Fraktell - Fractal Visualization in Haskell

Riley Weber
University of North Carolina at Charlotte
Under direction from Dr. Ali Sever
github.com/rileyweber13/fraktell

Introduction

Fractals are a fascinating mathematical phenomenon that can be explored by visualizing them. This task is iterative in nature and lends itself nicely to computer imaging. Furthermore, Haskell, a functional programming language, is well suited to perform the mathematical operations that are common in visualizing fractals. While many programs already exist with this functionality, there does not exist one written in Haskell. Therefore, I have chosen to leverage the functional nature of Haskell and its close association with mathematics to visualize the julia set. My goal was to demonstrate the chaotic nature of the julia, exhibit the property of fractals to be self-similar, and find speedup through parallelism.

Haskell

Haskell is a “purely functional” statically typed programming language with a robust set of features, including type inference, function currying (aka partial evaluation), lazy evaluation, concurrency, and parallelism [6]. It also has a well maintained package manager and package repository [14]. This language is closely tied to lambda calculus and combinatory logic, fundamental theories that form the underpinning of all functional programming languages [13]. In fact, the main page of the haskell wiki notes that the lambda symbol was chosen as the symbol for haskell because of the great importance of lamba calculus [7]. The language is named after Haskell Curry, who pioneered combinatory logic, a superset of lambda calculus that was key to making functional programming possible [2][4].

History

By 1980’s, an explosion of popularity surrounding functional programming had left programmers with numerous functional languages that all had effectively the same constructs, but unique syntaxes [2]. This included languages like ML, Hope, and Miranda, and many of these languages were proprietary, with closed source implementations that stifled progress [1]. In response to this, functional programmers convened in 1987 to remedy this situation. Their solution was Haskell, which was designed to be a unifying, standard language that they could all use and thereby easily share work and further the development of functional programming. In addition to this, the committee had 4 other goals associated with the development of this new

language. They declared that Haskell should be suitable for both education and practical use, that its grammar should be comprehensively and formally described, that it should be freely available, and that it should be based on principles that functional programmers agreed upon [2].

It is difficult to unify competitors, and history has shown that introducing a new “standard” frequently just adds one more competitor to the mix, rather than unifying users. However, with Haskell, this was not the case. As of November of 2019, Haskell is used much more widely than all functional programming languages except LISP. To be fair, there exist companies that use less popular functional languages like erlang, but Haskell still maintains more frequent public use and discussion [12].

Elements of Haskell

In haskell, reserved words are called “keywords”. In addition to operators, the primary reserved words are listed below:

- `do - used to indicate that the next block of code is a procedure`
- `data - used for introducing abstract data types, discussed below`
- `deriving - used to automatically generate instances for for operations that support it, notably “show” and “eq”`
- `import - used to import modules`
- `module - keyword that indicates the beginning of a module`
- `type - introduces an alias for another type`

Haskell supports the basic types as expected in other languages, and each starts with a capital letter:

- `Bool`
- `Char`
- `Double`
- `Float`
- `Int`
- `String`

`Double` is preferred over `Float`, as more time has gone into optimizing double. There also exists a type `Integer` that has no max size (besides the memory limits of the system), but it is slower than `Int` [9].

Additionally, `List`s and `Tuple`s are fundamental parts of haskell. Members of a `List` must all have the same type, whereas members of `Tuple`s may have varying types. Combining `List`s and `Tuple`s with themselves or each other is supported and trivial. Additionally, while `Tuple`s are quite primitive, `List`s have a great deal of built-in functionality. It is easy to index, append to, and pop from lists. Furthermore, using the `map` function, one may apply a function to all

members of a list, and the `fold` function allows lists to be reduced in arbitrary ways [7]. Haskell's lists are powerful and should be leveraged.

Syntax and Semantics

A Haskell program is a collection of modules, usually one per file, that make up a piece of software. Each module is made up of declarations that describe data types, values, functions, and so on. Central to haskell programming are expressions, which are most commonly seen in function declaration and application. Haskell programmers will most likely spend the majority of their time composing expressions [3]. This section will focus primarily on the declarations and expressions necessary to describe data types and functions.

User-defined Datatypes

Following is a simple example of how one may describe a data type:

```
data BookInfo = Book Int String [String]
                deriving (Show)
```

Taken from chapter 3 of Real World Haskell [9].

Our custom data type is an object that contains an integer, string, and list of strings. The names `BookInfo` and `Book` do not need to be different, and in practice they are frequently the same. However, we use two different words here to emphasize that later in our code, this data type will be referenced using the word after the `=` operator. The line `deriving (Show)` allows the function `show` to convert the object to a string that may be printed or saved.

The intention of this data type is to contain an ID number, title, and list of authors for this book. However, this is not immediately made clear by reading the source code. A programmer may make this intention clearer by using type synonyms [9].

```
type ISBN = Int
type Title = String
type Authors = [String]
```

This code declares several new identifiers (on the left of the `=`) and their synonyms (on the right). Therefore, an `ISBN` is just another name for an `Int`. We can leverage this to improve our declaration of `BookInfo` from earlier:

```
data BookInfo = Book ISBN Title Authors
                deriving (Show)
```

This makes the intention of the code clearer. It is possible to contain this same information in a tuple. Therefore, the following two lines are effectively equivalent, though their representations and usages are different.

```
myBook1 = (123456789, “My Book”, [“Programmer A”, “Programmer B”])
myBook2 = Book 123456789 “My Book” [“Programmer A”, “Programmer B”]
```

Choosing to use a tuple has some pitfalls, as the type checker cannot infer your intention from the datatypes of the parts of the tuple alone. To illustrate this, consider a program that supports both cartesian and polar coordinates, and has functions to handle both. A programmer declares two coordinates, one with each system.

```
cartesian = (1.0, (sqrt 3))
polar = (2.0, pi/3)
```

Assuming we are using radians to represent angle measurements, these points refer to the same location in 2D space. However, both `cartesian` and `polar` have the type `(Double, Double)`. This design would not prevent a programmer from passing polar coordinates to a function expecting cartesian coordinates. This is certainly not desirable behavior, and the programmer would not get their expected result at runtime! However, by defining types for both kinds of coordinates, we can leverage the type checking of the Haskell compiler to turn a (possibly silent) runtime error into a compile error.

```
data Cartesian2D = Cartesian2D Double Double
                   deriving (Eq, Show)

data Polar2D = Polar2D Double Double
               deriving (Eq, Show)
```

Taken from chapter 3 of Real World Haskell [9].

Notice how `Cartesian2D` and `Polar2D` are identical in the sense that they contain two doubles and derive the functions `Eq` and `Show`, but they are given distinct names. It is now possible to rewrite our values from earlier using these new data types:

```
cartesian = Cartesian2D 1.0 (sqrt 3)
polar = Polar2D 2.0 pi/3
```

With this extra information, the compiler is made aware of the fundamental difference between cartesian and polar coordinates. The programmer is now able to write functions that accept Cartesian2D as a parameter, and this hypothetical function would no longer accept polar coordinates, as the compiler would report an error.

