# par-fractal: Fractal Image Generation in Parallel Using Haskell

Max Zhang, Isabel Tu
mz2956, it2334

December 19, 2024

**Abstract**

This project implements a parallel fractal image generator using Haskell. Fractals, like the Mandelbrot and Julia sets, are computationally expensive to generate, making them ideal candidates for parallel processing. This report explores multiple parallelization strategies, including `parBuffer`, `parListChunk`,and Repa, to improve performance and scalability. Benchmarks, speedup graphs, and ThreadScope visualizations provide insights into the efficiency of each approach.

## 1 Introduction

The goal of this project is to generate high-resolution fractal images using parallel functional programming in Haskell. Fractals, such as the Mandelbrot and Julia sets, require extensive floating-point calculations for each pixel, making them computationally intensive. With the rise of multi-core processors, parallel programming can significantly reduce computation time.

The project includes the following objectives:

- Implement sequential and parallel versions of the fractal image generator.
- Leverage parallel Haskell strategies, `parBuffer`, `parListChunk`,and Repa arrays.
- Analyze performance improvements using speedup graphs and ThreadScope profiling.
- Compare the observed speedup against the ideal linear speedup.

This report discusses the implementation details, benchmarking results, and insights gained from parallelizing fractal computations.

## 2 The Mandelbrot and Julia Sets

Fractals are intricate geometrical shapes generated through mathematical iterations. Two famous fractals, the Mandelbrot and Julia sets, are defined in the complex plane and serve as the foundation for this project.

### 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.

## 3 Implementation

The generation of these fractals involves mapping each pixel in the image to a point in the complex plane. The computational challenge arises from iterating over millions of pixels and performing hundreds of iterations for each pixel. The following function demonstrates this for the Mandelbrot set:

1: Mandelbrot Iteration Function (Pseudo Code)

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

Similarly, the Julia set computation applies the same iterative formula but with a fixed $z_0$ value:

2: Julia Iteration Function (Pseudo Code)

```
function juliaIter(c, z0, maxIter):
    z = z0
    iterations = 0
    while |z| <= 2 and iterations < maxIter:
        z = z^2 + c
        iterations += 1
    return iterations
```

# 4 Generated Images

Using this implementation, we can visualize the generated Mandelbrot and Julia sets by rendering high-resolution images using our implemented fractal generator. The following figures showcase the output for different configurations:

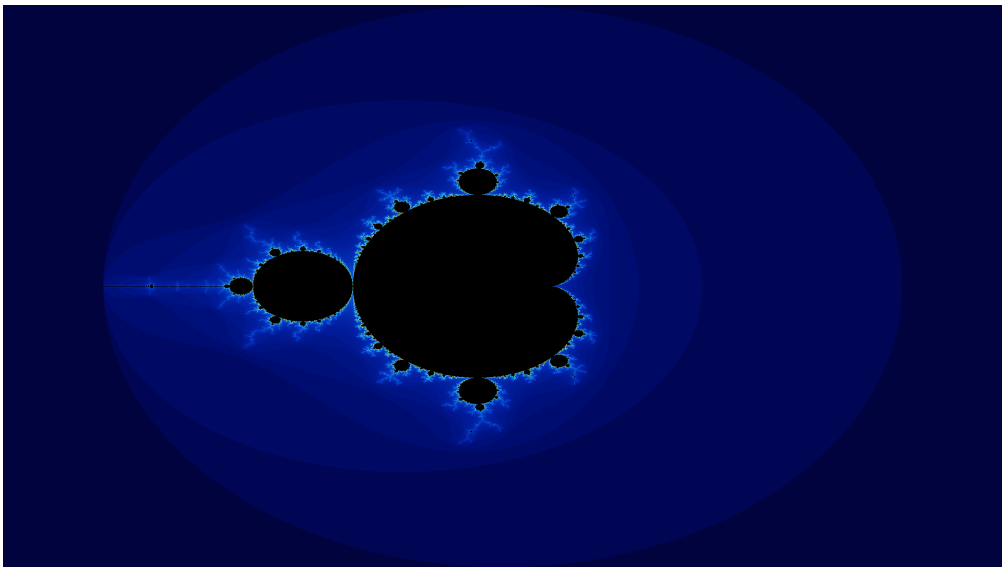

Figure 1: Mandelbrot Set generated using 1000 iterations

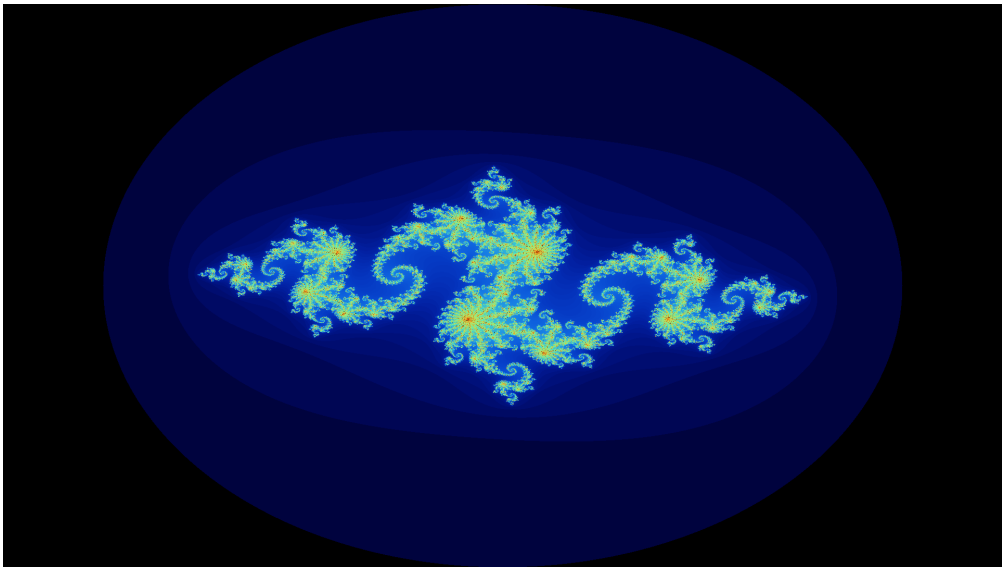

Figure 2: Julia Set with $c = -0.8 + 0.156i$ generated using 1000 iterations

# 5 Parallelization Strategies

This section analyzes the performance and behavior of different strategies used to compute the Mandelbrot and Julia sets. For each strategy, we analyze:

- **Execution Time**: The time taken to compute the fractal on a single core.
- **Speedup**: The ratio of sequential execution time to parallel execution time.
- **ThreadScope Analysis**: A visualization of thread activity to assess parallel workload distribution.

## 5.1 Sequential Implementation

- **Description**: The sequential implementation computes the escape time for each pixel without any parallelism.
- **Execution Time**:
  - Execution time of Mandelbrot on a single core: 1.959s
  - Execution time of Julia on a single core: 0.810s

## 5.2 Parallel Strategies with parListChunk and parBuffer

- **Description**: The parallel strategy divides the grid into chunks using either `parListChunk` or `parBuffer` to evaluate the chunks in parallel. `parListChunk` splits the workload into fixed-sized chunks, distributing these chunks to threads for evaluation. This allows for predictable and controlled parallelism but may lead to inefficiencies if the workload within chunks varies significantly. `parBuffer`, on the other hand, maintains a rolling buffer of tasks, dynamically allocating work to threads as they become available. This approach helps balance the workload more effectively, especially when computation is uneven across chunks.

  We observed that `parBuffer` performed better than `parListChunk` on Mandelbrot, as its dynamic task allocation helped reduce the impact of uneven workloads near the boundary of the fractals, improving overall parallel efficiency. However, neither strategy performed well for the Julia set due to its highly unpredictable iteration counts, which caused significant workload variability across the grid. This made it difficult to achieve balanced parallelism, regardless of the chunking strategy used.

- **Execution Time**:
  - Execution time of Mandelbrot on a single core: 3.156s
  - Execution time of Mandelbrot on 8 cores: 1.435s
  - Execution time of Julia on a single core: 1.086s
  - Execution time of Julia on 8 cores: 1.36s

