Planar Convex Hull Benchmarking in Parallel

Jay Idema
Rensselaer Polytechnic Institute
New York, USA
idemaj@rpi.edu

Gregory Saini
Rensselaer Polytechnic Institute
New York, USA
sainig@rpi.edu

Lucas Standaert
Rensselaer Polytechnic Institute
New York, USA
standl@rpi.edu

Chris Vanderloo
Rensselaer Polytechnic Institute
New York, USA
vandec7@rpi.edu

Abstract

In this paper, we explore existing convex hull algorithms, including Quicksort, Chan's Algorithm, and Monotone Chain. We characterize the performance of each of these algorithms in parallel in both strong and weak scaling when using massive (upwards of 2^{24} point) datasets, in addition to communications overhead. We arrive at an understanding of how divide-and-conquer approaches differ in real-world performance from algorithms that use more linear approaches.

1 Introduction

Being able to quickly find convex hulls has become more important in the last few decades. It is a fundamental problem of computational geometry, and finds applications in many fields, such as mathematics, statistics, combinatorial optimization, biology, and others. Being able to find convex hulls over large sets of data is important for all these fields, and luckily a lot of work has gone into finding efficient algorithms that solve for the convex hull. There are many, but we stuck to exploring three in this paper: Chan's Algorithm, Monotone Chain, and Quickhull.

A convex hull over a set of points S is equal to the intersection of all convex sets over that set of points S. Visually, and with a set of points in two dimensions, if you were to wrap a rubber band around all the points, the convex set of points would be touching the rubber band. We explored a dataset of randomly generated points in a 2 dimensional Euclidean space. The problem can be adapted to higher dimensions as necessary.

In this paper, we explore the effect of introducing high-performance computing to these algorithms in order to find convex hulls over enormous data sets. We evaluate the strong and weak scaling of each algorithm and parallelization approach. Additionally, we consider the communications overhead for each algorithm to better characterize how much effort is going into computation and which algorithms have bottlenecks when working in parallel. We determine which algorithms have varying performance based on the dataset, and which algorithms will give more consistent results on average.

2 Related Works

Previous works developed a variety of convex hull algorithms that we have to explore today. Some of the earliest convex hull algorithms were developed in the 1970s as complexity and the use cases of finding large hulls became a more interesting problem. In the 1972 "Efficient parallel convex hull algorithms" paper, it lays out the problem of solving for convex hulls efficiently, and states that it is not possible to solve for the hull faster than sorting, effectively bounding convex hull algorithms at a runtime of $O(n \log n)$ [14]. As we see later, many algorithms attempt to beat that runtime by using output-dependence. Deterministic and random methods were created to solve for the convex hull, such as solving for a 3D convex hull on a multiprocessing system in an asymptotically optimal $O(n \log np)$ time [6]. An algorithm based on the quicksort sorting mechanism, known as Quickhull, was later introduced. It was abstracted to higher dimensions by use of the Beneath-Beyond Algorithm [4]. Another algorithm, Graham's Scan, operates in $O(n \log n)$ time by searching through the dataset, consistently "rotating" through points that could be added to the convex hull until it can't find any more points [8]. An algorithm like Monotone Chain or Chan's makes use of sorting algorithms, such as SampleSort, which utilizes oversampling techniques to attempt to mitigate problems when doing any sort of sorting with a divide-and-conquer approach from a non-uniform data distribution [7]. In this paper, we use SampleSort with Monotone Chain and a oversampling value of 64 to get the most optimal performance. In 1996, Timothy Chan laid out a series of algorithms that were considered output-sensitive, in both the 2nd and 3rd dimensions. The article noted algorithms such as Graham Scan and Jarvis March, which had time complexities of $O(n \log n)$ and $O(nh)$. These algorithms would then see a series of improvements until obtaining an optimal time of $O(n \log h)$, where h is the size of the end convex hull. This was done by pre-processing a set of points before passing them to the gift-wrapping method, [5]. Another algorithm that achieves that time complexity is the aptly-named "Ultimate Planar Convex Hull Algorithm", which is a divide-and-conquer algorithm that uses a "marriage-before-conquest" modified

approach. It reverses the approach to search for the dividing lines between the divided sets, rather than adding to the hull and recursing. This algorithm is part of the class of output-sensitive algorithms, where the runtime depends on the resulting size of the hull, but is asymptotically better than algorithms with a $O(n \log n)$ runtime [11]. Chan's is similar in that it's runtime approaches $O(n \log h)$ if the selected m (number of subsets) is approximately equal to h. An assumption we made and explore is whether asymptotically better translates to real-world improvements in speed on these large datasets. As shown in a response paper to the "Ultimate Planar Convex Hull Algorithm", the algorithms that reach $O(n \log h)$ time may work well theoretically, but fall short in real-world use cases to simpler algorithms like Quickhull [13].

As we said before, convex hulls find applications in a variety of fields like mathematics, statistics, combinatorial optimization, and is useful for simulations and problem solving. Some papers adapt to higher levels, such as solving for convex layers instead of simply the outer convex hull [16]. In May of 2020, Möls, LI, and Hanebeck designed a plane extraction algorithm that utilized spherical convex hulls [15]. We noted real-world applications and use of convex hull algorithms in the fields of robotics. For example, in one paper, an array of Light Emitting Diodes are used in an environment, and the minimum bounded circle algorithm is used to draw a virtual circle from the information collected by the sensors [1]. Here, Chan's Algorithm is used to map the surrounding points, and provides a barrier for the robot to understand.

