Parallel QuickHull Algorithm in Haskell

George Morgulis (gm3138), Henry Lin(hkl2127)

December 2024

1 Introduction

The convex hull problem is a classical problem in computational geometry. The essence of the problem is to find the smallest convex polygon, the convex hull, that fully encloses a given set of points. The convex hull problem applies to point sets in any dimension, but for this project, we will focus on the planar convex hull.

The subject of this report will be the QuickHull algorithm, the implementation of which can be found in Section 2. The planar QuickHull algorithm was invented by Jonathan Scott Greenfield in 1990, whereas the N-Dimensional QuickHull algorithm was discovred Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa in 1996.

2 QuickHull Sequential

The QuickHull algorithm functions very much like QuickSort. A set of points is taken as an input. A line is made between the x-coordinate extremes, dividing the set of points in two and computing the upper and lower hulls. The extremes are added to the hull.

Let us focus on the upper hull (though the algorithm is exactly the same for both). We find the point with the longest distance from the line segment; this point is added to the hull. We use this point, along with the two extremities to create a triangle. All points within the triangle are eliminated. The remaining set is further divided into two: those to the left and the right of the maximum point. The same process is repeated until no points remain.

The result of the QuickHull algorithm is a set of points that make up the convex hull. Notably, intermediate co-linear points are never included in the set, as they are redundant information.

On average, QuickHull runs in $O(nlog(n))$ time, but in the worst case, it runs in $O(n^2)$ time. Below is the pseudocode for the QuickHull Algorithm.

Algorithm 1 Quick Hull Algorithm

```
1: function QuickHull(points)
2:     a1 ← findMinPoint(points)
3:     a2 ← findMaxPoint(points)
4:     h1 ← hullHelper(points, a1, a2)
5:     h2 ← hullHelper(points, a2, a1)
6:     return a1 : a2 : (h1 + +h2)
7: end function
```

Algorithm 2 Quick Hull Helper

```
1: function hullHelper(points, a1, a2)
2:     group ← groupLeftOf(points, a1, a2)
3:     if group is empty then
4:         return []
5:     else
6:         m1 ← findFurthestPoint(group, a1, a2)
7:         h1 ← hullHelper(group, a1, m1)
8:         h2 ← hullHelper(group, m1, a2)
9:         return m1 : (h1 + +h2)
10:    end if
11: end function
```

3 Parallelizing QuickHull

While the divide and conquer nature of the QuickHull algorithm seem very simple to parallelize, it actually requires a rather moderate and delicate approach.

3.1 Naive Approach

Our naive attempt at parallelizing QuickHull consisted of the following:

- Evaluate $h1$ and $h2$ in parallel at the recursive calls
- Parallelize $findMinPoint$, $findMaxPoint$, and $findFurthestPoint$ by breaking the sets up into smaller chunks, applying the functions on the chunks, and comparing the results to find the final values

3.2 Complications with the Naive Approach

We will start by addressing the parallelization of the smaller functions. Each of them ($findMinPoint$, $findMaxPoint$, and $findFurthestPoint$) is a linear

time function that passes through the input set exactly once. For this reason, even large sets of points, for example $4,000,000$ points, are computed incredibly quickly, and the overhead that comes from managing parallelization trumps any benefit gained from parallel evaluation. Granted, for increasingly large input sizes, this type of parallelization might indeed become effective, but our computer is not strong enough to handle such large amounts of data.

The second complication arises from parallelizing the recursive calls. The difficulty is that on average, the convex hull of a set of points is many times smaller than the input set. For example, I've found that the convex hull size for a set of $4,000,000$ randomly generated points rarely exceeds a 200 points. This, in turn, means that the depth of the tree rarely exceeds the single digits. Indeed, the QuickHull algorithm is already so efficient at eliminating point not in the hull, that even large input sizes are very quickly whittled down.

However, there are certain edge cases that greatly benefit from the recursive parallel treatment. Recall that the worst case scenario of QuickHull is $O(n^2)$, in which a significant percentage of the points lie on the boundary of an ellipse-like shape. In this case, parallelization is incredibly effective. The difficulty, then, is implementing a parallel algorithm that improves performance around the edge cases without worsening the performance on the average cases.

3.3 Note About Data Structures

Our initial implementation utilized the standardized Haskell Linked List, which created an immense overhead with regards to memory usage due to the storage of pointers at each node.

