:            1.4  GiB/s      1,504,137,217  bytes per second (of mutator time)
Maximum slop:               2.8  MiB            2,980,336  bytes
```

"Parallel" Version 1, traversal and visualizer output

Unfortunately, my first attempt at a parallel traversal managed to have absolute no parallelization taking place. The execution time was nearly the same as the sequential implementation, and only one core was doing any productive work.

## Hypothesis on Parallel V1 Failure

Assuming we have a list of IO Actions that we want to complete, it makes sense that we would probably want to execute them sequentially as this preserves any possible read/write ordering to files. However, in this particular usecase, a stat() call to a file in one subdirectory has no effect on a stat() call to a file in another subdirectory yet the default behavior of IO Actions still forces them to execute sequentially. I believe this is the primary cause behind the observed results from my first parallel implementation.

Another possible cause with looking into is how the FileInfo data for each file is propagated back up through the recursive calls. Constantly appending to a list could be introducing a barrier to parallelism as we are constantly waiting for list construction to finish.

In either case, the first attempt at a parallel implementation likely does not properly manage the parallel and lazy forces of Haskell.

**Important Note on Caching**
The first traversal over a root directory is always going to be the one that takens the longest. In all of my benchmarks, I have made sure to run the sequential traversal prior to the benchmarked traversal whether it be another sequential traversal, a parallel V1 traversal, or a parallel V2 traversal. In my presentation, I highlight the substantial performance differences that can be observed between the first and subsequent runs of any traversal methodology. All of the results here account for the effects of caching in the operating system's file system implementation.

**Parallel Implementation: Version 2**
After my failure with the first parallelization attempt, I decided to restructure the problem into two areas: the parallelization of the file tree traversal and the parallelization of the collection of file statistics via stat() calls to the operating system.

In both cases, the sequential execution nature of IO actions possesses a challenge. I discovered multiple possible solutions for this issue, and I was able to achieve true parallelization in my implementation.

The Rust implementation that inspired this project uses a pool of worker threads to traverse subdirectories and collect file system information. I decided that this would be the best way forward to get around the issue of IO Actions executing sequentially.

For the parallelization of the stat() calls, the modified the file statistic collection function to be the following implementation:

