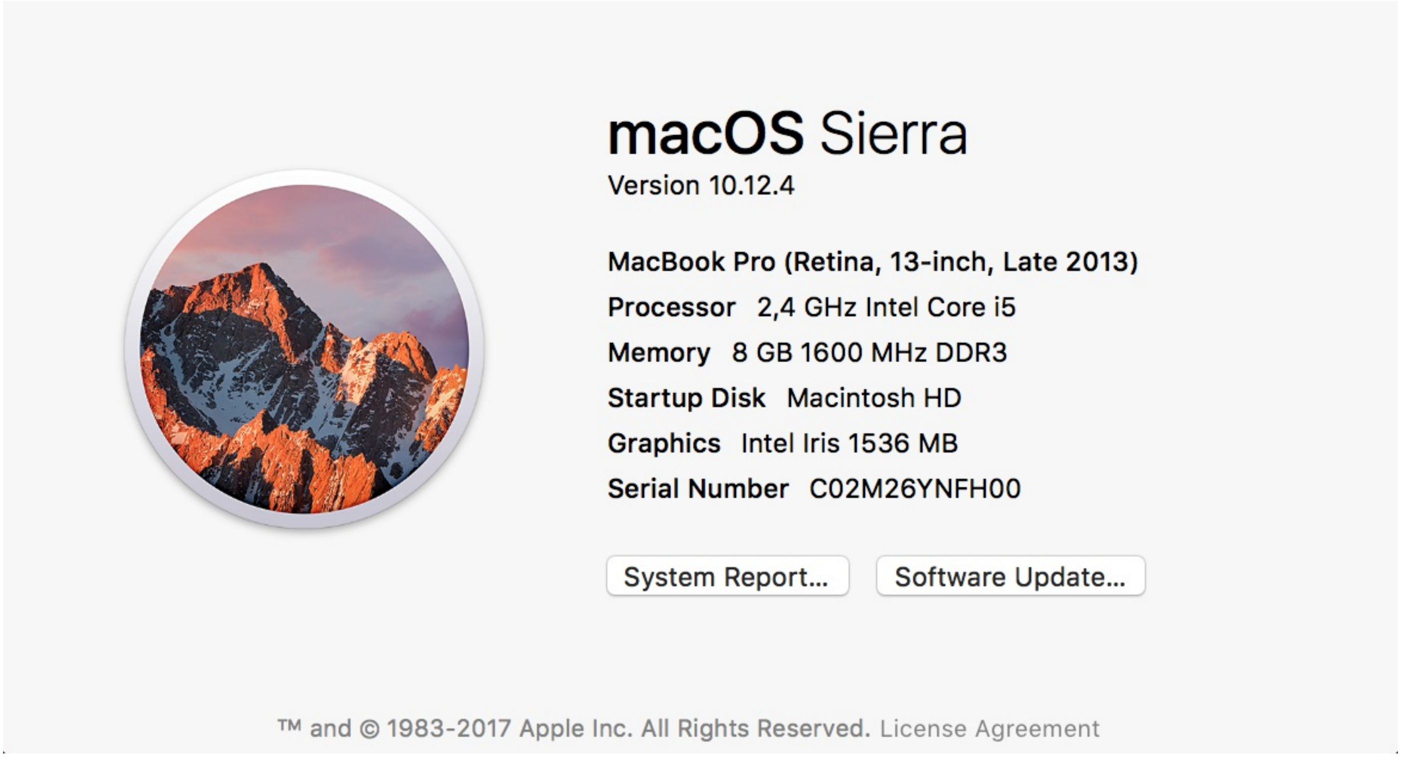
DAT280 – Lab A: “Parallel Programming in Haskell”

Olle Svensson (ollesv@student.chalmers.se)

Agazi Berihu (agazi@student.chalmers.se)

Setup

The benchmarks have been performed on the following machine:



macOS Sierra

Version 10.12.4

MacBook Pro (Retina, 13-inch, Late 2013)

Processor 2,4 GHz Intel Core i5

Memory 8 GB 1600 MHz DDR3

Startup Disk Macintosh HD

Graphics Intel Iris 1536 MB

Serial Number C02M26YNFH00

™ and © 1983-2017 Apple Inc. All Rights Reserved. License Agreement

The complete model name for the processor is **Intel(R) Core(TM) i5-4258U CPU @ 2.40GHz**. This processor has two physical and four logical cores.