For this reason, we resolved to use a random access data structure. The choice was between the RBB vector and the Unboxed Vector. The RBB vector was interesting because it supported $O(log(n))$ time insertion, which is very important for QuickHull as nearly every stage of the algorithm requires insertion and concatenation; this is compared to Unboxed vector in which all such operations take $O(n)$ time. However, the RBB Vector also had a significant memory footprint, making it rather infective for our algorithm. For this reason, we decided to use Haskell's Unboxed Vector as the data structure for QuickHull.

3.4 Parallel Implementation

The challenge of this project is to parallelize QuickHull such that the edge cases see a significant speedup without slowing down the average cases. For this reason, three checks must be passed before the program forks into two parallel threads.

Firstly, if the current group size is less than 250,000 points, there will not be parallelism. Secondly, parallelization will only take place down to certain depth

in the recursion tree Finally, if more than 30% of points are remaining at the end of a round (compared to the amount of points at the beginning of that particular round), then there will be a spark for parallel evaluation; otherwise, the evaluation will proceed sequentially.

Below is the pseudocode for the Parallel QuickHull Algorithm; the full Haskell implementation can be found at the end of the report.

Algorithm 3 Parallel Quick Hull Algorithm

function QUICKHULL($points$)
    $d \leftarrow threadCount$
    $a1 \leftarrow$ findMinPoint($points$)
    $a2 \leftarrow$ findMaxPoint($points$)
    $h1 \leftarrow$ hullHelper($points, a1, a2, d$)
    $h2 \leftarrow$ hullHelper($points, a2, a1, d$)
    return $a1 : a2 : (h1 + + h2)$, with parallel $(h1, h2)$
end function

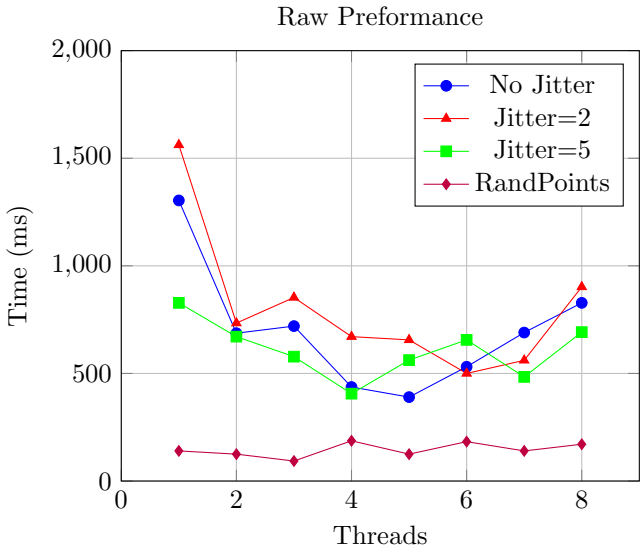
Algorithm 4 Parallel Quick Hull Helper

function HULLHELPER($points, a1, a2, d$)
    $group \leftarrow$ groupLeftOf($points, a1, a2$)
    $m1 \leftarrow$ findFurthestPoint($group, a1, a2$)
    $h1 \leftarrow$ hullHelper($group, a1, m1$)
    $h2 \leftarrow$ hullHelper($group, m1, a2$)
    if $group$ is empty **then**
        return []
    else if $d > 0$ and $length(group) > 250000$ and $\frac{length(group)}{length(points)} > 0.3$ **then**
        return $m1 : (h1 + + h2)$ with parallel $(h1, h2)$
    else
        return $m1 : (h1 + + h2)$
    end if
end function

