Jakub Slowinski 16319781

Assignment 1:
Report on parallelism of 2 sorting algorithms

Introduction:
I commenced the assignment by reading the Microsoft pdf at https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/parallel_haskell2.pdf .
I used the code included in the Microsoft slides page 13 as a starting point.
It gave me the implementation for `forceList`, which forces evaluation of a list as well as `secDiff` which makes a float from the comparison of 2 system times.

I used Haskell stack and therefore have my code split between app/Main.hs and src/Functions.hs. I will submit the 2 main file with this report along with the entire project folder as a zip file.

In the package.yaml I specified GHC's command-line flags
ghc-options:
- -threaded
- -rtsopts
- -eventlog
- -with-rtsopts=-N

I also experimented with the -O2 flag, which enables level 2 optimisation. The testing in the algorithms sections didn't contain this flag.

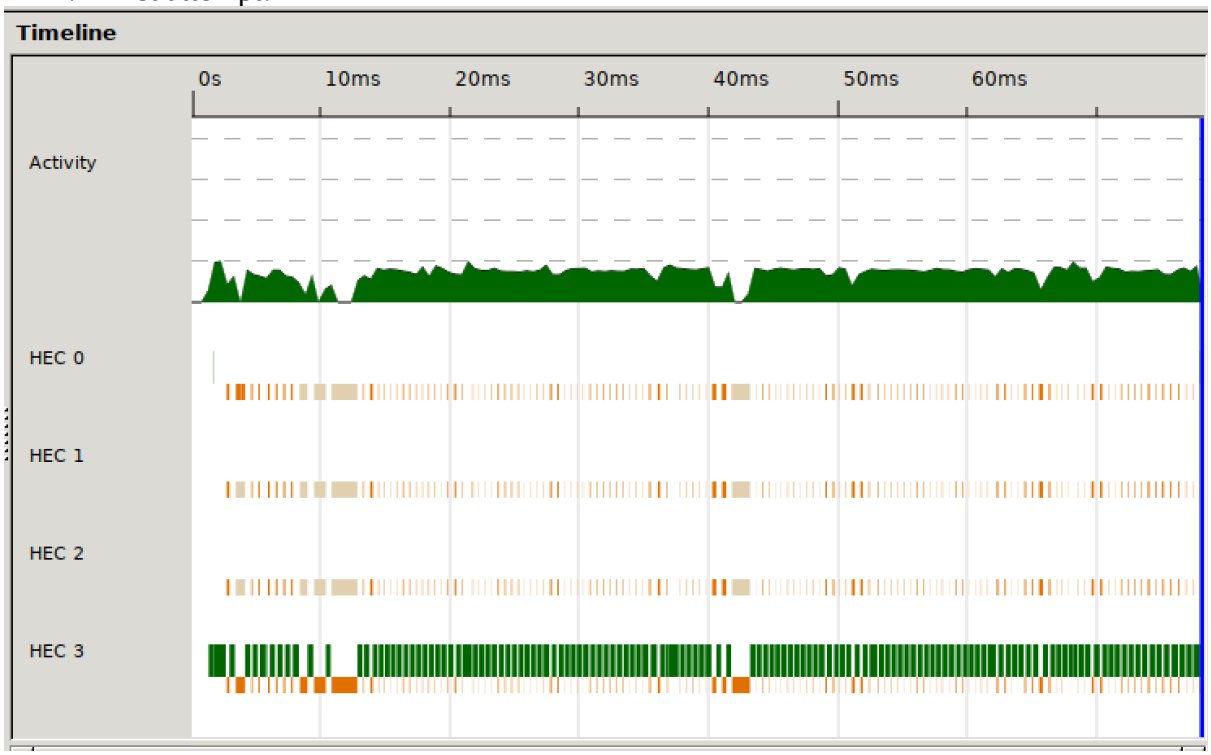
Command line arg:
I used `stack ghci` for quick testing(doesn't work with -threaded)
I used `stack build` to build the executable(and the test suite which I didn't make use of).
Executed using `stack exec -- assignment1-exe +RTS -N8 -ls `, the number beside N was changed depending on how many cores were used in that execution.
This created a .eventlog file which I analysed in threadscope.
The Main.hs sums the result of the algorithm in order to force the evaluation. This sum is then outputted for error checking along with the time taken to perform the sort.

Threadscope:
Threadscope was used on the lunbuntu VM provided, dragged assignment1-exe.EXE.eventlog to the desktop and opened with threadscope. It was very simple to use on a unix system. This report contains numerous snapshots of the logs passed into threadscope.

Algorithms:
I decided to code and then parallelise and optimise quicksort + merge sort as recommended. These are 2 recursive divide and conquer algorithms which naturally lend themselves to be parallelised.
I initially started with merge sort as the implementation was already known to me.
Below are the steps I took in order to achieve parallelism; the result can be seen in the algorithm conclusion section.

Quicksort:

1. First attempt:

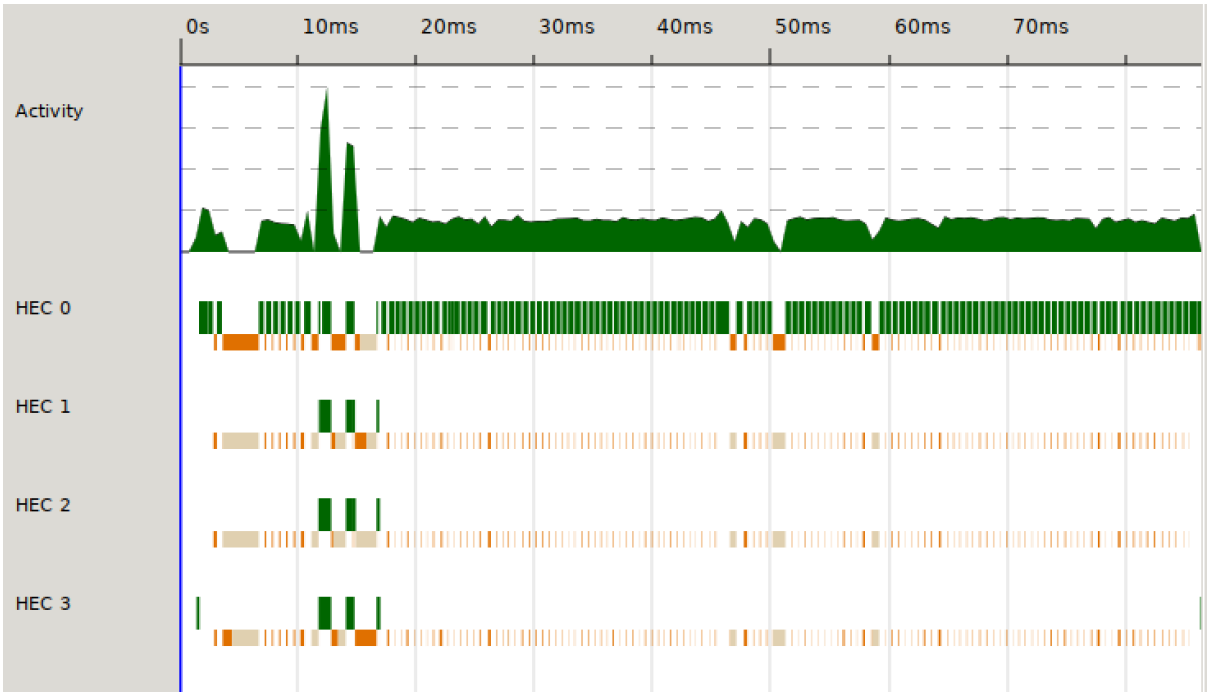


The cores 0,1,2 were just used for garbage collection.
The program was initially started and finished in core 0.

2. Using (sortLow `par` sortHi) `pseq` (sortLow ++ x : (hi `pseq` sortHi))

   Quick slow 9.3702994e-2 this time, but on average, with 4 cores, beats the sequential quicksort.
   If using 8 cores(and changing nCores=8 in code Main.hs of course)
   The algorithms nCores goes to 0 then only 1 core will perform the rest sequentially.

   $ stack exec -- assignment1-exe +RTS -N8 -ls
   Sum of quicksort: 998782
   Time to quicksort: 7.8106e-2
   Sum of quicksort w/ parallelism: 998782
   Time to quicksort w/ parallelism: 7.810795e-2



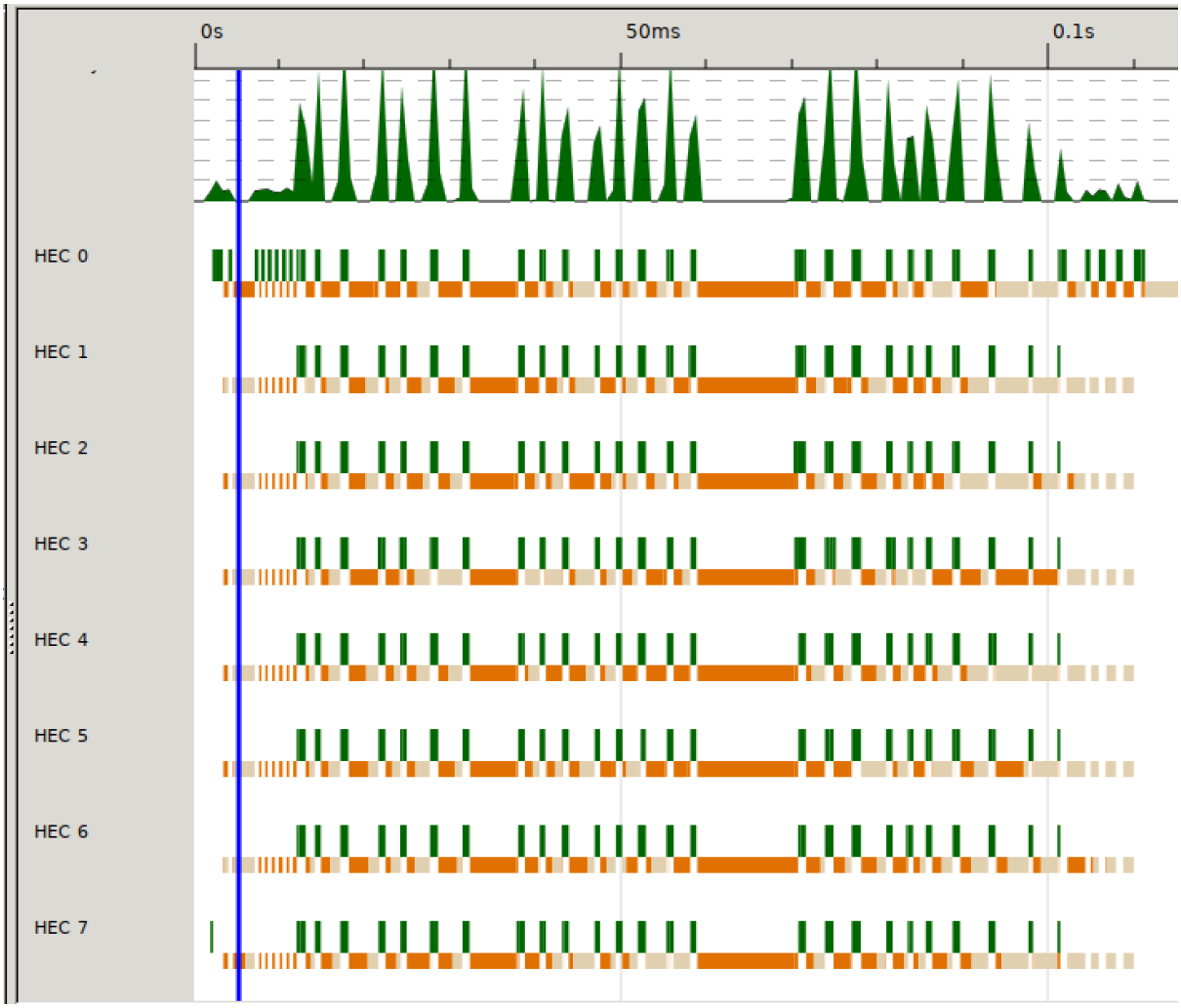
3. Attempt at telling the function how many cores were needed

```
quicksortPar :: Ord a => Int -> [a] -> [a]
quicksortPar _ [] = []
quicksortPar _ [x] = [x]
quicksortPar 0 x = squicksort x
quicksortPar nCores (x:xs) = (sortLow `par` sortHi) `pseq` (sortLow ++ x : (hi `pseq` sortHi))
        where
        lo = [y | y <- xs, y < x]
        hi = [y | y <- xs, y >= x]
        sortLow = quicksortPar (nCores-1) lo
        sortHi = quicksortPar (nCores-1) hi
```

With commented out: -- quicksortPar 0 x = squicksort x
$ stack exec -- assignment1-exe +RTS -N8 -ls
Sum of quicksort w/ parallelism: 998782
Time to quicksort w/ parallelism: 0.101487

With: quicksortPar 0 x = squicksort x
$ stack exec -- assignment1-exe +RTS -N8 -ls
Sum of quicksort w/ parallelism: 998782
Time to quicksort w/ parallelism: 8.6893e-2

Quite a lot slower with the line commented out
Having a look at threadscope:



The cores are much more active when you never force it to do quicksort sequentially, but so is the garbage collector
In my opinion, the parallelism fails here as all the cores are occupied and each core is waiting for a core to be free in order to perform computations on another core. This looks like a form of deadlock as they all look like they are executing at the same time and deadlocking at the same time.

