Project Report: Parallel Convex Hull

Andrei Coman, ac4808@columbia.edu

December 18, 2022

1 Introduction

This report presents a parallel Haskell implementation for the Convex Hull problem. More formally, given N points in the xy-plane, we want to find their convex hull - the smallest convex set containing all N points, or, alternatively, the intersection of all convex sets containing all N points.

An efficient solution to this problem is called Graham's Scan and is presented in Section 2. Then, the remaining sections present the parallelization of this algorithm along with several performance measures of the accompanying Haskell implementation. The complete code listing and usage are presented in the Annex.

2 Sequential Algorithm

Graham's Scan considers the convex-hull of the N points as composed of an upper-hull and a lower-hull. If we let L denote the leftmost point of this convex hull and R denote the rightmost point, the upper-hull contains all the vertices which lie above the $[LR]$ segment, whereas the lower-hull contains the ones below. To compute the upper-hull, the algorithm first sorts the points in increasing order of their x-coordinate. Then, it iterates through the points, maintaining at each step a "convex" stack - where, by convex, we mean that any three adjacent points on the stack form a clockwise turn. Whenever a point is inserted, the stack is popped as much as needed to preserve the convexity property (see Fig. 1 and Fig. 2). The computation of the lower-hull is symmetric, but a concavity property is now enforced. Once the upper and lower hull are computed, the convex hull is their union. This algorithm has complexity $O(NlogN)$ for the sort and $O(N)$ for the following sweeps. A Haskell implementation of this idea can be found in Annex B, under src/Lib.hs.

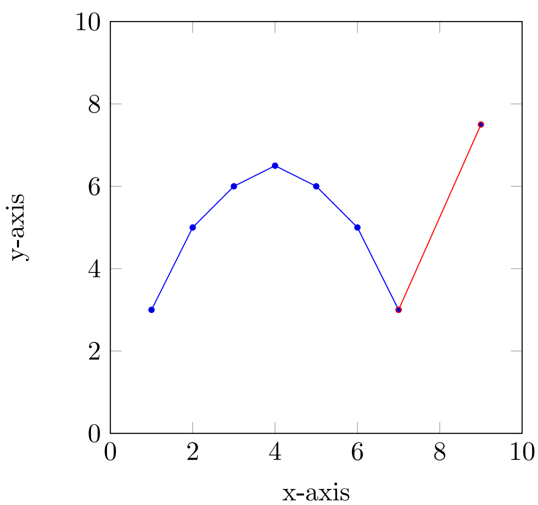

Figure 1: Stack Before Insertion (Convexity Violated)

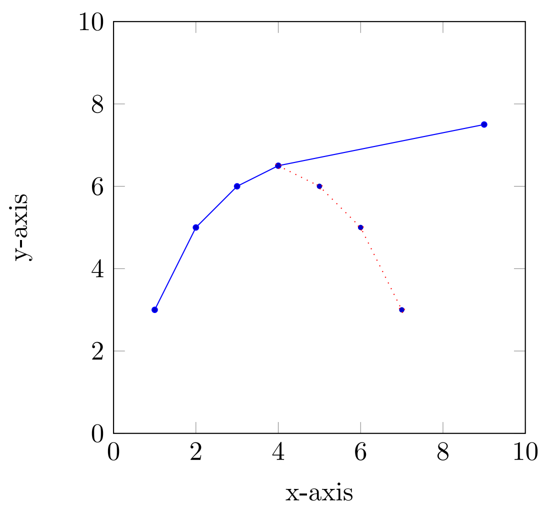

Figure 2: Stack Following Insertion (Convexity Restored)

3 Parallel Algorithm

3.1 Convex Hull Merging

To parallelize this algorithm, we can adopt the following divide and conquer approach: instead of computing the hull of the entire dataset directly, we split the dataset into two halves, compute their individual hulls in parallel, then merge them to obtain the overall hull. While merging two arbitrary possibly intersecting convex polygons (in particular, convex hulls) can be performed efficiently using more involved data structures, we can choose a split that minimizes the complexity of this operation.

In particular, let us choose some arbitrary vertical line in the xy-plane and partition the points into two subsets according to the side of the separating line they fall on. Then, the convex hulls corresponding to these two subsets will be linearly separated by the aforementioned line. Thus, we only need to handle the problem of merging two linearly-separated convex hulls. This operation can be performed efficiently as follows.

Consider, as in the sequential algorithm, that a convex hull is the union of an upper-hull and a lower-hull. Then, the convex hull resulting from the merge is determined by the common upper-tangent of the two upper-hulls and the common lower-tangent of the two lower-hulls (Fig. 3, Fig. 4).

