Parallelized Fractal Image Generation in Haskell

Mandelbrot and Julia Sets with Parallel Strategies

Max Zhang, Isabel Tu

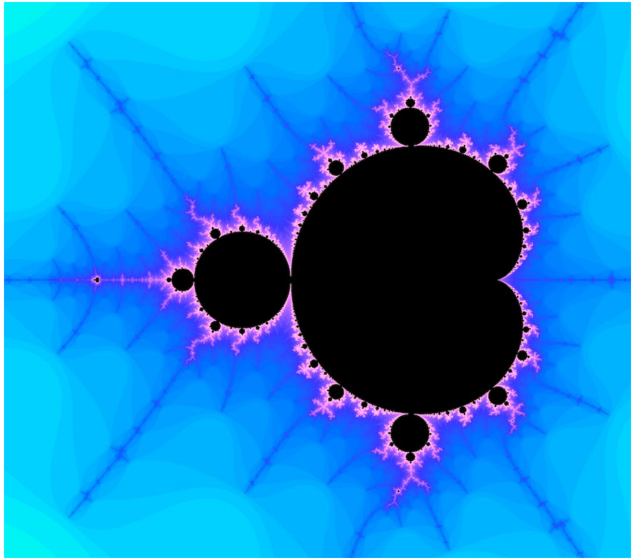
Introduction

Fractals like the Mandelbrot and Julia sets require **intensive computation** to generate images.

The goal is to leverage **parallel programming** in Haskell to reduce execution time using multi-core processors.

Objectives:

- Implement sequential and parallel versions of the fractal generator.
- Explore strategies: **parBuffer**, **Repa**, and more.
- Analyze performance with **speedup graphs** and **ThreadScope**.



The Mandelbrot and Julia Sets

2.1 Mandelbrot Set

The Mandelbrot set is defined as the set of complex numbers c for which the sequence defined by:

$$z_{n+1} = z_n^2 + c, \quad z_0 = 0 \tag{1}$$

remains bounded. A point is considered part of the Mandelbrot set if $|z_n| \leq 2$ for all iterations n. The number of iterations before $|z_n| > 2$ determines the color of the corresponding pixel in the fractal image.

2.2 Julia Set

The Julia set is generated similarly to the Mandelbrot set but with a fixed complex parameter c instead of varying it for each pixel. For a given complex parameter c, the Julia set includes all points z_0 where the sequence:

$$z_{n+1} = z_n^2 + c \tag{2}$$

remains bounded.

Fractal Generation - Core Computation

We are generating an image where each pixel maps to a point in the **complex plane**.

Escape Time: Number of iterations before $|z|>2$

Computation challenge:

- Millions of pixels.
- Many of iterations per pixel.

```
function mandelbrotIter(c, maxIter):
    z = 0
    iterations = 0
    while |z| <= 2 and iterations < maxIter:
        z = z^2 + c
        iterations += 1
    return iterations
```

Escape Time Computation

```haskell
mandelbrotIterations :: Double -> Double -> Int
mandelbrotIterations cr ci = go 0 0 0
  where
    go !i !zr !zi
      | i == maxIter || zr*zr + zi*zi > 4.0 = i
      | otherwise =
          let zr' = zr*zr - zi*zi + cr
              zi' = 2*zr*zi + ci
          in go (i+1) zr' zi'


juliaIterations :: (Double, Double) -> Double -> Double -> Int
juliaIterations (cr, ci) zr zi = go 0 zr zi
  where
    go !i !zr !zi
      | i == maxIter || zr*zr + zi*zi > 4.0 = i
      | otherwise =
          let zr' = zr*zr - zi*zi + cr
              zi' = 2*zr*zi + ci
          in go (i+1) zr' zi'
```

