Parallelized Haskell Mandelbrot Set Generation

Declan O'Reilly

Parallel Functional Programming

Abstract- This paper documents the Mandelbrot set generation procedures, codes and results when created using both the sequential and parallel facilities provided by the Haskell runtime system and its libraries.

I. Introduction

THe Mandelbrot set is the set of complex numbers c which the function $f_c(z) = z^2 + c$ does not diverge when iterated from $z = 0$, i.e. for which the sequence $f_c(0), f_c(f_c(0))$, etc. remains bounded in absolute value [1]. When the set is displayed, the complex plain is converted to its two-dimensional cartesian plane representation, with the real values along the horizontal axis and the imaginary values along the vertical axis. In this representation, the horizontal values usually go from -2 to 1 and the vertical axis goes from -1 to -1 as this range of values will allow us to generate an image as in Figure 1.

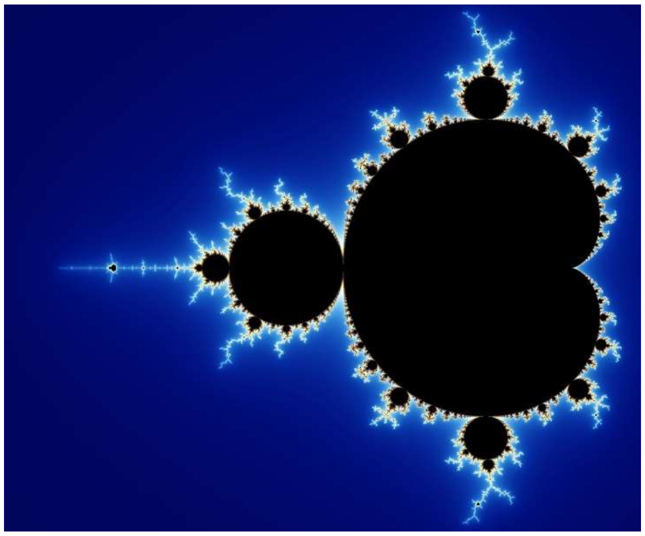


Figure 1: The Mandelbrot set within a continuously colored environment. [1]

II. Mandelbrot

When rendering the Mandelbrot set within the chosen coordinates, we must choose the number of pixels that lie between both the horizontal and vertical boundaries. Each pixel coordinate then represents a complex number c that is used in the function $f_c(z) = z^2 + c$, where z is initially set to 0. When the function is applied iteratively, if the value of $f_c(z)$ remains bounded then we color that pixel black. If the value of $f_c(z)$ is unbounded, then the color or the pixel is determined by the iteration number when the function became unbounded. This number is sometimes called the 'escape' number. Obviously, if the function never goes unbounded, then the number of iterations will be infinite. To avoid this situation, we use two filters to determine if the function remains bounded. First, we terminate our calculation after 100 iterations. In other words, if the function is still bounded after 100 iterations, we decide that it will remain unbounded. Second, if the magnitude of z goes above two, then the function will eventually become unbounded [1].

From a parallel programming perspective what is interesting about the Mandelbrot set; is that the bounded/unbounded calculation for a pixel is independent of all other pixels. Theoretically, if there is enough computing power available, calculating the 'escape' number for each pixel could be calculated in parallel. Of course, the computer power needed will depend on the number of pixels between the coordinate boundaries. If, as mentioned previously the horizontal boundary goes from -2 to 1, and the vertical boundary goes from -1 to 1 and if we choose 2048 pixels along the horizontal axis and 1024 pixels along the vertical axis, we will need to calculate the 'escape' number for 2,097152 pixels. Since calculating a value for this number of pixels in parallel belongs to the realm of GPU computing, while this study is restricted to solutions running on CPU's we will use the parallel functionality supplied by the Haskell system runtime and its libraries.

III. Mandelbrot (Baseline)

In his book 'Parallel and Concurrent Programming in Haskell' [2], Simon Marlow does use the GPU's power to calculate the Mandelbrot set in parallel (*github.com/simonmar/parconc-examples/tree/master/mandel)*. He uses the Accelerate library (www.acceleratehs.org) which is a Haskell Library built on top of the CUDA library from Nvidia. In 2013 when the book was written the Accelerate library provided two 'backends'. The 'interpreter' backend which ran calculations in serially and the CUDA backend, which obviously ran its calculation on the GPU. Interestingly since the book has written the Accelerate library has changed their range of 'backends'. The interpreter ‘backend’ is still available, but there are now two other backends, the *llvm-cpu* and the *llvm-ptx*, which have replaced the originally CUDA backend. The former utilizes the LLVM complier infrastructure running on multicore CPU’s and the latter again utilizing the LLVM complier infrastructure, but this time running on the GPU. The Accelerate library has gone from version 0.12.2.0 in 2013 to 1.3.0.0 today and there has been some ‘breaking changes’ in the library, and as mentioned the ‘backend’ infrastructure has been upgraded. So rather than using Marlow’s original Mandelbrot set code as a base line, we will use an updated version that is included in the examples package of the accelerate library *(hackage.haskell.org/package/accelerate-examples)*. This version had its genesis with Marlow’s version but has been updated over the years to utilize new functionality introduced into Accelerate as new versions were released.

Our baseline example consists of the Mandelbrot version running using the Accelerate interpreter backend (i.e., sequentially) and the version running against the LLVM CPU backend (i.e., in parallel). For the baseline example we will create and render the Mandelbrot set with a height of 1024 pixels and a width of 2048 pixels. The example code suffers in one respect. The code is written so the code does not need to change whether the example is running sequentially or in parallel. The only difference is the Accelerate library configuration. This means the sequential version must use the Accelerate library in places where it would not necessarily need to do so, if the code did not need to be agnostic with respect to the run time environment. There is an example of this in the signature of the ‘mandelbrot’ function.

```
mandelbrot
    :: forall a. (Num a, RealFloat a, FromIntegral I
nt a, Elt (Complex a))
    => Int                                          -
- ^ image width
    -> Int                                          -
- ^ image height
    -> Acc (Scalar a)                               -
- ^ centre x
    -> Acc (Scalar a)                               -
- ^ centre y
    -> Acc (Scalar a)                               -
- ^ view width
    -> Acc (Scalar Int32)                           -
- ^ iteration limit
    -> Acc (Scalar a)                               -
- ^ divergence radius
    -> Acc (Array DIM2 (Complex a, Int32))
```

To aid in the parallel version, the ‘mandelbrot’ function has a set of parameters using the `Acc (Scalar a)` datatype. If the function had been written without regard for the parallel version, these parameters, would no doubt have used the simple Haskell `Int or Double` datatype. So, while comparisons between the sequential and the parallel version maybe skewed, looking at the parallel version will be informative. Figure 2 shows the output from ThreadScope when the program was run using the interpreter backend, i.e., sequentially. The two points that stand out is a) it took 500 seconds to complete and b) it only utilized a single core, which is to be expected.

