Concurrency Final Project: Parallel Convex Hull Algorithms

Henry Liu and Raymond Hong

May 6, 2021

The code for this project is available at https://github.com/henryliu5/convex-hull. This lab took us about 70 hours to complete.

1 Objectives

1.1 Background

Computing the convex hull of a set of points is a well-studied problem in computational geometry. The convex hull is the minimum number of points that completely enclose the others such that a convex polygon is formed. An intuitive way to think about it would be the shape a rubber band would make if stretched out and placed around a bunch of pegs.

Convex hulls have a wide range of applications ranging from statistics and optimizations to geometric modeling. For example, the simplex algorithm for solving linear programming problems iterates along the vertices of the convex hull. Other use cases include robot motion planning (the optimal path around obstacles is constrained by the convex hull) and applications such as the "magic wand" tool in Photoshop.

1.2 Goals

Our main goal was to investigate the parallelization of two efficient 2D convex hull algorithms, namely Quickhull and Chan's Algorithm, in Golang. Our objective was to see if we could get reasonable speedup across various test cases and also learn more about the Go programming model and runtime. Both algorithms are in theory faster than quadratic, with Quickhull being $O(n \log n)$ and Chan's Algorithm being $O(n \log h)$, where n is the number of input points and h is the number of points on the hull. We chose these two algorithms because their serial versions provide the best known asymptotic run times. Chan's Algorithm uses two other convex hull algorithms, Jarvis March ($O(nh)$) and Graham Scan ($O(n \log n)$) as subroutines, so we looked at these algorithms as well during our investigation.

We also wanted to learn more about how these algorithms and their parallel implementations behaved for various input types. In particular we were curious about how the $\log h$ vs $\log n$ factors compared in practice. To this end we created a test cases with various n to h distributions.

We hypothesized that it should be possible to achieve good speedup with Quickhull since it naturally can be parallelized. We also predicted that a sequential implementation of Chan's Algorithm would perform worse than Quickhull despite its better theoretical asymptotic time complexity due to the fact it must make multiple guess and check iterations, throwing away work each step. However, we saw that the structure of Chan's Algorithm has many opportunities for concurrency that could potentially mitigate this overhead. We will evaluate the performance of our parallel implementations of Chan's Algorithm and Quickhull against each other on various inputs to provide context for the performance of our implementations.

2 Implementation

2.1 Quickhull

Quickhull follows a similar pattern to quicksort; that is, it is a divide and conquer algorithm that typically has a time complexity of $O(n\log(n))$, but in pathological cases can have $O(n^2)$ performance

Quickhull starts by first identifying the leftmost and rightmost point. These points must be included in the convex hull. Then, the algorithm adds a line between these 2 points and finds the furthest point from this line that is above the line and the furthest point from this line that is below the line. Every point that lies within the triangle formed by the 2 line endpoints and 1 of the furthest points cannot be included in the convex hull.

This process of finding the point that is furthest from the line and on the correct side is recursively performed using the edges on the triangle between one of the line end points and one of the furthest points. The base case here is when there are no points on the correct side of the line, in which case the end points of the line are both added to the convex hull. Note that it is possible for convex hull points to be added multiple times across this algorithm, so a set needs to be used to store the hull.

Parallelization of Quickhull was done by parallelizing the recursive calls. Recall that for each line examined, the furthest point on the desired side is found and then this process continues for the edge from the left endpoint to the furthest point and edge from the right endpoint to the furthest point. As a result, every recursive call has the potential to create 2 recursive calls. So to speed up the algorithm, each of the recursive calls created are done on a separate goroutine, allowing them to be done in parallel. The synchronization needs required for this are relatively small. Namely, a lock was added surrounding modifications to the set representing the convex hull and each recursive call sends a value to a channel when it is finished and each non-base case recursive call waits for its 2 recursive calls to send back a value through channels. This is done so that it is easy to identify when the algorithm has finished.

