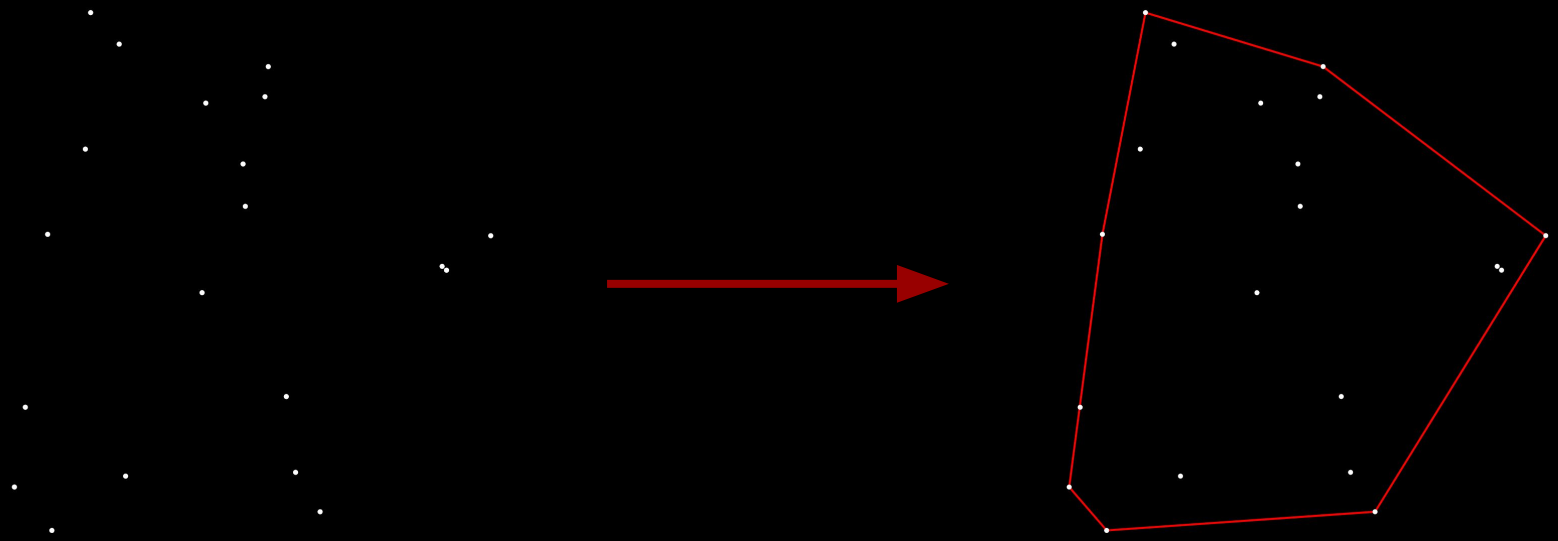
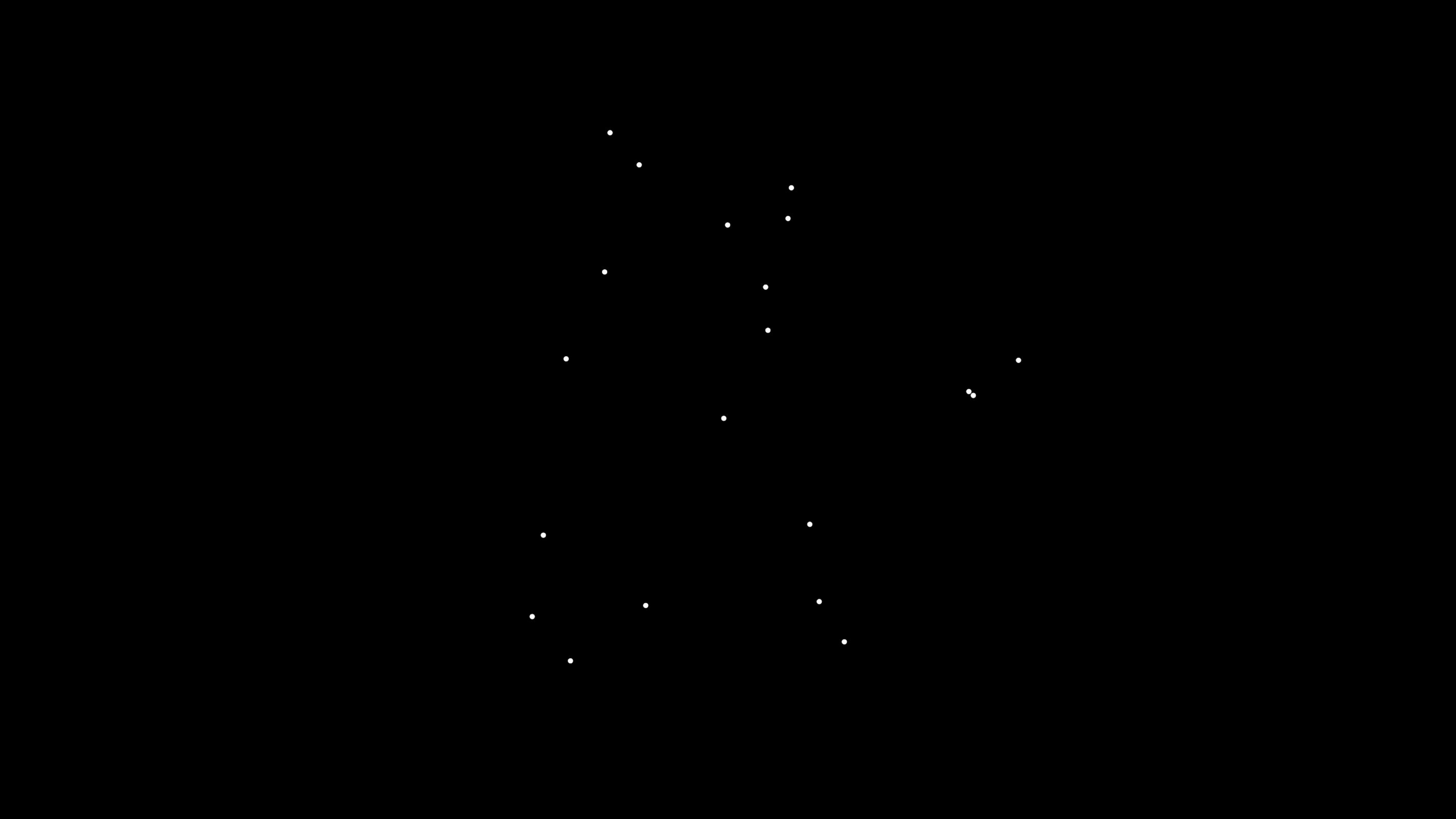