As with most algorithms, convex hulls benefitted from the development of extremely parallel computations. In 1988, Miller, Russ and Sout, and Quentin F. presented a simple convex hull algorithm that could be done in parallel, utilizing clockwise and anticlockwise warps [14]. In 1994, Amato, Nancy M and Goodrich, Michale T and Ramos, and Edgar A discussed a randomized parallel algorithm for the convex hull problem in the 3rd dimension. Through their work, they found a sequential algorithm that had an overall time complexity of $O(n \log_3 n)$ [2]. Work in the 3rd dimension would continue in 1995 with Dehne, Frank and Deng, Xiatie and Dymond, Patrick and Fabri, Andreas and Khokhar, and Ashfaq Q, who designed another randomized parallel algorithm [6]. In their case, the overall time complexity came out to be $O(n \log np + n; p)$, where n and p are the local memory per processor, and gamma np is time complexity of one phase. Other parallelization techniques such as the Concurrent Hull algorithm make use of divide and conquer approaches, where each parallel process will find a hull and then merge the end results at the end of the run [12].

Later implementations would begin to make use of GPUs and the CUDA programming model designed by NVIDIA, continuing into higher-dimensional spaces. In 2012, Stein, Ayal and Geva, Eran and El-Sana, and Jihad designed CudaHull, a parallel algorithm based upon QuickHull. They found that the algorithm was far faster than the existing CPU-based implementation [18]. In addition to this, Tang, Shao, Tong and Machoa created an algorithm that utilized both the GPU and CPU. In their implementation, the GPU removes boundary points and the convex hull is then calculated by the CPU. Their implementation saw speedups of 27 times on static point sets, and 46 times on deformed point sets [19].

The previous work shows all the possible implementations and use cases of convex hull. While others have explored speedups with CUDA and GPU, we attempt to classify the difference between the "idealized" $O(n \log h)$ algorithms and compare them against other $O(n \log n)$ algorithms when run in parallel.

3 Convex Hull Algorithms

3.1 Quickhull

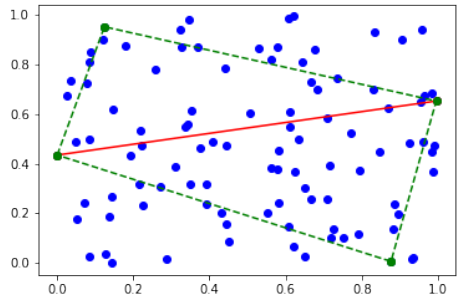

Figure 1. First Step of Quickhull

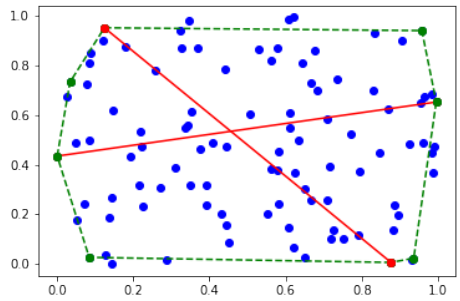

Figure 2. Second Step of Quickhull

3.1.1 Description. The Quickhull algorithm is a recursive, divide-and-conquer algorithm derived in name from quicksort and has an expected runtime of $O(n \log(n))$ [4]. Quickhull starts by finding the farthest left and farthest right points in the set to create a dividing line which defines the top and bottom subhulls. This is denoted in Figure 1 by the

solid red line between the farthest left and right points. The algorithm partitions the points between the top and bottom of this dividing line, and recurses on each partition. Given the left and right points, the recursive procedure finds the point furthest from the dividing line and creates a triangle with the provided left and right points (as seen in Figure 1). The distance of a point from the divinding line is defined as the length of a perpendicular line segment that starts at the line and ends at the point in question. Given points L, R, and p, the distance of p from the line formed by L and R is represented by the equation:

$$\frac{||(R-L)\times(L-p)||}{||R-L||}$$

Through a series of two partitions based on the new edges, the points inside the triangle are discarded, and two new sets with bottom dividing lines are created. The second recursion level of this process can be observed in Figure 2, where we have found 8 points to add to the hull so far and are divided into 4 partitions. The algorithm recurses and completes the same procedure on each set continuously until there are no new points left to consider.

3.1.2 Parallelization. Being a divide-and-conquer algorithm, a possibility for parallelization would be to simply assign a node every time it recurses down to another level. However, that could be considered "embarrassingly parallel" and would not be helpful to our experiments, which assume the computational effort is evenly divided over the algorithm. So, the program evenly divides the points amongst all available processors in linear time. To do this quickly, we don't make any assumptions about the ordering of points (or sort them) and simply split them up amongst the available nodes randomly. This means each node will end up running all of the same procedures as the others, just over a different partition of the dataset. We run the recursive procedure in an iterative fashion through unrolling [17]. This reduces control flow overhead and allows for better insight to where MPI communications should occur. Consider, for example, a random set of points split between a number of MPI ranks. A farthest left and farthest right point could be determined on each processor, and then they can communicate over MPI to determine which points are the global left and right points. Through each recursive pass (as described above), the farthest point from the previous barriers is agreed upon through a custom MPI reduction. This creates a method for determining the convex hull in a number of communications proportional to the size of the final convex hull, and while evenly splitting computational efforts. This is the method that we used during our experiments.