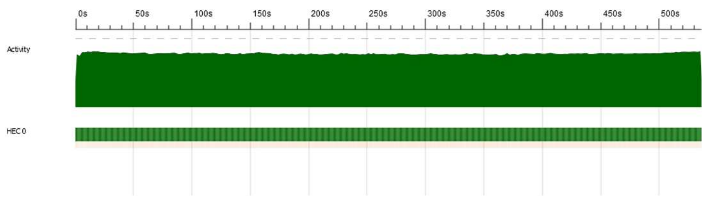


Figure 2: The Accelerate ‘examples’ Mandelbrot running under the interpreter backend

When the program was run using the LLVM CPU backend, running on 4 cores, Figure 3 shows that the statistics have vastly improved. The program now runs in 0.5 seconds and we can also see that, while not consistently, the 4 cores were utilized. The rest of the timeline shows the application setup and the image creation and its persistence to disk.

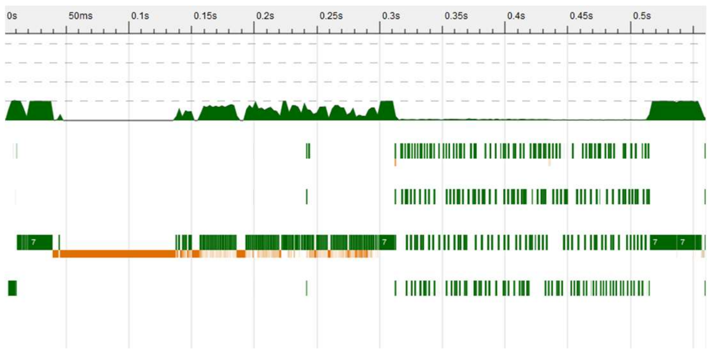


Figure 3: The Accelerate ‘examples’ Mandelbrot running under the LLVM CPU backend

IV. Mandelbrot (Sequential)

In the project examples, the code can be broken up into 5 steps.

1. Generate the complex plane values over the boundaries (-2 to 1 on the horizontal axis and -1 to 1 on the vertical axis). As in the baseline examples, all project examples

also have a height of 1024 pixels and a width of 2048 pixels (see code example 2).

2. Generate the Mandelbrot set for each Complex value in the values generated in step a (see code example 1).
3. Convert the Mandelbrot ‘escape’ number to an RGB value (see code example 3).
4. Collect the list of RGB values into a collection that can be processed to create an image. The study collects the RGB values into 3 different collections: a list, a vector and a Repa array. Each of these collection types provide their own distinct parallel opportunity.
5. Convert the collection of RGB values into an image and persist this image.

   Note: Step 5 provides the visual output validation, but as it tends to be a sequential process, it will not play a major part in the study.

The first example generates the complex plane values using a Haskell list comprehension. As part of the list comprehension the Mandelbrot set is generated, and then the results are converted to their RGB values (see code example 4). Figure 4 shows the ThreadScope graph output for this run. From this graph we can see that the sequential version took 10.53 seconds to run and as again would be expected only utilized a single core. The graph also shows that a small percentage of this time was taken up with garbage collections.

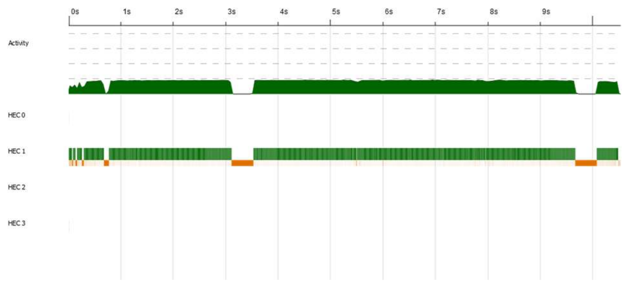


Figure 4: The Mandelbrot set generated as a list running ‘sequentially’

The next example repeats the Mandelbrot set generation while storing the set in a Haskell vector (see code example 5). In this test Figure 5 shows that the execution time was reduced to 8.05 seconds.

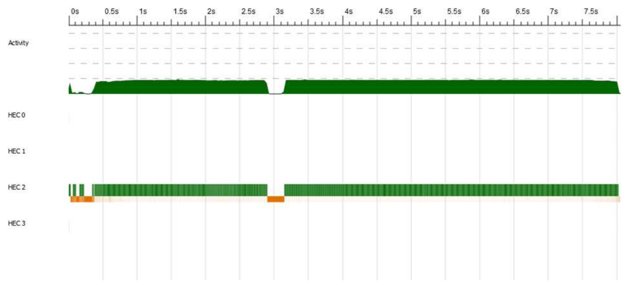


Figure 5: The Mandelbrot set generated as a vector running ‘sequentially’

The last sequential example stores the Mandelbrot set in a Haskell Repa array (see code example 6). This example is the quickest of the three examples at 7.36 seconds. This maybe a little surprising since a Repa array has more structure that a list or vector. Therefore, we might expect that this extra structure would result in a slower output, but obviously this structure creation time is an efficient operation within Repa or the overall efficiencies inherent in the Repa runtime makes up for any extra time that the array creation would take. Part of these efficiencies maybe as a result that during this run, zero garbage collections occurred (see Figure 6).