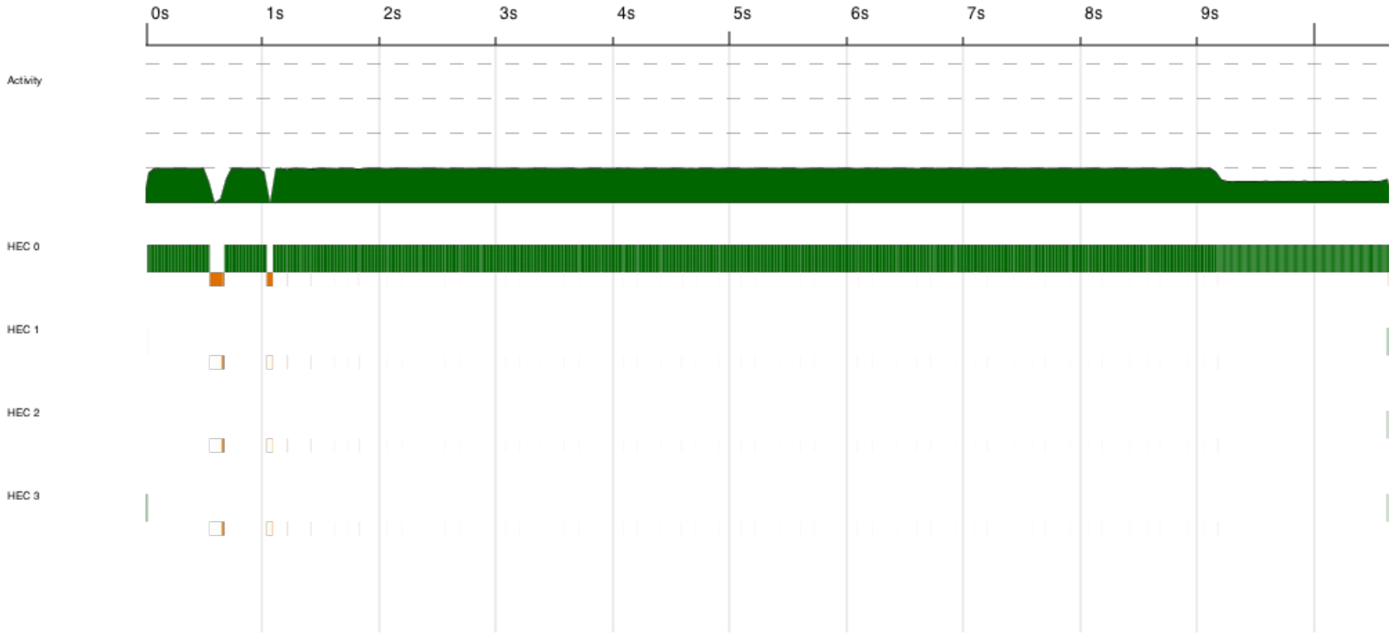
The executable has been run with the following additional flags to increase performance and reduce time spent garbage collecting: **-N4 -A100M -H2G**.

The executable was also compiled with the **-O2** flag to allow the GHC compiler to perform as many optimisations as possible.