So let's consider the unrolled procedure. After the first split, we search for a "farthest point" in two subhulls. Every time that point is found, we double the number of subhulls we're searching over. First 2, then 4, 8, 16, etc. Because we're doing this calculation amongst a number of processes, there may be nodes in a subhull on a particular process, while another process has no nodes in the same subhull. Using MPI, the processes communicate with each other as to whether they need to do another split of all remaining subhulls. If so, they split, and the processes have no remaining nodes submit a "NULL" response when reducing the global maximum point in each subhull.

By splitting the points evenly and considering the algorithm iteratively, with a number of MPI requests proportional to the size of the convex hull, this performs in expectation quite well. Due to the nature of splitting at the furthest point, it's expected that Quickhull will have recursive trees that go quite far, depending on the dataset. Quickhull may have good performance in expectation, but real-world performance that could be all over the place.

3.2 Monotone Chain

3.2.1 Description. The Monotone Chain algorithm for computing convex hulls uses a preliminary step of sorting the set of points, either by x component or by y component, with the opposite component used for tie-breaking [3]. In our implementation, we arbitrarily choose the x component to be the primary component of sorting. This sorted set of points is iterated over in order, and points are appended to vectors containing the pending upper and lower convex hulls, u_hull and l_hull respectively, such that u_hull bounds the data-set from the top and l_hull bounds the data-set from the bottom. An example of the upper and lower hulls found for a data-set by Monotone Chain can be seen in (Figure 3). During iteration over the sorted set of points, if a to-be-appended point causes a turn in the pending hull which is in the wrong direction (that is a "RIGHT" turn in the bottom hull and a "LEFT" turn in the top hull), the last last appended point is popped from the pending hull.

The Cross product equation

$$\begin{aligned}\text{Cross}(\text{O}, P_1, P_2) = (P_3.x - \text{O}.x) \times (P_2.y - \text{O}.y)\\ - (P_3.y - \text{O}.y) \times (P_2.x - \text{O}.x)\end{aligned}$$

is used to compute the direction of turn for three points in two-dimensions. A Cross value greater than zero representing a "RIGHT" turn and a value less than zero representing a "LEFT" turn. Once we've iterated through all the points, we reverse the bottom hull and remove the last element from each portion of the convex hull as these points contain respective duplicates in the other portion of the convex hull. A noteworthy property of the Monotone Chain algorithm is that it can only construct the convex hulls of a set of 2-dimensional points as opposed to the arbitrary number of dimensions in which quickhull can find a convex hull.

3.2.2 Parallelization. The second step of Monotone Chain is necessarily iterative; the sorted nature of points and in-order processing of the points is a requirement. This step

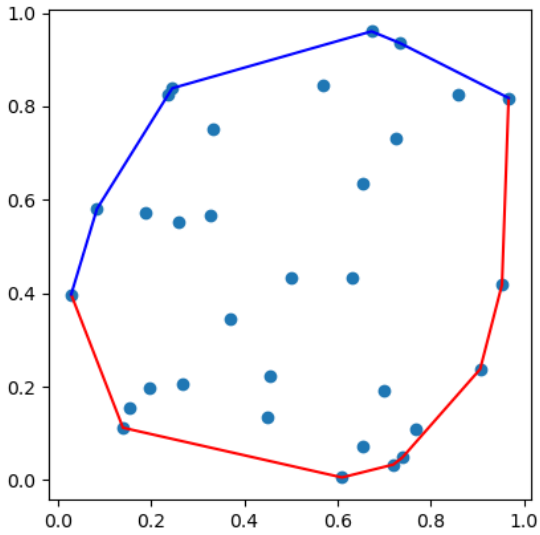

Figure 3. The upper and lower hulls of a 32 point data-set computed using Monotone Chain.

cannot be easily parallelized.
However, the initial step of the Monotone Chain algorithm, the sorting performed on points, can be parallelized as sorting lends itself well to sub-problems and divide-and-conquer algorithms. We choose Samplesort as the algorithm to implement for sorting our points. Samplesort is an extension of Quicksort which divides the data into a set of N bins based on a sorted set of $(N-1)$ pivots rather than 2 bins using a single pivot as Quicksort does [7]. Similarly to quicksort, all of given bin i's points have lower sorting priority than a bin $i-1$'s points and higher sorting priority than a bin $i+1$'s points.

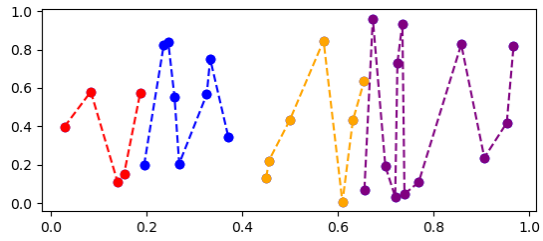

Figure 4. Order of processing points, left to right, for 32 points processed over 4 MPI ranks. Red points were owned and independently sorted by Rank 0, blue points were owned and sorted by Rank 1, yellow points by Rank 2, and purple points by Rank 3.