This illustrates a key part of haskell philosophy: compile-time errors are better than runtime errors. The Haskell compiler is built with a robust and powerful type checker that works hard to prevent unexpected behavior. In my experience, Haskell has type checking of a caliber far beyond most programming languages, and the design of the language facilitates this. The type checker was a feature that was sometimes frustrating but ultimately helpful and empowering as I wrote fraktell.

User-defined Functions

Functions have two parts: the type signature (optional) and the actual definition. Here's a simple example:

```
triple :: Int -> Int
triple x = 3 * x
```

The first line is the type signature, which indicates the name of the function (`triple`) as well as the type of the parameter and the type produced by the function. Type signatures are usually optional, because haskell can infer types in a large variety of situations. The second line defines what the function actually does. It declares a name for the parameter, which is `x` in this case, and then describes what the function actually does. Notice how closely this mirrors the algebraic syntax taught in public schools to teenagers: this is effectively the same as saying $f(x) = 3x$. This function can be applied as follows, and would produce 24.

```
triple 8
```

It is also possible to define piecewise functions. Consider the following continuous probability distribution:

$$f(x) = \begin{cases} x < 0 & 0 \\ 0 \leq x \leq 2 & 0.5 \\ 2 < x & 0 \end{cases}$$

In haskell, this can be written using "guards" as follows:

```
myDist :: Double -> Double
myDist x
  | x < 0 = 0
  | 2 < x = 0
  | otherwise = 0.5
```

To a programmer used to imperative languages, this is reminiscent of a switch statement, and it is probably the most commonly used control structure in haskell.

Haskell also supports a powerful feature called partial application. Consider a function that takes two parameters, one of which is a constant:

```
poly :: Double -> Double -> Double
poly c x = x ** 2 + c
```

In order to use this function, we can apply it as follows:

```
poly 3 4
```

This would produce $4^2 + 3$, which is 19. However, this is not ideal, as requiring the constant to be passed each time is error-prone. Alternatively, the programmer may “partially apply” this function by supplying only one argument, which gives a new function:

```
do
  let poly2 = poly 3
  poly2 4
```

The `do` keyword means that these lines will be executed one after another, like in imperative programming, and a `do` block may be thought of as a procedure. This snippet of code would produce the same output as the earlier example, 19, but it provides some assurance that the constant will not change as the function is applied across many values of x. This is especially useful because many functions in Haskell libraries take a function as a parameter, and that function is required to fit requirements as to the number and kind of parameters it takes. For instance, fraktell uses the `makeImageR` function from the Haskell Image Processing (HIP) Library, which uses a user-supplied function that takes an x and y pixel coordinate as input and returns a pixel value as an output [5]. However, these kinds of functions frequently need more parameters than just a location in an image. Consider the following example:

```
generateImage :: Int -> (Int, Int) -> Pixel RGB Double
generateImage colorCode (x, y) =
  -- implementation goes here
```

makeImageR would not accept `generateImage` as a parameter, as it has too many parameters. However, if we partially apply it first, then it is acceptable:

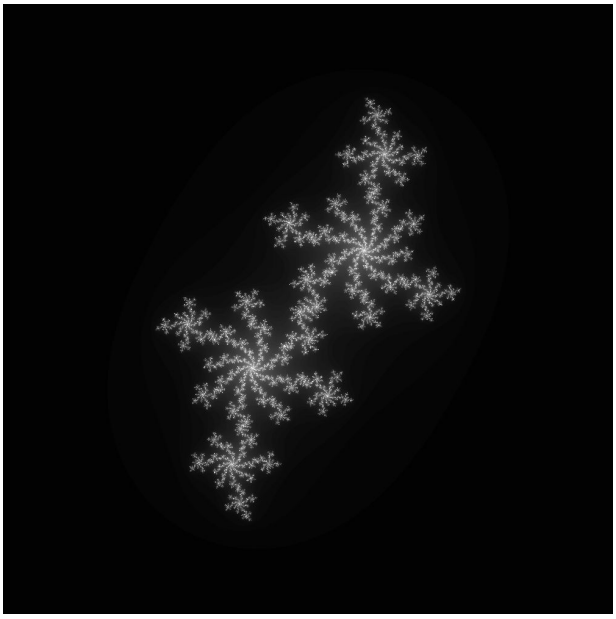
```
do
  let generateImage2 = generateImage 121212
  makeImageR VU (200, 200) generateImage2
```

Therefore, partial application provides a flexibility that would otherwise be impossible. Any function may now be passed to `makeImageR`, as long as one parameter of that function is a `Tuple` of two `Int`s and the rest of the parameters are partially applied beforehand.

An Evaluation of Haskell

The Good

Where haskell truly shines is in its ability to nearly always produce compiled code that behaves as the programmer expects. As an example, after working on fraktell for some time, I was ready to test it by generating an actual image. I added the code to produce and save an image to disk, and after working through some compile-time errors, I was finally able to run fraktell. The first time I ran it, this image was produced:



It turned out I had mixed up the axes, and the image was therefore rotated 90 degrees from what I expected, but otherwise the program worked flawlessly. It is quite impressive that a reasonable visualization was generated on the first run of a new piece of software. This is a result of fraktell's draconic type checker, whose stringent compile-time requirements allow for fewer runtime errors.

Additionally, unit testing is very easy in Haskell. I was able to begin without thinking of how I/O would work, but instead focused on the core of the project: that is, the actual function for which I would find a julia set and the procedure to iteratively visualize it. Using hspec, I could jump right in and focus on what mattered most.

Additionally, I was impressed with the intuitiveness and clarity of the project directory structure, made possible in great part by the ease with which new project creation is handled in the haskell stack. It generates three directories for haskell code: app, src, and test. Through just 30 seconds of reading the stack documentation, it is clear what those are used for: “app” for the main file and other executable-related things, “src” for the helper stuff like data types and functions, and the “test” folder for unit tests [11].

Furthermore, by following the conventions of stack, it is incredibly easy to build and test a project. As long as dependencies are specified in the “package.yaml” file, stack will automatically manage dependencies and ensure that the compiler is aware of them when you run “stack build”, “stack run”, or “stack test”. The ease with which I could use the stack tool suite and the speed with which I could understand the directory structure were remarkable to me, as I have often had to wrestle with dev tools. For example, after years of working with Python, I still sometimes have trouble importing the modules and packages I write into other parts of my code.