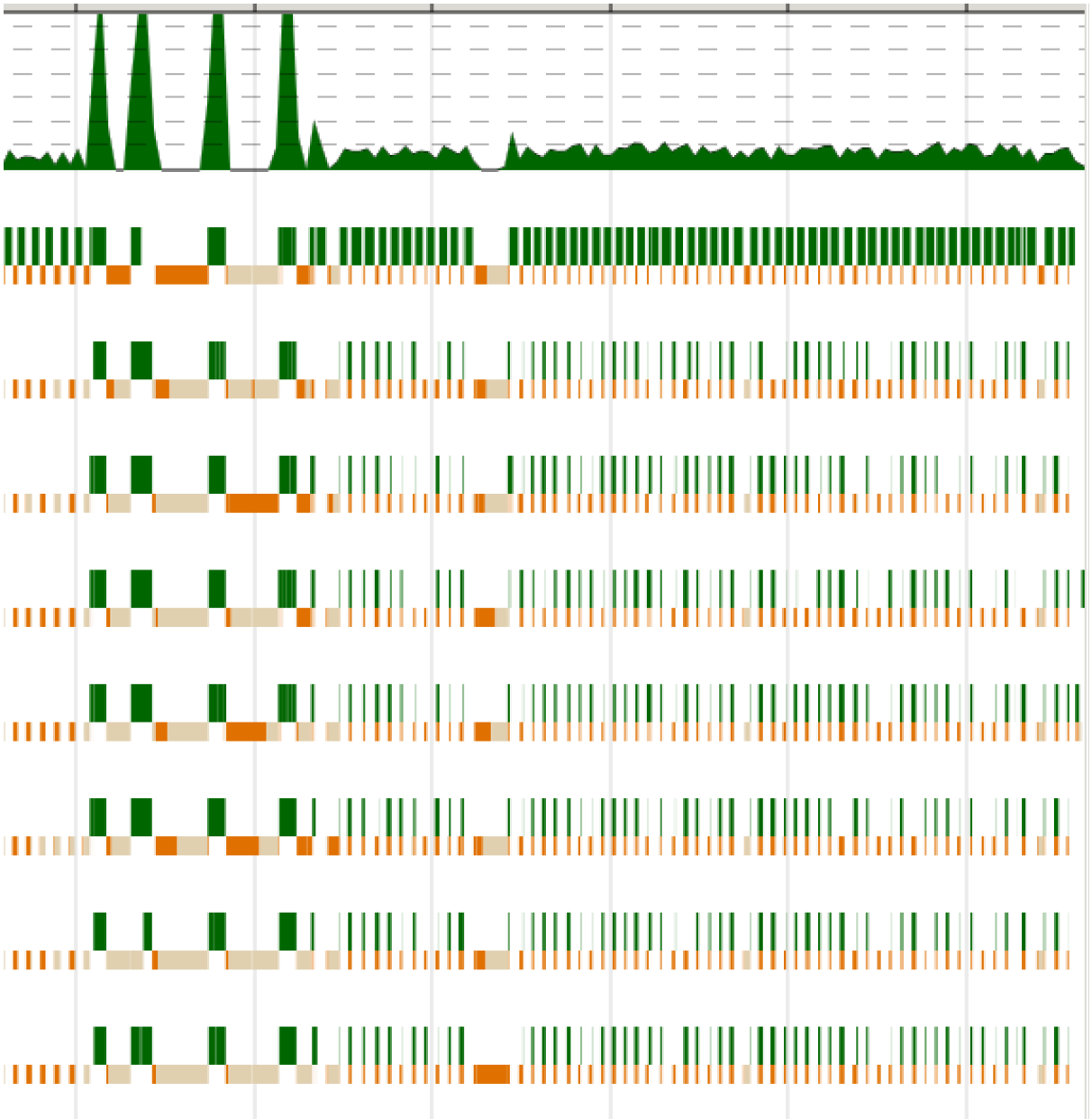
4. Attempt at never letting it go sequential:
   I decided that there must be n, a counter of how many cores left.
   When n goes to 0, run sequentially then make n=1

```
$ stack exec -- assignment1-exe +RTS -N8 -ls
Sum of quicksort2 w/ parallelism: 665344
Time to quicksort2 w/ parallelism: 9.1469e-2
```

```
quicksortPar :: Ord a => Int -> [a] -> [a]
quicksortPar _ [] = []
quicksortPar _ [x] = [x]
-- best so far 6 - got under 5 shit:3
quicksortPar 0 (x:xs) = losort ++ x : hisort
        where
        losort = quicksortPar 0 [y | y <- xs, y < x]
```

```
            hisort = quicksortPar 4 [y | y <- xs, y >= x]
-- quicksortPar 0 xs = squicksort xs
quicksortPar nCores (x:xs) = (sortLow `par` sortHi) `pseq` (sortLow ++ x : (hi `pseq` sortHi))
                where
                lo = [y | y <- xs, y < x]
                hi = [y | y <- xs, y >= x]
                sortLow = quicksortPar nminus lo
                sortHi = quicksortPar nminus hi
                nminus = nCores - 1
```

With 8 cores:



```
$ stack exec -- assignment1-exe +RTS -N8 -ls
Sum of quicksort2 w/ parallelism: 529952
Time to quicksort2 w/ parallelism: 8.6607e-2
```