2.2 Chan's Algorithm

From a high level, Chan's Algorithm achieves a $O(n\log h)$ runtime by computing estimates how many points are on the convex hull, splitting the input points into subsets based on the estimate, and computing the convex hulls of each of the estimates before joining them together. The convex hull of subsets of points, "subhulls" are computed using the Graham Scan algorithm and the subhulls are wrapped (joined) together using a variant of the Jarvis March algorithm. Our implementation of Chan's Algorithm enables the concurrent execution of operations within these three phases: hull-size estimation, subhull computation, and subhull wrapping. We hypothesized that despite it's better asymptotic runtime in theory, it was likely that a sequential Chan's Algorithm wouldn't be much faster than Quickhull in most use cases due to the fact that it has to run these multiple "guess and check" iterations. However, we also speculated that it was possible to mitigate this overhead with concurrency since the iterations could possibly be executed with good performance concurrently as discussed later. Additionally, the subroutines of subhull computation and subhull wrapping also had opportunities for parallelization that were different than Quickhull. Sections 2.2.1 and 2.2.2 focus on these subroutines, and Section 2.2.3 discusses the novel parts of Chan's Algorithm.

2.2.1 Graham Scan

The Graham Scan algorithm is an $O(n\log n)$ convex hull algorithm that chooses a extreme reference point and traces the convex hull based on polar angle relative to the chosen point. By default, our implementation chooses the point with lowest x and y coordinate as the reference point and then sorts all other points based on polar angle relative to this point. The points are then processed in this order using a slice-based stack, getting pushed on each iteration and popped off whenever "right turns" are found. This essentially ensures that no line segments that point inwards will remain in the stack. The Graham Scan is dominated by the sorting phase since processing the points afterwards is $O(n)$.

Our implementation of the Graham Scan has an iterative quicksort and a parallel quicksort. The parallel quicksort is fairly simple and spawns a goroutine for each recursive call into the upper and lower partitions. Our implementation stops spawning goroutines once the number of elements in the remaining portion of the array drops below a certain threshold to avoid basically fork-bombing a bajillion goroutines. With some empirical testing we found that switching to sequential recursion around 2000 elements remaining in the

array was reasonable. We chose to try out the iterative quicksort for the sequential implementation because of some quirks with the Go `sort.Slice` implementation discussed later.

2.2.2 Jarvis March

The Jarvis March AKA Gift-Wrapping algorithm is an $O(nh)$ convex hull algorithm that by default starts by finding the leftmost point in the dataset. This point is guaranteed to be on the convex hull, and adds it to the list of points on the hull. Then every point is scanned to determine its orientation relative to the last point added to the hull being constructed. The "leftmost" point is then chosen to be added to the hull, and the process repeats until the point chosen already exists on the convex hull. Thus for every hull point an $O(n)$ sweep of all points is required.

Our parallel implementation of the Jarvis March launches concurrent workers to build parts of the "ring" from different sources. Since the start of the Jarvis March ideally starts on a point on the hull, this meant we were constrained by the dimensionality of our points to practically 4 different points and searching in two directions (left/rightwards). Convergence for a particular goroutine then depends on whether the next point it selects has already been chosen globally by another goroutine. Technically in theory it may be possible to start by initializing the march at any arbitrary point and then chopping off a prefix of the points later that aren't part of the polygon, but this wouldn't be work efficient and would likely lead to a quadratic run time, since the leftmost/rightmost for an interior point isn't well defined.

2.2.3 Chan's Algorithm

Chan's Algorithm makes increasing estimates for what the value of h truly is and uses some clever implementation tricks to achieve its $O(n \log h)$ run time.

The algorithm begins by estimating the number of points that will be on the convex hull, call this estimate m and we can assume for now that $h = m$. The n input points are then partitioned into $\lceil \frac{n}{m} \rceil$ subsets and the convex hull of each subset is computed with a Graham Scan in $O(m \log m)$ time. Thus computing the subhulls takes $O(n \log m)$ overall.

A Jarvis March is then executed on the points of these subhulls to "wrap" them and find the final hull. Jarvis March is normally $O(nh)$, but finding the leftmost or rightmost tangent to a convex hull can be performed using binary search rather than a brute force scan. Chan's original paper conveniently leaves out the details of the binary search which produces lots of pain-in-the-butt casework regarding which points to pick next and what to do if the reference point is on the convex hull you are searching. The key idea boils down to the fact that you can compute the leftmost or rightmost tangent from an arbitrary external point to a convex hull whose points are sorted in some order by establishing an ordering based on the orientation of line segments relative to the external point. Thus for our subhulls of size m, computing the leftmost or rightmost tangent during a Jarvis March only takes $\log m$ time for each of the $\frac{n}{m}$ subhulls, so overall the process takes $O(\frac{nh}{m} \log m)$ time. So if $h = m$ then this takes $O(n \log h)$ time.