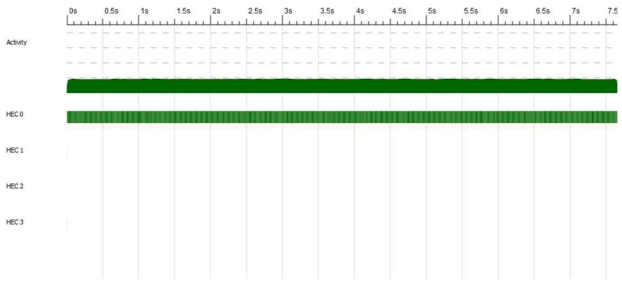


Figure 6: The Mandelbrot set generated as a Repa array running ‘sequentially’

V. Mandelbrot (Parallel)

The Haskell runtime has the concept of a *spark* as a feature that allows code to be parallelized. ‘The runtime collects sparks in a pool and uses this as a source of work when there are spare processors available, using a technique called work stealing. Sparks may be evaluated at some point in the future, or they might not—it all depends on whether there is a spare core available’ [3]. When we parallelize the code that generates the Mandelbrot set as a list and a vector, we will use code provided by the runtime that takes advantage of this feature by creating sparks underneath the covers. Since Haskell by default, is a ‘lazy’ language, ‘expressions are not evaluated when they are bound to variables, but their evaluation is deferred until their results are needed by other computations. In consequence, arguments are not evaluated before they are passed to a function, but only when their values are actually used’ [4]. This means if we can force eager evaluation and the creation of sparks, the eager evaluation can happen in parallel. One of the most straight forward means to cause parallelism is by using the `rpar` and `rseq` functions from the `Eval` Monad. When we use the `rpar` combinator before a section of code, it states that the `rpar` argument (i.e., the operation of interest) can be run in parallel. `rseq` is used when you want to execute an operation but wait until it completes. As stated previously, creating a Mandelbrot set that ranges over 1024 rows and 2048 columns will result in 2,097,152 possible independent operations, i.e., the function iteration $f_c(z) = z^2 + c$ can be run for the value assigned to each pixel. Though if we run 2 million parallel operations, the overhead would outweigh the benefit that would result from the parallelism. To avoid such a situation Haskell provides 'Evaluation Strategies' that are designed to separate an algorithm from its runtime parallelism. There is a subset of these strategies called 'Chunking Strategies' that can be utilized in cases like our Mandelbrot set problem. The idea is to 'chunk' or group the input data, so that the level of granularity can be sensibly managed by the Haskell runtime system. One such 'chunking' function is called `parListChunk`, which takes two arguments, the 'chunk' size and a parallel strategy. Examples of strategies are `rpar` and `rseq` that we have just see and the strategy used in the study code, `rdeepseq`. `rdeepseq` is a function that actively enforces evaluation to normal form (completely), and makes Haskell behave more like a strict programming language' [5]. When the code that generates the Mandelbrot set as a list is parallelized it is broken up in 1000 chunks and uses the `rdeepseq` strategy (see code example 7). The ThreadScope output from this is shown in figure 7. Here we see that the code now runs in 6.24 seconds as opposed to 10.53 seconds when it ran sequentially. This gives us a speedup of 1.69, which is quite acceptable considering we only added one function to the code. We also see that it used all four cores present on the machine while the garbage collection pattern is similar to the earlier sequential run.

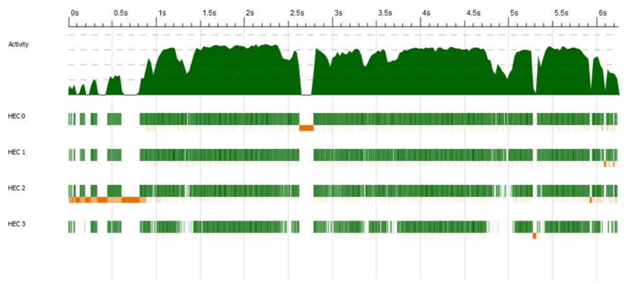


Figure 7: The Mandelbrot set generated as a list running in parallel with 'chunk size' of 1,000

In our example, the work was broken up into 1,000 chunks which caused 2000 Sparks to be created. As a comparison, when the work was broken up into 100,000 chunks the processing wall-clock time of 6.29 seconds was similar, but in this case only 21 Sparks were created. Interestingly, when the work was broken up into 1,000,000 chunks, the wall-clock time was 6.77 seconds again similar to the 1000 chunks case, but in this case only 3 Sparks were created and noticeably not all cores were in use for most of the run, see Figure 8.

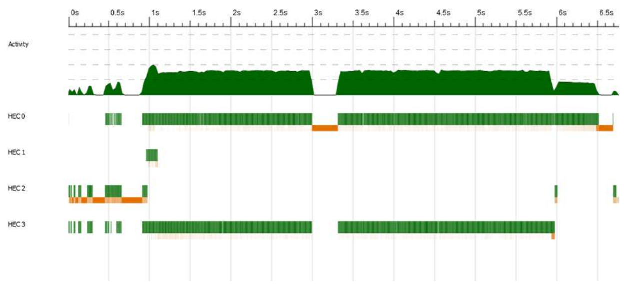


Figure 8: The Mandelbrot set generated as a list running in parallel with 'chunk size' of 1,000,000

When the chunk size is pushed to 2,000,000, we are in effect running sequentially again and, the time increases accordingly back to 9.18 seconds (see Figure 9).

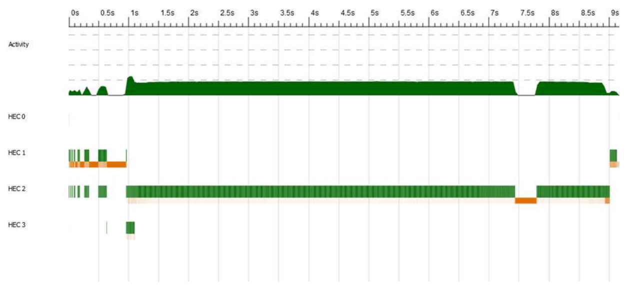


Figure 9: The Mandelbrot set generated as a list running in parallel with 'chunk size' of 1,000,000

When we go in the opposite direction and use a chunk size of 100, we end up trying to create 20,972 sparks. Since the spark pool is of a fixed size and if we try to create a spark when it is full the spark will be dropped or 'overflowed'. So, in this case 20,972 is too many sparks and 10,760 did in fact overflow. While we did successfully create roughly 10,000 sparks the overhead of the overflow sparks causes the process to slow down to 9.40 seconds and again, we see that not all cores are continually in use, see figure 10.