4 Results

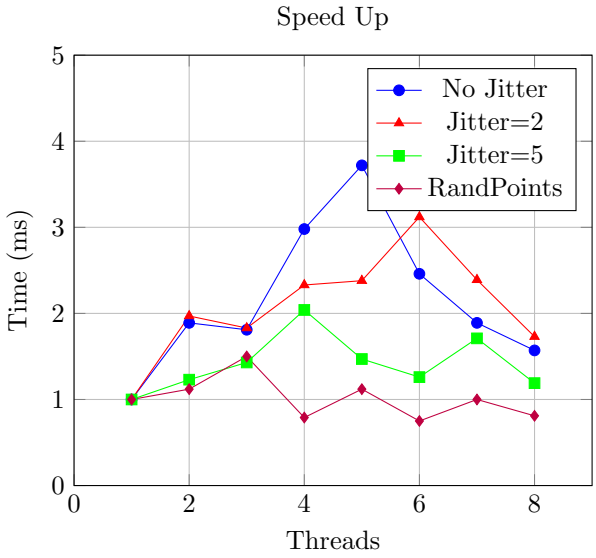
Time was computed by taking the difference between end time and start time of the method run, which was obtained through the *getCPUTime* method from the *System.CPUTime* package, as well as *deepseq* to force function evaluation. The reason that standard Haskell timing cannot be used is because the process of reading a file dominates the program runtime. Samples of four million points were used to test the parallel algorithm.

4.1 Raw Data



Threads	0 Jitter (ms)	2 Jitter (ms)	5 Jitter (ms)	Rand Points (ms)
1	1304	1562	828	140
2	687	734	671	125
3	720	853	578	93
4	437	671	406	187
5	390	656	562	125
6	531	500	656	187
7	690	651	484	140
8	828	902	692	171

4.2 Speedup Data



Threads	0 Jitter	2 Jitter	5 Jitter	Rand Points
1	1.00	1.00	1.00	1.00
2	1.89	1.97	1.23	1.12
3	1.81	1.83	1.43	1.50
4	2.98	2.33	2.04	0.79
5	3.72	2.38	1.47	1.12
6	2.46	3.12	1.26	0.75
7	1.89	2.39	1.71	1
8	1.57	1.73	1.19	0.81

4.3 Note About Data Collection

This data is slightly different from what was shown in our presentation. We added the jitter feature which creates a donut-like shape that the points occupy. For example if $j = x$, the points can by anywhere on the circumference of the circle or the circumference plus-minus x. A jitter of size 0 produced a hull of 4000000 points, a jitter of size 2 produced a hull 297393 point, and a jitter of size 5 produces a hull of size 99469. The fully random hull was of size 49

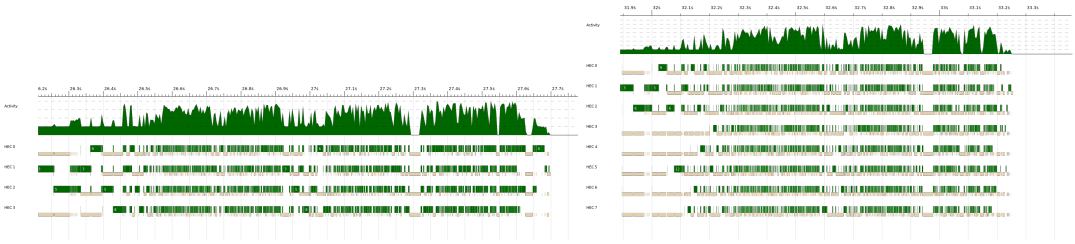
Furthermore, we ran the code in a slightly different way: we used cygwin instead of powershell, and we closed all other applications and turned off WIFI. This way, the code ran much faster, though the speed up itself remains unchanged.

In the next section, we show the thread scope graph for $-N4$ and $-N8$, which were separate runs from the data shown above. Again, minor fluctuations can be observed, but the bigger picture remains the same.

5 Threadscope Graphs

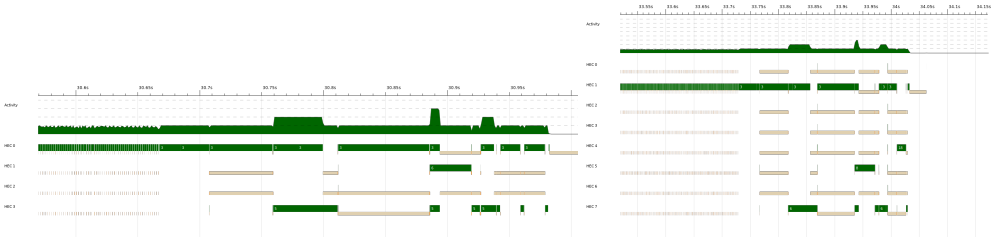
5.1 No Jitter

5.2 Jitter=2



5.3 Jitter=5

5.4 Fully Random Points



5.5 Amdahl's Law

When applying Amdahl's law, using the data from the sequential case, we estimate $P = 1/31$

$$limit_{N\to\infty} S = 31/30 = 1.033$$

We can therefore see that the algorithm is bottle-necked by the reading time. However, convex hull algorithms are typically used within the context of other tasks, so the impact of the one-time cost of reading would be reduced. Experimentally, we saw large variations in the reading time so we have no included a discussion about overall speedup.