To ensure the estimate is eventually and quickly correct, Chan's Algorithm uses a doubly exponential (squaring) stepping strategy, with m starting at 2^{2^t}, where t is the current iteration. If the wrapping step takes more than m steps, the algorithm restarts and moves on to iteration $t + 1$. There is a maximum of $\log \log h$ iterations, and with some log rules you can show that the summation of the $\log \log h$ iterations yields only $O(n \log h)$ total work. We initialize t to be 2, as starting as an overestimate is ok, and this enables skipping the first iteration which in many cases will be too small anyways.

Our implementation enables concurrent computation of the subhulls by spawning a limited number of worker goroutines to compute subhulls which send results to a central manager goroutine. The worker pulls "work" (points to be computed + metadata) off of a work channel. The manager assembles subhulls as they come into a large slice composed of all subhulls and tracks subhull sizes so indices can be determined later. Originally we were thinking some techniques from group-by such as an atomic counter for each group position could be useful, but the overhead of rescanning to figure out sizes then writing back into the right

positions in parallel made this unruly for this application. This worker/manager paradigm has the advantage of making it easy to limit the number of workers.

Additionally, the previously described technique for parallelizing the wrapping step is also used, just with binary search. Convergence of the goroutines wrapping concurrently is determined by whether the next point selected by a goroutine has already been chosen by any of the other goroutines, so this requires some global synchronization/communication. In this case, a global bitmap of selected points doesn't work since the indices of each of the points being selected are only known relative to the subhulls that the points lay on, since in the previous step they must be shuffled in a sorted order for binary search. Thus we use a concurrent hash set (map) with fine-grain locks to store the selected convex hull points.

We also allow a limited number of iterations (2) of Chan's Algorithm to run concurrently. The main idea here is that although the latency of each iteration will likely be worse, since the exponential stepping is so large there should be a significant enough difference between two subsequent iterations that having the faster one finish + some latency will be quicker than the sum of their times (especially since generally lower steps take longer, but you don't want to initially overshoot by too much).

One of the interesting ideas presented in Chan's paper for optimizations is having future iterations reuse some of the work performed by previous iterations by coalescing old subhull into newer ones. We were curious about how this optimization would play out in a concurrent regime. Thus we also look at an implementation where these two concurrent iterations of Chan's are able to share their state and attempt to reduce repeated computation. This is accomplished by having iterations write the subhulls for particular partitions of the input points into a global map. Then the thread distributing the work to the subhull workers will scan over the partition of points it is going to send to a worker with a stride of the previous group size and check if the subhull for that set of points has already been computed. If it has, then those subhull points will be packed up with the input to the worker, rather than the original points from the input set.

2.3 Challenges

Floating point error/roundoff proved to be a very unexpected and interesting/time-consuming challenge to debug. All of the convex hull algorithms we implemented depended on computing cross products and distances to determine the spatial relationship between two points. Generally this is not a major issue and correctness depends on being within a certain epsilon anyways, but there are interesting edge cases where this caused trouble with the algorithms themselves. For instance, the Jarvis March algorithm begins by selecting some extreme point (furthest left, right, bottom, etc.) as a starting point, as clearly this point must be on the actual convex hull, and the algorithm exits once it has "marched" around the hull points and returned to the starting point. However, if the starting point is indistinguishable from its neighbors relative to the true last point on the hull due to roundoff, then it becomes difficult to robustly ensure that the starting point is always chosen again and the algorithm can fail to converge.

This issue is somewhat also related to collinear points which also are an interesting challenge, as points that are collinear shouldn't all be part of the convex hull. However, it may become difficult to tell if things are truly collinear due to floating point error, so this can only be correct to within a certain error as well.

These issues with floating point errors became more apparent as we started to test larger cases to stress the performance of our implementations. This was because the likelihood of "almost collinear" points grew significantly as the number of total points increased. We found that many pseudocode implementations online were overly simple and failed to handle these cases (even those on Wikipedia!).

Another example of a hack that was used to solve this somewhat is that the Graham Scan implementation uses a two-pass approach in opposite directions. The Graham Scan relies on sorting all points based on polar angle from a single point and then stepping through to find "left" and "right" turns, but with large numbers of points this approach has trouble differentiating between extremely tight clockwise vs. counterclockwise turns and thus the hull can have an order of magnitude more points than it should. To solve this we run two passes of Graham Scan using different (leftmost vs. rightmost) points as the reference point for the polar angle sort. Any points that previously appeared almost collinear and were subject to this error now appear extremely far apart in polar angle from the new reference point. The second pass adds only minimal extra run time in practice because the first pass will still prune out a significant amount of the original points.

