1 Introduction

The Mandelbrot Set is a set of complex numbers. Every complex number can be represented as a linear combination of \( a + bi \), where \( a \) and \( b \) are real numbers. As such, every complex number can be plotted on a two-dimensional Cartesian plane with one axis representing the real component and one axis representing the complex component of a number.

The fractal shape of the Mandelbrot set comes from plotting the points in this set on the complex plane through the repeated iteration over the following equation for each point on the complex plane.

\[
f_c(z) = z^2 + c
\]

The value of \( z \) starts at 0 (\( a = b = 0 \)) and changes through each iteration of the function while \( c \) represents a complex number that remains the same through each iteration of the function. A point is considered inside the Mandelbrot set if the function doesn’t diverge or, in other words, the norm of the point under iteration must be bounded in a disk of finite radius. The norm of a complex number is calculated as its Euclidean distance from the origin on the complex plane:

\[
||a + bi|| = \sqrt{a^2 + b^2}
\]

Often the finite radius used in this calculation is 4. Points outside of the Mandelbrot set are often colored differently according to the number of iterations it takes for it to escape a certain radius. It’s along these boundaries that the complex shapes and self-recursive images are produced with infinite precision and detail.

2 Mandelbrot

Since the Mandelbrot set comprises complex numbers of the form \( a + bi \), or is a vector space of degree 2 over the real numbers, we can represent points as a tuple of two Floats—one representing the real component and one representing the imaginary component. This can also be interpreted as the \( x- \) and \( y- \)coordinate of the number as a point in the plane.

The complex plane can then be represented as a two-dimensional list of Points. Most complex numbers do not belong to the Mandelbrot set since they are large enough that they diverge quickly or start with a norm greater than 2. Therefore, the grid can be limited to having a complex domain of \( \{-10...1.0\} \) and a real domain of \( \{-2.0...1.0\} \). Also for the sake of ease of computation, we will limit our granularity to hundredths.

```haskell
grid :: [[Point]]
grid = [[(x/100 , y/100) | x <- [-200..100] | y <- [-100..100]]
```

Since a number in the Mandelbrot set will never diverge, it can be impossible to tell whether or not a point has yet to escape the finite disk of radius 2 or if will remain bounded forever. As a heuristic, we will simply cut off the number of iterations at 2000 and if the norm hasn’t diverged by then it will be included in this rendering of the Mandelbrot set. The ASCII encoding chosen will print an A, B, C, or space depending on how long it takes for the point to diverge under iteration.

```haskell
mandelbrot :: Point -> State (Int,Point) Char
```
mandelbrot z = d
    (c,p) <- get
    if c > 2000 || escaped p
      then return $ encode c
    else do
      modify $ \(i,p') \rightarrow (i+1,\text{iterPoint } z \ p')
      mandelbrot z
    where
      escaped (a,b) = a*a + b*b > 4.0
      iterPoint (a_z,b_z) (a,b) =
        (a*a - b*b + a_z, 2.0*a*b + b_z)
      encode count
        | count > 2000 = ' '
        | count > 50 = 'A'
        | count > 10 = 'B'
        | count > 1 = 'C'

-- render :: [[ Char ]]
-- Apply mandelbrot to each point in grid and
-- get the resulting ASCII rendering
main :: IO ()
main = do
  mconcat $ putStrLn <$> render

The iteration over each complex number is not only
recursive and self-contained, but also entirely inde-
pendent from the rest and results in easily embar-
rassingly parallel computation. Therefore the render
function that describes how the mandelbrot function
will be applied over the plane will be the primary
point of variation in parallelization.

3 Serial Mandelbrot

The fully serial version of the Mandelbrot set
render is shown below. In this implementation, the
mandelbrot function is simply mapped over the two-
dimensional list of Points, so each point is iterated
over in sequence.

render :: [[ Char ]]
render = map (map evalPoint) grid
  where
    evalPoint z =
      evalState (mandelbrot z) (0,(0.0,0.0))

This implementation of the Mandelbrot renderer
was used as the baseline for all relative statistics and
speedups when comparing with parallel implementa-
tions.