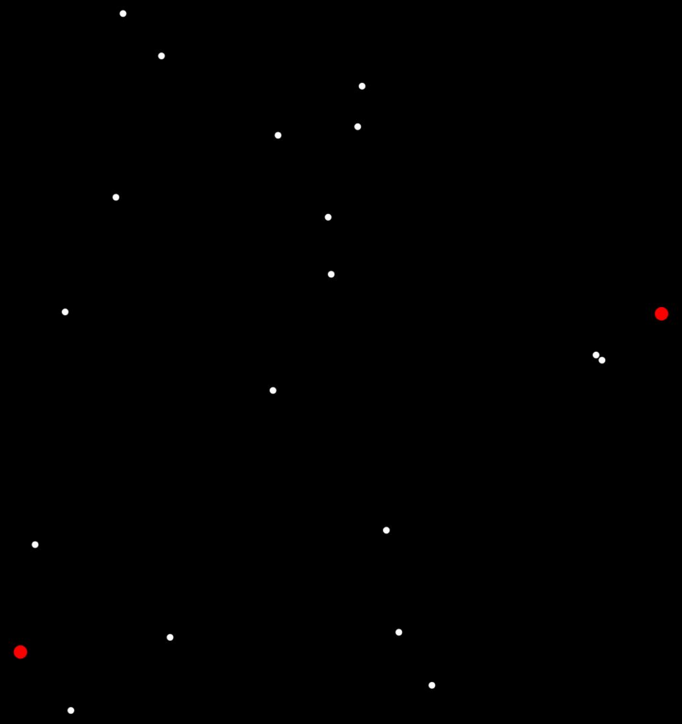
Quick Hull