The Inconvenient

GHC, the Glasgow Haskell Compiler, is the de facto standard compiler for Haskell, but the error messages it provides are often incredibly verbose and downright misleading. For instance, consider this example code:

```
module Main where

poly :: Double -> Double -> Double
poly c x = x ** 2 + c

main = do
  putStrLn show (poly 3 4)
```

Copy this to a file and attempt to compile it, and you will be greeted with approximately 20 lines of error reporting, most notably the report `Couldn't match type ‘a0 -> String’ with ‘[Char]’. Probable cause: ‘show’ is applied to too few arguments`. What is wrong with this code? It is nothing to do with `show`, in fact. `purStrLn` only accepts one argument, but in this case we have provided two, `show` and `(poly 3 4)`. The intent was to evaluate `(poly 3 4)` first, and so the code may be fixed by changing the final line to:

```
  putStrLn (show (poly 3 4))
```

This explicitly declares that `show (poly 3 4)` will be evaluated before being passed to `putStrLn`. Our error message did mention this somewhere, saying `The function ‘putStrLn’ is applied to two arguments, but its type ‘String -> IO ()’ has only one`, but this sentence was drowned out in 18 other lines of error reporting. As a

beginner to the haskell language, I found that I spent more time interpreting error messages than actually producing code.

Additionally, trying to do anything that required user input was difficult, and I imagine these difficulties extend to other kinds of I/O. Fraktell allowed for arguments to be supplied at runtime via the command line. This meant that the pure functions I was executing relied on input from the user; input that was only available after it was extracted from the command line arguments with an impure function. This dependency on I/O requires the programmer to ensure that first the parameters are parsed and then the function may be run. This sequence of actions is rarely thought of by programmers working in imperial languages, as it's entirely trivial. However, with Haskell's emphasis on pure functions and lazy evaluation, getting one thing to execute before another is not a core part of the language. In short, haskell does not make it easy to mix pure and impure functions.

Haskell has support for the `do` monad, which, as shown above, essentially allows for the programmer to write a procedure. This ensures order of operation, enabling actions like this:

```
do
  let args <- getArgs
  myFunc (args !! 0)
```

The `<-` operator causes the software to wait for the right hand side to be evaluated and sets the left hand side to be that value. The `!!` operator simply gets the value of the list at the index to the right of the operator. Therefore, we ensure that command line arguments are obtained from I/O before running the pure function `myFunc`. This seems simple enough at first, but this paper will show that `do` blocks behave entirely differently than haskell does otherwise, and the lack of uniformity increases programming complexity.

Guards (the use of the pipe symbol to create piecewise functions demonstrated earlier) do not work in do blocks, and so the programmer must either try to leverage unwieldy nested `if` statements, as below, or decompose their function into smaller functions.

```
if … then do
  doThing
else do
  if … then do
    doThing2
  else do
    doThing3
```

At first glance, these seem to follow the pythonic attitude of using whitespace as syntax. However, whitespace has no syntactical meaning in Haskell and instead if statements are

simply “`if ... then ... else ...`. Every `if` must have an `else`, and nested conditionals like this get very messy very quickly.

Additionally, if error messages were difficult outside of `do` monads, they are nearly impossible inside them. 20 or so lines of error reporting turned into pages and pages that I had to sift through. Haskell tries very hard to always keep its purely functional rules, and as a result, doing anything even remotely imperative feels like a hack. In refusing to break its own rules, Haskell requires the programmer to do so instead.

Regularity and Reliability

In this section, the the words “general”, “orthogonal”, and “uniform” as well as their various forms are used as defined by K.C. Louden and K.A. Lambert in Programming Languages.

Haskell is incredibly reliable. The autocratic type checker of the GHC ensures that as few things as possible are allowed to be run. The type checker is reliable and consistent, and it only allows functions that are absolutely typographically correct to be compiled. Therefore, a working haskell program will have many fewer edge cases than an equivalent program written in a language like C++.

Haskell is somewhat general. For one, it allows functions to be passed as if they were any other value. Additionally, as far as I can tell, everything in haskell is either a function or datatype, and most things are functions. For instance, the constant pi is exposed by a function of the same name in the `Prelude.Math` package.

However, the `do` monad damages the generality of Haskell, as described earlier. Additionally, the use of `=` is inconsistent. Outside a `do` block, function definition is simply indicated by the `=` operator, and intermediate values can be assigned by using that same operator inside a `where` block. However, inside a `do` block, using the `=` operator necessitates preceding it with the keyword `let`. As far as I have found, this is the only special case where this is required. Additionally, the inability to use guards inside a `do` block is an example of nonorthogonal behavior in Haskell.

Haskell is remarkably uniform. Functions only look like functions and data types only look like datatypes. However, again, the `do` monad is the ugly exception, where a `do` block looks somewhat like a `where` block but behaves entirely differently. Additionally, functions that use a `do` block are the only functions that are not described with the `=` symbol.

Fraktell - Fractal Visualization in Haskell

Fraktell is designed to visualize instances of the Julia set. It is intended to demonstrate a test-first development practice, leverage purely functional properties, utilize lazy evaluation, be

purely written in Haskell, demonstrate the chaotic nature of the julia set, and generate images large enough to demonstrate self-similarity in the julia set. Defining and generalizing the julia set is outside the scope of this paper, but suffice it to say that a julia set is associated with a function f that takes a single complex number as an input and produces a single complex number as an output. For that function, the julia set is the set of all points in complex space that satisfy some requirement when the function is applied to that point. In practical cases, only a small subset of points are considered.

Method

At the core of the program, a function must be described for which we will create a julia set. Each pixel in our final image is mapped to a point in complex space. Then, for each of these points, the function is applied to that point and then applied to itself recursively until it surpasses a given value, the “escape radius”. The number of iterations before diversion is recorded as the pixel value. This is repeated for every pixel in the image. The algorithm is written below in haskell-style functional pseudocode.

```
visualizeJuliaSet f outputFilename =
  writeImage f image outputFilename
  where
    image = makeImageR VU (1000, 1000) generatePixelF
    generatePixelF = generatePixel f

generatePixel f (x, y) = numIterationsToDivergance point 1 maxIter f
  where
    point = pixelToComplexPlane (x, y)
    maxIter = 100

numIterationsToDivergance point i max f
  | i > max = max
  | magnitude(point') > escapeRadius = i
  | otherwise = numIterationsToDivergance point' (i+1) max f
    where
      point' = f point
      escapeRadius = 1.5
```

This is effectively how fraktell works, but simplified by relaxing type constraints, assuming `pixelToComplexPlane` is implemented, and providing hardcoded values for the escape radius and maximum number of iterations. For a fully working haskell program, the reader is encouraged to view the source code provided in the references section [10].