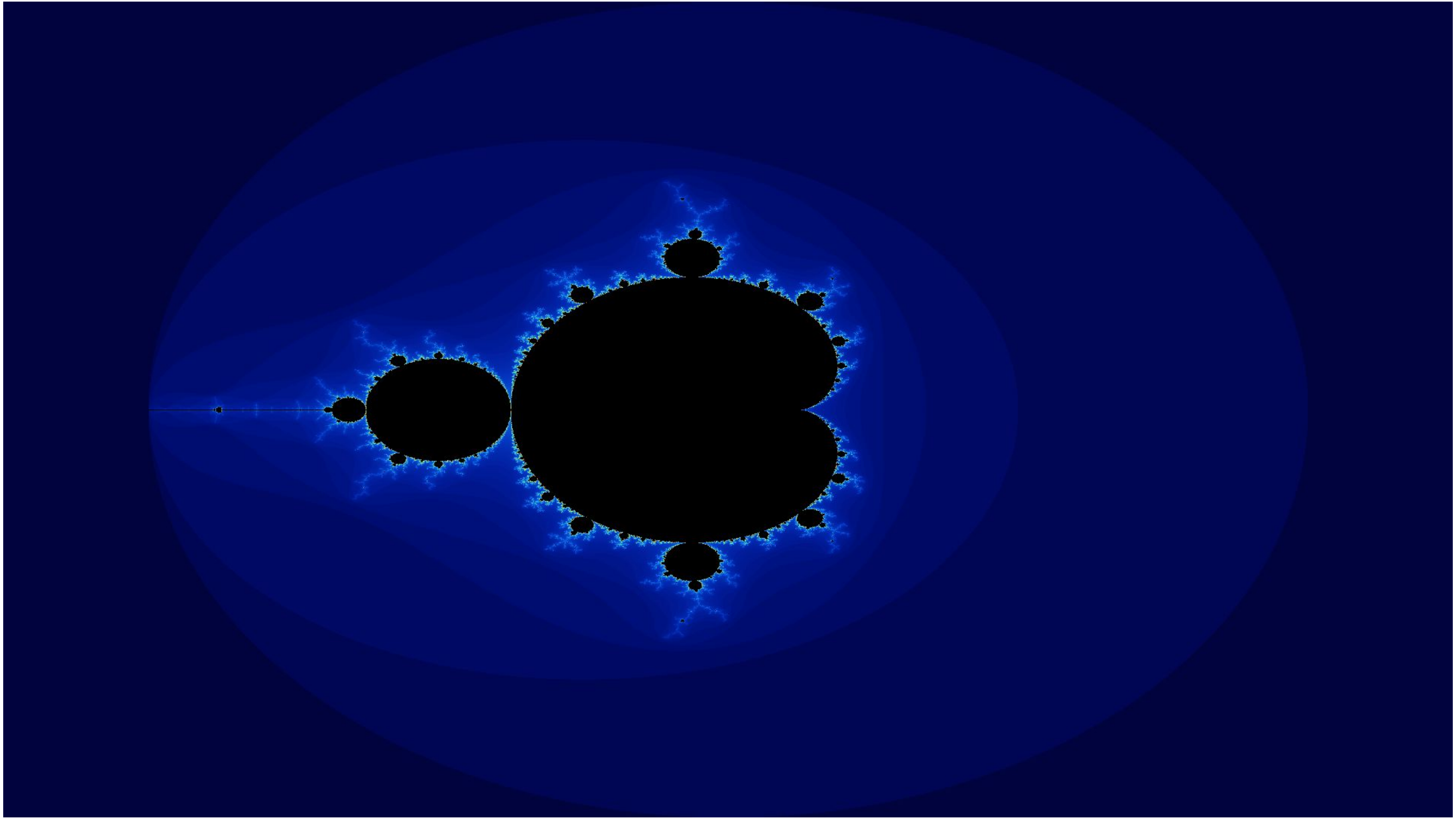
We found that a recursive approach greatly improved performance and parallelism.
Also avoiding computationally expensive sqrt by checking if square of the number > 4.

