Parallelism in Haskell

Quicksort and Mergesort

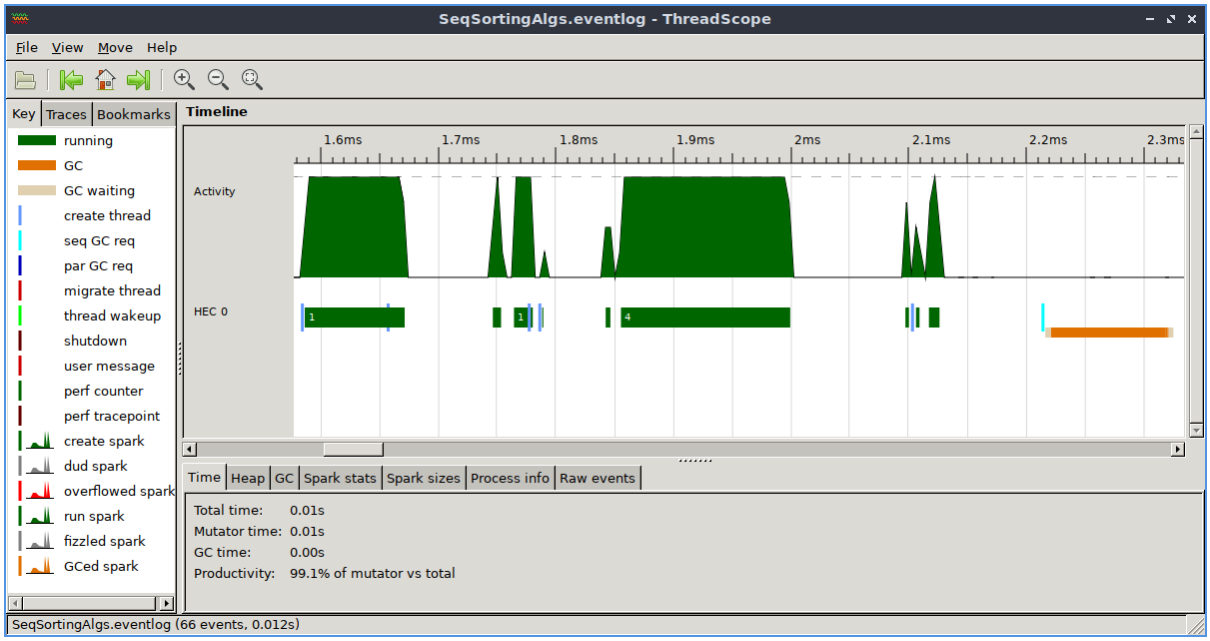
Introduction

I will be examining the implementation of parallelism in Haskell over the sorting algorithms Quicksort and Mergesort.
I have chosen to use these algorithms because

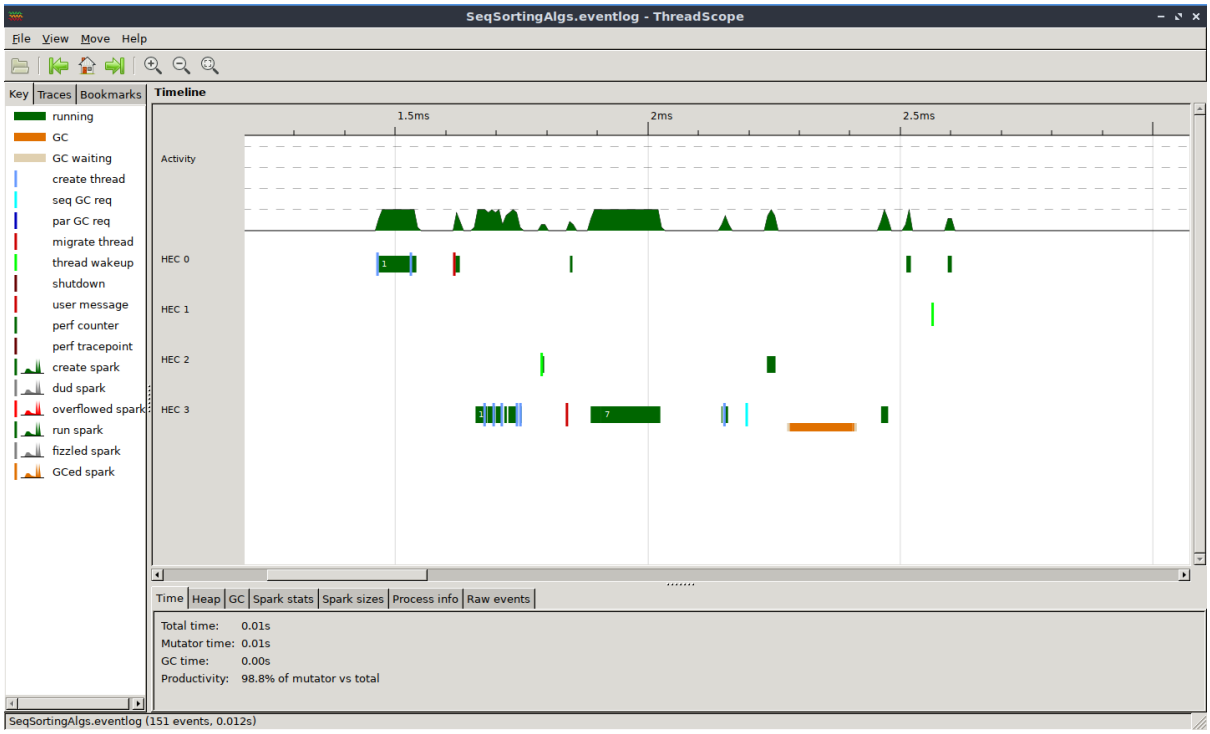
1. They were the recommended algorithms
2. They both can be implemented using recursion which should lend itself to parallelism.
3. They're straightforward algorithms meaning I can obtain examples from online and therefore focus on parallelising them and observing the results.