6 Analysis and Discussion

6.1 Analysis of Results

After completing our project, we have come to see that QuickHull is not as parallelizable as it initially seems. However, our results show a reasonable parallelization of QuickHull along the edge cases, without seeing a decrease in the average case (randomly generated points). Parallelization seems to be most effective at around 4 or 5 threads.

As Amdahl's law shows, the bottleneck of reading the points is quite severe, overshadowing the gains made from the algorithm itself. However, we see this issue to be somewhat of an inevitability; no matter what, the points must be read.

While the overall gains from parallelization are rather slim when taken in isolation, if this algorithm is used in a larger pipeline, where the data is read once and many operations are preformed, then this parallelization would be justified.

We observed no major issues with garbage collecting, and our implementation of the algorithm itself is very fast.

6.2 Discussion and Future Work

Our first consideration about further improving our algorithm is attempting to resolve the bottleneck with regards to file reading. This is indeed the greatest issue hampering speed up, and resolving it would make our algorithm much more efficient.

Another area that requires attention is depth control. Our current depth control makes sure that parallelization only occurs along the recursion tree, however, the recursion tree expands exponentially with a factor of 2. While our current implementation manages to keep the workload pretty even, it also means that the numbers of threads needs to grow exponentially (with the tree) to see consistent improvement. This means our parallelization might not utilize all the

threads between powers of 2; notably garbage collection will still be parallelized, leading to some continual speedups, but the main part of the algorithm will not parallelize.

Finally, we must consider the accumulation of the hull. Insertion and concatenation in Unboxed Vectors is an $O(n)$ operation, which is not optimal. While this issue does not impact the overall asymptotic complexity of QuickHull, as there are other operations at each step of the recursion that take $O(n)$ time, it is an undeniable downside of this parallel implementation.

7 Bibliography

Wikipedia contributors. "Convex Hull Algorithms." Wikipedia, October 9, 2024. https://en.wikipedia.org/wiki/Convex hull algorithms.

"Quickhull." Wikipedia, April 25, 2023.
https://en.wikipedia.org/wiki/Quickhull.

Chan, T. "Optimal Output-sensitive Convex Hull Algorithms in Two and Three Dimensions." Discrete and Computational Geometry 16 (1996): 361–68.
https://www.cs.umd.edu/class/spring2020/cmsc754/Lects/lect03-hulls-bounds.pdf.

Coman, Andrei. "Project Proposal: Parallel Convex Hull." Project proposal. COMS 4995 Parallel Functional Programming, November 27, 2022.
https://www.cs.columbia.edu/ sedwards/classes/2022/4995-fall/proposals/ConvexHull.pdf.