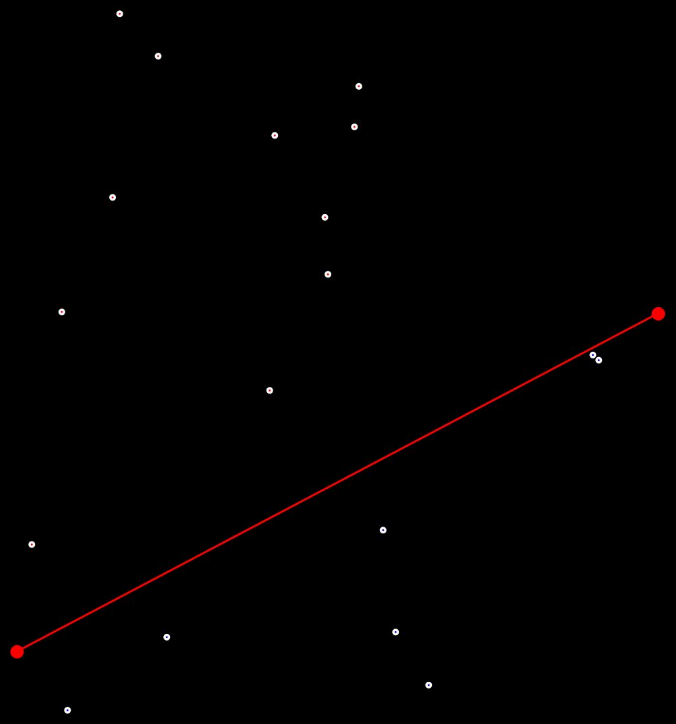
George Morgulis and Henry Lin

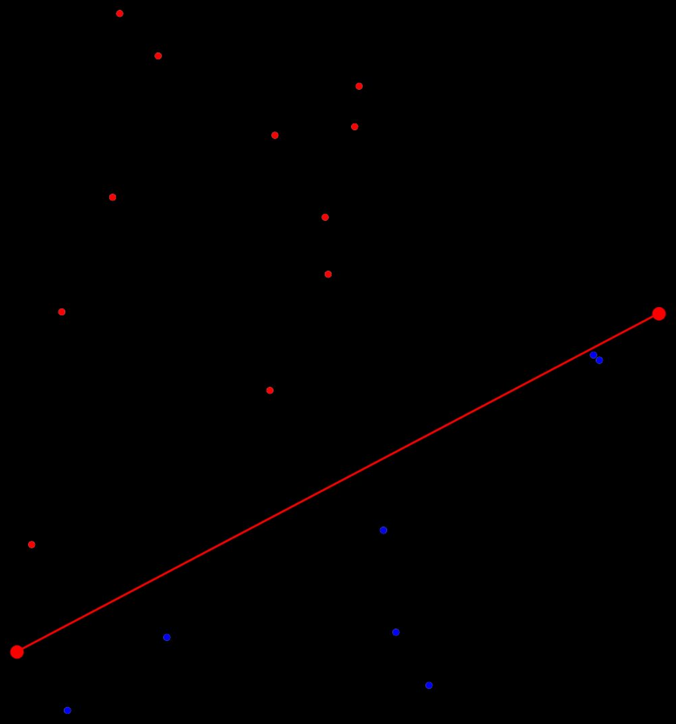
The Convex Hull Problem

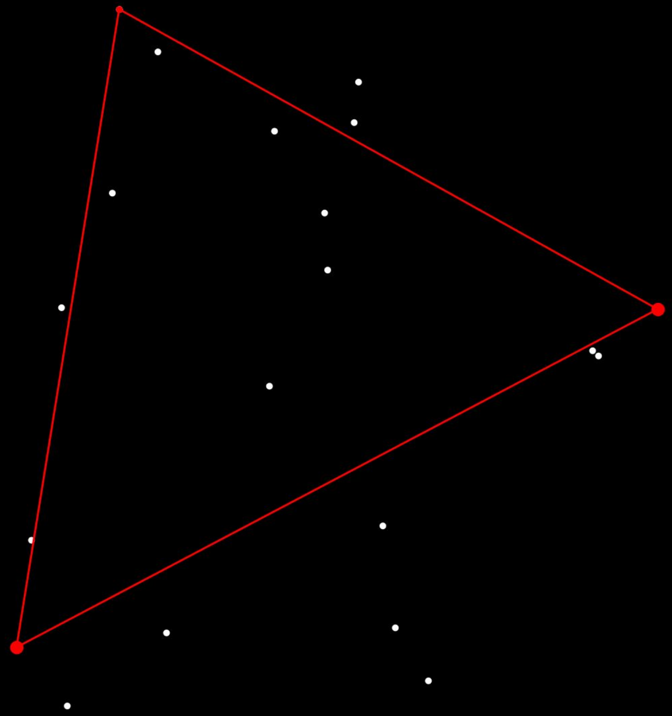


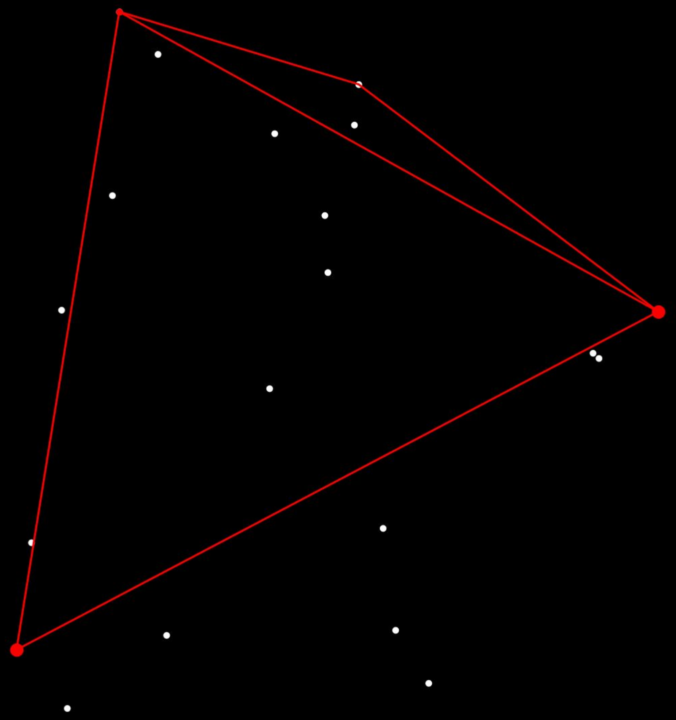


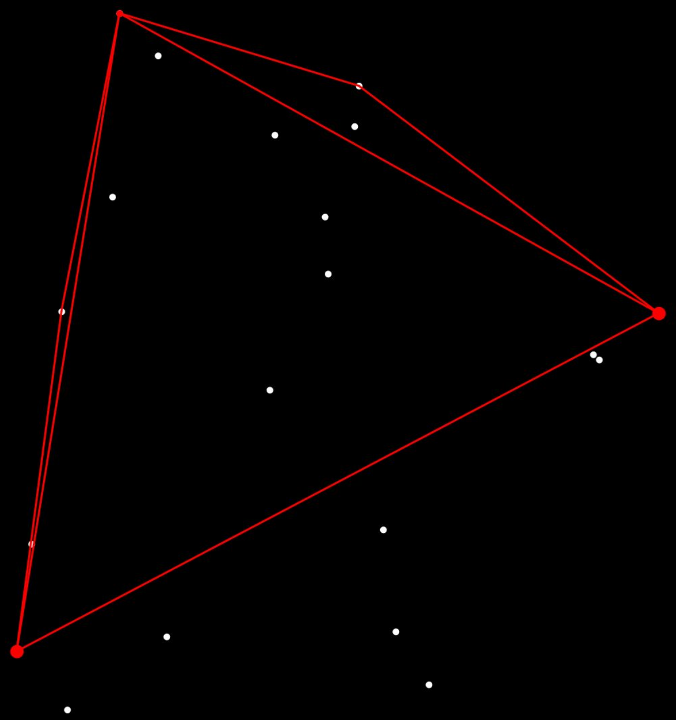




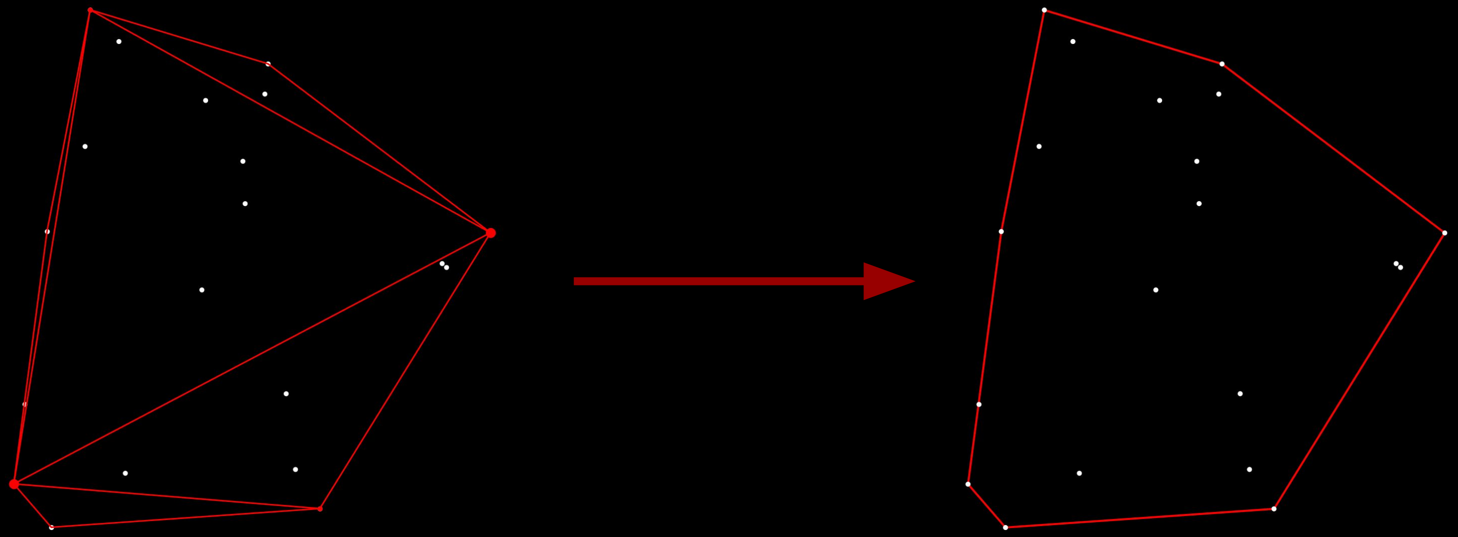








The Convex Hull Problem



Algorithm Part 1

Algorithm 1 Quick Hull Algorithm

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

Algorithm Part 2

Algorithm 2 Quick Hull Helper

```
function HULLHELPER(points, a1, a2)
    group ← groupLeftOf(points, a1, a2)
    if group is empty then
        return []
    else
        m1 ← findFurthestPoint(group, a1, a2)
        h1 ← hullHelper(group, a1, m1)
        h2 ← hullHelper(group, m1, a2)