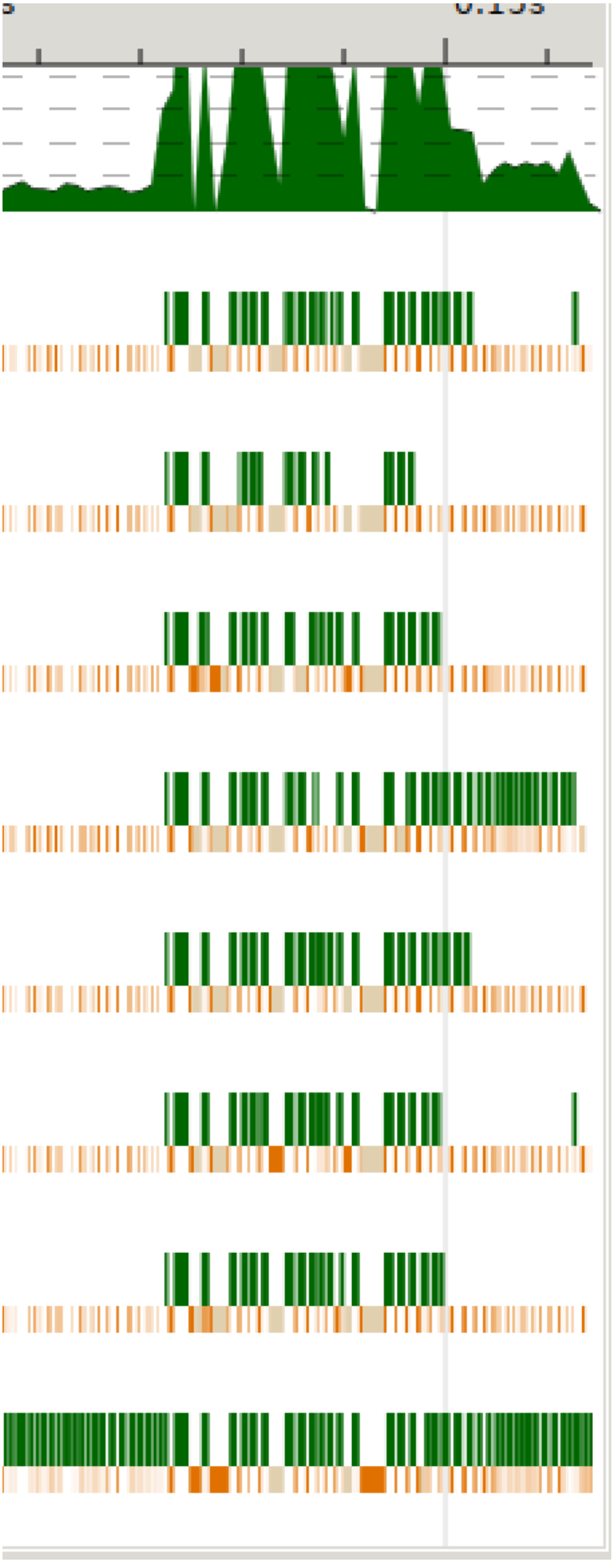
Looks good right?
This solution turned out to be regressive, so I ended up reverting to the previous code.

5. Eventually I decided to use the forceList function from the Microsoft slides (already being used in the main) in quicksort

   It turned out to be rapid and by far my best solution.
   The threadscope for 8 cores can be seen to the right.

   This parallel solution cuts the time of the sequential solution by a magnitude of 3.
   This is the version of quicksort included with my final code.

   $ stack exec -- assignment1-exe +RTS -N8 -ls
   Sum of quicksort: 998782
   Time to quicksort: 0.108841
   Sum of quicksort w/ parallelism: 998782
   Time to quicksort w/ parallelism: 4.1968003e-2



Merge sort:

1. First attempts at merge sort w/ parallelism:

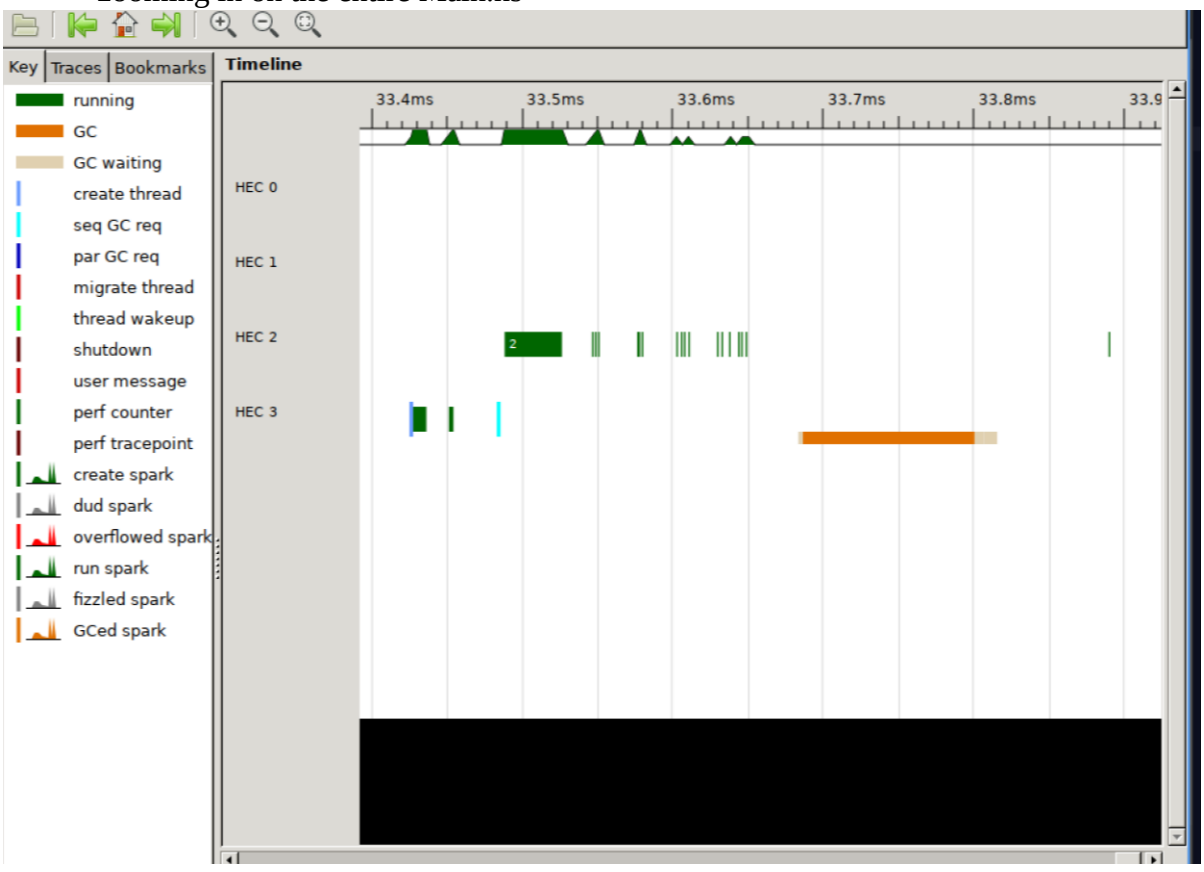
```
-- Non-sequential merge sort TODO::
mergesortPar :: Ord a => [a] -> [a]
mergesortPar [] = []
mergesortPar [x] = [x]
mergesortPar x = nf `par` merge2Lists nf (secondHalf `pseq`mergesortPar secondHalf)
        where
        firstHalf = take ((length x) `div` 2) x
        secondHalf = drop ((length x) `div` 2) x
        nf = (firstHalf `pseq` mergesortPar firstHalf)
```