8 Haskell Code for QuickHull

Parallel QuickHull Algorithm

```haskell
import Data.Ord (comparing)
import qualified Data.Vector.Unboxed as V
import Control.Parallel (par, pseq)

type V2 = (Double, Double)
type VV2 = V.Vector V2

{-Returns the convex hull-}
quickh :: VV2 -> Int -> VV2
quickh points d = V.cons a1 (V.cons a2 hpar)
  where
    a1 = minv points
    a2 = maxv points
    h1 = ph points a1 a2 (depthUpdate d)
    h2 = ph points a2 a1 (depthUpdate d)
    hpar =  h1 `par` (h2 `pseq` V.concat [h1, h2])

{-Helper function for quick hull-}
ph :: VV2 -> V2 -> V2 -> Int -> VV2
ph points a1 a2 d
  | V.length group == 0 = V.empty
  | d > 0 && V.length group > 250000 &&
     percentRemaining > 0.3 = V.cons m1 hpar
  | otherwise = V.cons m1 h
  where
    percentRemaining = fromIntegral (V.length group) /
        fromIntegral (V.length points) :: Double
    group = grouper a1 a2 points
    m1 = maxAreaPoint a1 a2 group
    h1 = ph group a1 m1 (depthUpdate d)
    h2 = ph group m1 a2 (depthUpdate d)
    h = V.concat [h1, h2]
    hpar = h1 `par` (h2 `pseq` V.concat [h1, h2])

{-Function to find the point with the maximum x-
    coordinate-}
maxv :: VV2 -> V2
maxv points = V.maximumBy (comparing fst) points

{-Function to find the point with the maximum x-
    coordinate-}
minv :: VV2 -> V2
```

```
minv points = V.minimumBy (comparing fst) points

{-Furthest point from a line segment defined by points
    a1, a2-}
maxAreaPoint :: V2 -> V2 -> VV2 -> V2
maxAreaPoint anchor1 anchor2 points = V.maximumBy (
   comparing (triArea anchor1 anchor2)) points

{-Groups by 2D determinants (cross product). Always
   return the points left of segment (a1,a2)-}
grouper :: V2 -> V2 -> VV2 -> VV2
grouper anchor1 anchor2 points = V.filter (\z ->
   determinant anchor1 anchor2 z > 0) points

{-Calculates 2d determinant, cross product-}
determinant :: V2 -> V2 -> V2 -> Double
determinant (z1, z2) (w1, w2) (u1, u2) = (w1 - z1) * (
   u2 - z2) - (w2 - z2) * (u1 - z1)

{-Area for triangle-}
triArea :: V2 -> V2 -> V2 -> Double
triArea (z1, z2) (w1, w2) (u1, u2) = abs ((z1 * (w2 -
   u2)) + (w1 * (u2 - z2)) + (u1 * (z2 - w2)))

{-Updates the depth of recursion-}
depthUpdate :: Int -> Int
depthUpdate d
  | d <= 1 = 0
  | otherwise = div d 2
```

Main Method

```haskell
module Main (main) where

import qualified Data.Vector.Unboxed as VU
import QuickHullV (VV2, quickh)
import System.CPUTime
import Control.DeepSeq
import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as BC
import GHC.Conc (getNumCapabilities)
import System.Environment (getArgs)

{-Basic main method. Times code-}
main :: IO ()
main = do
    args <- getArgs
    case args of
        [fileName] -> do
            threads <- getNumCapabilities
            putStrLn $ "Running with " ++ show threads
            print "Reading:"
            print fileName
            points <- readPointsFromFile fileName
            print (VU.length points)
            points `deepseq` putStrLn "Points have
                been fully read and evaluated"
            print "Starting Test"
            startT <- getCPUTime
            let parPoints = quickh points threads
            endT <- parPoints `deepseq` getCPUTime
            print (div (endT - startT) 1000000000)
            print (endT - startT)
            print (VU.length parPoints)
            print "done"
        _ -> putStrLn "Usage: program <input_file>"


{-Reads and parses the file into a points (vv2)-}
readPointsFromFile :: FilePath -> IO VV2
readPointsFromFile filePath = do
    content <- B.readFile(filePath)
    let linesOfFile = BC.lines content
        pointsList = map parseLine linesOfFile
    return $ listToVV2 pointsList
```

```
44
45 {-Parses a ByteString line into a tuple of doubles-}
46 parseLine :: B.ByteString -> (Double, Double)
47 parseLine line =
48     let [x, y] = map (read . BC.unpack) (BC.split ',' line)
49     in (x, y)
50
51 {-Converts a list of tuples to a vector of tupples (vv2)-}
52 listToVV2 :: [(Double, Double)] -> VV2
53 listToVV2 = VU.fromList
```