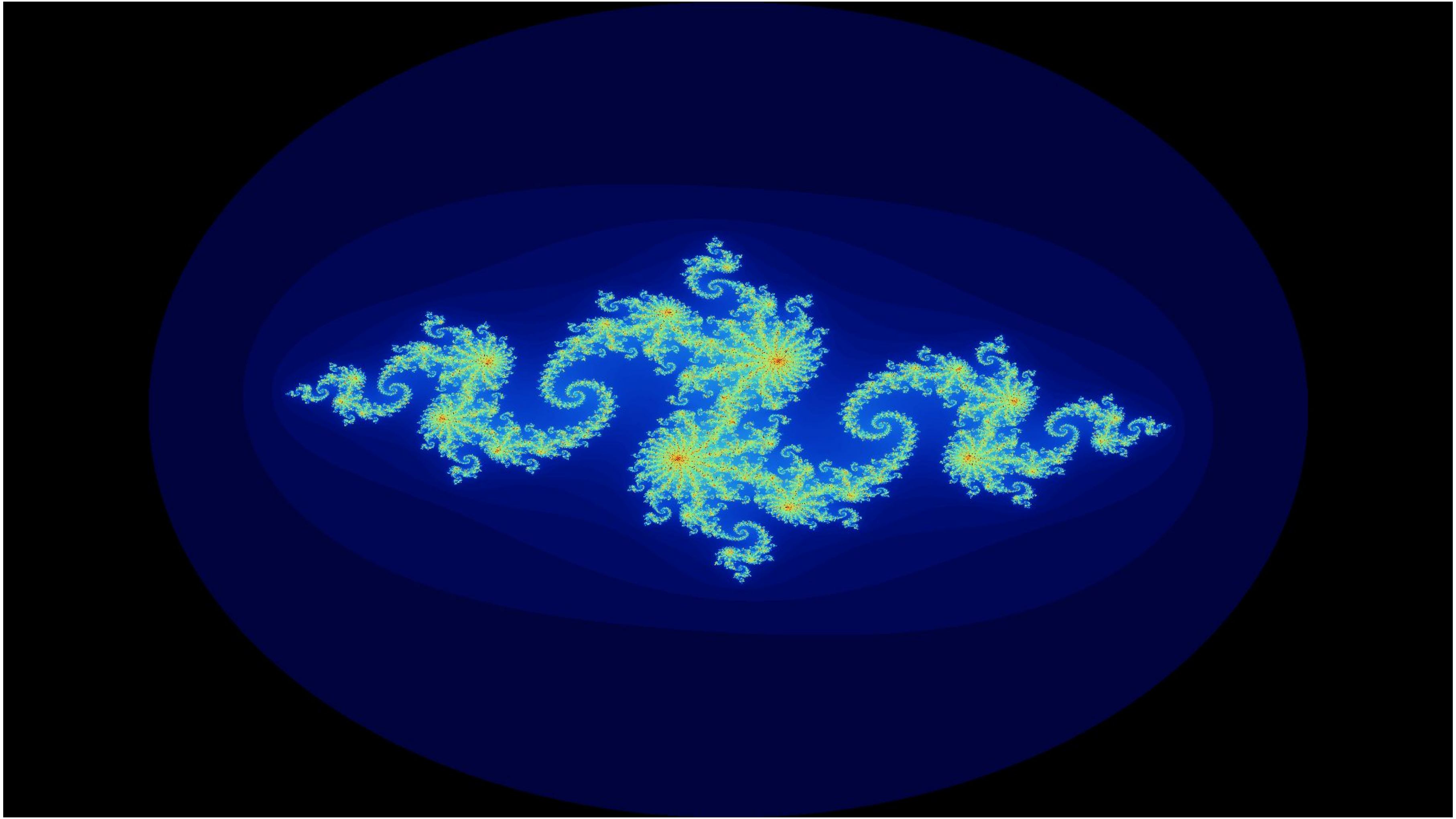
Computing with Parallel Strategies

```haskell
computeParallel :: String -> (Double -> Double -> Int) -> Int -> Int -> (Double, Double) -> [[(Word8, Word8,
Word8)]]
computeParallel strategy iterationFn width height bounds =
    let rows = [ [ iterationToColor (iterationFn re im)
                 | x <- [0 .. width - 1]
                 , let (re, im) = pixelToCoord x y width height bounds ]
               | y <- [0 .. height - 1] ]
    in case strategy of
        "seq"          -> rows
        "parListChunk" -> rows `using` parListChunk 8 rdeepseq
        "parBuffer"    -> rows `using` parBuffer 32 rdeepseq
        _              -> error "Invalid parallel strategy"
```

Computing Grid using REPA

```haskell
computeRepa :: (Double -> Double -> Int) -> Int -> Int -> (Double, Double) -> IO (R.Array R.U R.DIM2 (Word8,
Word8, Word8))
computeRepa iterationFn width height bounds =
    R.computeUnboxedP $ R.fromFunction (Z :. height :. width) $ \(Z :. y :. x) ->
        let (re, im) = pixelToCoord x y width height bounds
        in iterationToColor (iterationFn re im)
```





Sequential Implementation (Baseline)

Description:

- Computes each pixel’s escape time sequentially.
- No parallelism or load balancing.