   Results = 2x as slow as sequential (not worth even showing them here)

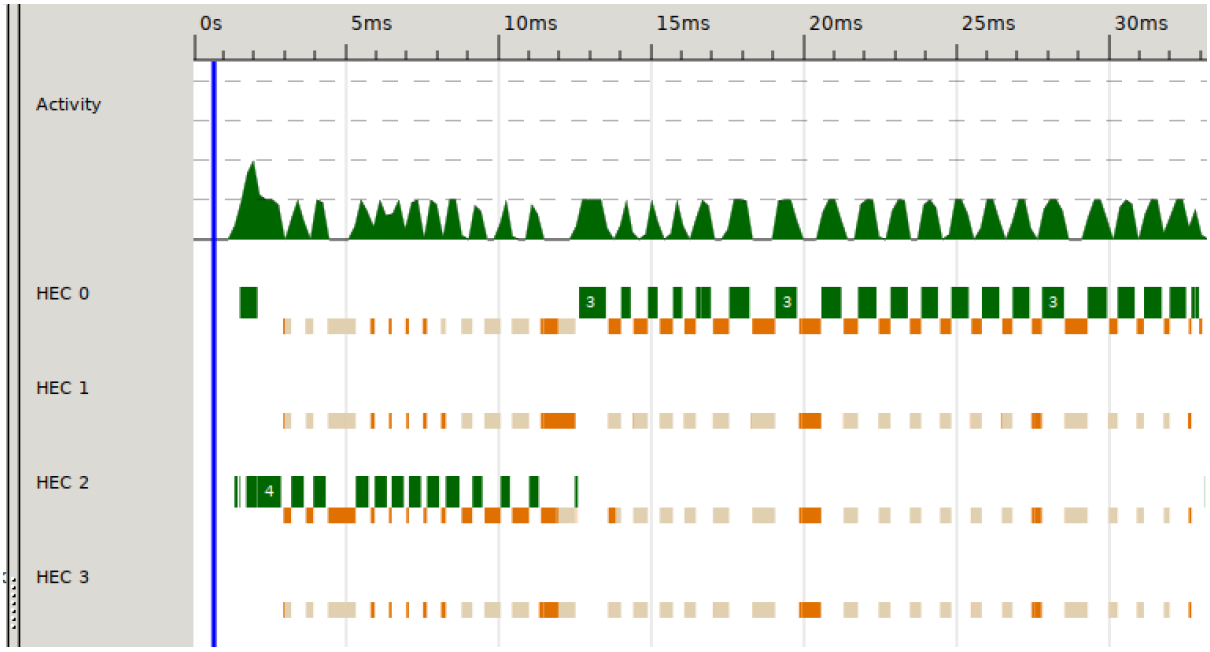
2. First showings of parallelism:
   A merge sort implemented similarly to the Fibonacci example from the lecture.

```
mergesortPar :: Ord a => Int -> [a] -> [a]
mergesortPar _ [] = []
mergesortPar _ [x] = [x]
mergesortPar 0 x = smergesort x
mergesortPar n x = par secondHalf (pseq secondHalf (merge (firstHalf) (secondHalf)))
        where
        firstHalf = mergesortPar (n-1) (take ((length x) `div` 2) x)
        secondHalf = mergesortPar (n-1) (drop ((length x) `div` 2) x)
```

   zooming in on the entire Main.hs

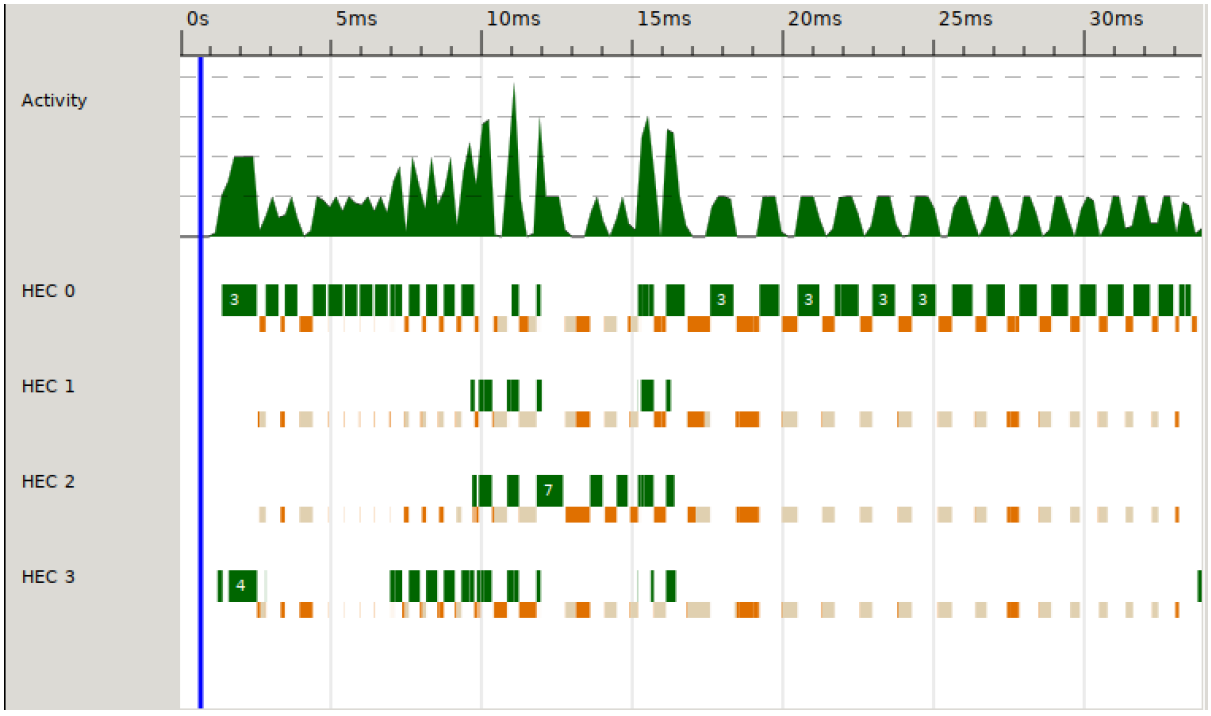


ran w/out any prints or timings(now has 2 cores in use …)