        return m1 : (h1 + +h2)
    end if
end function
```

Seems Simple to Parallelize

- Parallelize 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

Seems Simple to Parallelize

IT WAS NOT THAT SIMPLE.

Seems Simple to Parallelize

- Parallelization **findMinPoint, findMaxPoint**, and **findFurthestPoint** was useless for sets of points of size 4,000,000.
- The parallel implementation that I described only slowed down the code

Seems Simple to Parallelize

- Parallelizing the recursive calls is tricky!
- The difficulty is that on average, the convex hull of a set of points is many times smaller than the input set. For example, we 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.

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

Note about Data Structures

- 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 Unboxed Vector as the data structure for QuickHull.

Parallel Implementation

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

Parallel Implementation

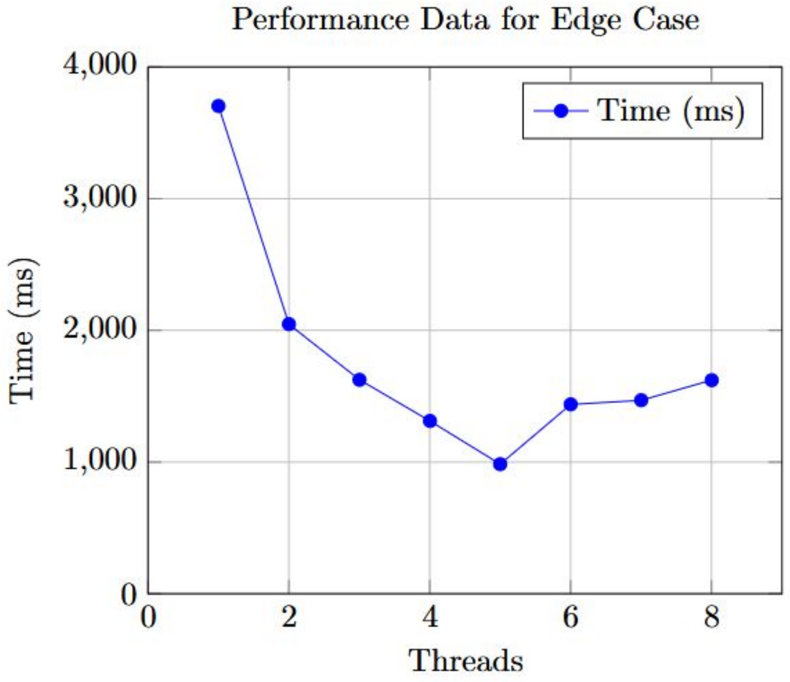
Algorithm 4 Parallel Quick Hull Helper

```
function HULLHELPER(points, a1, a2, d)
    group ← groupLeftOf(points, a1, a2)
    m1 ← findFurthestPoint(group, a1, a2)
    h1 ← hullHelper(group, a1, m1)
    h2 ← hullHelper(group, m1, a2)
    if group is empty then
        return []
    else if d > 0 and length(group) > 250000 and length(group)/length(points > 0.3 then
        return m1 : (h1 + +h2) with parallel (h1, h2)
    else
        return m1 : (h1 + +h2)
    end if
end function
```