3 Results and Discussion

To profile the performance of our implementations, we sought to create several types of test cases to analyze the impact of various workloads. These tests were all done by running 6 trials per sample point, throwing out the first trial and taking the average. These trials were done on the lovey UT lab machine, which is a 20 core lab machine with a Xeon E5-2650 @ 2.30GHz. The host's OS version is Ubuntu 18.04.

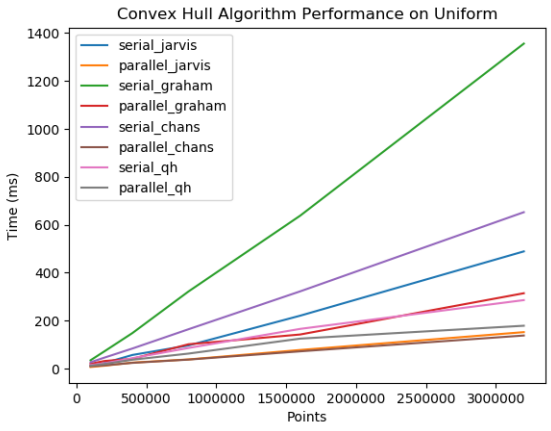
The first type of test case was a simple uniform distribution, where a specified number of points where generated within the 2 by 2 square centered at the origin. A second test case was similar to the uniform square test cases except the points were generated uniformly in the circle of radius 1 centered at the origin

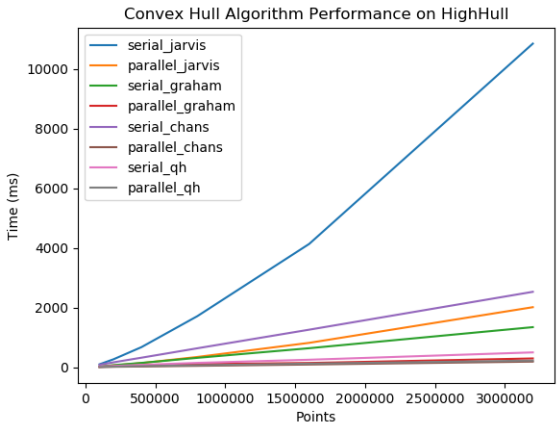
One problem with testing both of the examples listed above was that it was difficult to control how many points actually ended up on the hull. To help impact how many points showed up in the hull, we created another test case which is similar to the uniform circle test mentioned above except we could control what percentage of points were guaranteed to lie on the circumference (these points are much more likely to be in the convex hull than points further in the interior). Using tests generated by this approach, it would easier to gauge how much increasing the points in the hull impacts performance (recall that Jarvis march and Chan's Algorithm were specifically noted to be impacted by number of points in the hull).

3.1 End-to-End

The first aspect of performance investigated was end-to-end performance of each of our implementations. Note that we parsed in the input files once and then executed these multiple variations all on the same file, so this timing does not include file IO. In particular, we wanted to see how well each algorithm performed with respect to the number of points in the given test case. The performance visualized here includes only the running time for the algorithm, meaning it does not include the time spent parsing the input.

The first task was to just gauge how well the algorithms performed with respect to each other. To do this, we measured their end to end performance on 4 separate test cases: (1) a uniform square as described in the test case section (Uniform), (2) a circle with 10% of points on the circumference (LowHull), (3) a circle with 90% of points on the circumference (MedHull), (4) a circle with 60% of points on the circumference (HighHull).