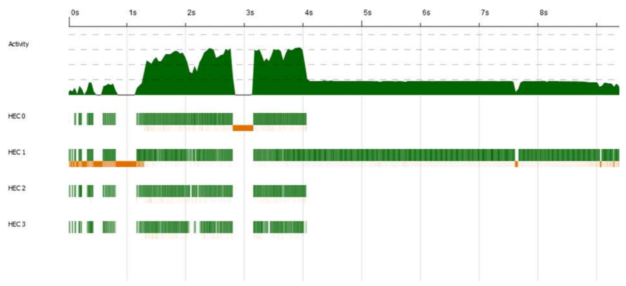


Figure 10: The Mandelbrot set generated as a list running in parallel with 'chunk size' of 100

When we used 100 chunks approximately half the sparks overflowed, therefore we might expect that if we increase the chunk size to 200 then all the sparks would be created successfully. In turns out that for a chunk size of 200, 1,255 sparks out of 10,486 sparks still overflowed. Though the overhead of the overflow sparks is not as large as we saw earlier, and the process finished in 6.53 seconds.

To achieve zero overflow sparks, the chunk size was increased to 250. In this case 8389 sparks were created successfully with no overflow sparks and the processed finished in 6.65 seconds and, all four cores were utilized while occurring a small garbage collection as seen previously. The variation in chunk size shows that the chunk size parameter can have a large variation and still produce similar results.

Earlier we saw two other strategies besides `rdeepseq`, namely `rpar` and `rseq.` It is instructive to see how the process performs using either of these strategies. In other words, if we used

```
`using` parListChunk 1000 rpar
```

rather than

```
`using` parListChunk 1000 rdeepseq
```

Surprisingly when we ran the process with both these strategies, it only took 2.91 and 2.75 seconds to complete. This would seem to be, not only a vast improvement over the sequential run, but also an impressive improvement over the parallel run using `rdeepseq.` On closer inspection though, the improvement in speed can be explained. When we use either `rpar` and `rseq` the result is either not evaluated at all or is only evaluated to *weak head normal form*. The `par` and `seq` function only evaluate its argument up to its first constructor. In fact, if we run the following command in `ghci,` (the second example is using the `force` command)

```
a = mandelbrotAsListsS 1024 2048
```

or

```
a = force $ mandelbrotAsListsS 1024 2048
```

and then look at the evaluation status of the variable *a* using the `ghci sprint` command, we see '`a = _`'. Since Haskell is lazy, *a* was not evaluated at all, as there was no need to evaluate the variable at this stage. If we use *Bang Patterns* and change the above to

```
a! = mandelbrotAsListsS 1024 2028
```

and look at the evaluation status of *a*, we still only see '**a** = (_,_,_) : _'. In this second case *a* has been evaluated to *weak head normal form.* If we are not 'using' the value of *a*, e.g., printing its value, and we want to evaluate it completely we need to use *Bang Patterns* and use the `force` command and run the following

```
a! = force $ mandelbrotAsListsS 1024 2028
```

So, while running the process using either `rpar` or `rseq` as a strategy might seem to improve the performance, under closer examination the work is in fact incomplete.

The second Mandelbrot set collection was stored in a vector. The `Vector.Strategies` module provides a function called `parVector` that is similar in functionality to `parListChunk`. Though it only takes one parameter the 'chunk' size (see code example 8). The strategy name is hidden and not documented, but one would assume that it is similar if not identical to `rdeepseq`. The vector collection was chosen to see if its parallel behavior and/or performance differed from the list collection parallel facilities. It turned out that for all the same the range of chunk sizes, i.e., 100 to 2,000,000 the behavior was almost identical.

The last Mandelbrot set collection was the Repa array (see code example 9). 'Repa provides high performance, regular, multi-dimensional, shape polymorphic parallel arrays. All numeric data is stored unboxed. Functions written with the Repa combinators are automatically parallel provided you supply +RTS -Nwhatever (sic) on the command line when running the program' [6]. Like the Haskell 'Strategies' facilities, the code can be paralyzed with small code changes. In the case of Repa, we use `computeP` rather than `computeS` (`computeP` though must run in a Monad, though any Monad will suffice). When running with Repa there are no parameters. That is, it does not use a chunk size nor a 'Strategy'. All the parallelization happens underneath the covers. Of all the tests the Repa process consistently gave the best performance. In Figure 11 we see that it completed in 4.19 seconds. Also noticeable is that fact that while 4 cores where initially in use, for at least half the run only two cores were utilized.

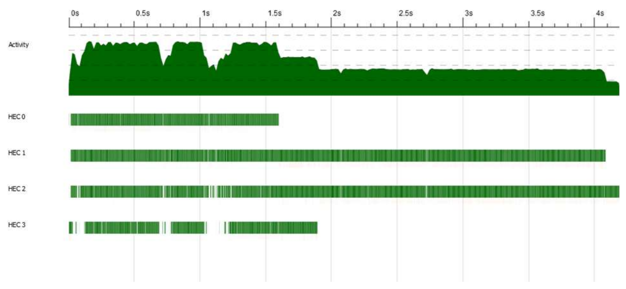


Figure 11: The Mandelbrot set generated as a Repa array running in parallel.

For Repa arrays, the concept of a *spark* is not applicable. Repa arrays process unboxed data which by its nature is not lazy. Likewise, when we looked at the Haskell Accelerate example version of the Mandelbrot set generation earlier, it also did not have any spark statistics. As we saw earlier, their parallelization features do not use the Haskell 'Strategy' features, but use a backend utilizing LLVM running on multicores CPU's or a backend utilizing LLVM running on GPU's.

One final task is to show the Mandelbrot image that was generated by the above processes. Each process generated the same image, which is to be expected. Figure 12 shows this image, though obviously shrunk.

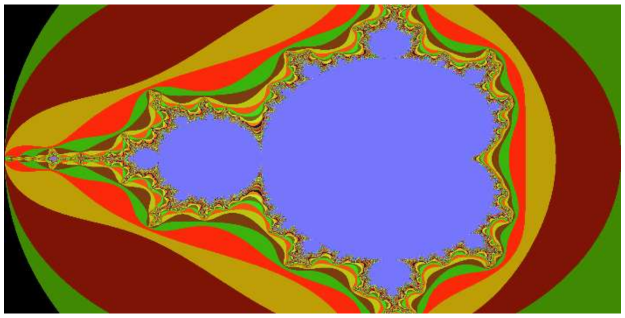


Figure 12: The Mandelbrot set image generated

This image does not match what we see in Figure 1. The reason for this is the `toRgb` function (see code example 3). implementation is quite basic and is not able to capture the variation that we see in Figure 1.