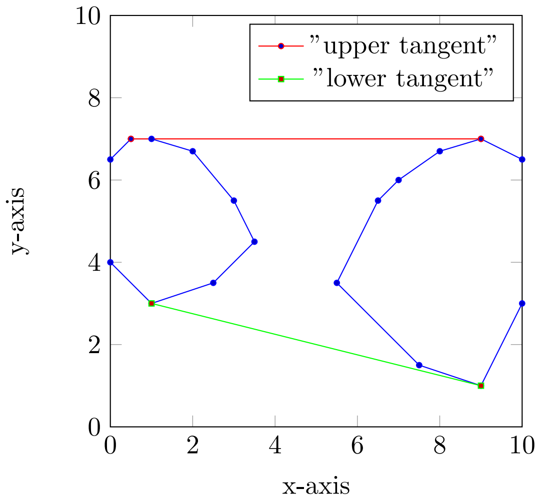

Figure 3: Upper/Lower Tangents

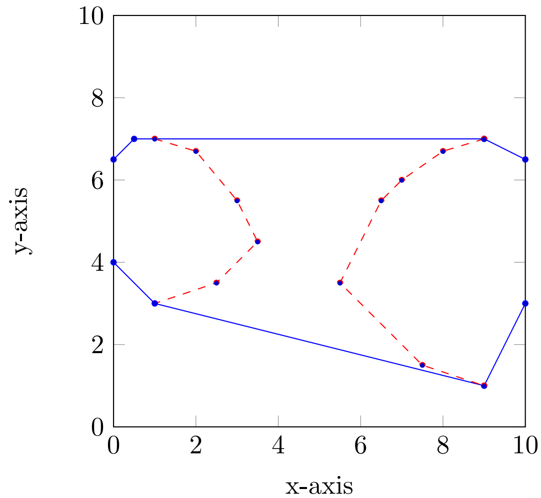

Figure 4: Merged Convex Hulls

Now, these two common tangents can be found by the following "iterative" algorithm. We will look at the process for finding the upper-tangent (the one for the lower-tangent is symmetric). Consider the segment between the left hull's rightmost point and the right hull's leftmost point as the first candidate for the upper-tangent. If shifting this segment to the next point on the left would create a concave angle (Fig. 5), shift the tangent to the left and continue the process recursively. If shifting the segment to the next point on the right would create a concave angle (Fig. 6), shift the tangent to the right and continue the process recursively. Otherwise, the current candidate is the common upper-tangent of the two hulls.

3.2 Parallelism Strategies

Having this merging algorithm for linearly-separated convex hulls, I considered several possible strategies for parallelizing the overall algorithm. The most obvious strategy is to first sort the points by their x-coordinate, then split them into several contiguous chunks and compute the convexHull of each chunk in parallel using parList. Since the points had been sorted before the split, the resulting convex-hulls are necessarily separated by some vertical lines and, so, the algorithm discussed above can be applied. The merging can either be applied as a fold over the resulting hulls or in a divide-and-conquer manner. However, since the convex hull has a logarithmic size in the number of points, this choice does not have a big impact.