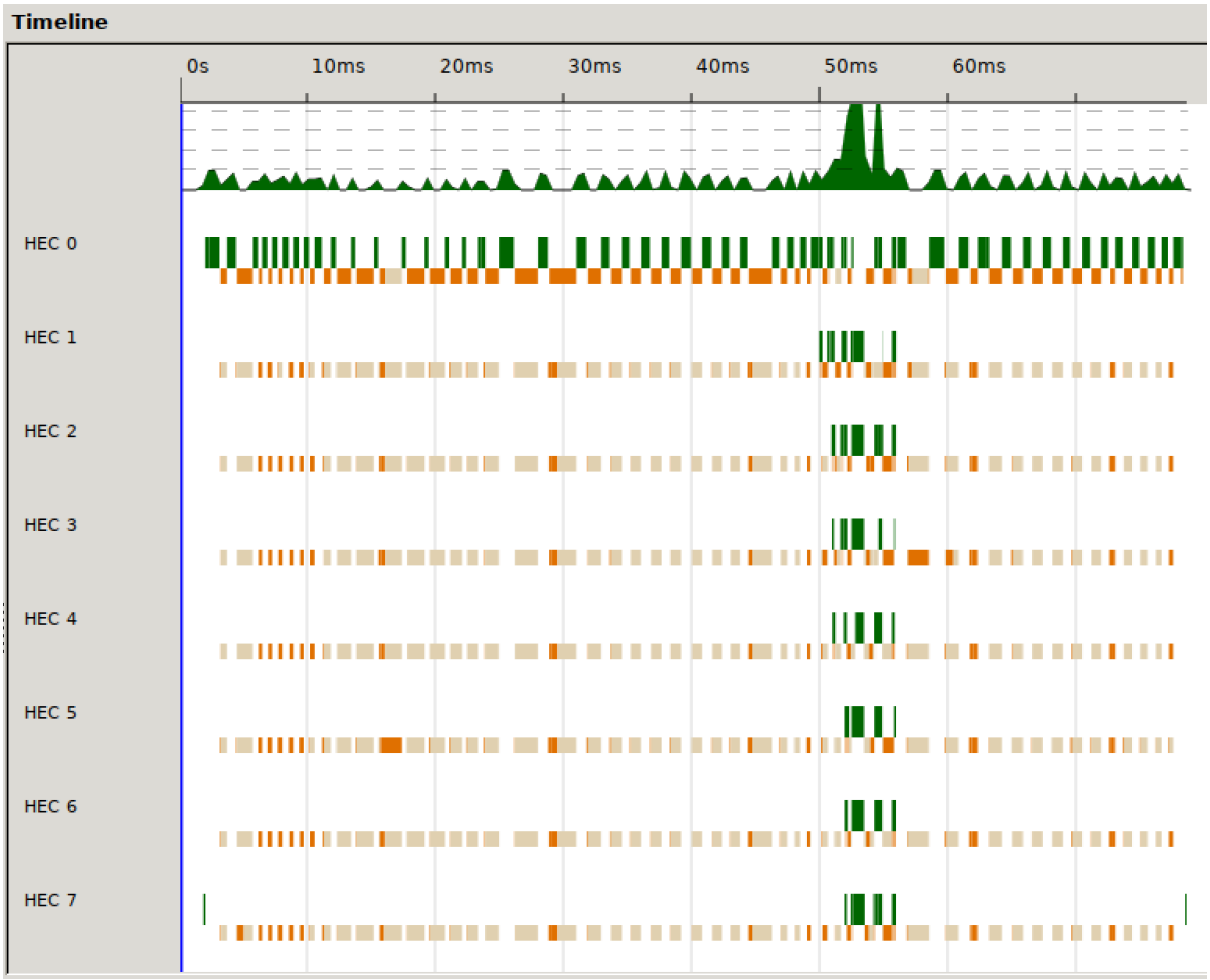
3. Success with 4 cores

```
mergesortPar :: Ord a => Int -> [a] -> [a]
mergesortPar _ [] = []
mergesortPar _ [x] = [x]
mergesortPar 0 xs = smergesort xs
mergesortPar n xs = (firstHalf `par` secondHalf) `pseq` (merge firstHalf secondHalf)
        where
        (left, right) = splitAt ((length xs) `div` 2) xs
        firstHalf = mergesortPar (n-1) left
        secondHalf = mergesortPar (n-1) right
```



Running the original sequential merge sort and the same parallel merge sort on 8 cores, the sequential version is visible on left side and the non-sequential can be seen in the cores utilised on the right side.

```
$ stack exec -- assignment1-exe +RTS -N8 -ls
Sum of mergesort: 998782
Time to mergesort: 3.901e-2
Sum of mergesort w/ parallelism: 998782
Time to mergesort w/ parallelism: 2.8303e-2
```



After I reached this point I decided to implement quicksort (the algorithm write up wasn't in order until now). After working on quicksort and making it as parallel as possible I realised that merge sort is much quicker even with less core utilisation.