To decide on a set of pivots, at least $P * N$ points are randomly selected from the overarching set of points where N is the number of bins and P is a positive constant for oversampling. This subset of pivot points is then sorted and each P-th element is used as a true pivot to divide all points into bins. This algorithm lends itself well to parallelization as we can select a bin count equal to the number of processes, allowing each to receive and process its own bin. In our implementation, each processes reads in distinct, un-ordered, and equal-sized chunks of points from a file which they independently select P points to construct the shared pivots from. These P points are shared with all other processes using MPI_Allgather. All processes then independently sort the $P * N$ points and retrieve the true $(N-1)$ pivots. The runtime complexity of this is

$$O(P \times N \log(P \times N))$$

where N is the number of processes and P is the amount of oversampling. The final pivot set is constructed by taking every Pth element of the sorted $N \times P$ size subset.
Oversampling is a key component of quicksort and samplesort with the intended effect of more uniformly dividing points into bins. For the case of our parallelized algorithm, achieving an even distribution of points removes jitter and allows much faster speeds even at the cost of some time during this step. It is also noteworthy that the the runtime of this step increases with the number of processes.
Processes then place all their points into bins based on the pivot set in

$$O(N \log(P))$$

and send to each process the bin of index equal to the process's rank (e.g. the first bin of every process would be sent to the first-rank process with rank equal to 0) using a coordinated MPI_AllToAllV call. If pivots were selected ideally, each process should receive roughly the same number of points. In our tests, we used an oversampling constant of 64 points per rank in our benchmarking.
Once the bins are collected, they are sorted independently using qsort in

$$O(S \log(S))$$

on average, where S is the number of points per rank which, on average, will be the total number of points divided by the number of ranks.
Every process with a given rank "I" now has a sorted subset of points, which are all of higher sorting order than any of the points owned by the processes with rank less than "I" and are all of lower sorting order than any of the points owned by the processes with rank greater than "I".
A lead process, with the obvious choice being the first process with rank 0, can then perform the serial, iterative top and bottom hull construction step of Monotone Chain, requesting the sorted subsets of points from processes in order. The runtime complexity of this step is

$$O(N)$$

where N is the number of total points.
This step is purely iterative, the leading process does not need to request a process's sorted subset until it has processed all points from processes with rank lower than that process,

nor does it need to remember any previously points not stored in the pending hulls. This permits notable memory advantages as the loading process only needs to request points the next process in order and need not remember the points of previously requested processes.

3.3 Chan's Algorithm

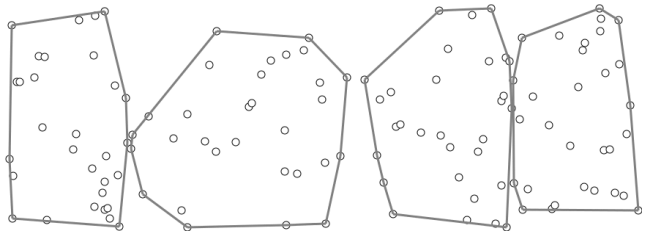

Figure 5. Subhulls Before Gift Wrapping [10]

3.3.1 Description. Chan's Algorithm is a two step algorithm to solve for the convex hull. It makes use of any $O(n\log n)$ algorithm to operate on a number of subsets. Then, it uses the Gift Wrapping algorithm (otherwise known as Jarvis March), which runs in $O(nh)$ time, where h is the resulting size of our convex hull[5][9]. In our implementation, we use Monotone Chain as the $O(n\log n)$ algorithm of choice. To start, Chan's Algorithm takes the points it is given and arbitrarily splits them into a series of $\lceil n/m \rceil$ different subsets. Monotone Chain then calculates the sub-hull before passing it off to the Gift Wrapping algorithm.

In the second stage of Chan's Algorithm, Gift Wrapping is used on the sub-hulls to compute the complete convex hull. At the start of Gift Wrapping, the points are sorted by x-value first. A second point is then chosen to calculate the minimum polar angle between the origin and a secondary point. At the end of each iteration, this minimum point is then set as the new origin and the process repeats until the algorithm hits the origin again. Given that there are M points on the complex hull, the algorithm will take at most $N(M+1)$ operations to find the convex hull [9].

Chan's Algorithm is considered an output-sensitive algorithm due to it's use of Gift Wrapping. As was previously stated, gift wrapping iterates through the hull searching for the minimum polar angle between points, setting this equal to the new origin [9]. If these points are preprocessed, the algorithm becomes much faster. In this scenario, there are far less points for Jarvis march to deal with drops the overall time complexity to

$$O(n(1 + \frac{h}{m}) \log m)$$

where n is the number of points, h the number of wrapping steps, and m the number of points per group [5]. In our implementation, we separate the points based upon the amount of MPI ranks. This means that as the number of ranks approaches h the algorithm starts to see an improvement in performance.

