PARALLEL CONVEX HULL ALGORITHMS

A PREPRINT

Matthew Molter Cockrell School of Engineering
University of Texas at Austin
Austin, TX
mmolter@utexas.edu

Joaquin Ambia Garrido
Cockrell School of Engineering
University of Texas at Austin
Austin, TX
ambia@utexas.edu

December 4, 2020

ABSTRACT

We have implemented, and compared two algorithms to find the convex hull of a set of points. Each of the two algorithms is representative of the two main tendencies to tackle this problem: *divide-and-conquer* and *iterative sequential*. In general we see a time reduction when more threads are used, only when the number of points is larger than a threshold (around 20,000), and only up to around 4 threads. Computational time can be cut by more than half.

1 Introduction

The convex hull algorithm has been fairly extensively examined in the literature dating back at least to the 1970s. The premise behind the problem is that given an arbitrary set of points, the convex hull is the smallest convex set that contains it. This means that given any two points within the hull, the entirety of the line that connects them is contained within the hull. It is an interesting problem in geometry with applications in numerous spaces, such as image processing, scientific computing (in particular quantum mechanics), and economic analysis.

It has been noted that despite the large number of serial algorithms for the computation of the convex hull, there are a surprisingly small number of parallel algorithms. Most parallel algorithms can be divided into one of two categories:

1. divide and conquer, where numerous smaller hulls are found and then combined into a single hull, as examined in [1] and [2]
2. parallelization of iterative sequential algorithms, as examined in [3] and [4]

We implement an example of each of these algorithms, and compare them to each other, as well as to the serial algorithm which will be presented below.

2 Serial Algorithm

In the serial algorithm, a stack is used to detect concavity and remove points from the hull. There is only one assumption made by this algorithm: that the points have already been sorted in increasing order by x-value, or in some cases by angle with respect to an arbitrary direction, and the center of all points.

2.1 TransparentStack

The stack used is for the most part one that the reader will be familiar with, but it has one addition that is worth mentioning for clarity ahead of time. The sole addition is that besides the familiar $push/pop/peek$ methods normally performed by a stack, this stack has an additional $second$ method, which will return not the top value, but the value underneath, hence the name $TransparentStack$. This is critical for allowing us to determine concavity in a simple fashion.

Our stack is implemented using the $ArrayList$ class in Java. This allows for push/pop/peek/second to execute in $O(1)$ time. The implementation is of the standard type that is easily found in text books and online.

2.2 Hull Algorithm

The algorithm begins by finding the left most and right most points, a trivial exercise that can be completed in $O(1)$ time since the points have already been sorted by their horizontal position. It then effectively draws a line between the two points, and points above that line are put into the group for what will be referred to as the $upperhull$ and points below that line are moved into the $lowerhull$. Once the upper/lower sets have been created, we can begin the algorithm to calculate the hull. The method is effectively the same for the upper and lower sets, so we will discuss only the upper hull algorithm in detail for brevity.

To begin with, the first two elements are pushed onto the stack. Once there are two elements, we can proceed with comparison. The next element in line to be pushed to the stack is taken with the $second$ element in order to form a virtual line. This is done simply by calculating the slope and the y-intercept point with the formula $slope = (y_2 - y_1)/(x_2 - x_1)$, and $intercept = y_1 - slope * x_1$, that many will remember from their introductory algebra course.

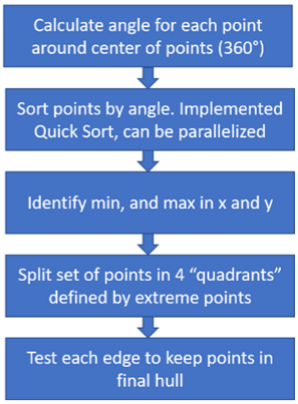
We then examine the top element in relation to this line, in our algorithm, via projection. This is done by finding the y value using the formula $y = slope * x + intercept$. If the y value of that point is greater than the y value of its projection, then it is above the line; if it is equal or lesser, then it is considered below the line. If the top element in the stack is above this line, then the surface formed by the three elements in comparison is convex. This can be easily visualized by drawing the resulting figure, examples of which can be seen in figure 1. This means that there is no action to be performed, so the next element is pushed onto the stack and the comparison repeated.

If the top element happens to be on or below the line formed by the $second$ element and the next element, then this is evidence of a concave surface. In order to remedy this, the top element is popped from the stack before pushing the next element on to the stack. Again, the comparison is then repeated until all points have been evaluated.