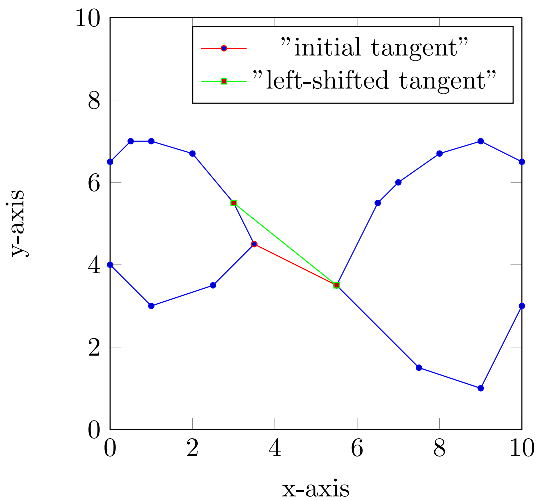

Figure 5: Left-shifting a Tangent

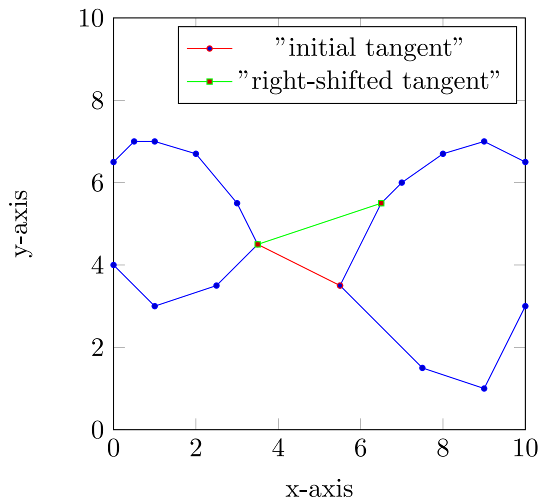

Figure 6: Right-shifting a Tangent

Parallelization using parList

```
convexHullHelper :: [Point] -> Hull
convexHullHelper xs =
  let chunks = chunksOf chunkSize xs in
  let hulls = map (force . convexHullNaive) chunks `using` parList rseq in
  List.foldl1 mergeHulls hulls
```

However, profiling this solution reveals that, when using several threads, the runtime becomes dominated by the sorting algorithm. This can be improved by implementing a simple parallel quicksort which partitions the array using some pivot, but then proceeds to sort the two partitions in parallel before merging them. To reduce the number of unnecessary (overflowed/fizzled/GC'd) sparks, we can also introduce a threshold for parallelism, after which the algorithm is just executed in series.

Parallel Naive QuickSort

```
quicksort :: (NFData a, Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x : xs) =
  let l = quicksort [y | y <- xs, y <= x] in
  let r = quicksort [y | y <- xs, y > x] in
  if length xs > chunkSize then
    let parRes = do l' <- rpar (force l)
                    r' <- rpar (force r)
                    _ <- rseq l'
                    _ <- rseq r'
                    return (l', r') in
    let (l', r') = runEval $ parRes in
    l' ++ [x] ++ r'
  else l ++ [x] ++ r
```

These two ideas already lead to good parallel performance. However, the runtime can still be improved by leveraging the similarity between the sorting algorithm and the divide-and-conquer approach for splitting and merging the convex hulls. More specifically, we can modify the parallel computation of the convex hull can be modified to follow the same structure as the sort: split the dataset into three partitions using some arbitrary pivot: elements to the left of the pivot, the pivot, elements to the right of the pivot. Then, depending on whether some threshold is exceeded, either compute the convex hull using the naive sequential algorithm or recursively compute the convex hull for all partitions recursively. Then, merge the left hull with the pivot and the right hull.

Divide-and-Conquer Parallelism (see Annex B for full code listing)

```haskell
{- recursively partition the input array; call naive
 - convex hull algorithm on leaves; merge left and right
 - hulls at internal nodes
 -}
convexHullHelper :: [Point] -> Hull
convexHullHelper xs =
  if length xs <= chunkSize
  then convexHullNaive $ quicksort xs
  else -- partition list of points into left/right halves
       let pivot = head xs in
       let (ls, _, rs) = partition xs pivot [] [] [] in

       -- compute the convex hulls of the two halves in parallel
       let lhull = convexHullHelper ls in
       let rhull = convexHullHelper rs in
       let parRes = do lhull' <- rpar (force lhull)
                       rhull' <- rpar (force rhull)
                       _ <- rseq lhull'
                       _ <- rseq rhull'
                       return (lhull', rhull') in
       let (lhull', rhull') = runEval $ parRes in

       -- merge the left hull, the pivot and the right hull
       lhull' `mergeHulls` (Hull (Cap [pivot] [pivot]) (Cap [pivot] [pivot]))
              `mergeHulls` rhull'
```

This implementation outperforms the others mentioned so far and is, therefore, the final parallel approach. For a full listing of the code with all the helper functions, see Annex B.