Despite the simplicity of these functions, they illustrate powerful features in Haskell; `visualizeJuliaSet` creates an image and performs I/O to disk and

`numIterationsToDivergance` demonstrates both recursive code and guards. This is a good example of how guards made it easy to write fraktell: recursive code lends itself nicely to piecewise functions, as recursive functions require one or more base cases. In this case, the base cases are `i > max` and `magnitude(point') > escapeRadius`. Additionally, notice that the parameter f is passed through each function. f is required to be a function and is finally applied to each point in the complex plane associated with each pixel. The function f is also partially applied inside `visualizeJuliaSet` to satisfy the requirement of `makeImageR` to be provided a function which only takes `(Int, Int)` as a parameter. Because of this work, it would be trivial to visualize julia sets for other functions, even though only one function, $f(x) = z^2 - c$, was considered here.

In order to color the images, pixels were represented using HSI, a method for describing pixels that records values for the Hue, Saturation, and Intensity. The number of iterations before divergence was normalized and used as the value of the Hue.

Functions were also written in the main module to parse command line arguments and print usage text. Additionally, a function was written to benchmark the piece of software using the criterion package, but the results were far from accurate and were discarded. Benchmarking haskell code is difficult because of lazy evaluation: the runtime may not run the thing you want to benchmark, even when it seems that it must be executed. A reasonable benchmark was obtained by writing a python script that can be found in “benchmark/benchmark.py”, which records the entire time to run fraktell. Unfortunately, this means that the time to write to disk is also included, which is something I have less control over.

Basic Usage

The primary usage of fraktell is listed below:

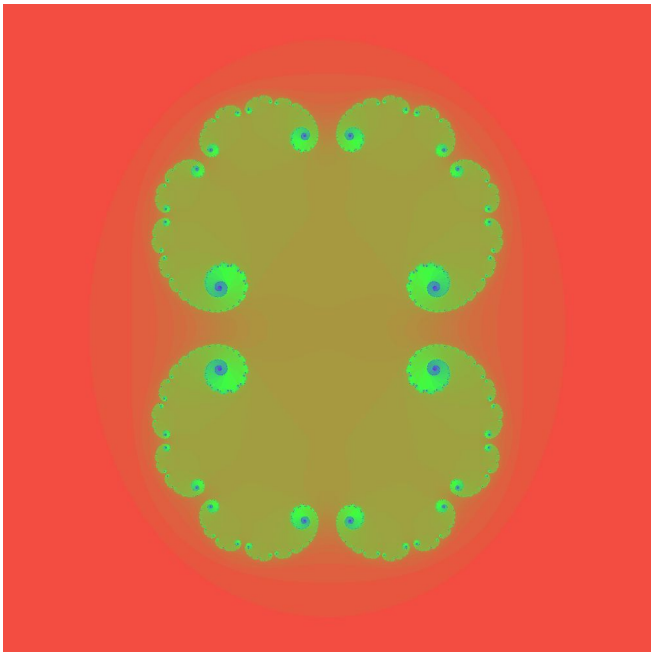
```
stack run escape_radius width height max_iter output_filename
          [arrType] [func_num [constant1 [constant2] ... ]]
```

`stack run` is used instead of directly calling the executable `fraktel-exe`. This way, dependencies are automatically managed and builds are continually updated.

To generate a visualization, you could call the program like this:

```
stack run 1.5 1000 1000 100 "images/output.png" RPU 1 "(0.285 :+ 0)"
```

This uses function 1, which is defined internally as $f(x) = z^2 - c$. In this case, (0.285 + 0i) is provided for c. Running fraktell with these parameters produces the following image at images/output.png:



Experiments

This project had three primary goals: to demonstrate the chaotic nature of the julia set, to exhibit self-similarity in fractals, and to show speedup through parallelism. In order to illustrate chaos, we will increment c by small amounts. To show self similarity, we will generate extremely large images and zoom in on them. To show speedup, the program is benchmarked with an external script and the times are graphed for various image sizes.

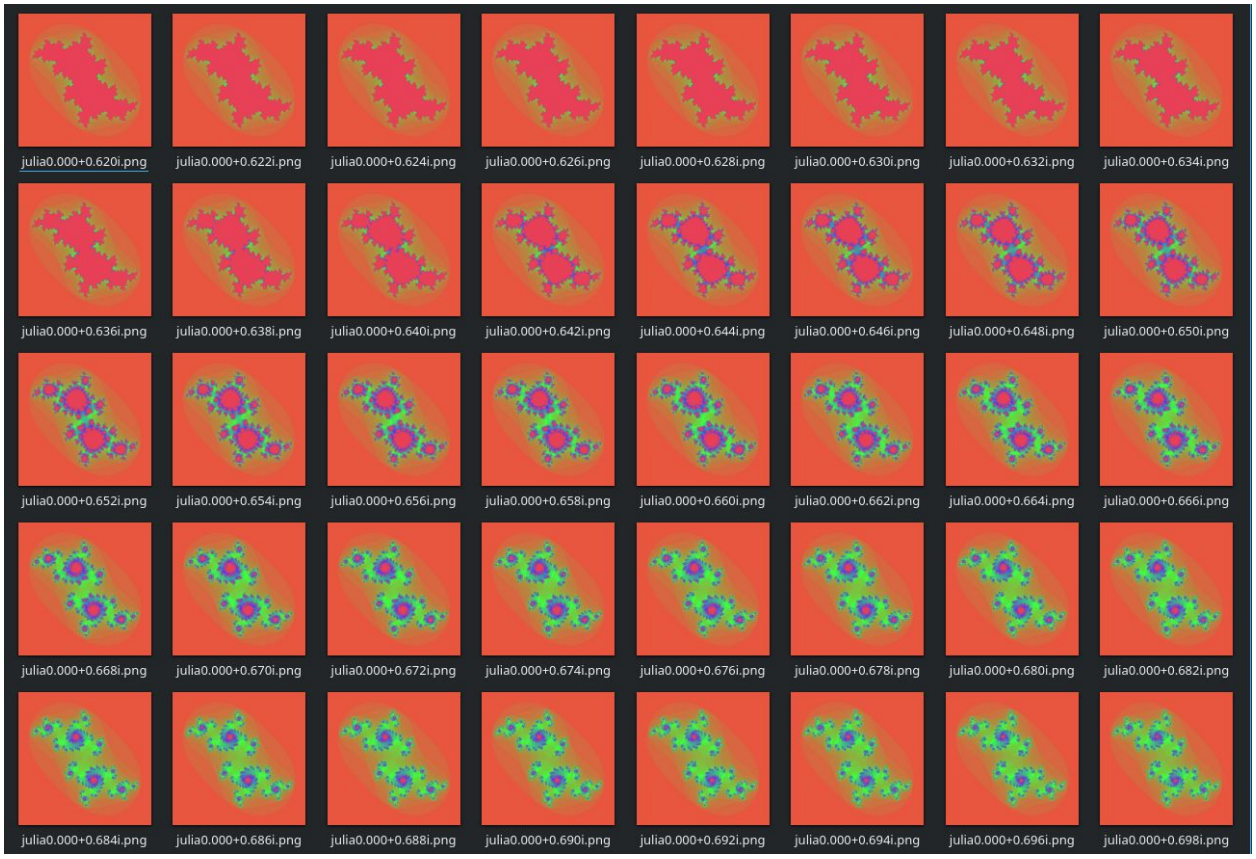
Chaos in the Julia Set

Our choice of c will range from (0 + 0.620i) to (0 + 0.698i), inclusive, while the other parameters remain constant. The width and height were set to 1000, the escape radius to 1.5, and the maximum number of iterations to 100. Then, a simple while loop in a python script was used to run the program repeatedly:

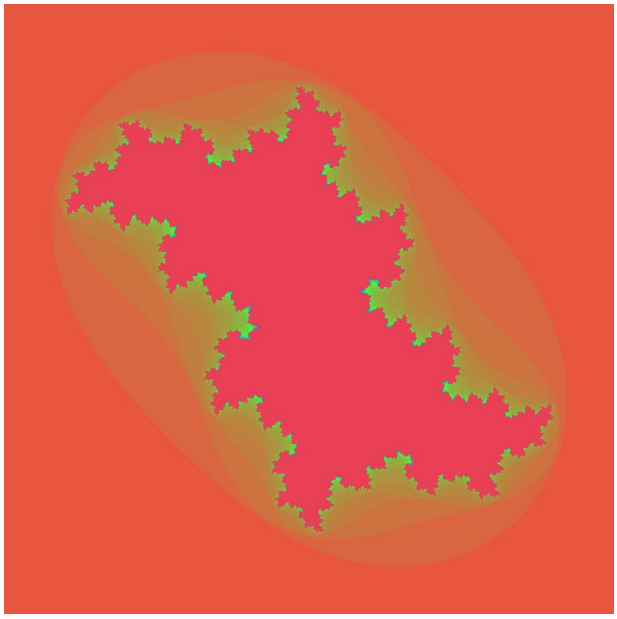
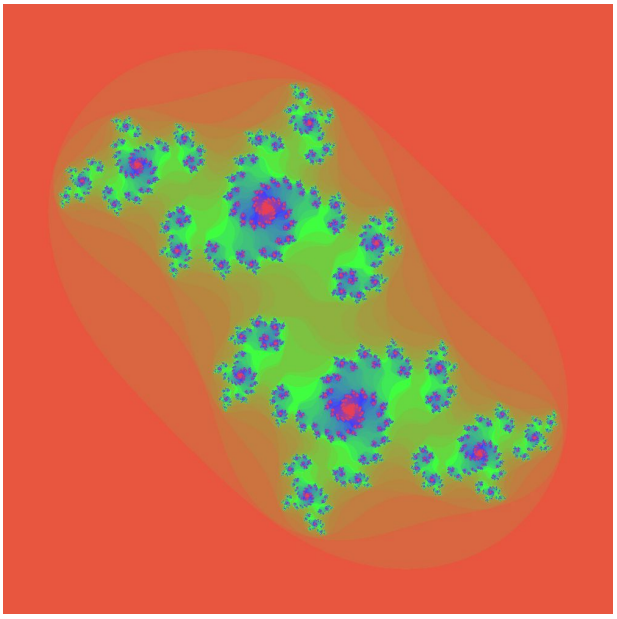
```
a = 0.62
size = 1000
max_iter = 100
while a < 0.70:
    c_str = "{:.3f}+{:.3f}i".format(0, a)
    subprocess.run([
        'stack', 'run', '1.5', str(size), str(size), max_iter,
        'images/julia-0.6i_to_0.66i' + c_str + '.png', 'RPU', '1',
        '(0 :+ ' + str(a) + ')'
```

```
])
a += 0.002
```

Notice that “subprocess.run” uses constant parameters besides a, which is used as the imaginary part of the constant c. Also notice that a is only changed by two thousandths every iteration. Running this script produces the following images as output:



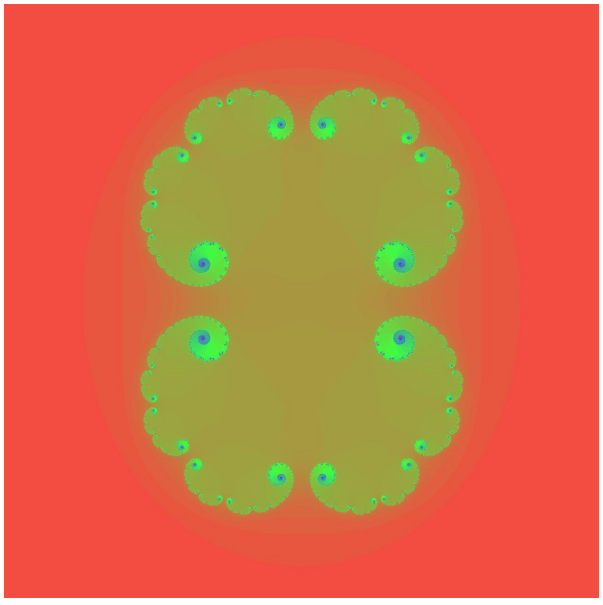
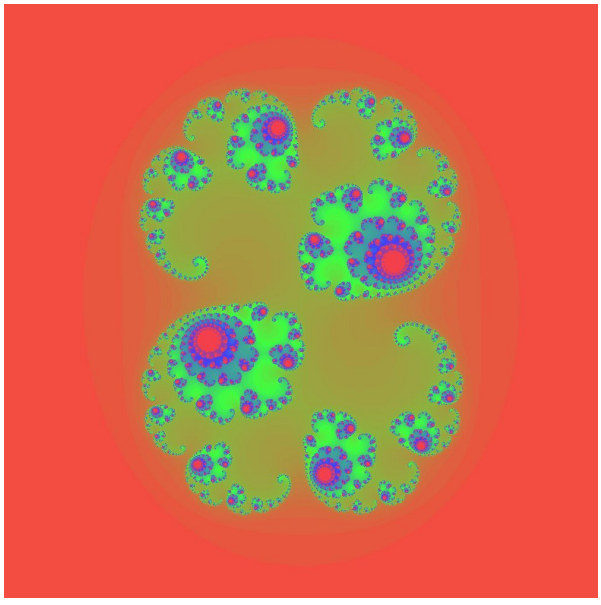
It is immediately apparent how dramatically different this set of images is from the one produced earlier where c = (0.285 + 0i). Additionally, while having the same basic shape, the first and last images produced are dramatically different:

Remember that in this case, the only change between iterations was to increase the imaginary part of c by 0.002. The difference in parameters between the first and last image is only 0.078, less than one tenth.