VI. Conclusions

This paper looked at the Mandelbrot set generation using various parallelization facilities provided by either the Haskell runtime or its libraries. In all cases that we looked at; parallelization was added to the program with just a little extra code. While utilizing the 'Strategy' module, we saw a speedup of 1.69, but the Repa array code saw a speedup of 1.76 where its 'sequential' run was also faster than the other 'sequential' runs. The one downside of Repa arrays is that they can only contain unboxed data, so they cannot hold arbitrary data types, which may restrict their use. In our case we only used the array storage for the final number, i.e., the RGB value, which was a tuple of 3 `Word8` variables. However, we were able to use the Repa parallelization features while converting a Tuple of 2 `Int`'s to a `Complex Double`, to a `Word8.` In other words, it was only the final representation that needed to be unboxed data, the data that was generated during the process did not suffer from this restriction.

Future Work

By far, the most significant speedup was achieved by the Accelerate Mandelbrot example. Though the 'sequential' version did seem to be hampered by the design choices made to help the code run under Accelerate. Nevertheless, the parallelized version did run in 0.5 seconds. This makes a further investigation of the Accelerate implementation running on top of LLVM multicore an interesting project. Likewise, there would seem to be many lessons to be learnt from their LLVM GPU project.

One last project that would be interesting to tackle, is to remedy the results produced by the RGB function. While its implementation was not particularly important to the paper, there is the saying that says 'a picture paints a thousand words', so surely a more accurate image can only increase this wordcount.

References

[1] Wikipedia contributors. "Mandelbrot set." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 1 Dec. 2020. Web. 1 Dec. 2020..

[2] Marlow, Simon. "'Parallel and Concurrent Programming in Haskell." O'Reilly Media, 2013, chapter 6.

[3] Marlow, Simon. "'Parallel and Concurrent Programming in Haskell." O'Reilly Media, 2013, chapter 2.

[4] "Lazy evaluation." *HaskellWiki,* . 3 Sep 2015, 03:52 UTC. 10 Dec 2020,

https://wiki.haskell.org/index.php?title=Lazy_evaluation&oldid=60051>.

[5] "A tutorial on Parallel Strategies in Haskell." *Parallel Functional Programming class,*. 11 Dec 2020,

http://www.cse.chalmers.se/edu/year/2015/course/pfp/Papers/strategies-tutorial-v2.pdf

[6] Repa Hackage page. "*repa: High performance, regular, shape polymorphic parallel arrays"*. 11 Dec 2020,

https://hackage.haskell.org/package/repa

Build Instructions

1) Uncompress `final.gz`

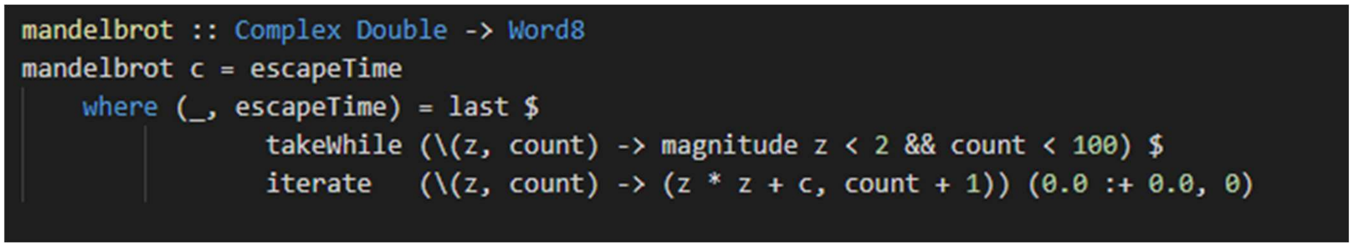
2) In the `final` directory run **`stack build`**

3) The usage and an example follows. Choices are between 1 to 12 (see `Main.hs`) where choice 1 to 6 saves the Mandelbrot set image using one of the different options outlined in the paper and choices 7 to 12 create the Mandelbrot set (again using the different options outlined in the paper) witout creating the final image.

Usage: final <choice> <row count> <column count>

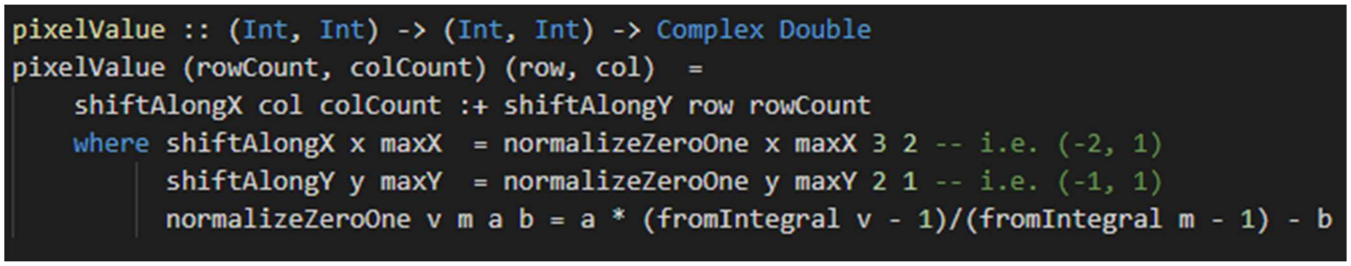
```
stack exec -- final-exe -RTS 6 1024
2048 +RTS -N4 -ls
```

Code examples referenced in the study: (Build instructions outlined at end of final page above)