4 Parameter Fitting

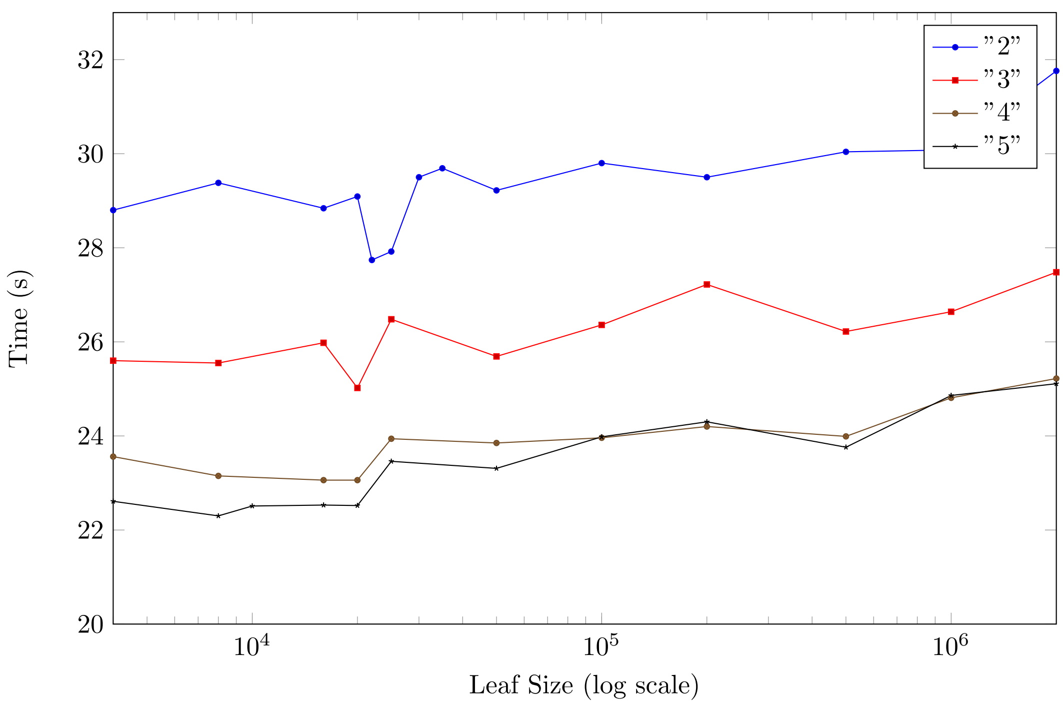

Figure 7: Runtime for different numbers of threads at different granularities (10^7-point dataset)

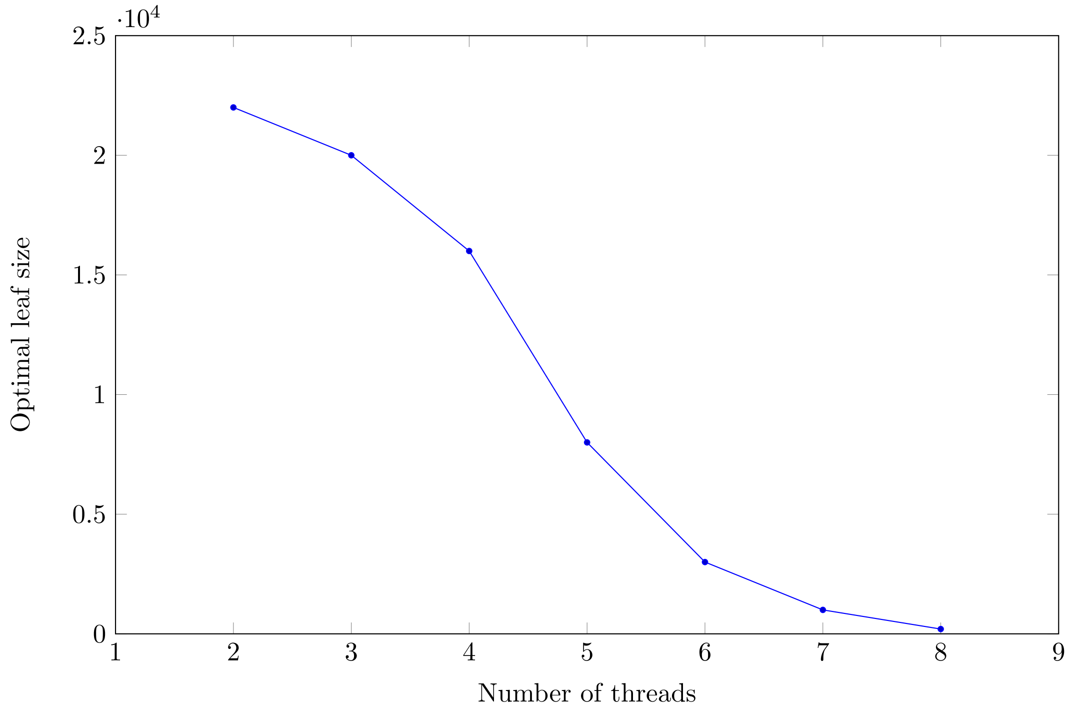

Figure 8: Optimal leaf size for different numbers of threads

The approach detailed above requires the user to set one parameter: the chunk size or leaf size - the number of points beyond which employing parallelism in the convex hull computation becomes less efficient than the sequential method. Fig. 7 shows how the runtime of the algorithm differs for different leaf sizes given different numbers of threads. Taking the optimal size for each number of threads, we obtain Fig. 8. The optimal chunk size follows a function resembling a reverse-sigmoid. So, if one were to automate setting this parameter, fitting this graph with a reverse-sigmoid should yield good results.