Another example of chaos in the julia set is shown when we compare images generated with c = (0.285 + 0i) and c = (0.285 + 0.01i). First the commands used are printed, followed by the images produced:

```
stack run 1.5 1000 1000 100 "output1.png" RPU 1 "(0.285 :+ 0)"
stack run 1.5 1000 1000 100 "output2.png" RPU 1 "(0.285 :+ 0.01)"
```

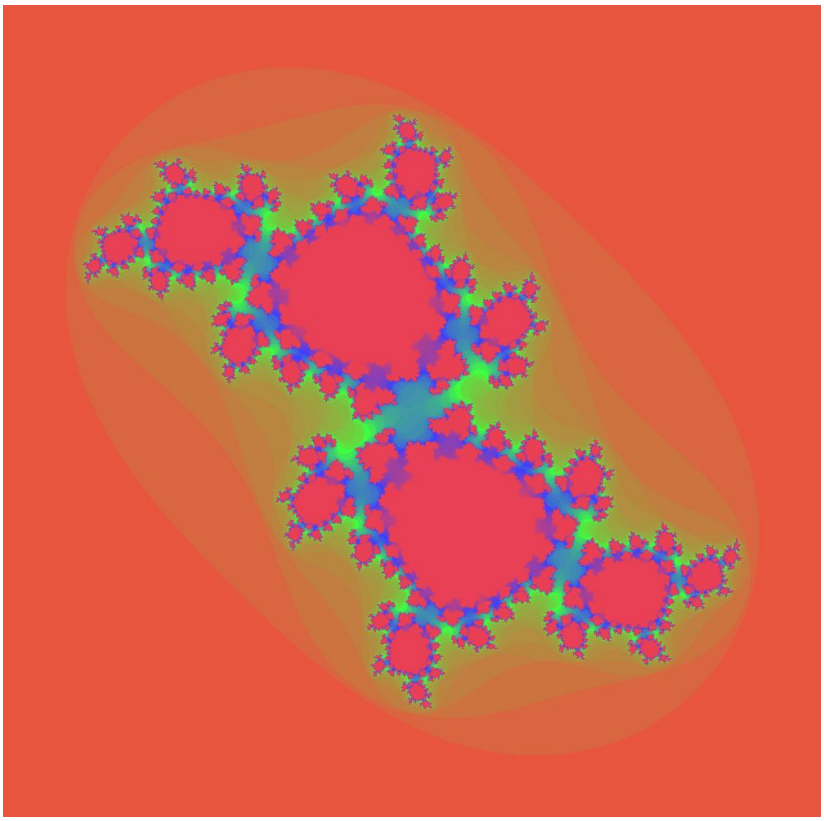
An increase of merely one one-hundredth in just the imaginary part of c has created a very different image.

Self-Similarity in the Julia Set

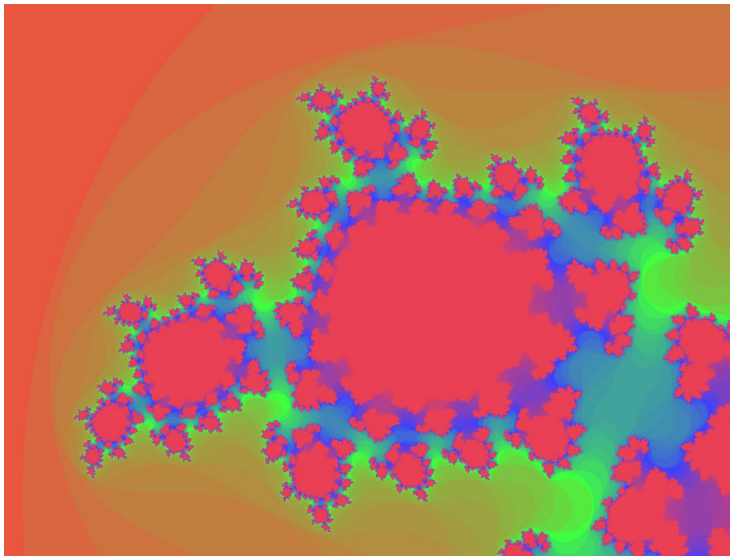
To demonstrate self-similarity, two fractals were generated at a resolution of 100 Megapixels, and several smaller images were cropped out of each. One example is the fractal generated with c = (0 + 0.646i), using the following command:

```
stack run 1.5 10000 10000 50 "images/output.png" RPU 1 "(0 :+ 0.646)"
```

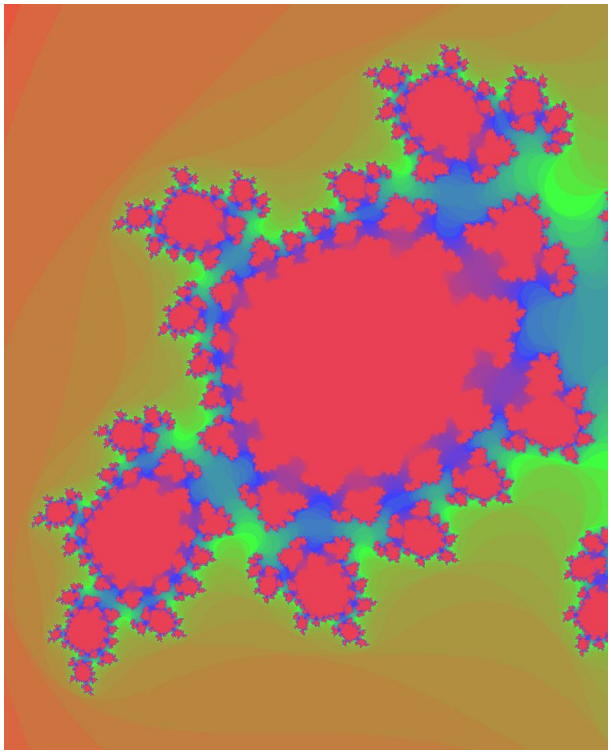
Notice the increased dimensions: 10,000 by 10,000 pixels. Following is the image produced by this command, resized to fit into this document.