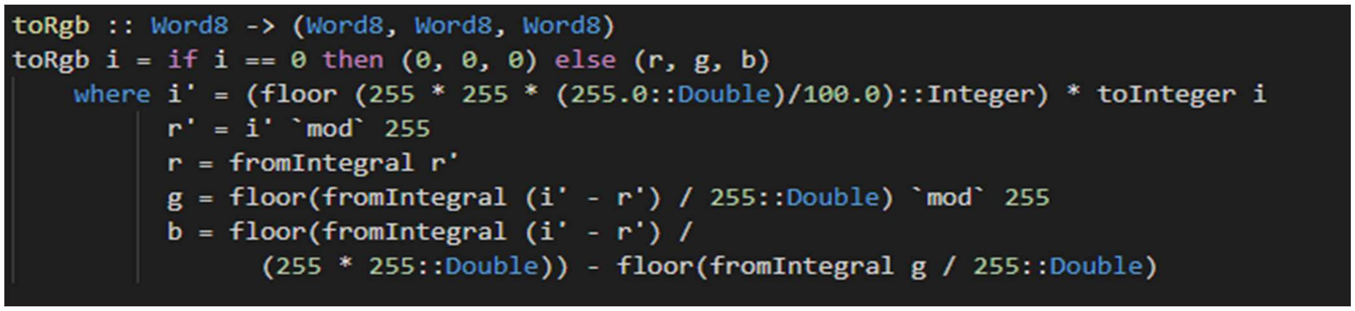
```haskell
mandelbrot :: Complex Double -> Word8
mandelbrot c = escapeTime
    where (_, escapeTime) = last $
                takeWhile (\(z, count) -> magnitude z < 2 && count < 100) $
                iterate   (\(z, count) -> (z * z + c, count + 1)) (0.0 :+ 0.0, 0)
```

Code example 1: The mandelbrot set generating function. The input complex number is converted to its 'escape' number



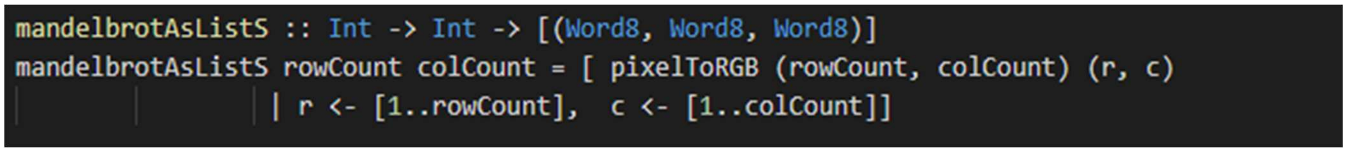
```haskell
pixelValue :: (Int, Int) -> (Int, Int) -> Complex Double
pixelValue (rowCount, colCount) (row, col)  =
    shiftAlongX col colCount :+ shiftAlongY row rowCount
    where shiftAlongX x maxX  = normalizeZeroOne x maxX 3 2 -- i.e. (-2, 1)
          shiftAlongY y maxY  = normalizeZeroOne y maxY 2 1 -- i.e. (-1, 1)
          normalizeZeroOne v m a b = a * (fromIntegral v - 1)/(fromIntegral m - 1) - b
```

Code example 2: Given the number of rows and columns between the boundaries and a specified row and column number, `pixelValue` returns its complex number representation



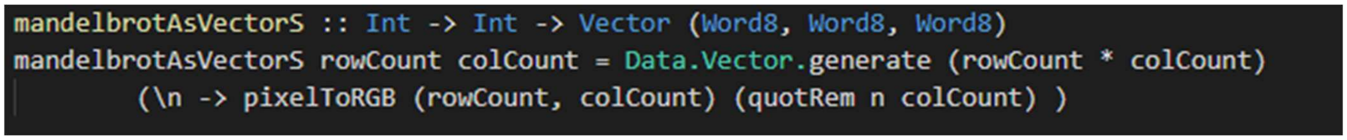
```haskell
toRgb :: Word8 -> (Word8, Word8, Word8)
toRgb i = if i == 0 then (0, 0, 0) else (r, g, b)
    where i' = (floor (255 * 255 * (255.0::Double)/100.0)::Integer) * toInteger i
          r' = i' `mod` 255
          r = fromIntegral r'
          g = floor(fromIntegral (i' - r') / 255::Double) `mod` 255
          b = floor(fromIntegral (i' - r') /
                (255 * 255::Double)) - floor(fromIntegral g / 255::Double)
```

Code example 3: Given an 'escape' number `toRgb` returns an RGB representation



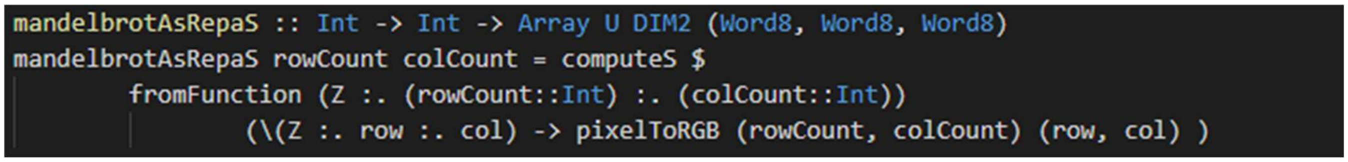
```haskell
mandelbrotAsListS :: Int -> Int -> [(Word8, Word8, Word8)]
mandelbrotAsListS rowCount colCount = [ pixelToRGB (rowCount, colCount) (r, c)
                  | r <- [1..rowCount],  c <- [1..colCount]]
```

Code example 4: The Mandelbrot set stored in a Haskell list



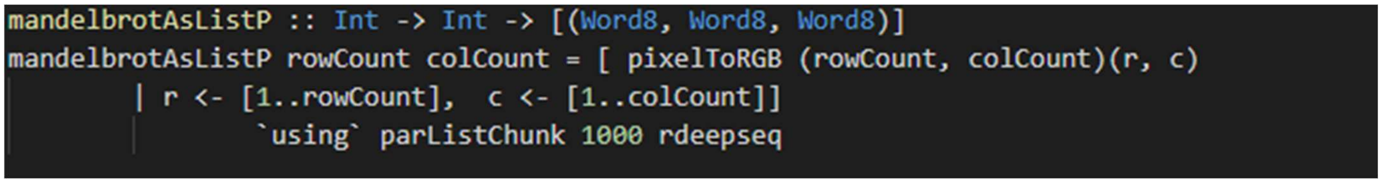
```haskell
mandelbrotAsVectorS :: Int -> Int -> Vector (Word8, Word8, Word8)
mandelbrotAsVectorS rowCount colCount = Data.Vector.generate (rowCount * colCount)
        (\n -> pixelToRGB (rowCount, colCount) (quotRem n colCount) )
```