Phase 1. Sequential implementation

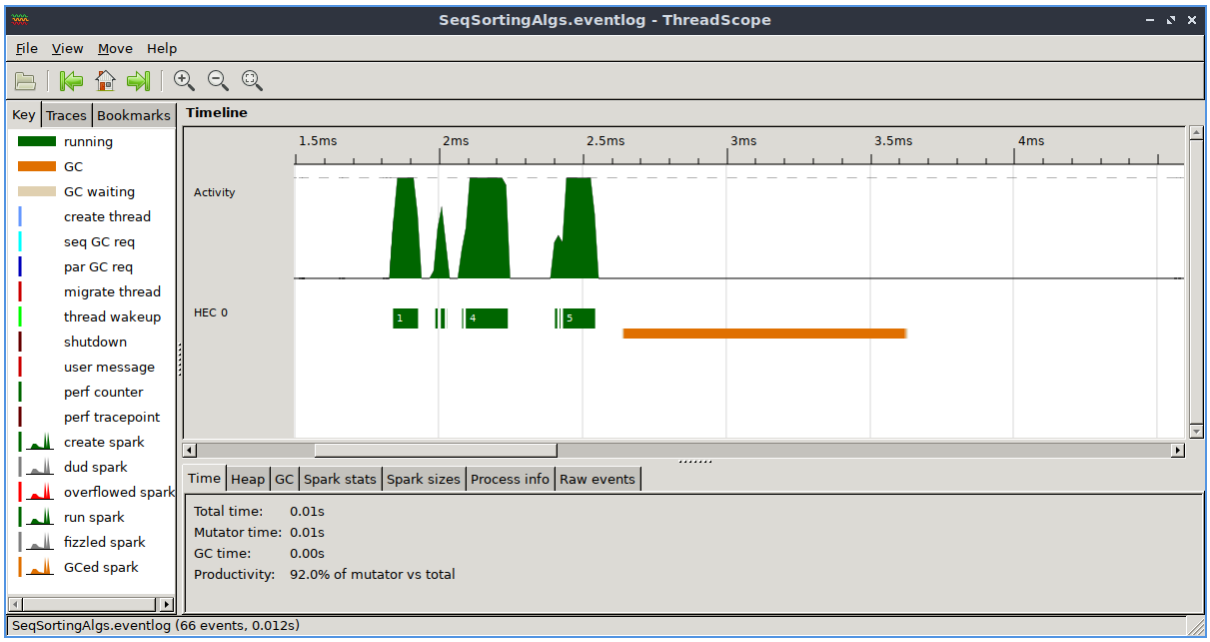
The first step was to implement a standard version of both algorithms used. I obtained versions online from "http://learnyouahaskell.com/recursion" for the quicksort algorithm and from "https://riptutorial.com/haskell/example/7552/merge-sort" in the case of mergesort. These are seen in SeqSortingAlgs.hs. As a benchmark we test these implementation using a list of 100 numbers between 1 and 500 and have them sorted by each algorithm. The sorting is run using 1 core initially for both algorithms then using 4 cores to compare the performance increase we might get without any explicit parallelisation.