3.3.2 Parallelization. To parallelize Chan's Algorithm, we utilize our parallel implementation of Monotone Chain in combination with a parallel form of the Gift Wrapping algorithm. After splitting the points amongst the MPI ranks, each rank runs it's own Monotone Chain calculation. After the hulls have been computed, they are gathered in rank 0, the main process. This is done through the use of MPI_gather and MPI_gatherv. Once the preprocessed points have been gathered, they are scattered, using MPI_Bcast and MPI_scatter, to each rank. MPI_Bcast tells each rank how many points are being scattered, while MPI_scatter sends the points out. Any points that have not been scattered are collected in rank 0. At the beginning of Gift Wrapping, each rank finds its furthest left point and sends this back to the main process. After finding the furthest left point amongst them, it broadcasts this point and begins to iterate. During an iteration, each process finds its local best pivot for the next point in the hull and reports this back to the main process. These pivots are then compared at rank 0, where the correct pivot is decided on. This process repeats until the entire hull has been created.

4 Experiments

4.1 Benchmarking

For our experiments, we ran each algorithm on point sets ranging from 2^{24} points to 2^{30} points (multiplying by 2 each time). These points are randomly generated in a normal distribution with x and y coordinates between 0 and 1, stored as a 32-bit float. Each point set is run through each algorithm with MPI ranks ranging from 1 to 64, where the 64 MPI ranks are split between two nodes on AiMOS. For the purposes of these experiments, we did not implement each algorithm in CUDA, although we would expect a boost in performance by utilizing GPU technology. Instead, we're specifically investigating the MPI overhead with each technique and the impact of parallelization across multiple nodes.

4.2 Verifying Correctness

We confirmed our algorithms were implemented correctly by computing the convex hulls of sets of points of size 2048 for both a set of points bounded within a square with a lower left coordinate of (0,0) and an upper right coordinate of (1,1) and a set of points bounded within a circle of radius 2. Each algorithm was run over 32 MPI ranks for this test. The results of each of the algorithms matched the expected hull; with some variation in order, the points selected for the convex hull by each algorithm were identical and accurate. We constructed a python mat-plot graphing program to graph the data sets of points used and highlighted the points selected as members of the convex hull in red. The program takes

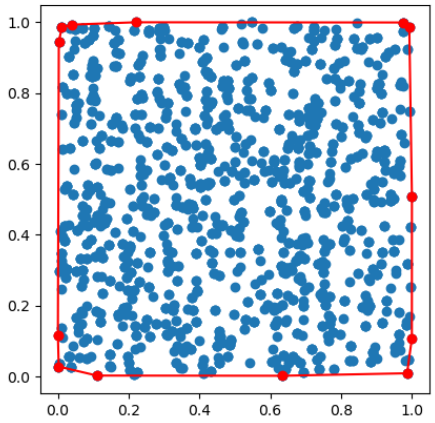

Figure 6. Convex Hull of Square Set of 2048 Points

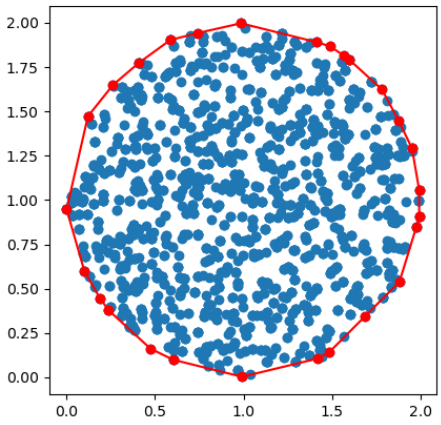

Figure 7. Convex Hull of Circle Set of 2048 Points

two plain-text files of line separated points, the first being all points and the second being the convex hull, and outputs a scatter plot with x and y as its axis.

5 Results

5.1 Quickhull

Quickhull shows off decent strong and weak scaling performance. With the same number of points per processor, generally speaking, performance stayed in the same ballpark. By increasing the number of processors over the same set of points, speed increases by a factor of 80 at the high end. All the experiments are shown in the performance graph. As expected, less nodes with the same number of points resulted in longer time to completion (Figure 8). Quickhull is an algorithm that parallelized well and was able to get speedups linear with the number of processors applied to the problem. However, the performance is more erratic because of the recursive nature of the algorithm.

5.2 Monotone Chain

Monotone Chain showed similar performance on lower ranks, but poorer scaling performance when compared to Quickhull and Chan's. This can be observed with an increase in processors realizing a higher compute time on the weak scaling result, indicating that massive parallelization was slowing down the algorithm. This resulted in strong scaling showing a linear speedup with exponential increase in processors (Figure 9). The cause of this is likely the linear time non-parallelizable component of the algorithm, which results in diminishing speedup for weak scaling.

5.3 Chan's Algorithm

Amongst the different algorithms tested, Chan's Algorithm saw the best performance. Chan's Algorithm had consistent improvement throughout the overall performance, weak scaling, and strong scaling. This included the highest threshold for speedups, incredibly strong weak scaling, and impressive performance boosts between ranks. As was mentioned earlier, Chan's Algorithm is considered an output sensitive algorithm due to the use of Gift Wrapping [5]. By splitting the points amongst the ranks, we slowly start to approach the optimal amounts of points per hull, based off the amount of vertices, which positively impacts performance.

6 Discussion

In this section we'll be discussing the performance of each convex hull algorithm in regards to overall performance, weak scaling, and strong scaling. We'll also explore the differences in MPI communications overhead.