As shown in Figure 1, this implementation of the
Mandelbrot renderer results in a simple event time-
line with all work being done on one processor. The
total average run time was 80.27s with about 5.5% of
that time dedicated to garbage collection.

4 Parallel Mandelbrot

In order to test the effects of different parallelization
methods on the runtime performance of the ASCII
Mandelbrot renderer, all implementations were run
on a virtual environment with 1,2,4, and 8 processors
and 8GB of RAM. The statistics listed below in this
section are averaged after running multiple trials on
4 CPUs.

4.1 seq Parallel Traversal

The implementation below represents the first at-
tempt at parallizing the Mandelbrot computations.
The innerPar map parallelizes the computation per-
formed on each point in one line of the complex plane
with a constant complex value (y-coordinate). The
map of innerPar parallelizes the per-line computa-
tion over all the constant y-coordinate lines in the
plane. This attempt tries to parallelize computation
of every point in the grid.

render :: [[ Char ]]
render = do
  let evalPoint z =
    evalState (mandelbrot z) (0,(0.0,0.0))
  let innerPar l =
Since both maps attempt to parallelize using the strategy `rseq`, the parallelized computation attempts to reduce the call to weak head normal form. For the inner map that maps a computation on a `Point` returning a `Char`, reducing to weak head normal form actually completes the computation. The outer map, however, maps a function that operates on a list of `Points` and returns a list of `Chars`. This reduction into weak head normal form will only reduce the result into a thunk head containing the full Mandelbrot iterative computation to be evaluated cons-ed onto a thunk representing the tail.

As shown in figure 2, the resulting event log shows that the load on the processors is extremely unbalanced. This is likely due to the fact that vast majority of much of the iterative calculations performed on each point in the grid is actually still serialized and only the first computation of each row is fully sparked and parallelized. This attempt results in an average run time of 39.49s which is still a 2.033x speedup with about 9.0% of that time dedicated to garbage collection. However, in order to more fully take advantage of the four cores, more point computations should be distributed in parallel.

**4.2 deepseq Parallel Traversal**

This parallel implementation of the Mandelbrot renderer uses `deepseq` which will force the full evaluation of its argument in parallel.

```haskell
let evalPoint z = evalState (mandelbrot z) (0,(0.0,0.0))
let innerPar l = map evalPoint l 'using' parList rdeepseq
map innerPar grid 'using' parList rdeepseq
```

Since `rdeepseq` reduces the given computation to normal form, each inner map is evaluated in parallel, resulting in a much more balanced load for the processors, as seen in the event timeline.

This implementation resulted in an average run time of 11.56s or a 6.94x speedup with about 9.2% of that time dedicated to garbage collection. Due to the large number of sparks that either fizzled or were garbage collected, it’s likely better performance can be achieved by parallelizing computations in a more controlled manner.

**4.3 Real Axis Parallel Traversal**

In this implementation of the parallel Mandelbrot renderer, only the inner map is a parallel traversal. This means that within one line of the complex plane
(holding constant the y-coordinate or complex component of the complex number), iteration over each point is being computed in parallel with the others.

\[
\text{render} :: \{\text{Char}\}
\]
\[
\text{render} = \text{do}
\]
\[
\text{let evalPoint} \ z =
\]
\[
\text{evalState} (\text{mandelbrot} \ z) (0, (0.0, 0.0))
\]
\[
\text{let innerPar} \ l =
\]
\[
\text{map evalPoint} \ l \ \text{‘using’ parList rdeepseq}
\]
\[
\text{map innerPar} \ \text{grid}
\]

In per-processor breakdown it can be seen that certain processors ran a significantly greater number of sparks than others. This is largely due to the way the computation was parallelized. Since all computations were parallelized pointwise, certain points outside the Mandelbrot set will escape earlier and terminate almost immediately compared to points inside that Mandelbrot set that will fully iterate 2000 times before terminating.