Tester Class

```
1  import qualified Data.Vector.Unboxed as VU
2  import System.Random (randomRIO)
3  import QuickHullV (V2, VV2, quickh)
4  import Data.List (sort, nub)
5  import Andrew (convexHull)
6  import Qhseq (qh)
7
8  {-
9  George Morgulis(gm3138)
10 Henry Lin(hkl2127)
11
12 This class simply checks for the correctness of the QuickHull Algorithm by comparing it to our correct
13 sequential implementation, and the correct implementation "Andrew Monontone Chain", which we found on wikepedia
14 -}
15
16 {-Generate a random point (x, y) where x and y are between -x and x-}
17 vRandomPoint :: Double -> IO V2
18 vRandomPoint x = do
19     xCoord <- randomRIO (-x, x)
20     yCoord <- randomRIO (-x, x)
21     return (xCoord, yCoord)
22
23 {-Generate a list of random points (n points)-}
24 vGeneratePoints :: Int -> IO VV2
25 vGeneratePoints n = VU.replicateM n (vRandomPoint 100000000)
26
```

```
{-Convert VV2 to a list of tuples-}
vv2ToListOfTuples :: VV2 -> [(Double, Double)]
vv2ToListOfTuples = VU.toList

{-This test checks simply for correctness of my
    algorithm by running it on a random set of points
  and checking with a correct implementation
-}
main :: IO ()
main = do
    print "Starting Point Generation"
    points <- vGeneratePoints 1000

    let convertedPoint = vv2ToListOfTuples points
    let seqPoints = sort (nub (qh convertedPoint))
    let parPoints = vv2ToListOfTuples (quickh points
        8)
    let andrewPoints = sort (nub (convexHull
        convertedPoint))

    print (length parPoints)
    print (length seqPoints)
    print (length andrewPoints)

    if (sort(nub parPoints) == sort(nub(seqPoints)))
        then print "Test Passed!"
        else print "Test Failed!"

    if (sort(nub parPoints) == sort(nub(andrewPoints))
        )
        then print "Andrew Passed"
        else print "Andew Failed"

    print "Complete!"
```

Original Sequential QuickHull Algorithm (deprecated)

```
    module Qhseq
    (C2, qh
    ) where

{-
George Morgulis(gm3138)
Henry Lin(hkl2127)
COMS 4995 Parallel Functional Programming

```

```
This is our very first sequential implmentation of Quick Hull.
(!!!) This code is not meant to be run as the project greatly changed since we first we wrote this
implementation. This simply exists to show how far the project has progressed.
-}

import Data.List (maximumBy, minimumBy, nub)
import Data.Ord (comparing)

{- Type used to represent points-}
type C2 = (Double, Double)

qh :: [C2] -> [C2]
qh points = (helper1 points [])
    where
        helper1 [] hull = hull -- starter
        helper1 (x : xs) hull =
            let m1 = maxAreaPoint a1 a2 group1
                m2 = maxAreaPoint a1 a2 group2
                group1 = fst (grouper a1 a2 (x:xs))
                group2 = snd (grouper a1 a2 (x:xs))
                a1 = mind (x:xs) 0
                a2 = maxd (x:xs) 0
            in a1 : a2 : helper2 a1 a2 m1 (keepOuter a1 a2 m1 group1) (m1 : hull) ++ helper2 a2 a1 m2 (keepOuter a1 a2 m2 group2) (m2 : hull)

        helper2 _ _ _ [] hull = hull -- calculates lower and upper hull
        helper2 o1 o2 pm (y:ys) hull =
            let m1 = maxAreaPoint o1 pm group1
                m2 = maxAreaPoint o2 pm group2
                group1 = fst (grouper o1 pm (y:ys)) -- always picking the left
                group2 = fst (grouper pm o2 (y:ys)) -- alwyas picking the left
            in helper2 o1 pm m1 (keepOuter o1 pm m1 group1) (m1 : hull) ++ helper2 pm o2 m2 (keepOuter o2 pm m2 group2) (m2 : hull) -- important: note the order of points

{-Computes the maximum of points by specified
```

```
    dimension-}
maxd :: [C2] -> Int -> C2
maxd points 0 = maximumBy (comparing fst) points
maxd points 1 = maximumBy (comparing snd) points
maxd _ _ = error "Invalid arguements."

{-Computes the minimum of points by specified
    dimension-}
mind :: [C2] -> Int -> C2
mind points 0 = minimumBy (comparing fst) points
mind points 1 = minimumBy (comparing snd) points
mind _ _ = error "Invalid arguements."

{-Calculates the maximum area given a line segment-}
maxAreaPoint :: C2 -> C2 -> [C2] -> C2
maxAreaPoint _ anchor2 [] = anchor2
maxAreaPoint anchor1 anchor2 points = maximumBy (
    comparing (triArea anchor1 anchor2)) points

{-Determines whether a point lies to the right or left
     of a vector. The first memeber of the return
tuple are the points to the left of the vector, the
    second are those to the right-}
grouper :: C2 -> C2 -> [C2] -> ([C2],[C2])
grouper anchor1 anchor2 points = helper anchor1
    anchor2 points [] []
    where
        helper _ _ [] group1 group2 = (group1, group2)
        helper (x1, y1) (x2, y2) (z:zs) group1 group2
            | closeEnough ((x2 - x1) * (snd z - y1) -
               (y2 - y1) * (fst z - x1)) 0 = helper (
               x1, y1) (x2, y2) zs group1 group2 --
               collinear points added to neither group
            | (x2 - x1) * (snd z - y1) - (y2 - y1) * (
               fst z - x1) > 0 = helper (x1, y1) (x2,
               y2) zs (z : group1) group2 -- left of
            | otherwise = helper (x1, y1) (x2, y2) zs
               group1 (z : group2) -- right of


{-Keeps all points outside the triangle. Works for
    everything other than points on triangle itself-}
keepOuter :: C2 -> C2 -> C2 -> [C2] -> [C2]
keepOuter t1 t2 t3 points = helper t1 t2 t3 points []
    where
        helper _ _ _ [] keep = keep
```

```
        helper p1 p2 p3 (x : xs) keep
            | pointInTriangle p1 p2 p3 x = helper p1 p2 p3 xs keep
            | otherwise = helper p1 p2 p3 xs (x : keep)

{-Finds the area of a triangle-}
triArea :: C2 -> C2 -> C2 -> Double
triArea (x1, y1) (x2, y2) (x3, y3) =  abs ((x1 * (y2 - y3)) + (x2 * (y3 - y1)) + (x3 * (y1 - y2)))

{-Checks if point lies inside triangle-}
pointInTriangle :: C2 -> C2 -> C2 -> C2 -> Bool
pointInTriangle t1 t2 t3 p =
    closeEnough (triArea t1 t2 p + triArea t1 t3 p + triArea t2 t3 p) (abs (triArea t1 t2 t3))

{-Epsilon value to mitigate floating point error-}
epsilon :: Double
epsilon = 1e-9

{-Method to mitigate floating point error-}
closeEnough :: Double -> Double -> Bool
closeEnough a b = abs (a - b) < epsilon
```