Assignment 1

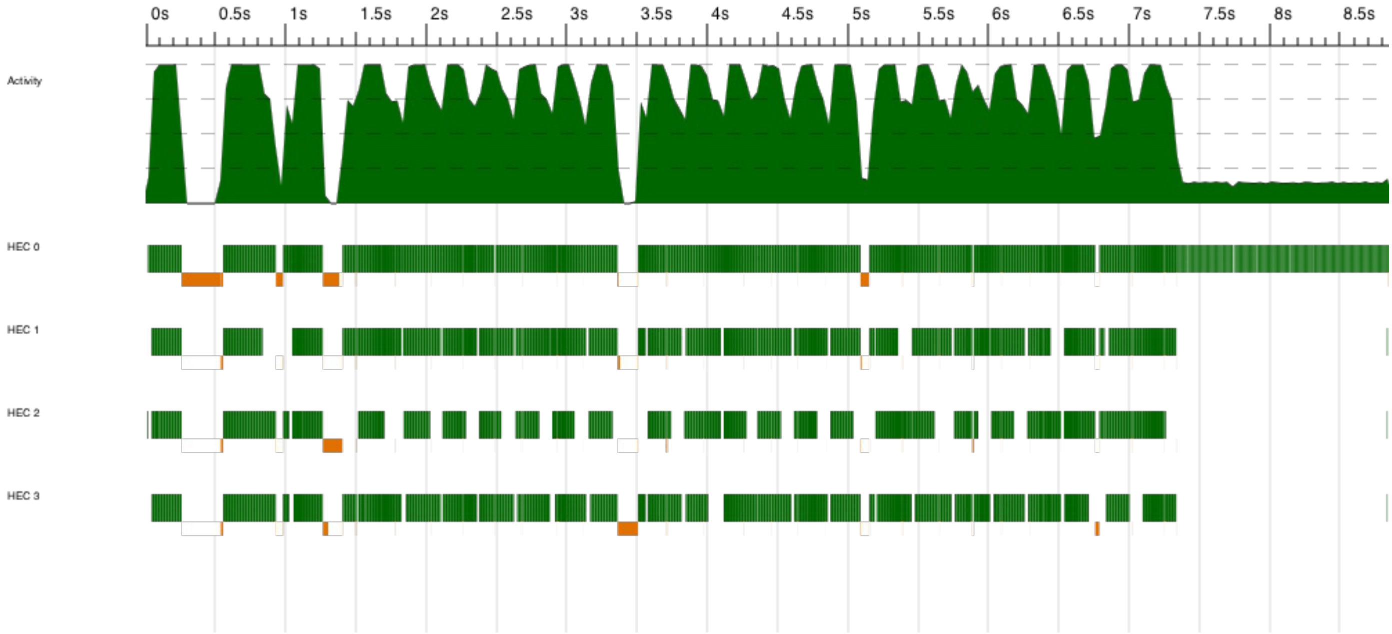
We start by running a benchmark on the standard `map` function as a starting point for our comparisons.

Benchmark 1.1 – map



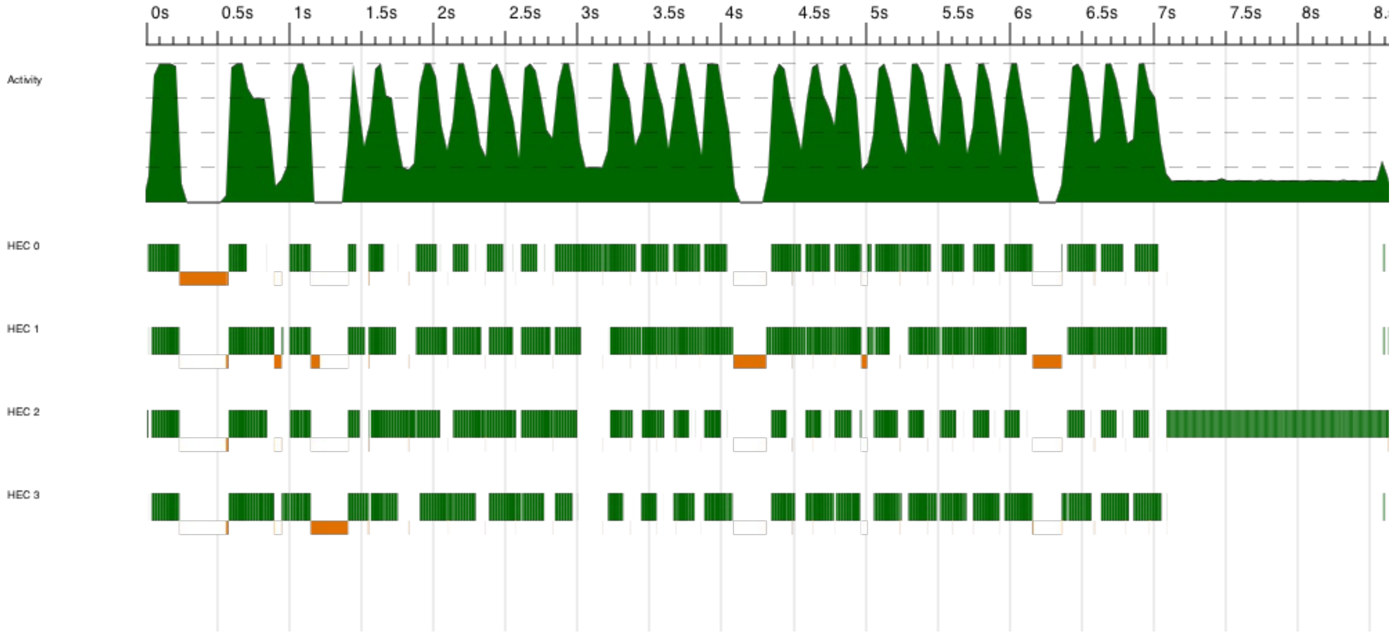
The second benchmark is done on `Main.parMap` which uses `par` and depth. It does not use `pseq` to wait for sub results since we found that the runtime were faster without. This leads to a slight increase in performance from a runtime of approximately 11 seconds in benchmark 1.1 to 9 seconds in 1.2.