- **Speedup**

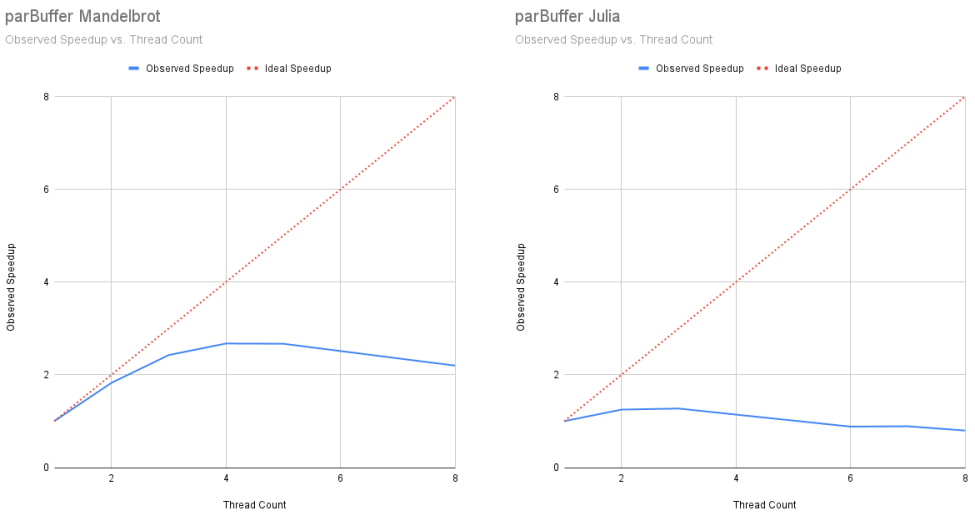


Figure 3: Speedup Graphs for `parBuffer`

  - Mandelbrot max speedup: 2.67 with 4 cores
  - Julia max speedup: 1.25 with 2 cores
  - Mandelbrot achieves better speedup with `parListChunk` and `parBuffer` because its workload is more uniform and predictable, with many points escaping quickly, especially far from the set. In contrast, Julia's iteration count varies significantly based on the starting point $z_0$ and fixed parameter $c$, resulting in more unpredictable and imbalanced workloads. This makes it harder for both `parListChunk` and `parBuffer` to distribute the work evenly across threads, as neither strategy fully compensates for the variability in Julia set computations.

- **ThreadScope Results**:

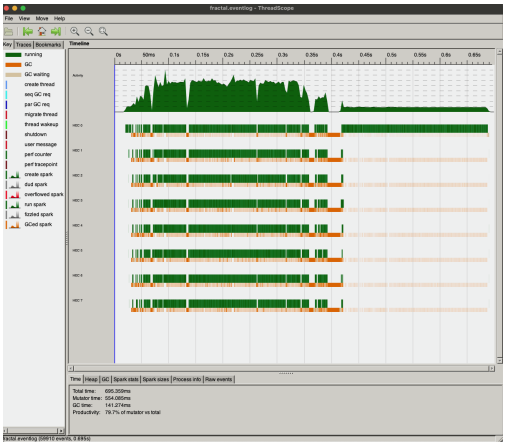

Figure 4: Threadscope for Mandelbrot using `parBuffer`

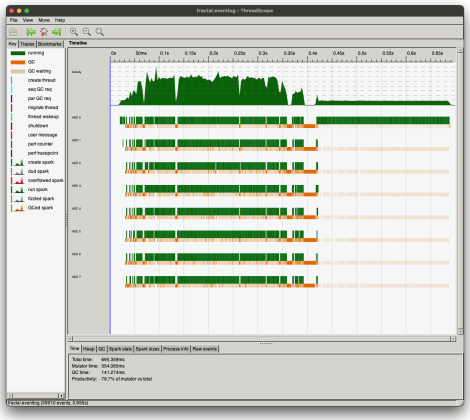

Figure 5: Threadscope for Julia using `parBuffer`

- The ThreadScope graphs show one thread running longer than the rest due to the IO step, which is not parallelized. While the Mandelbrot and Julia computations benefit from parallelism, the sequential rendering of the final image introduces a bottleneck, causing one thread to appear disproportionately active at the end. Since the Julia set computation is inherently faster, the unparallelized image generation step constitutes a larger portion of the total runtime, further limiting the overall speedup achieved.

## 5.3 Parallel Strategies with Repa Library

- **Description**: The Repa implementation uses `fromFunction` to define the grid, where each element is mapped to a computation based on its pixel coordinates, using the provided iteration function to determine the fractal's color. The `computeUnboxedP` function then evaluates the entire array in parallel, ensuring efficient processing of the grid. This approach minimizes load imbalance by evenly distributing the computation across threads, leveraging Repa's strict data-parallel model for uniform and efficient parallel execution.