```
getFileInfosPar :: [FilePath] -> Int -> IO [FileInfo]
getFileInfosPar filePaths fileDepth = forConcurrently filePaths ( \indivFilePath -> do
 indivFileStatus <- getFileStatus indivFilePath
 let toFileInfo :: FilePath -> FileStatus -> FileInfo
     toFileInfo filePath fileStatus = FileInfo { fPath = filePath
                                               , stats = FileStats { devID = (deviceID
fileStatus)
                                                                   , fSize = (fileSize
fileStatus)
                                                                   , isDir =
(isDirectory fileStatus)}
                                               , depth = fileDepth}
 return (toFileInfo indivFilePath indivFileStatus))
```

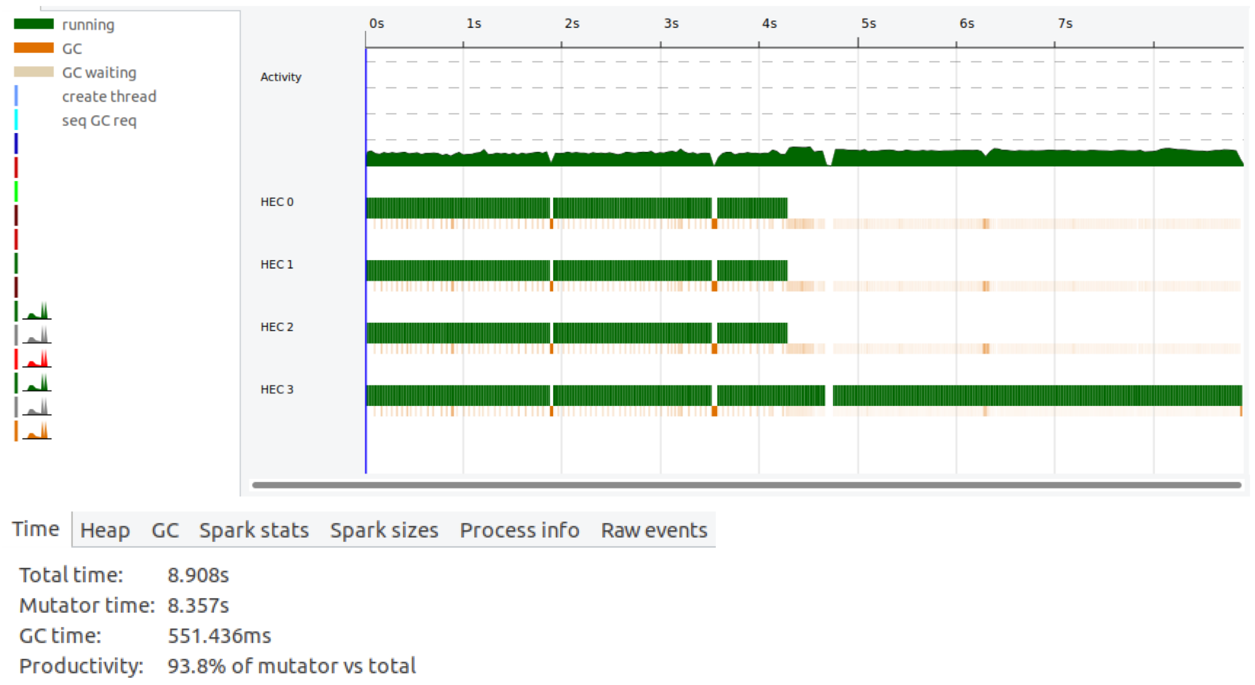
The forConcurrently function is provided in the Control.Concurrent.Async package. This function executes all of the given IO actions in parallel, and returns a convenient list of the results. This means for a given list of files in a directory, all of the stat() calls are able to run in parallel.

The modified getFileInfosPar function is integrated in a largely similar traversal implementation from before.

The Async package provides a layer of abstraction over the concurrency mechanisms provides in Haskell via the Control.Concurrent package. Async spawns threads which execute IO actions in a manner that protects against hidden exceptions and leaked threads. In using the Async package, the parallelization achieved in parallel version 2 of my implementation is closely related to the mechanism used by the Rust utility.

**Parallel Version 2 Benchmarks**

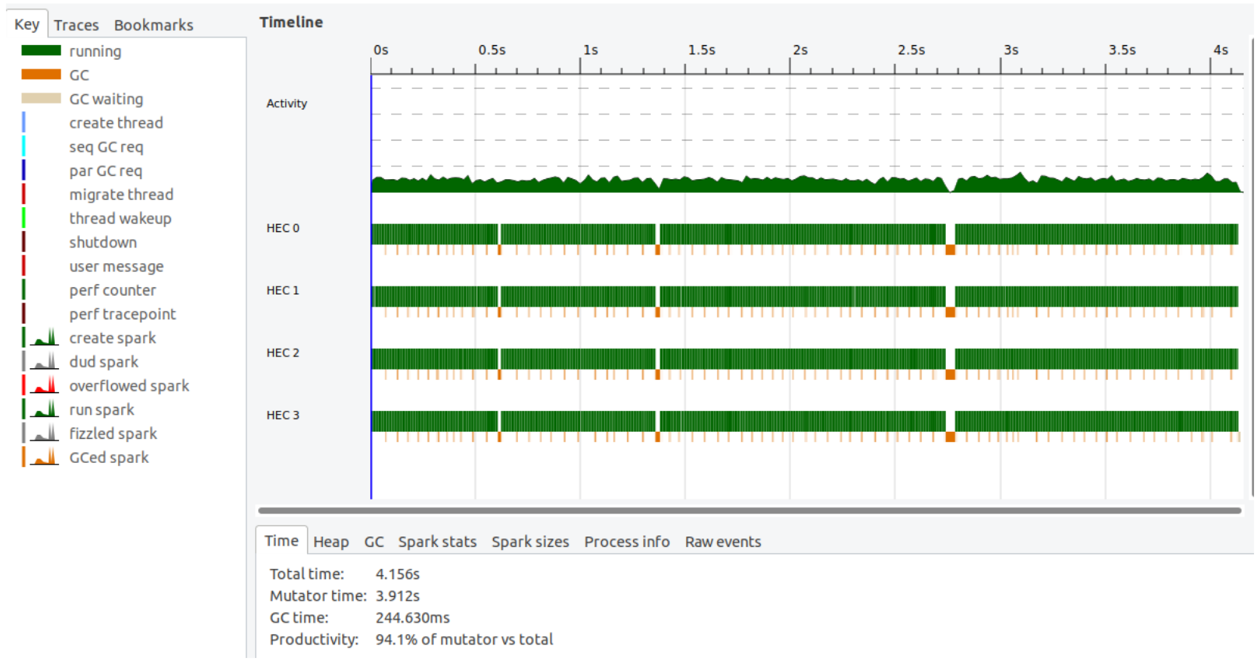
The work of outputting the visual representation of the file system statistics is unfortunately difficult to parallelize. To my knowledge, I have yet to see an appropriate mechanism by which any program outputs to stdout concurrently, so I've accepted this is a challenge I did not tackle in my implementation. With that said, the following benchmarks show both the threadscope output from a traversal with a visualizer output and an isolated traversal.