Results:

- Mandelbrot (1 core): **1.959s**
- Julia (1 core): **0.810s**

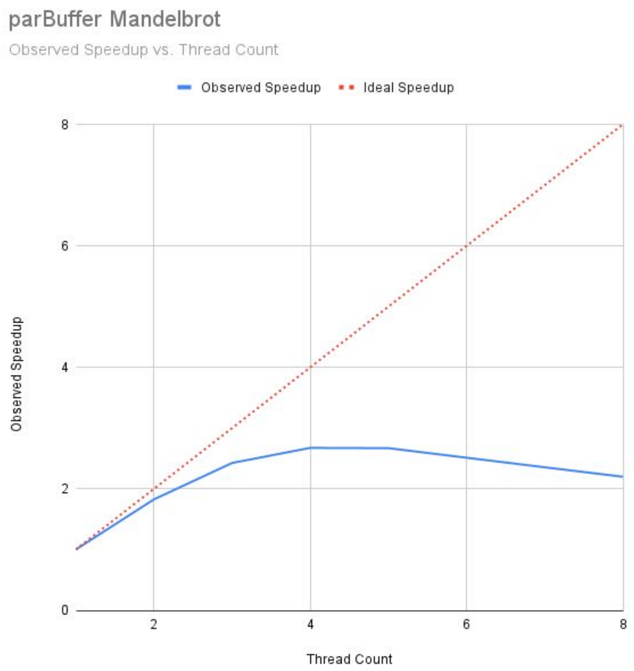
Parallel Strategy - parBuffer

Description:

- Divides the pixel grid into chunks using parBuffer
- Dynamically allocates tasks to threads, allowing chunks to be processed in parallel as threads become available.
- This dynamic scheduling helps mitigate load imbalance more effectively compared to fixed chunking strategies.

Execution Times:

- Mandelbrot:
  - 1 core: **3.156s**
  - 4 cores: **1.18s**
  - 8 cores: **1.435s**
  - Best speedup: **2.67x**



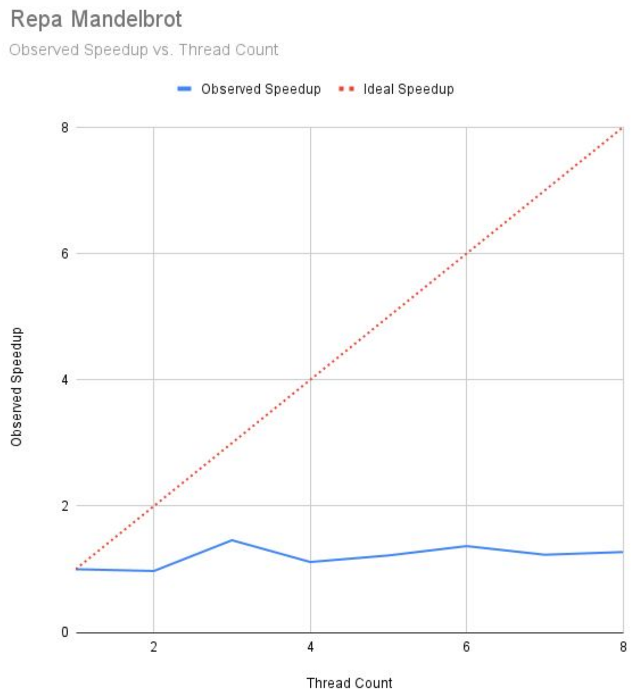
Parallel Strategy - Repa

Description:

- Defines the pixel grid using fromFunction, where each element is computed based on its coordinates.
- Uses computeUnboxedP to evaluate the entire grid in parallel.
- The data-parallel approach processes the grid uniformly but struggles with workload variability and sequential bottlenecks, leading to no significant improvement over the sequential implementation.

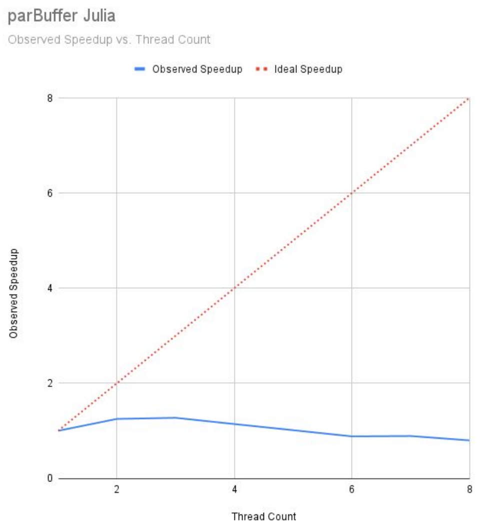
Execution Times:

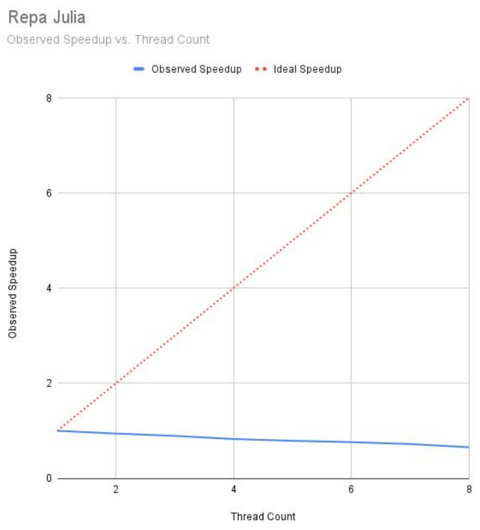
- Mandelbrot:
    - 1 core: **3.083s**
    - 4 cores: **2.767s**
    - 8 cores: **2.424s**
    - Best speedup: **1.46x**



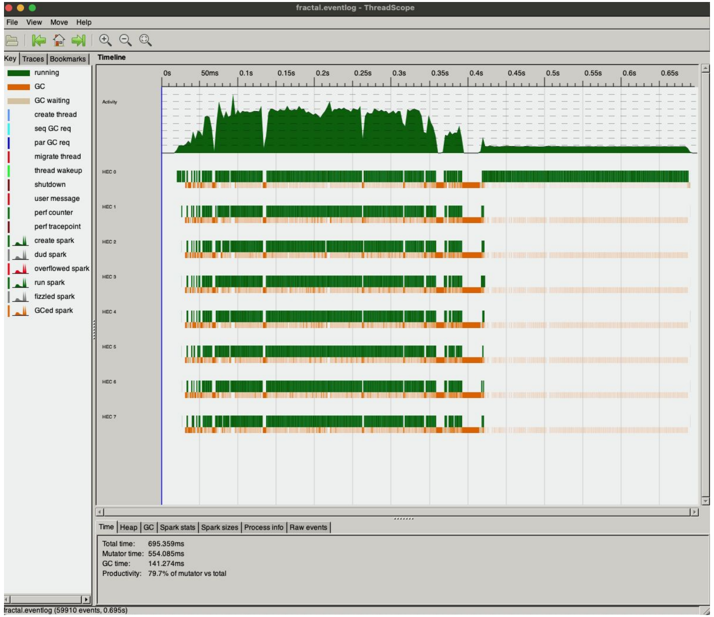
Issues with Parallelization of Julia

- **Unpredictable workloads**: Iteration counts vary across the grid.
- **Sequential bottlenecks**: Faster base runtime amplifies unparallelized steps.
- **High overhead**: Parallel overhead outweighs runtime gains.

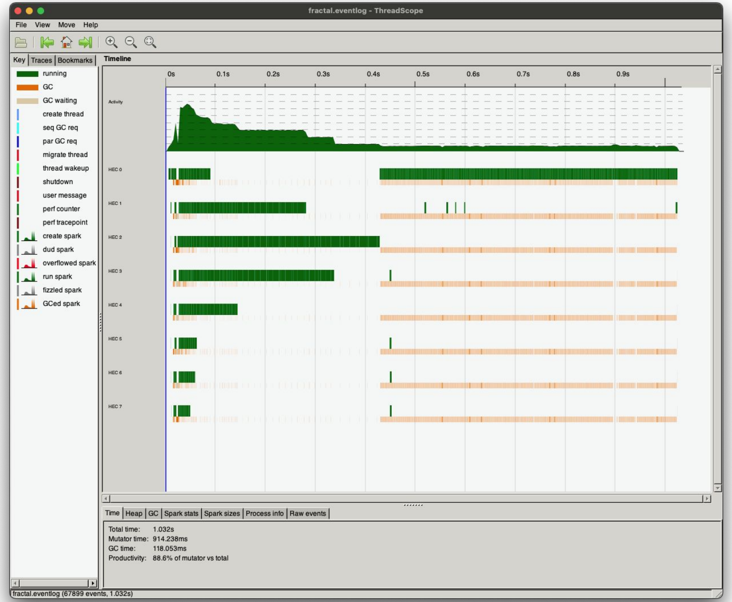




Threadscope



parBuffer



Repa

references

https://complex-analysis.com/content/mandelbrot_set.html

https://complex-analysis.com/content/julia_set.html