- **Execution Time**:
  - Execution time of Mandelbrot on a single core: 3.083s
  - Execution time of Mandelbrot on 8 cores: 2.424s
  - Execution time of Julia on a single core: 1.147s
  - Execution time of Julia on 8 cores: 1.755s

- **Speedup**

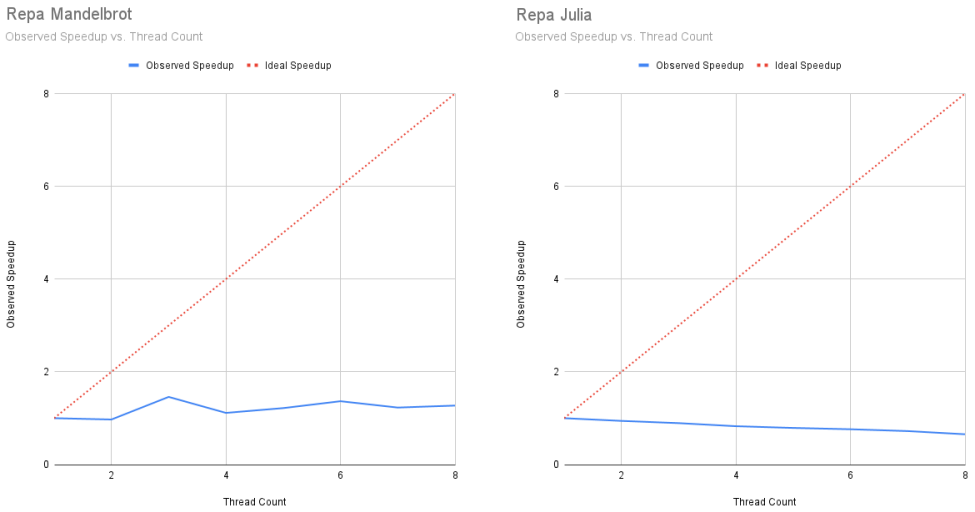


Figure 6: Speedup Graphs for Repa

  - Mandelbrot max speedup: 1.46 with 3 cores
  - Julia max speedup: no observed speedup
  - Unlike in `parBuffer`, where Mandelbrot parallelizes better due to its more predictable workload, Repa shows limited speedup for Mandelbrot and no speedup for Julia, which actually becomes slower with more cores. While Repa's data-parallel approach processes the grid uniformly and minimizes workload variability, the inherently faster Julia computation is dominated by the sequential image generation step, negating the benefits of parallelism and leading to poorer scaling.

- **ThreadScope Results**:

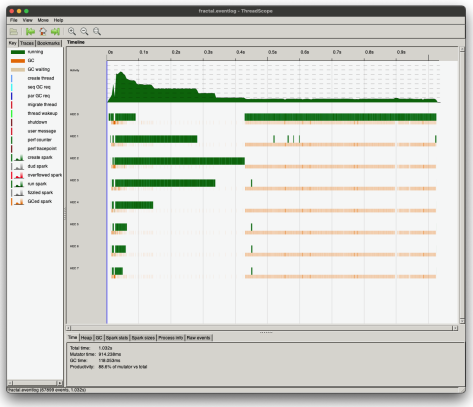

Figure 7: Threadscope for Mandelbrot using `Repa`

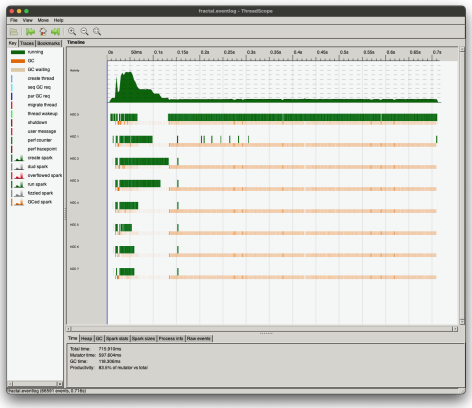

Figure 8: Threadscope for Julia using `Repa`

# 6 Conclusion

This project demonstrates the benefits and challenges of parallelizing fractal image generation in Haskell using three strategies: `parBuffer`, `parListChunk`, and the Repa library. While `parBuffer` and `parListChunk` improve performance over the sequential implementation, the Repa library fails to achieve any meaningful speedup, highlighting key limitations in its approach.

Both `parBuffer` and `parListChunk` effectively parallelize the Mandelbrot set, with `parBuffer` dynamically balancing uneven workloads and `parListChunk` providing predictable task allocation. However, Repa's data-parallel array model shows no improvement over the sequential implementation, as the inherently faster Julia computation is dominated by the unparallelized image generation step, and the grid processing lacks the flexibility to address workload variability.

Overall, this comparison underscores the strengths of `parBuffer` and `parListChunk` in achieving practical parallel performance, while highlighting the inefficiencies of Repa in handling fractal computations where workload variability and sequential bottlenecks remain significant challenges.