5 Results

Running the sequential implementation and the parallel implementation (with different numbers of threads, always with the optimal chunk/leaf size) yields the following runtimes and speedups:

No. threads	RT	Speedup
sequential	41.37	-
2	27.74	x1.49
3	25.02	x1.65
4	23.06	x1.79
5	22.30	x1.86
6	21.37	x1.94
7	20.97	x1.97
8	21.13	x1.96
9	21.71	x1.90

Table 1: Speedup

Here, we can see that the optimal speedup is close to x2 and is obtained for 7 threads.

Using Threadscope, we can visualize how the load is balanced throughout the entire run:

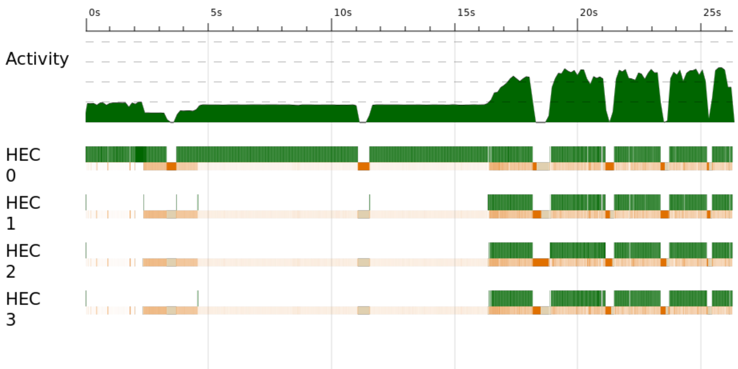

Figure 9: Overall Runtime of Algorithm (Threadscope)

and just throughout the parallel section:

Figure 10: Parallel Section of the Algorithm (Threadscope)

We can see that, after the data is read from disk (which still takes a considerable amount of time due to the size of the input), the parallel section is fairly-well balanced. In the case depicted above, which uses only 4 threads, all threads are performing work consistently throughout the parallel section of the execution. However, the overall profile is not smooth because of the large amount of garbage collection - garbage is generated by the partitioning of the dataset, the sorting, and the stack in Graham's algorithm.

6 Sample Input/Output

Sample Input/Output

```
2.0 0.0
0.0 2.0
1.0 3.0
0.0 4.0
3.0 3.0
2.0 6.0
4.0 2.0
4.0 4.0
```

```
(0.0, 2.0)
(2.0, 0.0)
(4.0, 2.0)
(4.0, 4.0)
(2.0, 6.0)
(0.0, 4.0)
```

Note: the implementation presented here produces only the vertices of the convex hull. Other tools must be used for visualization.