Benchmark 1.2 – Main.parMap (a)



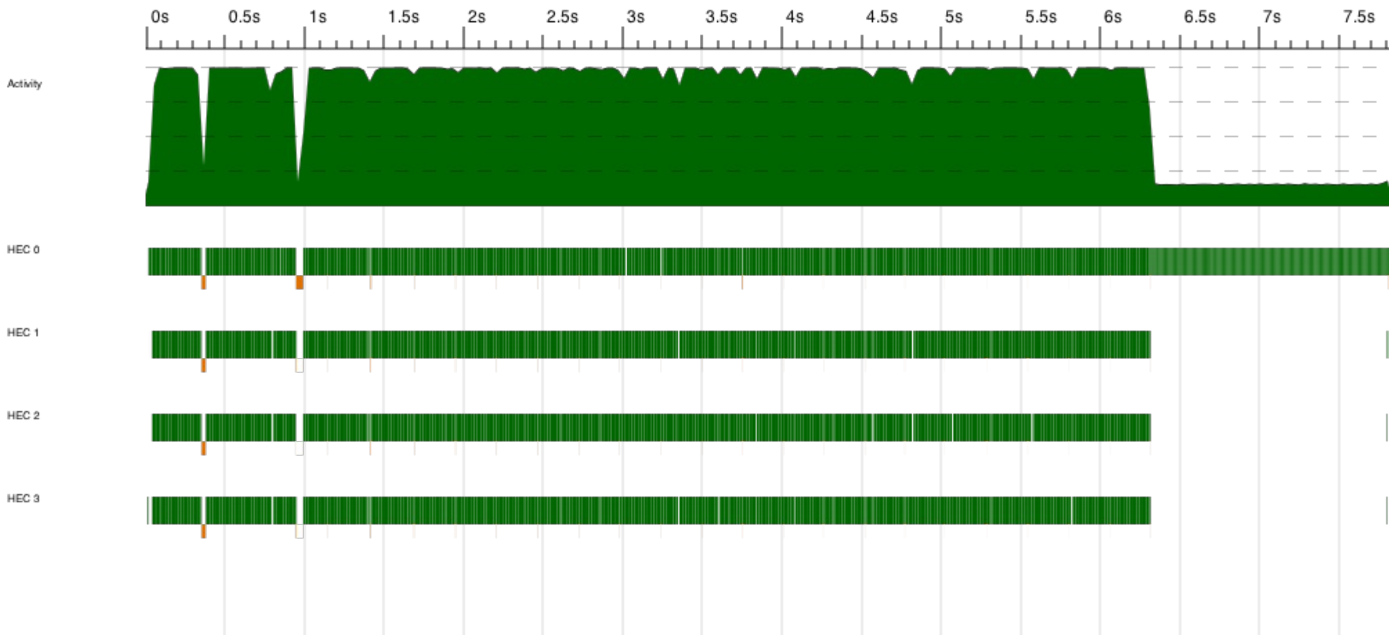
The third benchmark is done on `parMapRD` which is implemented in a similar way as `Main.parMap` but uses the Eval-monad instead. The runtime is around 9 seconds here as well as in 1.2, which is expected since it is basically the same function but wrapped in a monad instead.