We start by examining the overall performance. Examining the results for 1 node, 1 rank we can see similar completion time for each number of points, taking around 900 seconds to complete 2^{30} points. Differences start emerge as more processes are introduced. Chan's Algorithm sees marginal drops in compute time with vast improvements between the initial ranks, with compute of 2^{30} points only taking about 400 seconds with 2 ranks compared the serial results. A similar trend is seen in Quickhull, however the time to compute is far less consistent. Despite an apparent trend, compute times can be seen to jump between nodes in both directions. With 8 ranks, a jump in performance can be seen at 2^{29} points, while 4 ranks shows a drop in performance at the same number of points. This erratic behavior will be expanded upon in the weak scaling and strong scaling analysis, which show similar patterns. Monotone chain performs the worst, with the highest times to compute per number of points at each rank. Monotone chain also shows

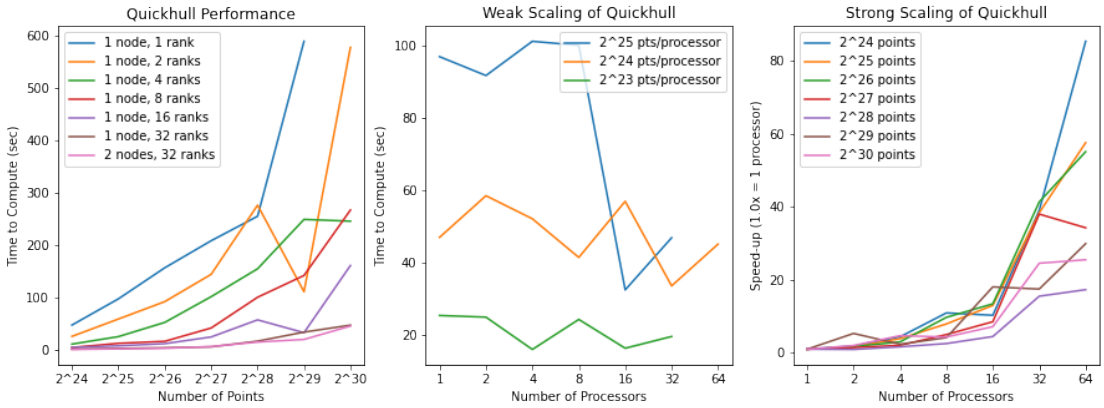

Figure 8. Performance Graphs for Quickhull

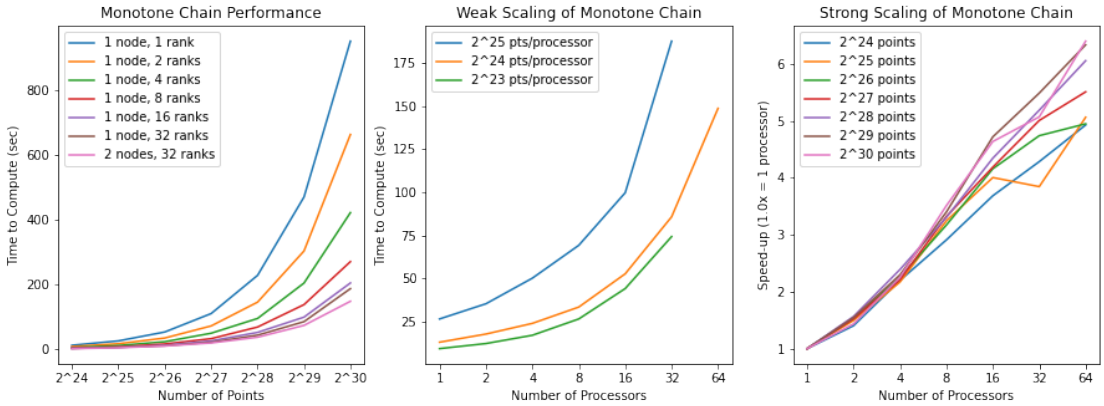

Figure 9. Performance Graphs for Monotone Chain

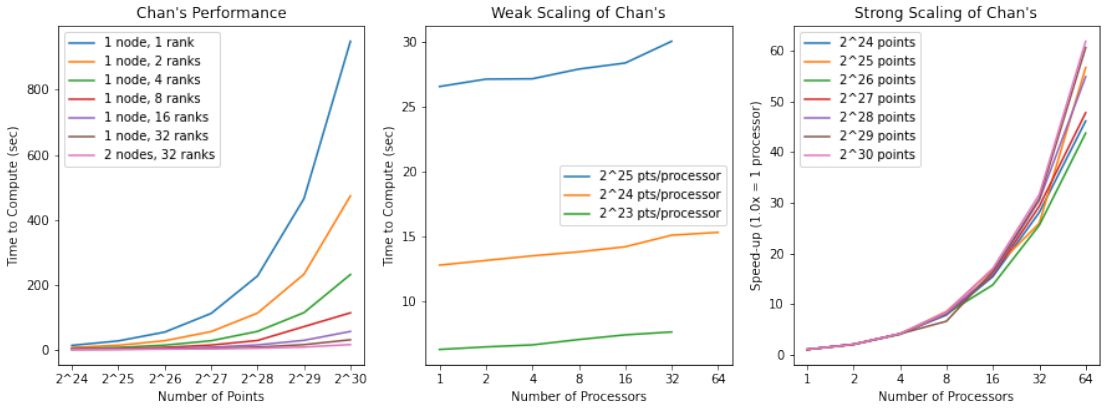

Figure 10. Performance Graphs for Chan's Algorithm

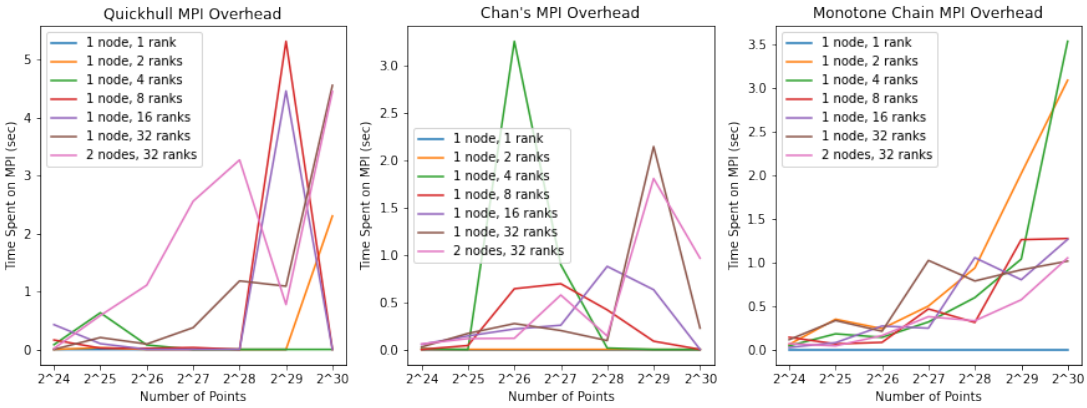

Figure 11. MPI Overhead Results for All Algorithms

the worst improvement between ranks, with very minimal gaps in performance at higher at processes counts.

The weak scaling analysis of the algorithms shows a similar pattern to the overall performance. Chan's shows the best results for weak scaling, with low compute times across the amount of processors and very little variance in time at each level, with the largest in performance being at the 2^{25} points per processor level. Quickhull is far more erratic. While the time to compute is overall less than Monotone Chain, the results are highly inconsistent. This mainly at the highest level of points per processor, where there the time to compute between 8 and 16 processors drops from 100 seconds to approximately 30 seconds. Monotone chain shows the worst performance in terms of weak scaling. Despite having similar results at 1 and 2 processors, the compute time grows exponentially as the number of processors increase at each amount of points per processor, taking approximately 175 seconds at 32 processors.

Our strong scaling analysis shows a slight difference in results compared to our weak scaling analysis and overall performance. Chan's Algorithm shows the best speedup per processor at higher amounts of points, with speedups of 50 to 60 times at 2^{28}, 2^{29}, and 2^{30} points. We also see an exponential curve appear for each set of points, further confirming Chan's strong performance. With Quickhull we see different