```
Total time:     8.908s
Mutator time:   8.357s
GC time:        551.436ms
Productivity:   93.8% of mutator vs total
```

*Parallel Version 2, traversal and visualizer output*

The distinct sections of work are evident in the above threadscope output. Although the file information collection is parallelized, the massive output to stdout for each file in the output tree still must run sequentially.



```
Total time:     4.156s
Mutator time:   3.912s
GC time:        244.630ms
Productivity:   94.1% of mutator vs total
```

Parallel Version 2, traversal and no visualization

Isolating only the file information collection and traversal, we can see that their is substantial parallelism among the stat() calls.

However, despite being blessed with parallelism on the second goaround, the performance of the parallel implementation has actually decreased substantially compared to sequential implementation. The cost of parallelism on 4 cores is approximately 1.6x as costly as running the sequential implementation.

| Implementation | 1 Core | 4 Cores |
|---|---|---|
| Sequential | | |
| Parallel V1 | | |
| Parallel V2 | | |

**Inspiration Environment vs Test Environment**

It's important to note that the environment these types of utility programs run on will play a role in how useful they are or not. In the instance of the forum discussion I referenced at the beginning of this report, being able to parallelize traversals across multiple disks on a large file server likely poses a much greater opportunity than being limited to a test environment involving only one disk. I'd imagine that executing the utility in parallel across multiple directories contained on separate disks would provide a much greater performance benefit.

**Future Work and Considerations**
Given then poor performance gains from parallelizing the stat() call interactions with the OS, I would be surprised if a parallelized directory traversal on a device containing only a single storage device would result in a speed gain compared to the sequential implementation counterpart. Parallization of the traversal would be accomplished to executing the IO Actions corresponding to each traverseFSParV2 call within a thread pool similar to the modified getFileInfosPar implementation.

**Visualizer Implementation**
The visualizer utility and traversal implementations can largely be treated as separate tasks. The visualization utility accepts input in the form of IO [FileInfo], and outputs a visual representation of the file tree. For each file, the file path and it's size are output. The data structures are setup though so that the devices the files are stored on could also be accounted for in the output.

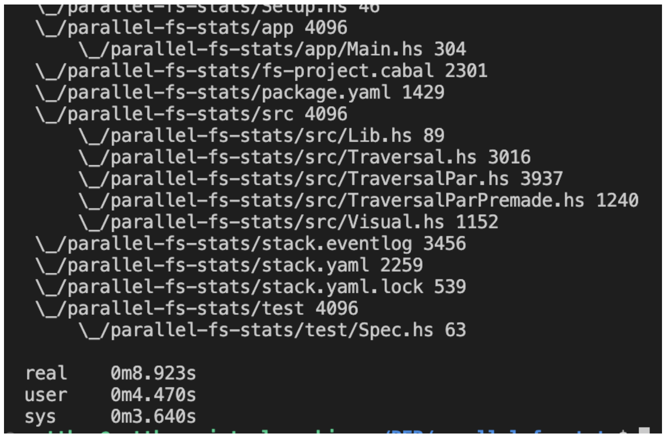
```
displayLines :: [FileInfo] -> String -> [String]
displayLines infoList prefix = strList
 where trimmedList = [x | x <- infoList]
       sortedList = sortBy (\x y -> compare (show (fPath x)) (show (fPath y)))
trimmedList
       strList = [(getElegantOutput x prefix) | x <- sortedList]

getElegantOutput :: FileInfo -> String -> String
getElegantOutput info prefix = (getPrefix info (depth info) (depth info)) ++
(rmPrefix (prefix) (fPath info)) ++ " " ++ (show $ fSize (stats info))

getPrefix :: FileInfo -> Int -> Int -> String
getPrefix info currDepth initDepth
 | currDepth == 0 = " "
 | currDepth == initDepth = (getPrefix info (currDepth - 1) initDepth) ++ "\\_"
 | otherwise = (getPrefix info (currDepth - 1) initDepth) ++ "    "