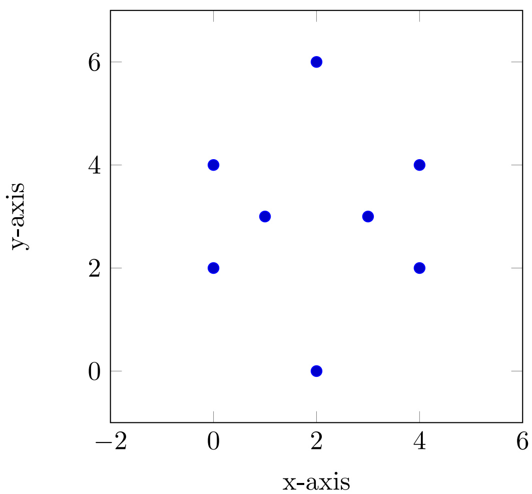

Figure 11: Dataset Visualization

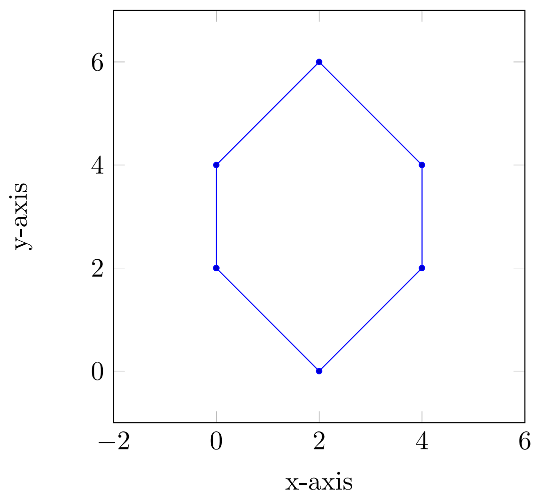

Figure 12: Convex Hull Visualization

7 Conclusion

This report presents how Graham's scan can be efficiently parallelized, resulting in an almost-x2 speedup over the entire runtime of the program. This shows that, even though Graham's scan has a linear complexity, the stack based algorithm preceeded by a sorting of the dataset can be made much more efficient by introducing parallelism at all stages (besides I/O). The best performance was obtained for 7 threads and a parallel/sequential threshold of 1,000 points. This allowed the processing of 10,000,000 points in 20.97 seconds.

The workload is distributed well amongst the working threads, which leads to fairly good balancing. However, the garbage collection is a noticeable setback to this approach, constantly fragmenting the execution of the threads and resulting in long, frequent garbage collection breaks. The question of how garbage production could be minimized could be considered as further work.

A Usage

The following command runs the parallel algorithm with n threads:

```
stack run -- +RTS -Nn -s -ls -RTS p <input-file> <output-file>
```

As a side effect, this command also prints runtime statistics to standard error and produces an associated .eventlog file which can be inspected with Threadscope. To run the sequential implementation mentioned in the beginning of this report, use the command:

```
stack run -- +RTS -Nn -s -ls -RTS s <input-file> <output-file>
```

Input files must contain a sequence of points, each represented as a space-separated pair of doubles per line. To test the implementation, run:

```
stack test
```

B Code Listing

app/Main.hs

```haskell
module Main (main) where

import qualified Data.Text as B (words, lines)
import qualified Data.Text.IO as B (readFile)
import qualified Data.Text.Read as B (double)
import Types (Point(..))
import qualified Lib (convexHull)
import qualified ParLib (convexHull)
import System.Environment (getArgs, getProgName)
import System.Exit (exitFailure)
import System.IO (hPutStrLn, stderr, withFile, IOMode(WriteMode))

main :: IO ()
main = do
  args <- getArgs
  case args of
    [mode, inputFilename, outputFilename] -> do
      contents <- B.readFile inputFilename
      let pts = map (\p -> case p of
                          [xstr, ystr] -> (case (B.double xstr, B.double ystr) of
                                             (Right (x, _), Right (y, _)) ->
                                               Point x y
                                             _ -> error "error while parsing input"
                                          )
                          _             -> error "malformed input"
                    )
                    (map B.words $ B.lines contents)
      let hull = case mode of
                   "p" -> ParLib.convexHull pts
                   "s" -> Lib.convexHull pts
                   _   -> error "invalid mode"
      withFile outputFilename WriteMode
               (\h -> do mapM_ (\p -> hPutStrLn h $ show p) $ hull)

    _ -> do
      pn <- getProgName
      hPutStrLn stderr $ "Usage: " ++
                         pn ++
                         "<mode(p/s)> <input-filename> <output-filename>"
      exitFailure
```

src/Types.hs

```haskell
module Types
    ( Point(..)
    , Cap(..)
    , Hull(..)
    ) where

import Control.Parallel.Strategies
import Control.DeepSeq
-- a cap is either the top or the bottom half of a hull
-- consists of its left-right and its right-left traversal
-- a hull is made out of its upper-hull/cap and its lower one
data Cap = Cap [Point] [Point]
data Hull = Hull Cap Cap
instance NFData Hull where
  rnf (Hull (Cap upperLR upperRL)
            (Cap lowerLR lowerRL)) = (rnf upperLR) `seq`
                                     (rnf upperRL) `seq`
                                     (rnf lowerLR) `seq`
                                     (rnf lowerRL)

data Point = Point Double Double
instance Eq Point where
  (Point ax ay) == (Point bx by) = (ax == ay) && (bx == by)
instance Ord Point where
  (Point ax ay) `compare` (Point bx by) =
    case (ax `compare` bx) of
      LT -> LT
      GT -> GT
      EQ -> ay `compare` by
instance Show Point where
  show (Point x y) = "(" ++ (show x) ++ ", " ++ (show y) ++ ")"
instance NFData Point where
  rnf (Point x y) = (rnf x) `seq` (rnf y)
```