4. Using the forceList strategy from my final quicksort implantation:

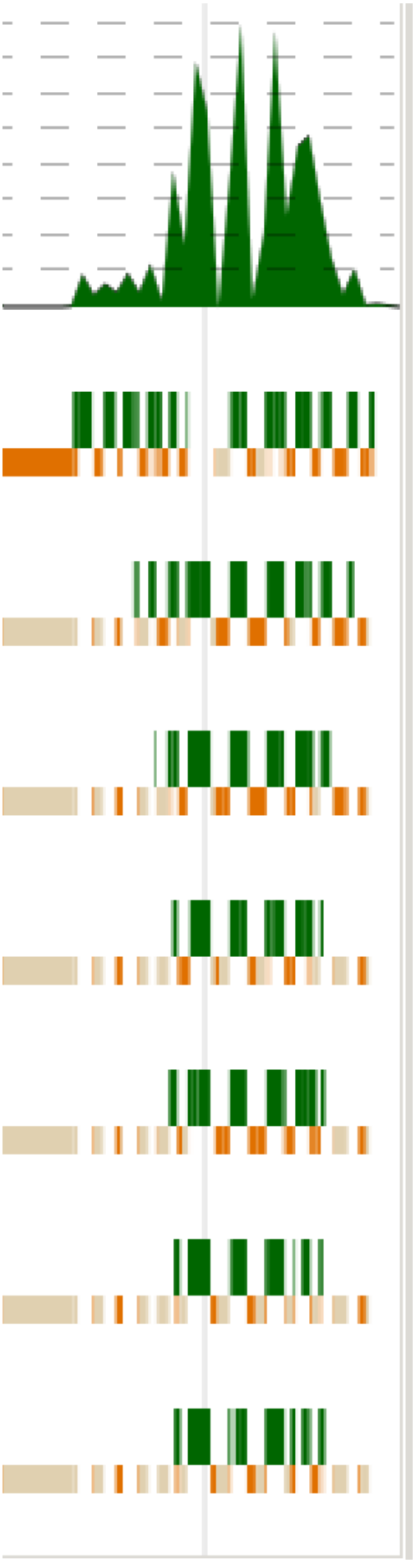
```
$ stack exec -- assignment1-exe +RTS -N8 -ls
Sum of mergesort: 998782
Time to mergesort: 5.0295003e-2
Sum of mergesort w/ parallelism: 998782
Time to mergesort w/ parallelism: 1.0746e-2
```

Very quick, 5 times faster than the unoptimised implementation with no parallelism.

The threadscope is to the right, you can see from the small width that this algorithm is very quick and efficient.

Can get quicker than 9.752001e-3 at times, depending on random list input.

This was my final version of merge sort which can be seen in my final code submission in Functions.hs.

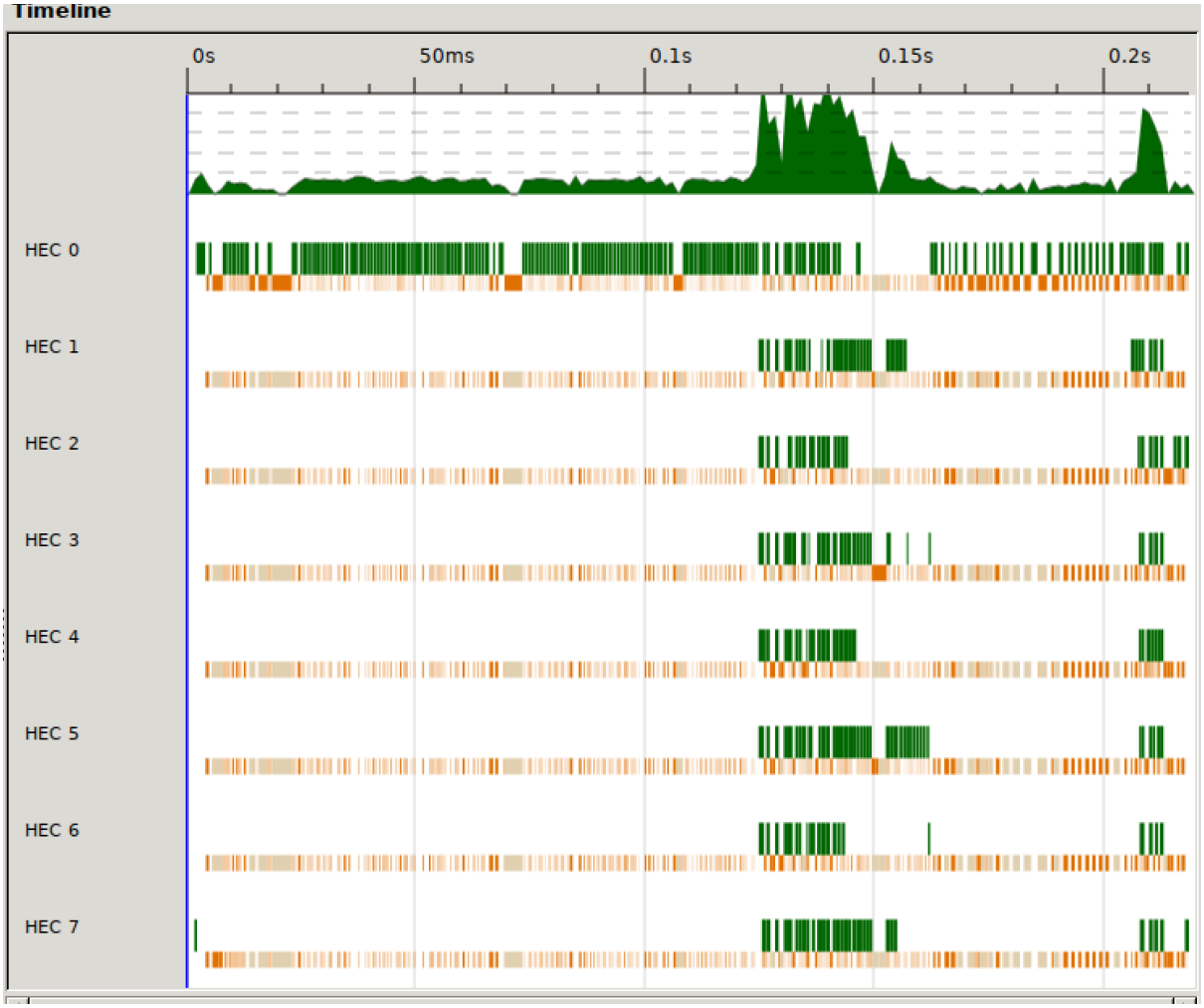


Algorithm conclusion:

Output of the 4 algorithms running one after the other on 8 cores:

```
$ stack exec -- assignment1-exe +RTS -N8 -ls
Sum of quicksort: 998782
Time to quicksort: 0.110311
Sum of quicksort w/ parallelism: 998782
Time to quicksort w/ parallelism: 3.7124e-2
Sum of mergesort: 998782
Time to mergesort: 4.3896e-2
Sum of mergesort w/ parallelism: 998782
Time to mergesort w/ parallelism: 1.1714e-2
```

The parallel quicksort can be seen in the middle and the parallel merge sort can be seen on the very right.