results, with the best results at lower point counts. This can be seen with speedups of approximately 60 times at 2^{25} and 2^{26} points, and an 80 time speed up at 2^{24} points. However, we again see the erratic nature of the results, as the speedup tends to jump between the number of processors. While an upward trend is more apparent, we can see these jumps appear at different amounts of processors for different point counts. Monotone chain again performs the worst of the convex hull algorithms. Here we see a maximum speed up of only 6 times, compared to the 60 and 80 of Chan's Algorithm and Quickhull. In addition to this we also see a drop in speedup at 32 processors with 2^{25} points. Overall, the performance boost that can be seen is negligible compared to Chan's Algorithm and Quickhull.

The overhead results were collected by timing the blocking MPI functions and the waits following non-blocking MPI functions to get a picture of the communications overhead. In Figure 11, we see that Quickhull has erratic overheads in most scenarios, except for the two node case. In the two node case, there's a more consistent several second overhead cost, likely to do with the hardware split. Chan's algorithm is similarly erratic in overhead, but generally speaking, fairly low and not trending consistently upwards given high numbers of points. With Monotone Chain, the erractic pops in communication stop, as it has a more consistent rise in overhead with more points. We expect this trend would continue with Monotone Chain, leading to worse communication overhead over time. This could be an explanation for poorer scaling performance, as well.

From these results, we can see that Chan's Algorithm has the most potential as a parallel convex hull algorithm. Across the many tests we ran, Chan's showed consistent results in a downward trend, something that could not be replicated by neither Quickhull nor Monotone Chain. Chan's the saw best performance at high point levels and incredible weak and strong scaling. In addition to this, Chan's also has negligible slowdowns due to MPI overhead. While Quickhull does have low times and impressive strong scaling at lower point counts, the algorithm showed incredibly erratic and inconsistent results throughout the different test. Without a pattern to these inconsistencies, Quickhull may be dependent upon the point set itself, which could produce random drops in performance. Amongst the algorithms, it was clear that Monotone Chain performed the worst. Not only did Monotone Chain have the worst overall performance, weak scaling, and strong scaling, but also saw an upward tend in MPI overhead. This may potentially have lead to the slowdowns we

saw in our testing. From our testing, we can conclude that Chan's Algorithm would be the best fit due to it's impressive performance and consistency.