Code example 5: The Mandelbrot set stored in a Haskell vector



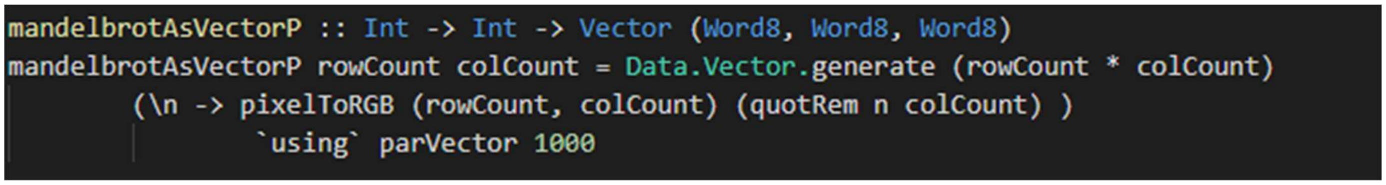
```haskell
mandelbrotAsRepaS :: Int -> Int -> Array U DIM2 (Word8, Word8, Word8)
mandelbrotAsRepaS rowCount colCount = computeS $
        fromFunction (Z :. (rowCount::Int) :. (colCount::Int))
                (\(Z :. row :. col) -> pixelToRGB (rowCount, colCount) (row, col) )
```

Code example 6: The Mandelbrot set stored in a Haskell Repa array



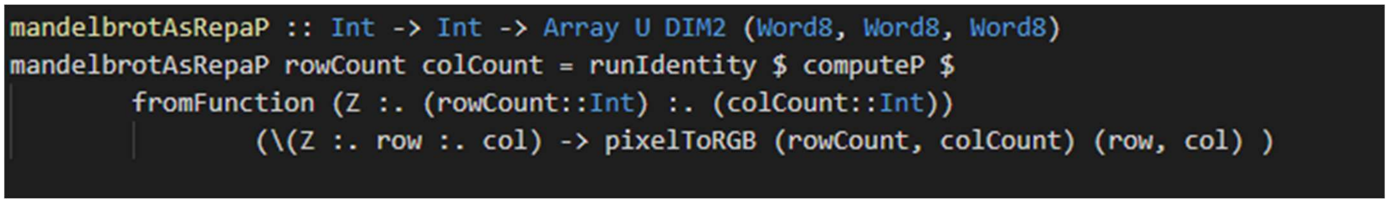
```haskell
mandelbrotAsListP :: Int -> Int -> [(Word8, Word8, Word8)]
mandelbrotAsListP rowCount colCount = [ pixelToRGB (rowCount, colCount)(r, c)
        | r <- [1..rowCount],  c <- [1..colCount]]
                `using` parListChunk 1000 rdeepseq
```

Code example 7: The Mandelbrot set stored in a Haskell listwhile run in parallel



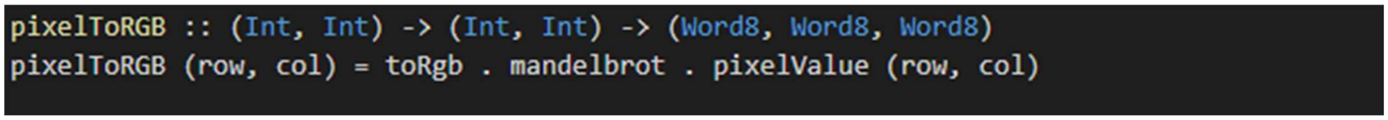
```haskell
mandelbrotAsVectorP :: Int -> Int -> Vector (Word8, Word8, Word8)
mandelbrotAsVectorP rowCount colCount = Data.Vector.generate (rowCount * colCount)
        (\n -> pixelToRGB (rowCount, colCount) (quotRem n colCount) )
                `using` parVector 1000
```

Code example 8: The Mandelbrot set stored in a Haskell vector while run in parallel



```haskell
mandelbrotAsRepaP :: Int -> Int -> Array U DIM2 (Word8, Word8, Word8)
mandelbrotAsRepaP rowCount colCount = runIdentity $ computeP $
        fromFunction (Z :. (rowCount::Int) :. (colCount::Int))
                (\(Z :. row :. col) -> pixelToRGB (rowCount, colCount) (row, col) )
```

Code example 9: The Mandelbrot set stored in a Haskell Repa array while run in parallel



```haskell
pixelToRGB :: (Int, Int) -> (Int, Int) -> (Word8, Word8, Word8)
pixelToRGB (row, col) = toRgb . mandelbrot . pixelValue (row, col)
```

Code example 10: `pixelToRGB` converts a pixel value to a complex number to an ‘escape’ number to a RGB color