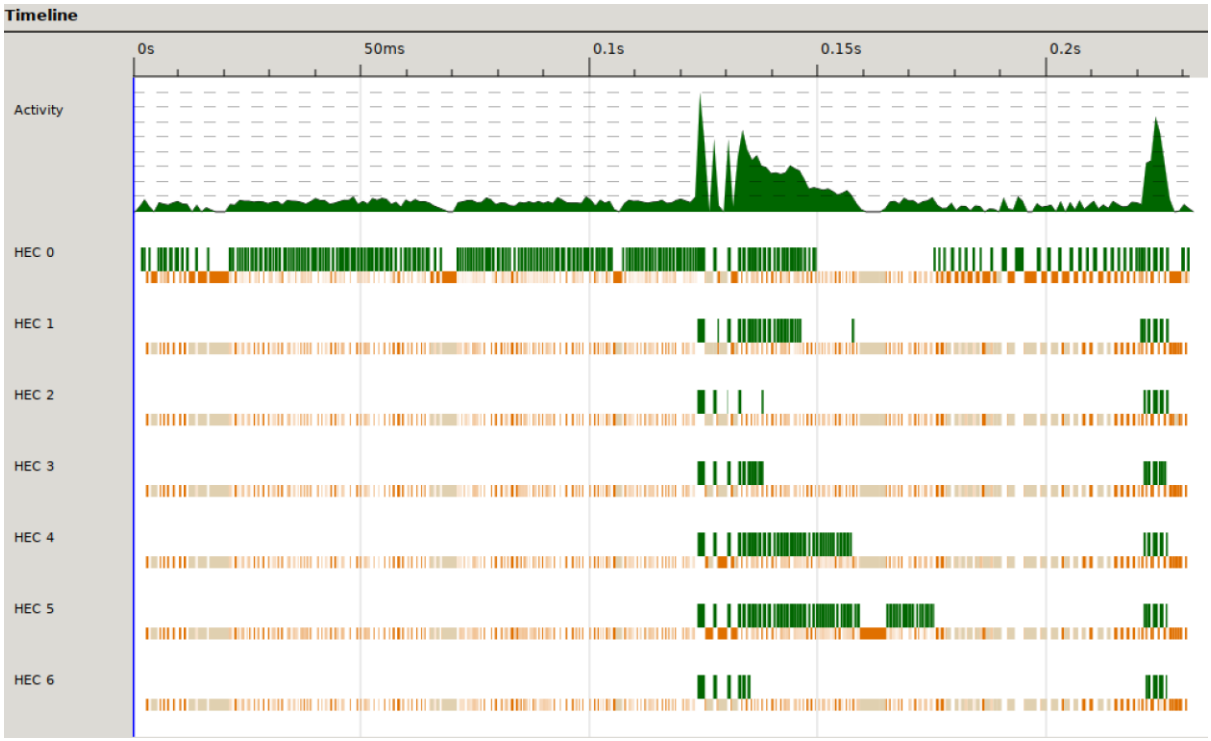


As you can see from the image quicksort takes longer than merge sort even unparallised, but also the initial pseudo-random list of pseudo-random numbers took some time.

Running with -O2 flag:

Execution with -O2 enabled:

$ stack exec -- assignment1-exe +RTS -N8 -ls
Sum of quicksort: 998782
Time to quicksort: 0.111494996
Sum of quicksort w/ parallelism: 998782
Time to quicksort w/ parallelism: 5.2722e-2
Sum of mergesort: 998782
Time to mergesort: 4.4891e-2
Sum of mergesort w/ parallelism: 998782
Time to mergesort w/ parallelism: 1.0743001e-2



I theorise that -O2, on average, makes my quicksort marginally slower, but I am not able to prove this.

Observations:

A pseudo random number generator generates the list to be sorted. The more sorted the list, the quicker the algorithms will be. I believe that this causes the spread of time from one execution to another.

Merge sort is much more suited to parallelism in my opinion as it achieves a bigger speed-up from the sequential version.

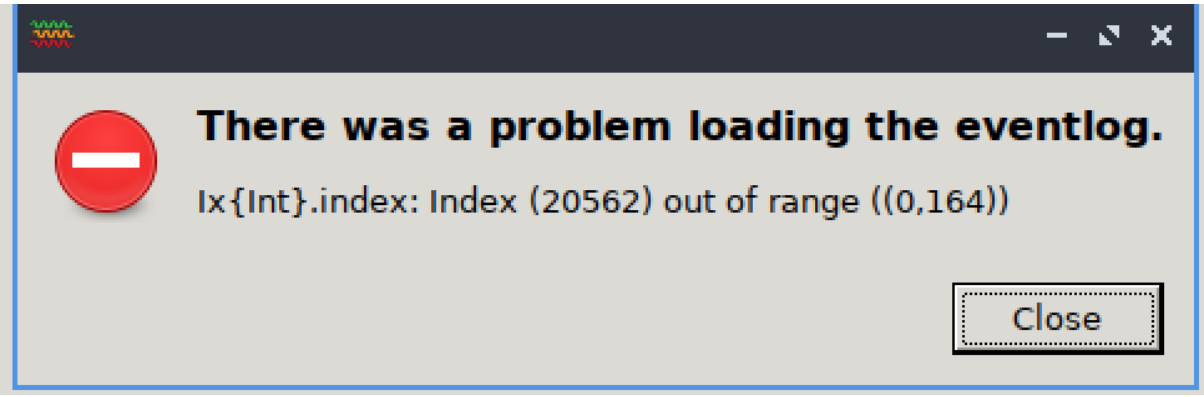
Errors & bugs & miscellaneous

- Exec stack needs -- before executable in order to pass in parameters

e.g. stack exec -- assignment1-exe +RTS -N4 -ls

- If you already have an existing .eventlog and you execute the program e.g. by running command: stack exec -- assignment1-exe +RTS -N4 -ls

, then try to open said eventlog with threadscope, it causes this error:



- I failed to install threadscope on windows even though I spent 5 hours attempting to do so, I was fortunately saved by Glenn's VM.
- At one stage I introduced a bug into my quicksort which removed 1/3 elements as it traversed the array. I had already started to document that solution, so I went down a bit of a rabbit hole. It made my solution pre-forceList roughly 50% faster but obviously it was faulty.
- If testing my code, please edit Main.hs and let nCores = (currently 7) number of cores desired.