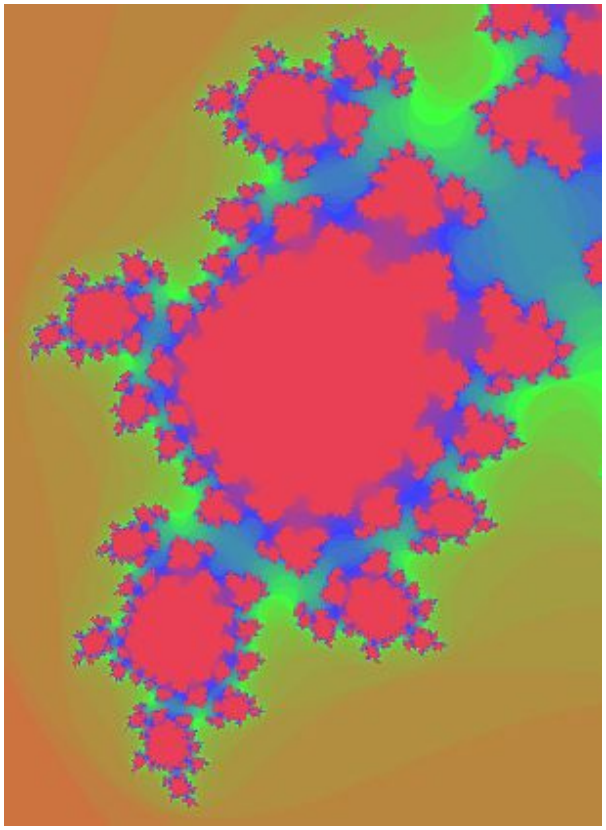
The primary feature of this image is three large blobs with small, similarly-shaped blobs surrounding each. This is rotationally similar around the center, with two different “arms” that have these properties. We will focus on one of these arms, and we begin by zooming in on the smaller two main blobs of the top arm:



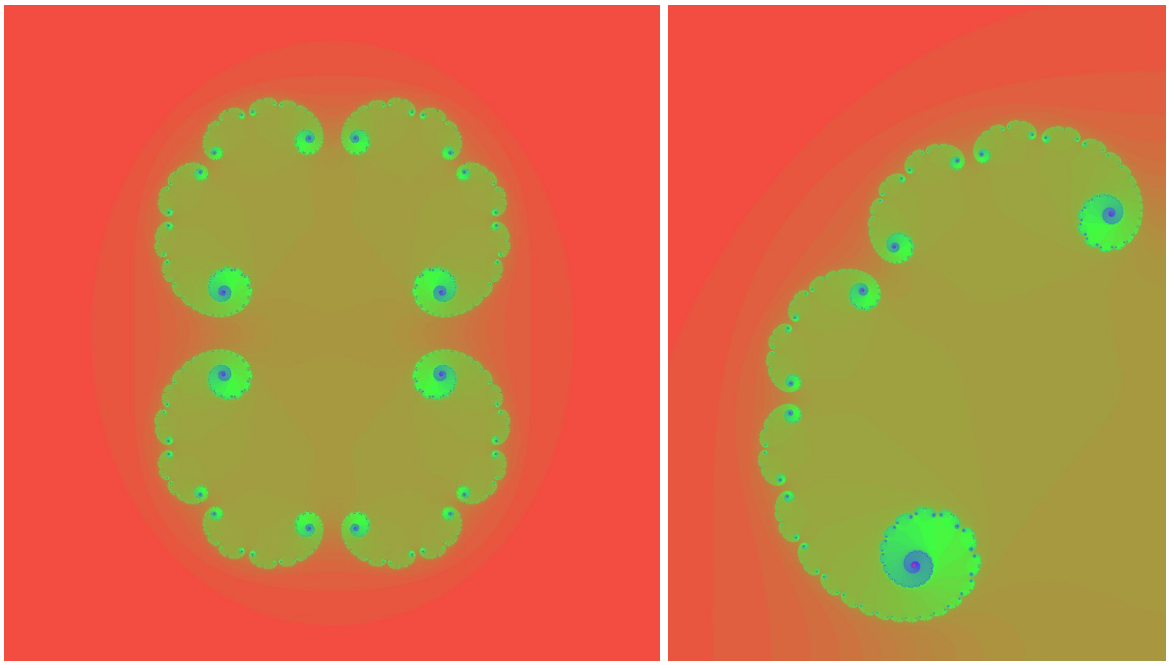
Once zoomed in, this looks fairly similar to the first image. One of the very small blobs from the first has grown to replace a larger one, and each is surrounded in a similar fashion. Let's zoom in again:

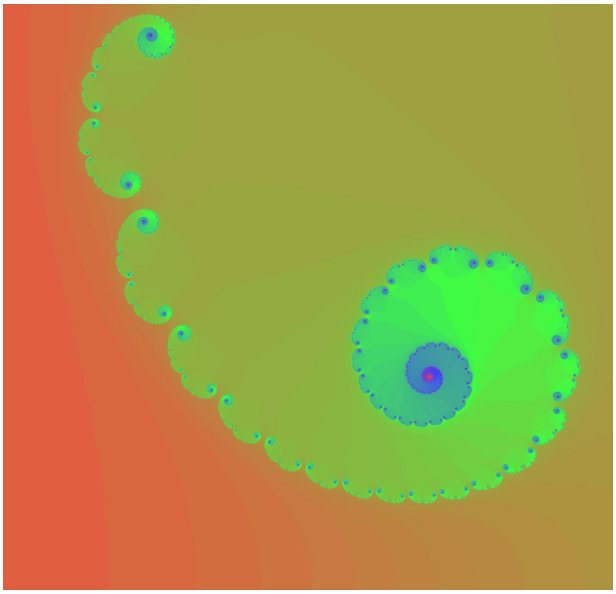


And, finally, one last time:



Notice that each iteration the image is zoomed in looks increasingly similar to the previous. This extremely large image was successful in demonstrating self-similarity in the julia set. Now, we explore this property in the julia set generated with c = (0.285 + 0i).





The factor by which the image is zoomed increases as the reader moves from left to right, top to bottom. Notice how the original image is made up of four swirls, each with a big end and small end. As we zoom in, we find that each of these swirls is made up of smaller swirls that also have a big and small end.

Speedup Through Parallelism

Fraktell performs math for each pixel of an image. The computations per pixel cannot be parallelized, as each iteration is dependent on the previous. However, it is possible to parallelize across pixels. The “MakeImageR” function calls a pixel-generating function for each pixel. Therefore, if we are to parallelize this program, we must investigate “MakeImageR”. Thankfully, this function supports various data structures to represent images, some of which implement parallel operations. This means that all that we need to do to use parallelism is to tell `MakeImageR` to use a data structure that supports parallel operations.