Lib.hs: Full code listing

```haskell
module Lib
    ( mandelbrotAsListS, mandelbrotAsListP, mandelbrotAsVectorS,
      mandelbrotAsVectorP, mandelbrotAsRepaS, mandelbrotAsRepaP)
    where

import Data.Complex ( magnitude, Complex((:+)) )
import Data.Vector ( generate, Vector )
import Data.Array.Repa ( fromFunction, computeS, computeP, Z (..), (:.)(..), Array, U, DIM2 )
import Data.Functor.Identity ( runIdentity )
import Control.Parallel.Strategies ( parListChunk, rdeepseq, using)
import Data.Vector.Strategies ( parVector )
import Data.Word ( Word8 )

mandelbrot :: Complex Double -> Word8
mandelbrot c = escapeTime
    where (_, escapeTime) = last $
                takeWhile (\(z, count) -> magnitude z < 2 && count < 100) $
                iterate   (\(z, count) -> (z * z + c, count + 1)) (0.0 :+ 0.0, 0)

pixelValue :: (Int, Int) -> (Int, Int) -> Complex Double
pixelValue (rowCount, colCount) (row, col)  =
    shiftAlongX col colCount :+ shiftAlongY row rowCount
    where shiftAlongX x maxX  = normalizeZeroOne x maxX 3 2 -- i.e. (-2, 1)
          shiftAlongY y maxY  = normalizeZeroOne y maxY 2 1 -- i.e. (-1, 1)
          normalizeZeroOne v m a b = a * (fromIntegral v - 1)/(fromIntegral m - 1) - b

toRgb :: Word8 -> (Word8, Word8, Word8)
toRgb i = if i == 0 then (0, 0, 0) else (r, g, b)
    where i' = (floor (255 * 255 * (255.0::Double)/100.0)::Integer) * toInteger i
          r' = i' `mod` 255
          r = fromIntegral r'
          g = floor(fromIntegral (i' - r') / 255::Double) `mod` 255
          b = floor(fromIntegral (i' - r') /
                (255 * 255::Double)) - floor(fromIntegral g / 255::Double)



pixelToRGB :: (Int, Int) -> (Int, Int) -> (Word8, Word8, Word8)
pixelToRGB (row, col) = toRgb . mandelbrot . pixelValue (row, col)
```

```haskell
---Sequential mandelbrot

mandelbrotAsListsS :: Int -> Int -> [(Word8, Word8, Word8)]
mandelbrotAsListsS rowCount colCount = [ pixelToRGB (rowCount, colCount) (r, c)
                 | r <- [1..rowCount],  c <- [1..colCount]]



mandelbrotAsVectorS :: Int -> Int -> Vector (Word8, Word8, Word8)
mandelbrotAsVectorS rowCount colCount = Data.Vector.generate (rowCount * colCount)
        (\n -> pixelToRGB (rowCount, colCount) (quotRem n colCount) )


mandelbrotAsRepaS :: Int -> Int -> Array U DIM2 (Word8, Word8, Word8)
mandelbrotAsRepaS rowCount colCount = computeS $
        fromFunction (Z :. (rowCount::Int) :. (colCount::Int))
                (\(Z :. row :. col) -> pixelToRGB (rowCount, colCount) (row, col) )


---Parallel mandelbrot

mandelbrotAsListP :: Int -> Int -> [(Word8, Word8, Word8)]
mandelbrotAsListP rowCount colCount = [ pixelToRGB (rowCount, colCount)(r, c)
        | r <- [1..rowCount],  c <- [1..colCount]]
                `using` parListChunk 1000 rdeepseq


mandelbrotAsVectorP :: Int -> Int -> Vector (Word8, Word8, Word8)
mandelbrotAsVectorP rowCount colCount = Data.Vector.generate (rowCount * colCount)
        (\n -> pixelToRGB (rowCount, colCount) (quotRem n colCount) )
                `using` parVector 1000


mandelbrotAsRepaP :: Int -> Int -> Array U DIM2 (Word8, Word8, Word8)
mandelbrotAsRepaP rowCount colCount = runIdentity $ computeP $
        fromFunction (Z :. (rowCount::Int) :. (colCount::Int))
                (\(Z :. row :. col) -> pixelToRGB (rowCount, colCount) (row, col) )
```

Main.hs: Driver code to run the exported functions from Lib.hs

```haskell
module Main where


import System.Environment ( getArgs )
import System.Exit ( die )
import Data.Word ( Word8 )
import Data.ByteString (pack )
import Data.Array.Repa ( extent)
import Data.Array.Repa.IO.BMP ( writeImageToBMP )
import Data.Time ( getCurrentTime, diffUTCTime )
import Control.DeepSeq ( force )
import Codec.BMP ( writeBMP, packRGBA32ToBMP, BMP )
import Lib   ( mandelbrotAsListS, mandelbrotAsListP, mandelbrotAsVectorS,
               mandelbrotAsVectorP, mandelbrotAsRepaS, mandelbrotAsRepaP )



main :: IO ()
main = do
    let errorMessage =  "Usage: final <choice> <row count> <column count>"
    args <- getArgs
    case args of
        [ch, rc, cc] ->  do
            let (choice, rowCount,colCount) = (read ch :: Int, read  rc :: Int, read  cc :: Int)

            beforeT <- getCurrentTime
            case choice of
                c
                    | c >=  1 && c <=  6 -> mandelbrotSaveImage  choice rowCount colCount
                    | c >=  7 && c <=  9 -> mandelbrotSequential choice rowCount colCount
                    | c >= 10 && c <= 12 -> mandelbrotParallel   choice rowCount colCount

                    | otherwise -> die errorMessage

            afterT <- getCurrentTime
            print $ show choice ++ " (" ++ show (diffUTCTime afterT beforeT) ++ ")"

        _ -> die errorMessage
```

```
mandelbrotSaveImage :: Int -> Int -> Int-> IO ()
mandelbrotSaveImage choice rowCount colCount = do

     case choice of
        c
            | c ==  1 -> writeBMP        "mandelbrotAsListS.bmp"   $ collectionToBmp     rowCount colCount
                                                                    $ mandelbrotAsListS   rowCount colCount
            | c ==  2 -> writeBMP        "mandelbrotAsVectorS.bmp" $ collectionToBmp     rowCount colCount
                                                                    $ mandelbrotAsVectorS rowCount colCount
            | c ==  3 -> writeImageToBMP "mandelbrotAsRepaS.bmp"   $ mandelbrotAsRepaS   rowCount colCount

            | c ==  4 -> writeBMP        "mandelbrotAsListP.bmp"   $ collectionToBmp     rowCount colCount
                                                                    $ mandelbrotAsListP   rowCount colCount
            | c ==  5 -> writeBMP        "mandelbrotAsVectorP.bmp" $ collectionToBmp     rowCount colCount
                                                                    $ mandelbrotAsVectorP rowCount colCount
            | c ==  6 -> writeImageToBMP "mandelbrotAsRepaP.bmp"   $ mandelbrotAsRepaP   rowCount colCount

            | otherwise -> print "unexpected"


mandelbrotSequential :: Int -> Int -> Int-> IO ()
mandelbrotSequential choice rowCount colCount = do

    case choice of
        c
            | c == 7    -> print $ Prelude.length $ force $ mandelbrotAsListS   rowCount colCount
            | c == 8    -> print $ Prelude.length $ force $ mandelbrotAsVectorS rowCount colCount
            | c == 9    -> print $ extent $                 mandelbrotAsRepaS   rowCount colCount

            | otherwise -> print "unexpected"
    return ()


mandelbrotParallel :: Int -> Int -> Int-> IO ()
mandelbrotParallel choice rowCount colCount = do

    case choice of
        c
            | c == 10   -> print $ Prelude.length $ mandelbrotAsListP   rowCount colCount
            | c == 11   -> print $ Prelude.length $ mandelbrotAsVectorP rowCount colCount
            | c == 12   -> print $ extent         $ mandelbrotAsRepaP   rowCount colCount

            | otherwise ->  print "unexpected"
    return ()
```

```haskell
-- conversion functions

collectionToBmp :: Foldable t => Int -> Int -> t (Word8, Word8, Word8) -> BMP
collectionToBmp rowCount colCount xs = packRGBA32ToBMP colCount rowCount $
        Data.ByteString.pack ( Prelude.concatMap (\(r, g, b) -> [r, g, b, 255])  xs)
```