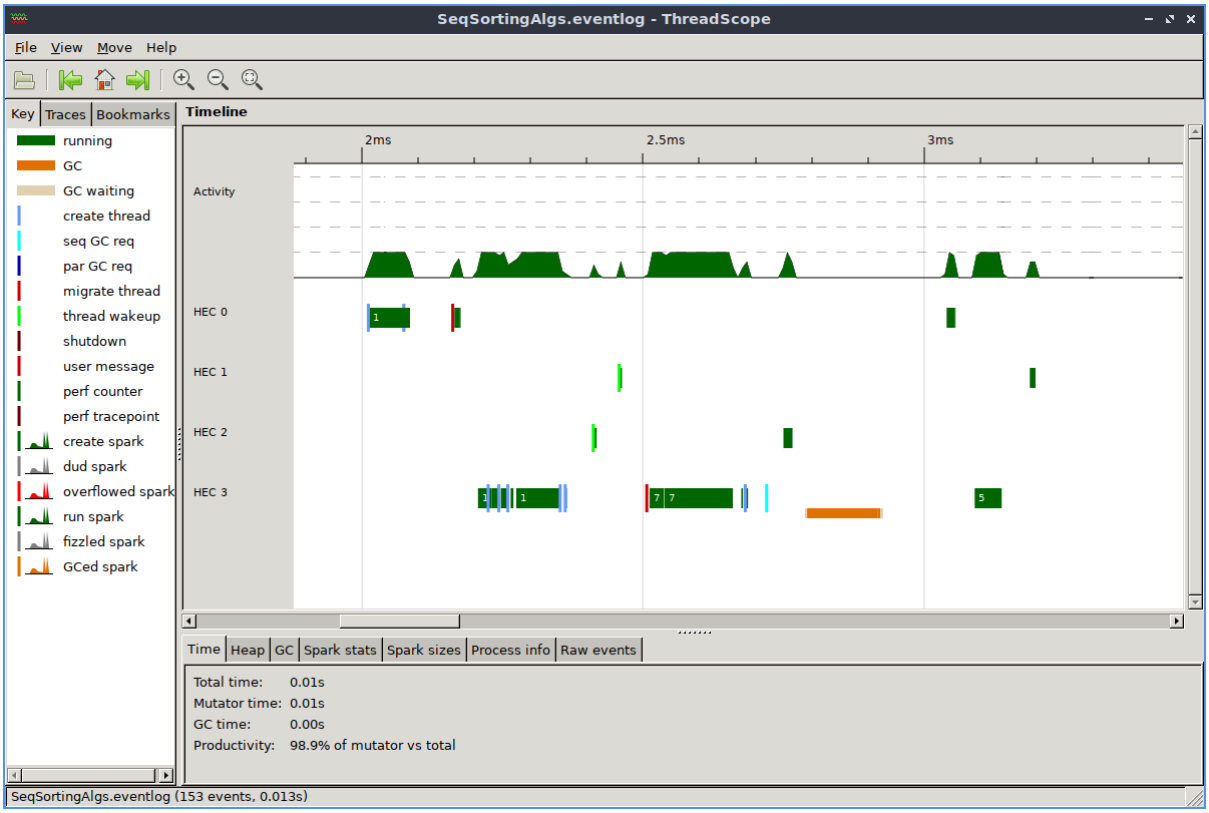
Quick sort run on a single thread without explicit parallelisation.



Quicksort run on 4 threads without explicit parallelisation, notice its performance degraded.



Mergesort run on a single thread without explicit parallelisation.



Mergesort run on 4 threads without explicit parallelisation. Performance seems to have suffered slightly.

Worth noting from the above screenshots is that while one would expect performance to increase given more processes to do calculations however in this instance we can see that the work is generally done sequentially across different threads due to poor allocation of the work load this means that we don't see an expected performance increase and due to having to manage more threads we even see a degradation due to increased overhead costs.

Phase 2. Implementing Parallelism

The compiler was unable to produce significant speeding up of the algorithms using parallelism by itself so now we will attempt to give it some guidance. This is achieved using the Control.Parallel import. Using this we apply "par" and "pseq" instructions to tell the compiler where to implement parallelism. The idea behind this is that the programmer explicitly organising the parallel execution can do a much better job than a compiler.

Due to issues getting up and running with threadscope and further issues with stack I was unable to get a working version of the algorithms running parallelised. As such I can only briefly discuss the steps I should have taken going forward, some of the considerations to be made and what I would have hoped to have seen.

As mentioned above by using par and pseq to designate when lines of code should have been broken down into parallel processes one would hope to allow faster execution. Parallelisation is best used on independent events of computation, as such mergesort provides an ideal opportunity to use this as it operates in a divide and conquer method which then is recursively executed one could instruct the program to process the two

separate sub arrays in parallel and then wait for them both to return before combining them together again and returning the new sorted array.
Consideration should be given to the amount of processes we are spawning by recursively executing in parallel, as such it is sensible to use a sequential execution once the level of computation becomes so little that it is not worth incurring the overhead of spawning new processes to handle the computation and just proceeding sequentially. Further consideration should be given to attempting to balance the workload given to any parallel processes such that no other process is waiting idle for another to return before sequential execution can resume.