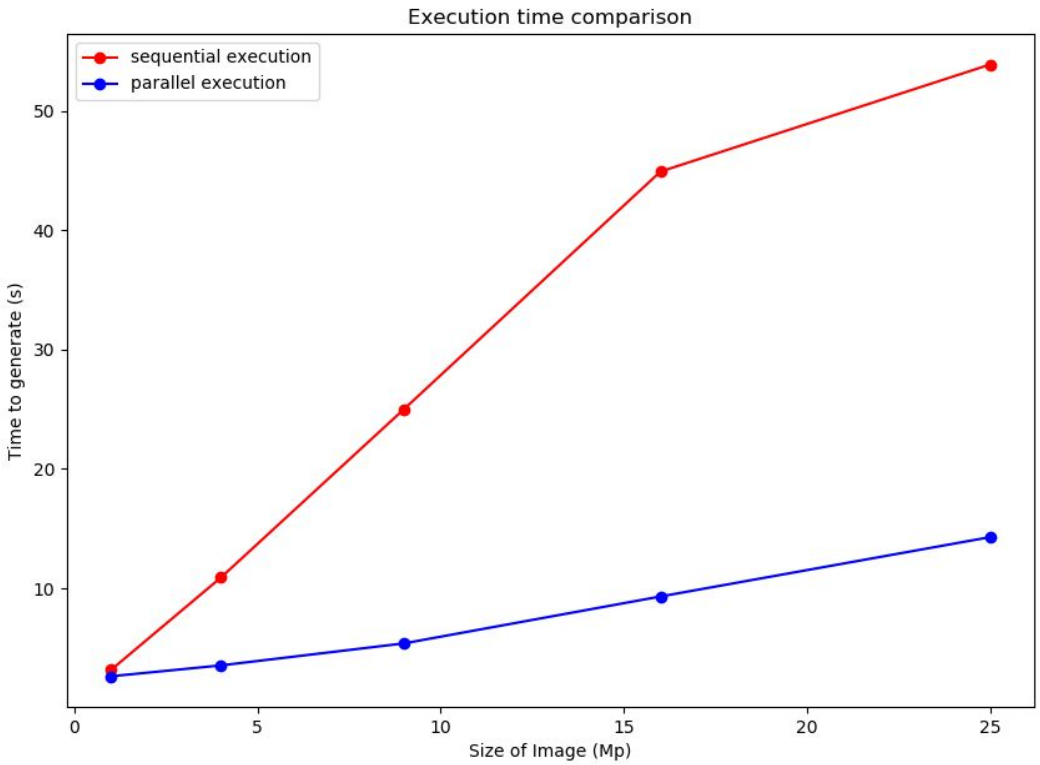
Benchmarking was done by comparing the time needed to generate an image for various sizes. These tests were run on a machine with 12 cores using a python script. The heart of this script is the function `run_julia_hs`, included here:

```
def run_julia_hs(size, num_procs, array_type):
    start_time = time.time()
    subprocess.run(['stack', 'run',
                    '+RTS', '-N' + str(num_procs),
                    '-RTS', '1.5', str(size), str(size), '100',
                    'images/output.png', array_type, '1',
                    '(0.285 :+ 0)'])
    end_time = time.time()
```

```
duration = end_time - start_time
num_pixels = size ** 2

pixels_to_mpixels = 1e-6
print(num_pixels * pixels_to_mpixels, array_type, num_procs,
      duration, num_pixels/duration * pixels_to_mpixels,
      sep=', ')
```

We will run this function for images of dimension 1000 x 1000 through 5000 x 5000, inclusive, incrementing each dimension by 1000 each time. This process is repeated for two types of arrays: “RSU” (Repas sequential) and “RPU” (Repas parallel). A csv is generated which can then be used to plot time for each image size:



There is a dramatic improvement of performance when using a data structure that supports parallel options. This experiment summarizes parallel programming in haskell. It is difficult to expose all the possible performance, as it is not immediately clear what exactly the machine is doing and in what order. However, there are many tools and libraries that are very easy to

incorporate because of the functional nature of haskell. Knowing that the function that generated each function was pure made it trivial to avoid race conditions.

Conclusions and Future Work

Haskell is well suited to this problem, despite the difficulties encountered with I/O and the `do` monad. The basic functionality of Haskell was demonstrated, and it was seen how a programmer can reduce runtime errors by leveraging the powerful type system of Haskell. The chaotic nature of the julia set was investigated by iterating values for the constant c, and images 100 MP in resolution were generated to explore self-similarity. Finally, speedup was shown by leveraging Repas parallel arrays. The ease with which good parallelism can be obtained and the difficulty associated with extracting excellent parallelism were discussed.

In the future, it would be advantageous to investigate reducing the need for the `do` monad and impure functions. Furthermore, criterion is a powerful package that should be explored for benchmarking, even though initial results were incorrect. Learning more about how the haskell runtime works may allow for proper benchmarking with criterion. Furthermore, the function `visualizeJuliaSet` in fraktell has hardcoded keywords for each array representation, and it may be possible to simply pass the array representation as a parameter. Additionally, more control over parallelism is possible if functions are written to replace `makeImageR` and `writeImage`. There are many more functions that can be explored, and fraktell could be made to support arbitrary functions passed as command-line arguments. A GUI to simplify use would be greatly beneficial, as well as the ability to regenerate just a small part of an image so that immense images do not need to be generated.

Bibliography

1. FutureLearn, “Brief History of Haskell - Functional Programming in Haskell,” FutureLearn. [Online]. Available: https://www.futurelearn.com/courses/functional-programming-haskell/0/steps/27218. [Accessed: 29-Nov-2019].
2. “Haskell 98 Language and Libraries,” The Haskell 98 Language Report. [Online]. Available: https://www.haskell.org/onlinereport/index.html. [Accessed: 23-Nov-2019].
3. *Haskell 2010 Language Report*, 20-Jul-2010. [Online]. Available: https://www.haskell.org/onlinereport/haskell2010/. [Accessed: 01-Dec-2019].
4. “Haskell Brooks Curry,” Haskell Curry (1900-1982). [Online]. Available: http://mathshistory.st-andrews.ac.uk/Biographies/Curry.html. [Accessed: 28-Nov-2019].
5. “Haskell Image Processing (HIP) Library: Graphics.Image,” *Graphics.Image*. [Online]. Available: https://hackage.haskell.org/package/hip-1.5.4.0/docs/Graphics-Image.html. [Accessed: 01-Dec-2019].
6. “Haskell Language,” Haskell Language. [Online]. Available: https://www.haskell.org/. [Accessed: 29-Nov-2019].
7. “Introduction,” Introduction - HaskellWiki. [Online]. Available: https://wiki.haskell.org/Introduction. [Accessed: 28-Nov-2019].
8. K. C. Louden and K. A. Lambert, *Programming languages: principles and practice*. Boston, MA: Course Technology/Cengage Learning, 2012.
9. B. OSullivan, J. Goerzen, and D. Stewart, *Real world Haskell*. Beijing: OReilly, 2009.
10. rileyweber13, “rileyweber13/fraktell,” *GitHub*. [Online]. Available: https://github.com/rileyweber13/fraktell.git. [Accessed: 01-Dec-2019].
11. “The Haskell Tool Stack,” *Home - The Haskell Tool Stack*. [Online]. Available: https://docs.haskellstack.org/en/stable/README/. [Accessed: 01-Dec-2019].
12. “TIOBE Index for November 2019,” *TIOBE*. [Online]. Available: https://www.tiobe.com/tiobe-index/. [Accessed: 01-Dec-2019].
13. D. A. Turner, “Some History of Functional Programming Languages,” Lecture Notes in Computer Science Trends in Functional Programming, pp. 1–20, 2013.
14. “Welcome to Hackage!,” Hackage. [Online]. Available: https://hackage.haskell.org/. [Accessed: 30-Nov-2019].