# Future Work

Outline future directions:

- Parallelize IO process for further speedup potential.
- Optimizing chunk size for `parListChunk`.
- Optimizing buffer size for `parBuffer`.
- Extending the implementation to GPU acceleration using `Accelerate`.

# 7 Full Code

## 7.1 Fractal

3: Fractal.hs

```
{-# LANGUAGE BangPatterns #-}
module Fractal (iterationToColor, pixelToCoord, mandelbrotIterations
   , juliaIterations) where

import Data.Word (Word8)

maxIter :: Int
maxIter = 1000

iterationToColor :: Int -> (Word8, Word8, Word8)
iterationToColor i
  | i == maxIter = (0, 0, 0)
  | otherwise    =
      let t = sqrt (fromIntegral i / fromIntegral maxIter)
          r = floor (9 * (1 - t) * t * t * t * 255) -- polynomials
             for colors found online
          g = floor (15 * (1 - t)*(1 - t)*t*t * 255)
          b = floor (8.5 * (1 - t)*(1 - t)*(1 - t)*t * 255)
      in (fromIntegral r, fromIntegral g, fromIntegral b)


pixelToCoord :: Int -> Int -> Int -> Int -> (Double, Double) -> (
   Double, Double)
pixelToCoord x y width height (minRe, maxIm) =
    let re = minRe + (fromIntegral x / fromIntegral width) * (5)
        im = maxIm - (fromIntegral y / fromIntegral height) * (4)
    in (re, im)


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'
```

## 7.2 Parallel Backend

4: ParallelBackend.hs

```
module ParallelBackend (computeParallel) where

import Data.Word (Word8)
import Control.Parallel.Strategies
import Fractal (iterationToColor, pixelToCoord)

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"
```

## 7.3 Repa Backend

5: RepaBackend.hs

```
module RepaBackend (computeRepa) where

import Data.Word (Word8)
import qualified Data.Array.Repa as R
import Fractal (iterationToColor, pixelToCoord)
import Data.Array.Repa ((:.)(..), Z(..))

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)
```

## 7.4 IO Handler

6: IOHandler.hs

```
module IOHandler (writePPM) where

import System.IO
import Data.Word (Word8)

writePPM :: FilePath -> [[(Word8, Word8, Word8)]] -> IO ()
writePPM file image = withFile file WriteMode $ \h -> do
    let height = length image
        width  = length (head image)
    hPutStrLn h "P3"
    hPutStrLn h (show width ++ " " ++ show height)
    hPutStrLn h "255"
    mapM_ (\row -> mapM_ (\(r,g,b) -> hPutStr h (show r ++ " " ++
       show g ++ " " ++ show b ++ " ")) row >> hPutStrLn h "") image
```

## 7.5 Main Program

7: main.hs

```haskell
-- compile with ghc -O2 -threaded -eventlog -outputdir build -o par-
   fractal main.hs IOHandler.hs Fractal.hs ParallelBackend.hs
   RepaBackend.hs

-- to run: ./fractal [mandelbrot/julia] [parallel/repa] [for
   parallel: seq, parBuffer, parListChunk] [for repa: just include a
    _]

import System.Environment
import Fractal
import ParallelBackend
import RepaBackend
import IOHandler
import qualified Data.Array.Repa as R
import Data.Array.Repa ((:.)(..), Z(..))

main :: IO ()
main = do
    args <- getArgs
    case args of
        ["mandelbrot", backend, strategy] -> runFractal
           mandelbrotIterations backend strategy
        ["julia", backend, strategy]      -> runFractal (
           juliaIterations (-0.8, 0.156)) backend strategy
        _ -> putStrLn "Usage: main <mandelbrot|julia> <parallel|repa
           > <strategy>"

runFractal :: (Double -> Double -> Int) -> String -> String -> IO ()
runFractal iterationFn "parallel" strategy = do
    let width = 1920; height = 1080
    let image = computeParallel strategy iterationFn width height
       (-2.5, 2.0)
    writePPM "output.ppm" image

runFractal iterationFn "repa" _ = do
    let width = 1920; height = 1080
    arr <- computeRepa iterationFn width height (-2.5, 2.0)
    R.deepSeqArray arr $ writePPM "output.ppm" [[ arr R.! (R.Z :. y
       :. x) | x <- [0 .. width - 1]] | y <- [0 .. height - 1]]
```

# References


1. https://complex-analysis.com/content/mandelbrot_set.html
2. https://complex-analysis.com/content/julia_set.html