Benchmark 1.3 – parMapRD (b)

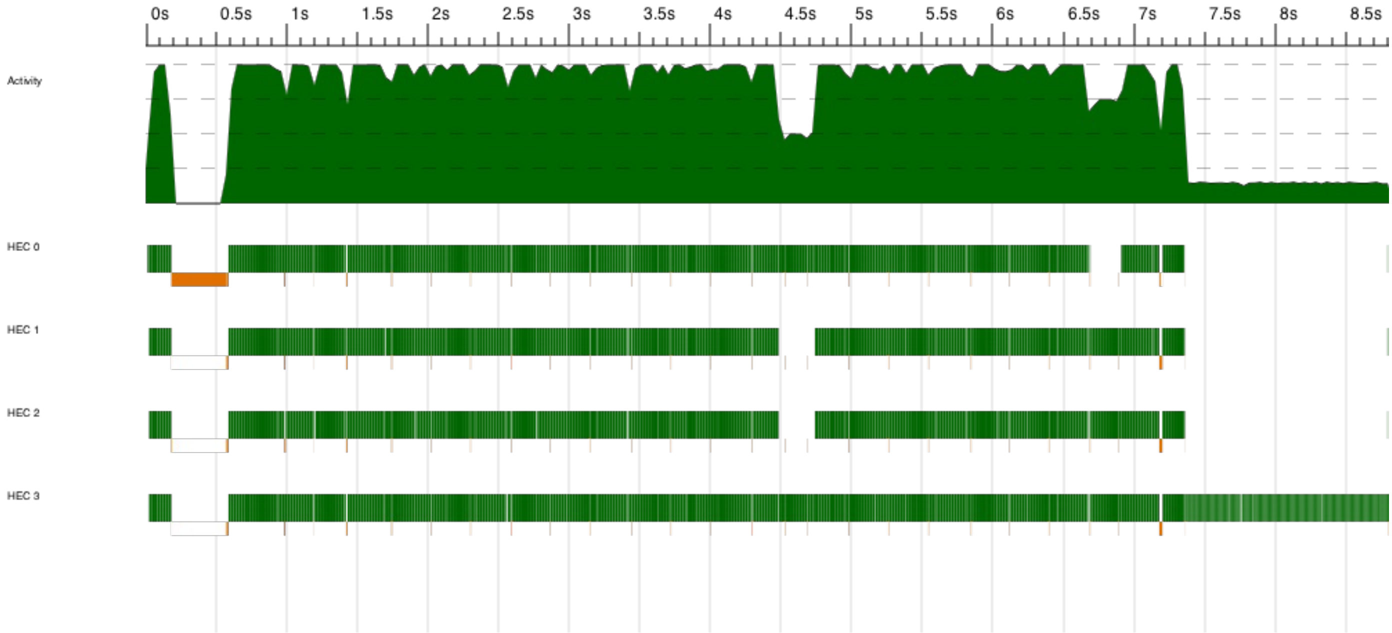


Benchmark 1.4 is done on `parMapS` which uses the Strategies library. Different from 1.2 and 1.3 is that it does not use depth but creates a spark for each element in the list that could theoretically be run in parallel. This yields a further increase in performance to a runtime of around 8 seconds.