This implementation resulted in an average run time of 13.531s or a 5.93x speedup with about 8.65% of that time dedicated to garbage collection. This implementation results in fewer troughs in activity due to garbage collection in addition to having a per-processor distribution with a more similar number of sparks.

4.5 Buffer-Limited Complex Axis Parallel Traversal

This approach to parallelizing the Mandelbrot renderer implements the same parallel map using deepseq on the outer map (holding the complex component of the number constant), but this time uses a buffer limit of 100 sparks. This will further limit the excessive creation of sparks.

\[
\text{render} :: \{\text{Char}\}
\]
\[
\text{render} = \text{do}
\]
\[
\text{let evalPoint} \ z =
\]
\[
\text{evalState} (\text{mandelbrot} \ z) (0, (0.0, 0.0))
\]
\[
\text{let innerPar} \ l = \text{map evalPoint} \ l
\]
\[
\text{map innerPar} \ \text{grid ‘using’ parList rdeepseq}
\]
Figure 6: Timeline of events by processor for the parallel Mandelbrot renderer implemented with one outer buffer-limited parallel map with the strategy rdeepseq and a buffer size of 100 sparks.

```ocaml
let evalPoint z =
  evalState (mandelbrot z) (0,(0.0,0.0))
let innerPar l = map evalPoint l
map innerPar grid
'using' parBuffer 100 rdeepseq
```

As the event timeline in Figure 6 shows, there are still steep troughs due to garbage collection, but they are much narrower (hence briefer) than in previous runs.

This implementation resulted in an average run time of 9.59s or a 8.59x speedup with about 6.8% of that time dedicated to garbage collection.

### 5 Discussion

For a basis of comparison, the speedup for each method of parallelization was calculated as the number of times faster it ran than the serial program for that number of cores. The results are shown in Figure 7. While all parallelization methods result a \( \geq 4x \) speedup, the increase from 4 to 8 cores yields a smaller acceleration than previous increases in available cores for across all methods. Additionally, the buffer-limited complex parallel map denoted Deep2 outperforms all methods up until 8 cores, when it’s speedup becomes comparable to that of the parallelized map across the complex range.

In order to normalize for small implementation level differences between the different methods parallelization, speedup was also normalized as the number of times faster it ran than the one core run for the serial program for that number of cores. The results are shown in Figure 8.

These metrics for speedup similarly confirm the results discussed above. Interestingly enough, as the sequential program was run with more cores it actually saw a run time slow down by several factors.

The methods that require an inner parallelized map over the complex line of the grid result in similar total numbers of sparks generated, as shown in Figure 9. This makes sense since this inner map creates sparks at the Point granularity while the other two methods map at the complex line level (constant y-coordinate).

In order to describe the utility of the sparks created and the efficacy of the parallelization methods, the percentage breakdown of what happened to the sparks produced by each method was analyzed. These results are show in Figure 10. No parallelization methods resulted in duds. The first parallelization attempt making use of both parallel maps and rseq results in a large fraction of overflowed sparks. The second parallelization attempt that used both parallel maps but with the rdeepseq strategy that forces complete evaluation to normal form results in
a number of fizzled sparks. Finally, the outer parallel map over the complex range and the depth limited parallel map over the complex range result in fully converted sparks.

To analyze the load distribution across the available cores, the standard deviation of sparks of that type across each core was calculated and normalized into a percentage according to the total number of sparks of that type. The results are shown in Figure 11.

The parallelization methods that used two parallel maps of rseq and one parallel map across the real plane resulted in the largest percent standard deviations by a wide margin. Interestingly, the method that similarly used two parallel maps only with rdeepseq rather than seq has significantly smaller deviations. The final two methods that involve mapping across the complex plane with rdeepseq (one buffer limited and one not) have nearly no deviation across cores and are thus the most well load-balanced.
Figure 11: The standard deviation of the given spark metrics between the four processors for each parallelization method. This shows how well distributed work was between CPUs.