This algorithm ends when we reach the furthest most right point of the set, which means that the entire set has been evaluated since it is sorted. Once that occurs, the hull is considered found. As stated above, the lower hull algorithm proceeds in a nearly identical manner, just with opposite orientation. In the serial algorithm, the lower hull is found after the upper hull has finished. Each hull is found in $O(n)$ time. This should be obvious, as each element can be at most pushed once, and at most popped once. Once both hulls have been found, the connecting point is the left/right extremities, so combination can occur in $O(n)$ time as well if copying elements.

A further improvement to this algorithm called $Quarter - Hull$ was proposed by Liu, in which instead of dividing the points in 2 sets, they are divided in 4, separated by all 4 extremes in the x, and y direction. Then the sorting is done in the angle, and the iterative process to determine concavity just follows the original algorithm. This modification allows for further parallelization.

It is important to notice that these algorithms do not require any further merger after the hulls have been calculated, since concavity will be satisfied locally, and the 4 extreme points are always part of the final hull.



As can be seen above, given a sorted set of elements, a convex hull in $O(n)$ time is trivial. However, sorting is a $O(nlog(n)$ problem at best, which dominates the run time on large datasets. As a result, comparisons will be made to the parallel algorithm only in the portion in which the hull is actually calculated. Comparison of sorting algorithms is beyond the scope of this paper.

3 Parallelization Approach - Divide-and-Conquer

The divide-and-conquer method is a relatively straightforward approach to parallelization, and likely the first idea that one would have if they were assigned the task of making a parallel algorithm for convex hull computation. Our approach is found in [1] and [2] in slightly different presentations. [1] gives the method by dividing the upper/lower hull into horizontal subsets, which is the exact method which we implemented, and is described below.

3.1 Finding Sub Hulls

Given a set S of an arbitrary number of points, and k processors, S is divided into n equal sets. The upper and lower hulls of each set are found, and then the hulls are connected to each other in order to find a complete upper and lower hull of the entire set. The upper and lower hulls are then combined in much the same manner as in the serial algorithm. Again, the upper/lower hull algorithms are nearly identical, so we will describe only the upper hull algorithm for brevity.

To begin with, the set of points is divided among its extreme edges into upper and lower sets, exactly as in the serial algorithm in $O(n)$ time. Once the upper and lower sets have been found, we can begin breaking them up into k disjoint subsets. This allows each to be handled with no interference. Since the set has already been ordered before beginning the algorithm, the split is trivial and can be done in place in $O(1)$ time, or in $O(n)$ time if copying. This allows a series of neighboring hulls to be found.

3.2 Combining Sub Hulls

Once the neighboring hulls have been found, the trick is to combine them in a manner that takes $O(n)$ time or better. Luckily, there is a fairly simple approach which can be used that combines the sub hulls in $O(nlog(n))$ time, which will be presented below.

In order to connect two hulls, a common tangent must be found. When connecting two hulls, there must be one point on each hull that is used to connect the two. The two points must form a line under which all other points in each hull fall for them to be considered within the hull. This is trivial to find generally, as the hulls should be horizontally disjoint. However, for the resulting hull to remain convex, an additional consideration must be made: that the line connecting the two hulls is a common tangent. Failing to take this into consideration, as shown in figure 1a, will create a hull that is not convex.

If two points, one on each hull, form a common tangent, then this means that 1) all points between them are contained underneath the tangent line, and 2) by the definition of convexity, a line is always convex. Again, this is easy to visualize as seen in figure 1b.

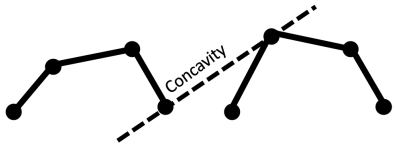


Figure 1a

Figure 1b

The question remains, what is a simple way to find this common tangent that is also efficient? Our approach is taken from [1]. First, we must describe how to determine if a line is a tangent. We will assume that there are two sub hulls that have already been found, which we will refer to as p and q. We have two points p_i and q_j, one one each hull, and want to determine if the line connecting them is a tangent to p_i. For the upper hull, the determination is as follows (and is reversed for the lower hull):

1. Examine $p_i - 1$, the point to the left of the point of interest on the hull. If it is below the line connecting p_i and q_j, then continue. If it is above, the line is clearly not a tangent as it intersects the hull at more than on point.
2. Examine $p_i + 1$. If it is below, then the line connecting p_i and q_j is a tangent to p_i. If it is above, then the line is not a tangent as it intersects the hull at more than one point.