Benchmark 1.4 – parMapS (c)



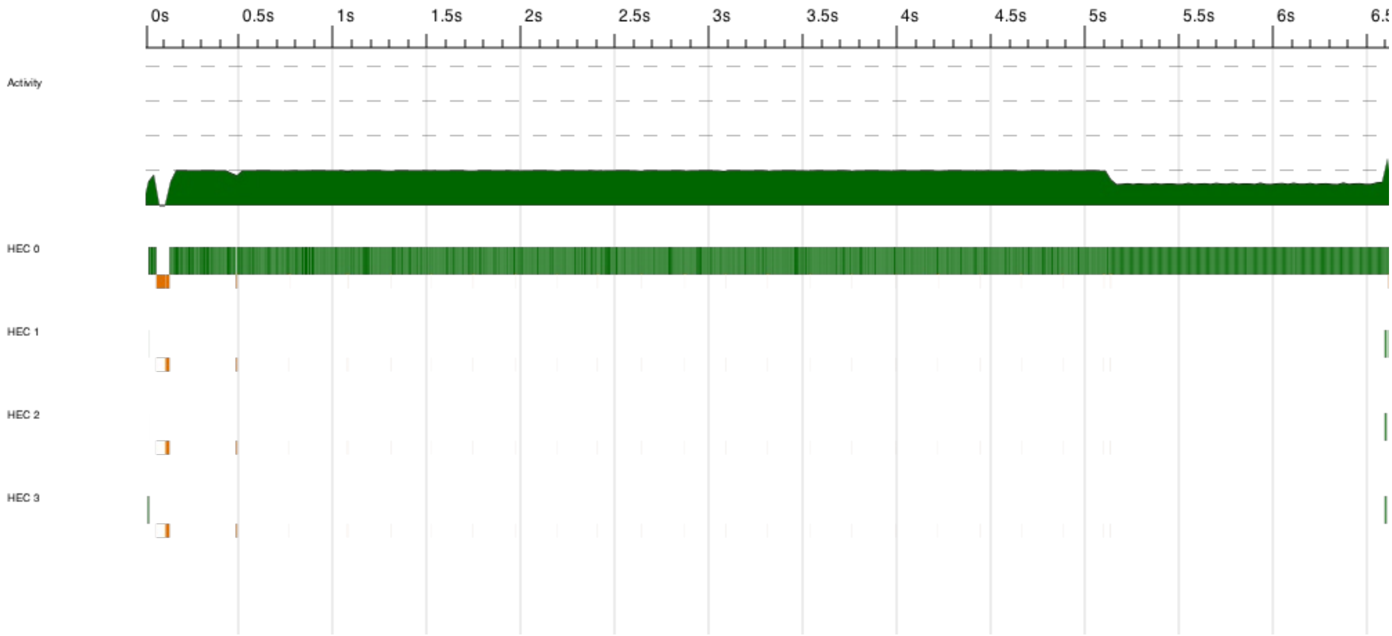
Finally, the fifth benchmark is done on `parMapP` which uses the Par-monad without depth. The runtime is around 9 seconds. **Benchmark 1.5 – parMapP (d)**