A note about timing

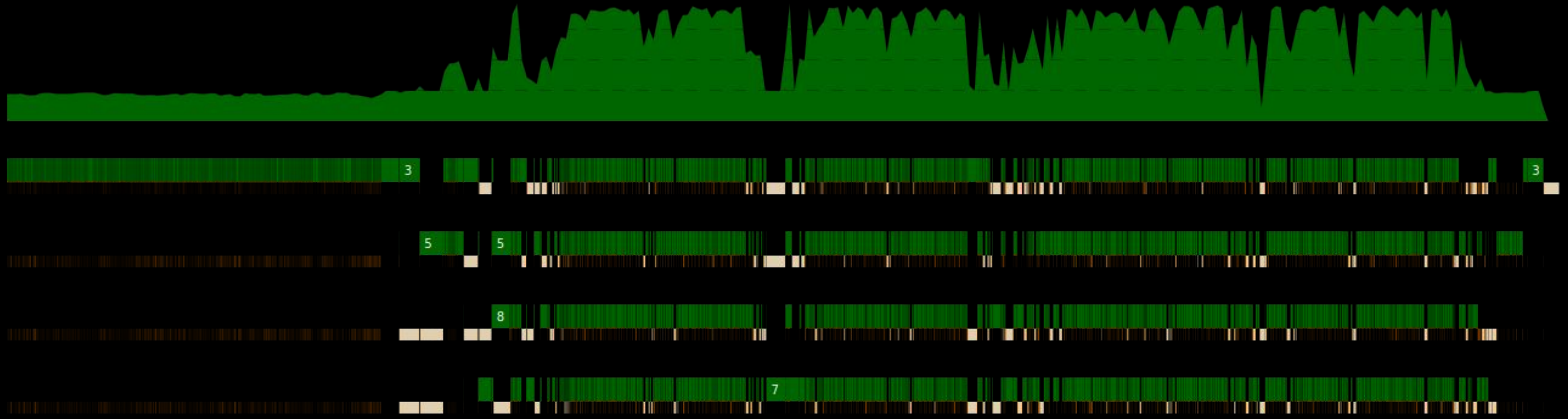
```
  SPARKS: 1 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)

  INIT    time    0.000s  (  0.003s elapsed)
  MUT     time   27.172s  ( 49.761s elapsed)
  GC      time    0.422s  (  1.099s elapsed)
  EXIT    time    0.000s  (  0.000s elapsed)
  Total   time   27.594s  ( 50.864s elapsed)

  Alloc rate    3,410,975,710 bytes per MUT second

  Productivity  98.5% of total user, 97.8% of total elapsed

PS C:\Users\georg\HaskellProjects\final\convex-hull>
```