The same method is used to determine if a point is tangent to q_j. Now we must determine how to move from points that do not form a tangent to points that do form a tangent. This is actually a straightforward process. If after 1), $p_i - 1$ is known to be above the line connecting p_i and q_j, then the tangent point lies to the left of p_i, and to the right if after 2), $p_i + 1$ is known to be above the connecting line. The amount moved is determined by the size of the set and the

iteration. For a set of size s, the first iteration would move $s/4$, the second $s/8$ and so on until the tangent points have been found. The same process is done for q_j, and once a line has been found which satisfies the tangent condition on both p_i and q_j, then a common tangent line has been found.

Once this common tangent line has been found, the two hulls can be easily connected. All points with x-values equal to or between the two tangent points can be removed, and the remaining points are the entirety of the convex hull which contains both sub hulls. This leaves both the finished upper and lower hulls, however they are still separated and not a complete convex hull for the set of planar points.

Luckily, the combination of the hulls is simple. Much like the method we used to separate the original set in the first place, we only need to know the extreme horizontal points, which are shared between the upper/lower hulls. These can be found in $O(1)$ time since our points are sorted by horizontal position. In our algorithm, we take the upper hull, and append the lower hull minus these common points to it, which is a $O(n)$ operation.

4 Results

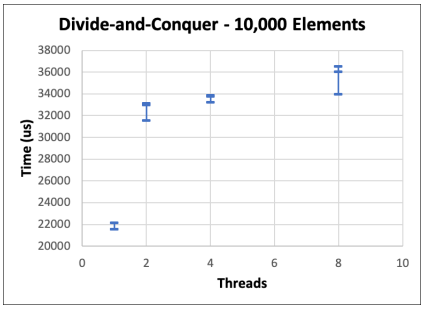
For testing both algorithms, we ran 3 separate tests of the same type, but with differing numbers of elements. First was a small set of 10,000 points, second a larger set of 100,000 and finally a set of 200,000 elements. The elements were randomly generated in each test with x and y values between a given range. In our tests, this was -1,000,000 to 1,000,000. The inputs were presorted using built in Java methods by x-value.

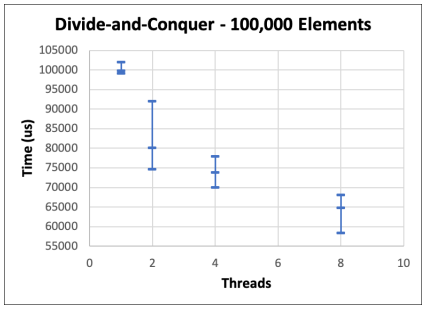
The divide-and-conquer approach was tested on a 2019 MacBook Pro with an 8-core Intel processor running at 2.2GHz, while the iterative sequential was a Windows machine with an intel i7-7700k processor. Tests were limited to 2, 4, and 8 cores for two reasons. One, we did not care to test on an oversubscribed processor, as this would obviously be at best equal, and potentially significantly worse, when compared to a test with 8 threads since portions of the input would be broken up, but then worked on essentially in serial. The second was that eliminating threads that were not equal to a power of 2 made it simpler to test, and we did not have to account for cases where there are an odd number of threads.

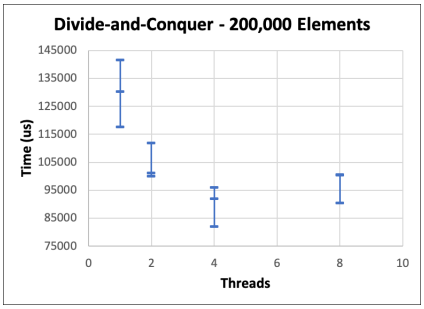
Finally, times were taken immediately before and after running the ConvexHull algorithm using Java's built in nanoTime function, and the delta between those two is used as our run time. Each test was run three times, with all three results displayed in the charts to follow. As can be seen in the first test with 10,000 points, the sequential method with a single processor is actually the fastest algorithm. Our initial belief was that this is due to the increased overhead from the parallel algorithm, and that small datasets may not benefit from parallelization since the overhead costs are more significant than the savings in hull calculation time.

The second test with 100,000 points shows behavior more inline with what one would expect from a parallel algorithm vs a serial algorithm. The serial method clearly takes the longest, with a strong improvement with 2 threads, and a smaller but noticeable improvement with 4 and 8 threads. This is in line with our prior stated belief that there is an increase in overhead that is only overcome with larger datasets.