Assignment 2

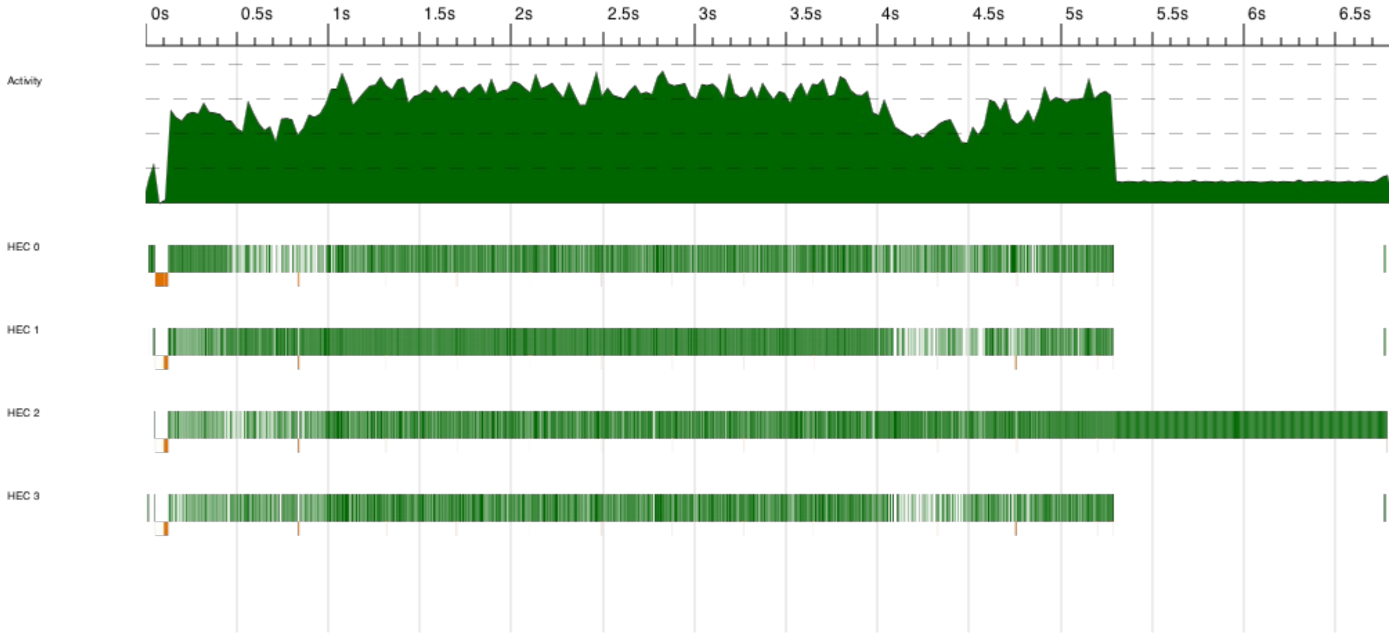
As in the previous case we start with a benchmark of a standard implementation of merge sort. The runtime is around 6.5 seconds.

Benchmark 2.1 – mergesort



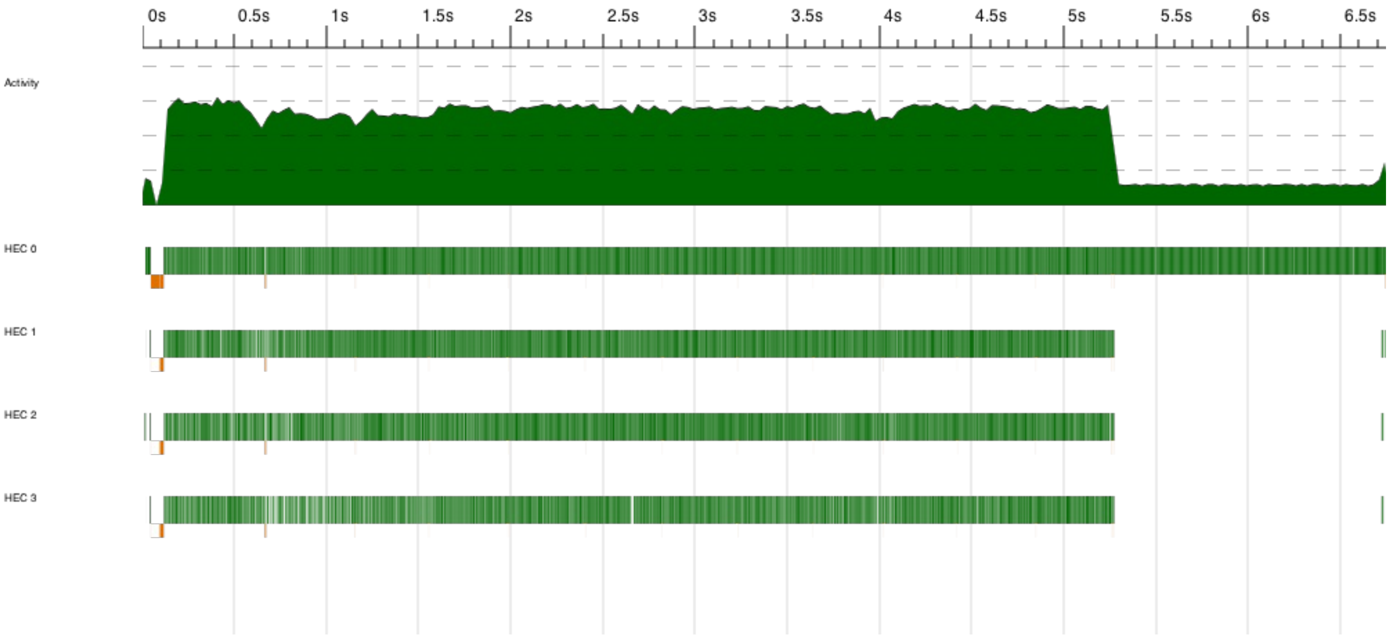
In the first parallelised version that we benchmark uses `pseq` but unfortunately the runtime is still around 6.5 seconds.