Parallel Implementation Results

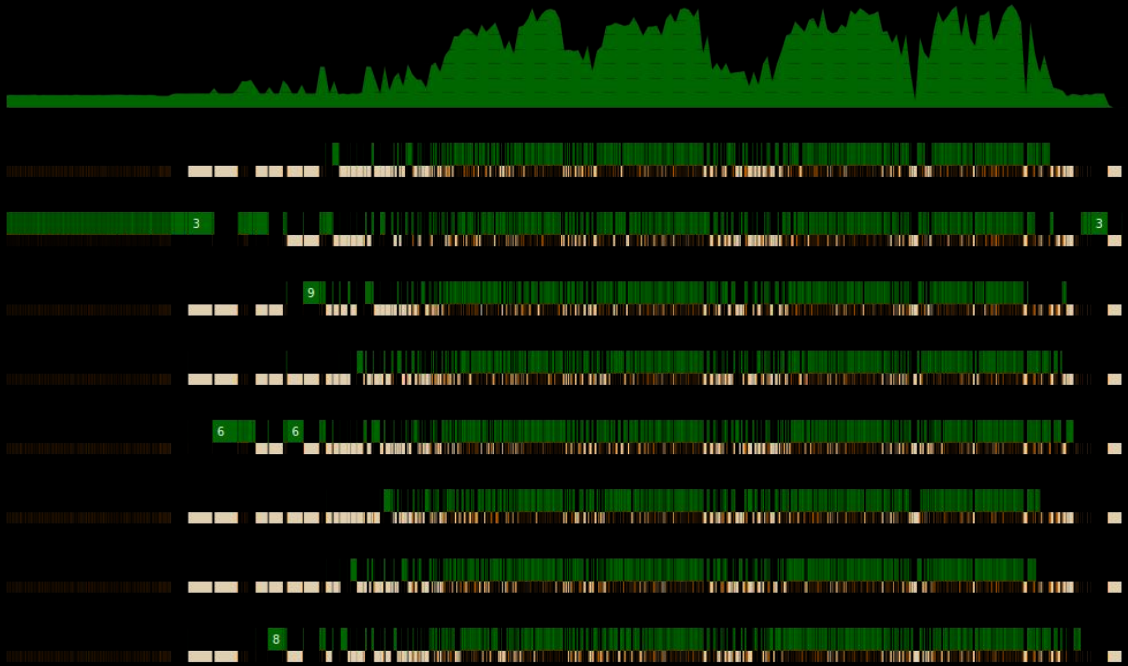


Threads	Time (ms)
1	3703
2	2046
3	1625
4	1312
5	984
6	1437
7	1468
8	1622

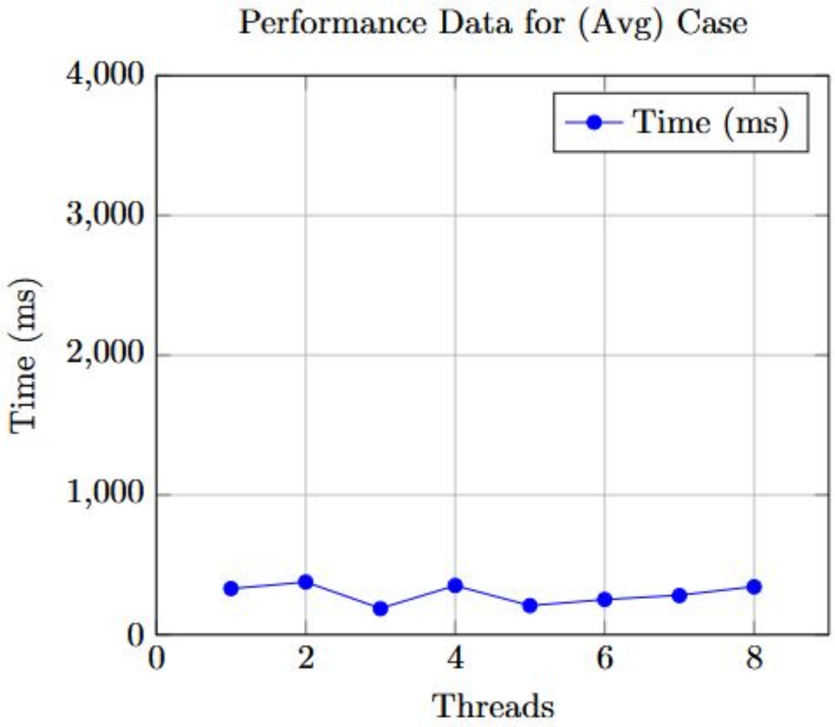
Parallel Implementation Results



Parallel Implementation Results



Parallel Implementation Results



Threads	Time (ms)
1	328
2	375
3	187
4	350
5	207
6	250
7	281
8	343

End