app/ParLib.hs

```haskell
module ParLib
    ( convexHull
    ) where

import qualified Data.List as List
import Control.Parallel.Strategies
import Control.DeepSeq
import Types (Point(..), Cap(..), Hull(..))

{- get the convexity of an ordered triplet of points
 - by computing a quantity proportional to the signed
 - area of the triangle formed by these points
 - this is > 0 when convex, < 0 when concave and
 - = 0 for collinear points
 -}
convexity :: Point -> Point -> Point -> Double
convexity (Point ax ay)
          (Point bx by)
          (Point cx cy) = (ax * by + bx * cy + cx * ay) -
                          (ax * cy + bx * ay + cx * by)

{- find the upper hull of an xy-sorted set of points
 - maintain a stack and enforce convexity at insertion
 -}
upperHull :: [Point] -> [Point]
upperHull xs = List.foldl insertConvex [] xs -- for every point
  where insertConvex :: [Point] -> Point -> [Point]
        insertConvex stack@(a : pop@(b : _)) p =
          if convexity b a p < 0
```

```
          then p : stack           -- add on top of stack
          else insertConvex pop p -- pop and retry
        insertConvex stack p =
          p : stack

{- find the lower hull of an xy-sorted set of points
 - maintain a stack and enforce cocavity at insertion
 -}
lowerHull :: [Point] -> [Point]
lowerHull xs = List.foldl insertConcav [] xs -- for every point
  where insertConcav :: [Point] -> Point -> [Point]
        insertConcav stack@(a : pop@(b : _)) p =
          if convexity b a p > 0
          then p : stack           -- add on top of stack
          else insertConcav pop p -- pop and retry
        insertConcav stack p =
          p : stack

{- compute the lower and upper hull using the naive
 - Graham's scan algorithm and then wrap them into
 - a "Hull" instance
 -}
convexHullNaive :: [Point] -> Hull
convexHullNaive xs =
  let lower = lowerHull xs in
  let upper = upperHull xs in
  Hull (Cap (reverse lower) lower) (Cap (reverse upper) upper)

{- find the common upper tangent of two hulls
 - the input consists of the list starting at the
 - rightmost point of the left hull, followed by
 - the list starting at the leftmost point of the
 - right hull
 - algorithm: shift the tangent incrementally
 - until the optimum is reached
 -}
upperTangent :: [Point] -> [Point] -> ([Point], [Point])
upperTangent xl xr
  -- can shift tangent to the right
  | (l : _) <- xl
  , (r : xrs@(rnext : _)) <- xr
  , convexity l r rnext > 0 = upperTangent xl xrs
  -- can shift tangent to the left
  | (l : xls@(lnext : _)) <- xl
  , (r : _) <- xr
  , convexity lnext l r > 0 = upperTangent xls xr
  -- optimum reached
  | otherwise = (xl, xr)

{- similar to upperTangent, but checks for concavity
 - instead of convexity
 -}
lowerTangent :: [Point] -> [Point] -> ([Point], [Point])
lowerTangent xl xr
  -- can shift tangent to the right
  | (l : _) <- xl
  , (r : xrs@(rnext : _)) <- xr
  , convexity l r rnext < 0 = lowerTangent xl xrs
  -- can shift tangent to the left
  | (l : xls@(lnext : _)) <- xl
  , (r : _) <- xr
  , convexity lnext l r < 0 = lowerTangent xls xr
  -- optimum reached
  | otherwise = (xl, xr)

{- to merge two hulls, compute their common upper tangent
 - and lower tangent, then reconstruct the resulting hull
 -}
```

```haskell
mergeHulls :: Hull -> Hull -> Hull
mergeHulls (Hull (Cap _ lowerRL) (Cap _ upperRL))
           (Hull (Cap lowerLR _) (Cap upperLR _)) =
  -- compute upper/lower tangents
  let (lowerL, lowerR) = lowerTangent lowerRL lowerLR in
  let (upperL, upperR) = upperTangent upperRL upperLR in
  -- combine into a `Hull` instance
  let lower = (reverse lowerL) ++ lowerR in
  let upper = (reverse upperL) ++ upperR in
  Hull (Cap lower $ reverse lower) (Cap upper $ reverse upper)

{- recursively partition the input array; call naive
 - convex hull algorithm on leaves; merge left and right
 - hulls at internal nodes
 -}
convexHullHelper :: [Point] -> Hull
convexHullHelper xs =
  if length xs <= 200
  then convexHullNaive $ quicksort xs
  else -- partition list of points into left/right halves
       let pivot = head xs in
       let (ls, _, rs) = partition xs pivot [] [] [] in

       -- compute the convex hulls of the two halves in parallel
       let lhull = convexHullHelper ls in
       let rhull = convexHullHelper rs in
       let parRes = do lhull' <- rpar (force lhull)
                       rhull' <- rpar (force rhull)
                       _ <- rseq lhull'
                       _ <- rseq rhull'
                       return (lhull', rhull') in
       let (lhull', rhull') = runEval $ parRes in

       -- merge the left hull, the pivot and the right hull
       lhull' `mergeHulls` (Hull (Cap [pivot] [pivot]) (Cap [pivot] [pivot]))
              `mergeHulls` rhull'

{- very simple quicksort method used in the leaves of
 - the helper above; for some reason, this is faster
 - than the library List.sort
 -}
quicksort :: (NFData a, Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x : xs) =
  let l = quicksort [y | y <- xs, y <= x] in
  let r = quicksort [y | y <- xs, y > x] in
  l ++ [x] ++ r

{- partition function used in the convexHullHelper;
 - splits a list into elements less than, equal to,
 - and greater than a given pivot
 -}
partition :: Ord a => [a] -> a -> [a] -> [a] -> [a] -> ([a], [a], [a])
partition [] _ lt eq gt = (lt, eq, gt)
partition (x : xs) pivot lt eq gt =
  case x `compare` pivot of
      LT -> partition xs pivot (x : lt) eq gt
      EQ -> partition xs pivot lt (x : eq) gt
      GT -> partition xs pivot lt eq (x : gt)

{- convert a Hull to list-of-point representation
 - since the upper/lower hulls have two common endpoints,
 - we have to remove these duplicates before concatenating
 - their associated lists; because of Haskell laziness,
 - the use of `init` and `++` should not decrease the
 - performance by a lot
 -}
toList :: Hull -> [Point]
```

```haskell
toList (Hull (Cap lowerLR _) (Cap _ upperRL)) =
  (init lowerLR) ++ (init upperRL)

{- entry point into the convex-hull library
 - calls the underlying helper method and converts the output to
 - a list-of-point format
 -}
convexHull :: [Point] -> [Point]
convexHull xs =
  toList $ convexHullHelper xs
```