Benchmark 2.2 – mergesortPseq



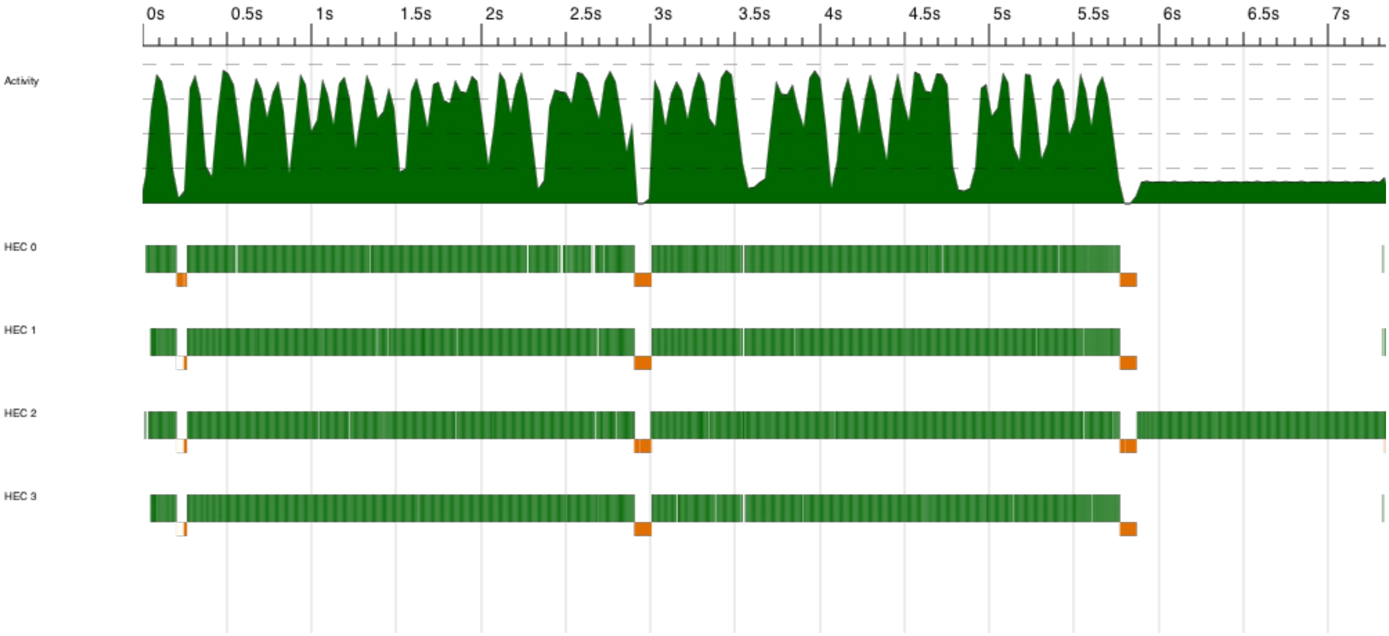
The third benchmark uses uses `rpar` and depth. Runtime is still around 6.5 seconds.

Benchmark 2.3 – mergesortRD



Finally we tried a version using the Par-monad which gave a runtime of roughly 7 seconds.

Benchmark 2.4 – mergesortP



Conclusions