--
https://stackoverflow.com/questions/18554083/haskell-program-to-remove-part-of-
list-and-print-the-rest
rmPrefix :: Eq a => [a] -> [a] -> [a]
rmPrefix [] ys = ys
rmPrefix _ [] = []
rmPrefix xs ys =
 if xs == take n ys
 then drop n ys
 else ys
 where n = length xs
```

Since the lexicographical ordering of the absolute file paths results in the correct line ordering for outputting the file tree, the visualization utility uses the absolute path to obtain a proper ordering for each file-line output.

**Conclusion**
Although I successfully parallelized the IO Actions for collecting file stats, I was not able to achieve a truly parallel traversal of the directories itself. I believe that this utility would be best applied on a larger file system where independent devices would provide more fertile group for performance gains due to parallelism

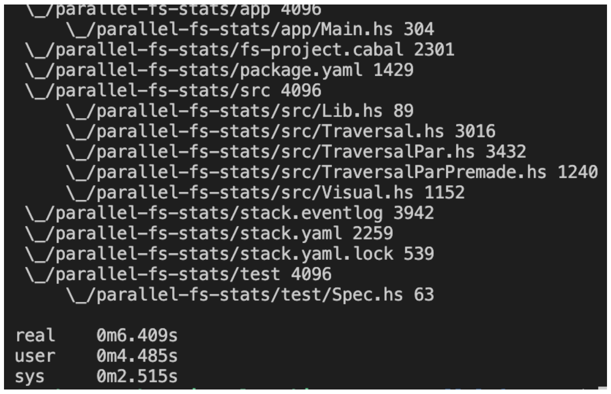
**Appendix**
Source Code:
https://github.com/nelsonm2991/parallel-fs-stats

Performance Comparison: Caching
Below are outputs from a first run and second run of the sequential traversal to show the effect caching has on the performance of subsequent traversals.



```
  \_/parallel-fs-stats/Setup.hs 46
  \_/parallel-fs-stats/app 4096
      \_/parallel-fs-stats/app/Main.hs 304
  \_/parallel-fs-stats/fs-project.cabal 2301
  \_/parallel-fs-stats/package.yaml 1429
  \_/parallel-fs-stats/src 4096
      \_/parallel-fs-stats/src/Lib.hs 89
      \_/parallel-fs-stats/src/Traversal.hs 3016
      \_/parallel-fs-stats/src/TraversalPar.hs 3937
      \_/parallel-fs-stats/src/TraversalParPremade.hs 1240
      \_/parallel-fs-stats/src/Visual.hs 1152
  \_/parallel-fs-stats/stack.eventlog 3456
  \_/parallel-fs-stats/stack.yaml 2259
  \_/parallel-fs-stats/stack.yaml.lock 539
  \_/parallel-fs-stats/test 4096
      \_/parallel-fs-stats/test/Spec.hs 63

 real    0m8.923s
 user    0m4.470s
 sys     0m3.640s
```

*First run*



```
 \_/parallel-fs-stats/app 4096
     \_/parallel-fs-stats/app/Main.hs 304
 \_/parallel-fs-stats/fs-project.cabal 2301
 \_/parallel-fs-stats/package.yaml 1429
 \_/parallel-fs-stats/src 4096
     \_/parallel-fs-stats/src/Lib.hs 89
     \_/parallel-fs-stats/src/Traversal.hs 3016
     \_/parallel-fs-stats/src/TraversalPar.hs 3432
     \_/parallel-fs-stats/src/TraversalParPremade.hs 1240
     \_/parallel-fs-stats/src/Visual.hs 1152
 \_/parallel-fs-stats/stack.eventlog 3942
 \_/parallel-fs-stats/stack.yaml 2259
 \_/parallel-fs-stats/stack.yaml.lock 539
 \_/parallel-fs-stats/test 4096
     \_/parallel-fs-stats/test/Spec.hs 63

real    0m6.409s
user    0m4.485s
sys     0m2.515s
```

*Second run*