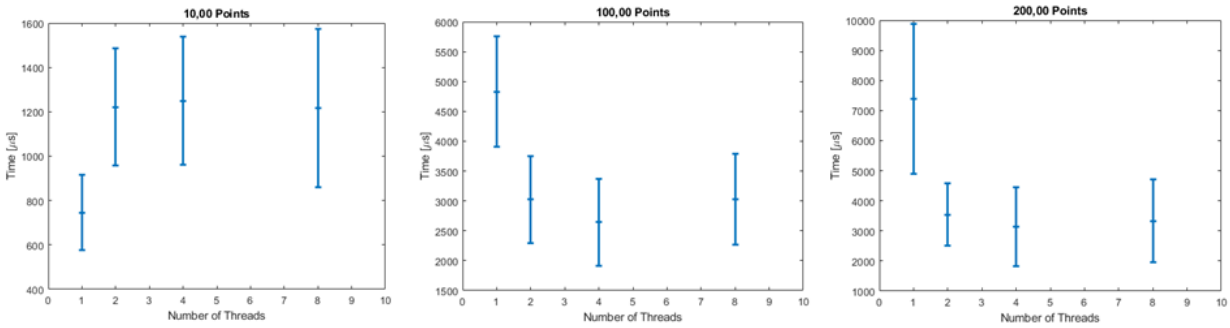
The third test with 200,000 points is similar to the 100,000 point test, with one notable exception. Again, the serial algorithm is clearly the slowest, with a noticeable improvement in the 2 thread test. The 4 thread test continues to show improvement, but the 8 thread test actually started to take longer. We have not isolated the cause of this, but have two potential causes. First, we believe there may be an oversubscription issue with background processes on our computer, leading to wait times on certain threads, and second, there may be a bug or error in implementation.







The same test were run for the iterative sequential algorithm, but the exercise was done 10 times. In the following plots we display the median, and the standard deviation of the tests



One additional item of note for the divide-and-conquer algorithm. On additional testing run after our initial data collection, we showed occasional outliers in performance for set ups with 2 or greater threads. On examination, this appears to be due largely to the final distribution of points in the set, and how the sets are broken into upper/lower sets. Depending on where the extreme horizontal points land, this can lead to a severe imbalance in the upper/lower set size, which leads to less effective parallelization. This leads us to the conclusion that the division method used was sub optimal, and we may need to look into a better method for dividing the sets into two vertical components. Also, a different choice of test data, such as one in which the data lays within a disc rather than a square distribution would also likely eliminate this anomaly.

5 Conclusion and Future Works

In conclusion, the costs of parallelization must be carefully weighed when selecting a convex hull algorithm. As can be seen clearly above, the dataset must be of a reasonably large size, otherwise the single threaded algorithm is clearly superior. This is consistent with our prior understanding of parallelizing algorithms, but serves as an important reminder.

As far as improvements and future work, an obvious improvement would be to add support and testing to both parallel algorithms for odd numbers of cores. This should be a trivial addition, but we did not have the time to implement and properly test a solution.

Also, as previously mentioned, an immediate area for consistency improvement to the divide-and-conquer algorithm would be to improve our vertical division method. Our current method tends to average fairly similar sizes, with most splits being around 1:2 to 2:3, but occasionally will split sets into ratios as bad as 9:1. A more effective split would look at vertical distribution and split among extreme points within some range of the average. This would increase overhead at the beginning if done incorrectly, but if we were to create our own sort function, and then keep track of max/min y-values during the x-value sort, we would likely be able to implement a more consistent split function. The other option would be to make some sort of assumption about distribution of the data, such as we know for a fact that our distribution should be centered around 0 with the function we used for our test set creation, so we could split positive/negative values into upper/lower sets respectively with what would likely be a very good consistency.

Lastly, there is the issue of 8 thread performance at 200,000 points for the divide-and-conquer method. We would need to look into this more to validate one of our theories about the performance issue.

6 References

References

[1] M. Nakagawa, D. Man, Y. Ito, and K. Nakano, “A Simple Parallel Convex Hulls Algorithm for Sorted Points and the Performance Evaluation on the Multicore Processors,” Hiroshima University. [Online]. [Accessed: 11-Nov-2020].

[2] J. Ramesh and S. Suresha, “Convex Hull - Parallel and Distributed Algorithms,” Stanford.edu, 2016. [Online]. [Accessed: 11-Nov-2020].

[3] R. Miller and Q. F. Stout, “Efficient parallel convex hull algorithms,” IEEE Transactions on Computers, vol. 37, no. 12, pp. 1605–1618, 1988.

[4] J. Liu, “Parallel Algorithms for Constructing Convex Hulls,” dissertation, LSU Historical Dissertations and Theses, Baton Rouge, LA, 1995.