7 Conclusion

In this paper, we introduced 3 algorithms that attempt to solve the convex hull problem. Monotone Chain sorts points by their x and y values and calculates two separate hulls, an upper and lower hull. These are then combined to compute the complete convex hull. In Quickhull, a series of lines are recursively found that link the farthest points upon the current hull. This process repeats until the entire convex hull has been found. In the final algorithm, Chan's Algorithm, Monotone Chain is combined with Gift Wrapping to create a set of preprocessed points to speed up the Gift Wrapping algorithm. Our goal was to compare the results of parallel implementations each algorithm and seeing which would produce the best results.

Chan's Algorithm was by far one the best across the many different tests we ran. Chan's Algorithm saw the best overall performance at each level, with a consistent downward trend in compute time and a massive increase in strong scaling. In addition to this, the overall overhead due to MPI was negligible compared to the other algorithms. Monotone chain was clearly the worst choice due to it's higher compute times and far worse strong scaling, only reaching up a 6 times speed up. Monotone also saw the worst overhead for MPI making it even less appealing. Quickhull was a bit odd, as while it outperformed Monotone Chain, it was erratic in its overall performance, with drastic high lows at different numbers points and processors.

One aspect that may have affected our results was testing upon a single set of points. This may have been part of the reason for results seen in Quickhull, which was filled with inconsistencies. This also may have affected Chan's, as the time complexity of Chan's Algorithm changes as you introduce more and more sub-hulls[5].

For future testing, we would recommend utilizing a variety of different uniform and random points as well as expanding the amounts of points used. Classifying non-uniform point sets could help widen the gap between some algorithms, and testing with various shape dataset (perhaps where the convex hull is nearly the size of the original set) could better characterize which algorithms will consistently outperform in edge cases. Another useful test could be to use algorithms that require a "guess" for optimal search or sampling values, such as the parameter m for Chan's or the size of the oversampling set when using SampleSort inside a convex hull algorithm. Sweeping over more parameters and having a heuristic to determine more optimal parameters would be a useful direction to ensure high performance can be maintained with future datasets.

References

[1] Al-Forati, I. S. A. and Rashid, A. (2019). Design and implementation an indoor robot localization system using minimum bounded circle algorithm. In *2019 8th International Conference on Modeling Simulation and Applied Optimization (ICMSAO)*, pages 1–6.

[2] Amato, N. M., Goodrich, M. T., and Ramos, E. A. (1994). Parallel algorithms for higher-dimensional convex hulls. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 683–694. IEEE.

[3] Andrew, A. (1979). Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219.

[4] Barber, C. B., Dobkin, D. P., and Huhdanpaa, H. (1996). The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483.

[5] Chan, T. M. (1996). Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete and Computational Geometry*, 16(1):361–368.

[6] Dehne, F., Deng, X., Dymond, P., Fabri, A., and Khokhar, A. A. (1995). A randomized parallel 3d convex hull algorithm for coarse grained multicomputers. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 27–33.

[7] Frazer, W. D. and McKellar, A. C. (1970). Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM*, 17(3):496–507.

[8] Graham, R. L. (1972). An efficient algorithm for determining the convex hull of a finite planar set. *Info. Pro. Lett.*, 1:132–133.

[9] Jarvis, R. (1973). On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2(1):18–21.

[10] Ji, S. (2020). File:chanalgdemo.gif — wikimedia commons, the free media repository. [Online; accessed 2-May-2021].

[11] Kirkpatrick, D. G. and Seidel, R. (1986). The ultimate planar convex hull algorithm? *SIAM journal on computing*, 15(1):287–299.

[12] Masnadi, S. and LaViola, J. (2020). Concurrenthull: A fast parallel computing approach to the convex hull problem. In *ISVC*.

[13] McQUEEN, M. M. and Toussaint, G. T. (1985). On the ultimate convex hull algorithm in practice. *Pattern Recognition Letters*, 3(1):29–34.

[14] Miller, R. and Stout, Q. F. (1988). Efficient parallel convex hull algorithms. *IEEE transactions on Computers*, 37(12):1605–1618.

[15] Möls, H., Li, K., and Hanebeck, U. (2020). Highly parallelizable plane extraction for organized point clouds using spherical convex hulls.

[16] Rufai, R. A. and Richards, D. S. (2017). A simple convex layers algorithm.

[17] Rugina, R. and Rinard, M. C. (2000). Recursion unrolling for divide and conquer programs. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, LCPC '00, page 34–48, Berlin, Heidelberg. Springer-Verlag.

[18] Stein, A., Geva, E., and El-Sana, J. (2012). Cudahull: Fast parallel 3d convex hull on the gpu. *Computers & Graphics*, 36(4):265–271.

[19] Tang, M., Zhao, J.-y., Tong, R.-f., and Manocha, D. (2012). Gpu accelerated convex hull computation. *Computers & Graphics*, 36(5):498–506.

Appendices

A Running the Code

A `README.txt` is included with our code repository. This file details running the Points_Creator program to create uniform random datasets of arbitrary sizes, and describes how to run each algorithm in parallel using these generated points. All code is assumed to be run with the current working directory set to the `scratch` folder. The code will not work correctly in the `barn` directory.

B Team Contributions

Each team member contributed in a few ways.

- Jay wrote the Monotone Chain Algorithm code, the points generation code, and the python convex hull graphing code.
- Lucas wrote the Chan's Algorithm code.
- Chris wrote the Quickhull Algorithm code and created the results visualizations.
- Greg gathered all related works and citations to prepare our introduction and related work documents.