9 Python Helper Code

Python Code to generate data around a circle with optional gitter

```
    import numpy as np
import random

# George Morgulis(gm3138)
# Henry Lin(hkl2127)
# Functions to create edge cases that look like donuts (use jitter=0 for perfect circle)
# Based on this stack overflow post:
# https://stackoverflow.com/questions/8487893/generate-all-the-points-on-the-circumference-of-a-circle
def generate_circle_points(radius, num_points, jitter):
    angles = np.linspace(0, 2 * np.pi, num_points, endpoint=False)
    x_rad = []
    y_rad = []
```

```
    for i in range(num_points):
        x_rad.append(radius + random.randrange(-jitter
            ,jitter))
        y_rad.append(radius + random.randrange(-jitter
            , jitter))
    x_coords = np.multiply(np.array(x_rad), np.cos(
        angles))
    y_coords = np.multiply(np.array(y_rad) ,np.sin(
        angles))
    return x_coords, y_coords

def write_points_to_file(filename, x_coords, y_coords)
    :
    with open(filename, 'w') as file:
        for x, y in zip(x_coords, y_coords):
            file.write(f"{x},{y}\n")

radius = 50000000
num_points = 4000000
jitter = 5
x_coords, y_coords = generate_circle_points(radius,
    num_points,jitter)

filename = "d4m5j.txt"
write_points_to_file(filename, x_coords, y_coords)

print(f"Points have been written to {filename}")
```

Python Code to Generate Random Points

```
    import random

def generate_random_points(num_points, x_range,
    y_range):
    points = []
    for _ in range(num_points):
        x = random.uniform(*x_range)
        y = random.uniform(*y_range)
        points.append((x, y))
    return points

def write_points_to_file(points, filename):
    with open(filename, 'w') as file:
        for point in points:
            file.write(f'{point[0]}, {point[1]}\n')

```

```
num_points = 4000000
x_range = (-50000000, 50000000)
y_range = (-50000000, 50000000)

random_points = generate_random_points(num_points,
    x_range, y_range)
write_points_to_file(random_points, 'r4m.txt')
```

10 Bash Helper Code

Runs our code in a batch.

```
#!/bin/bash

echo "here"

run_command() {
  local n=$1
  stack run -- +RTS -N$n -s -ls -RTS r4m.txt
}

for n in {1..8}; do
    echo "Iteration $iteration, running with -N$n"
    run_command $n
done
```