app/Lib.hs

```haskell
module Lib
    (convexHull
    ) where

import qualified Data.List as List
import Types (Point(..))

{- get the convexity of an ordered triplet of points
 - by computing a quantity proportional to the signed
 - area of the triangle formed by these points
 - this is > 0 when convex, < 0 when concave and
 - = 0 for collinear points
 -}
convexity :: Point -> Point -> Point -> Double
convexity (Point ax ay)
          (Point bx by)
          (Point cx cy) = (ax * by + bx * cy + cx * ay) -
                          (ax * cy + bx * ay + cx * by)

{- find the upper hull of an xy-sorted set of points
 - maintain a stack and enforce convexity at insertion
 -}
upperHull :: [Point] -> [Point]
upperHull xs = List.foldl insertConvex [] xs
  where insertConvex :: [Point] -> Point -> [Point]
        insertConvex stack@(a : pop@(b : _)) p =
          if convexity b a p < 0
          then p : stack
          else insertConvex pop p
        insertConvex stack p =
          p : stack

{- find the lower hull of an xy-sorted set of points
 - maintain a stack and enforce cocavity at insertion
 -}
lowerHull :: [Point] -> [Point]
lowerHull xs = List.foldl insertConcav [] xs
  where insertConcav :: [Point] -> Point -> [Point]
        insertConcav stack@(a : pop@(b : _)) p =
          if convexity b a p > 0
          then p : stack
          else insertConcav pop p
        insertConcav stack p =
          p : stack

{- compute the lower and upper hull and concatenate
 - them; the two hulls have two common endpoints, so
 - remove these from the upper hull before appending
 -}
convexHull :: [Point] -> [Point]
convexHull xs =
  let xsSorted = quicksort xs in
  let lower = lowerHull xsSorted in
  let upper = upperHull xsSorted in
```

```
  combine lower upper
  where combine :: [Point] -> [Point] -> [Point]
        combine as bs =
          let ta = List.reverse .tail $ as in
          let tb = init bs in
          ta ++ tb

quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x : xs) =
  let l = quicksort [y | y <- xs, y <= x] in
  let r = quicksort [y | y <- xs, y > x] in
  l ++ [x] ++ r
```

References

[1] Miller Russ Chen, Weiyang. Parallel Implementation of the Convex Hull Problem. https://cse.buffalo.edu/faculty/miller/Courses/ CSE633/Weiyang-Chen-Spring-2020.pdf.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[3] Petr Felkel. Convex Hulls. https://cw.fel.cvut.cz/b181/_media/courses/ cg/lectures/04-convexhull.pdf.