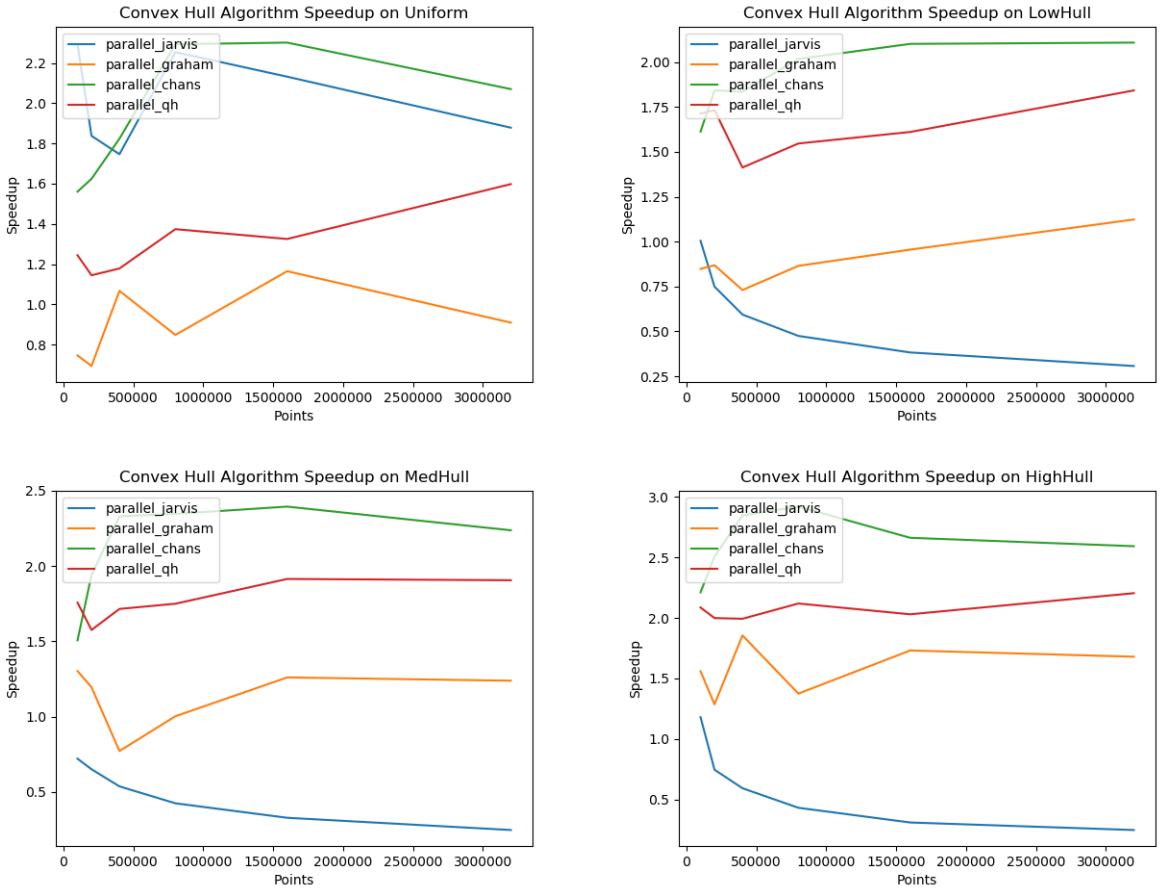




Pictured above are the results for Uniform and HighHull. We found results for LowHull and MedHull to appear very similar to HighHull, with the Jarvis March performing marginally better those two compared to HighHull. These two instances demonstrate the relative performance of the algorithms given datasets with low and high values for h. For context, at 32,000,000 points there are roughly 50 hull points on Uniform and roughly 2000 hull points on HighHull. As expected, the Jarvis March performs extremely poorly when h is high, while the Graham Scan performs poorly for smaller h. However, what is particularly notable is that the serial implementation of Chan's Algorithm performs the 2nd worse in terms of run time out of all algorithms, despite having the best theoretical asymptotic time complexity. This confirmed our hypothesis

that the iterative stepping technique did not lead to the best real-world performance.

For all of our tests we found that the most effective serial implementation was the serial implementation of Quickhull. This meant for our speedup calculation we used the serial Quickhull as the best serial algorithm and compared all of our parallel implementations to serial Quickhull when calculating speedup. So using serial Quickhull as a baseline, we created the following speedup graphs from the previous results.



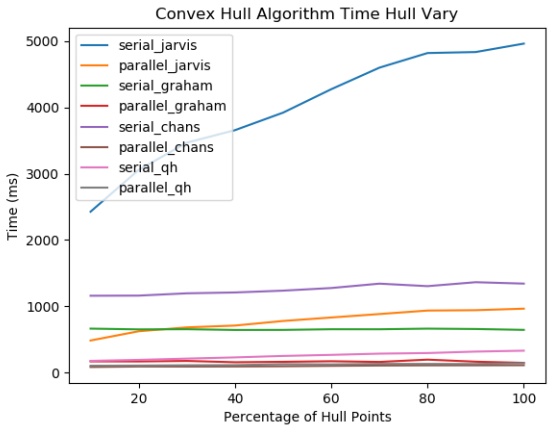
From these speedup graphs, we noticed that across the test cases chosen only the parallel Quickhull and parallel Chan's Algorithms were only really able to offer a consistent, competitive speedup over the serial implementations. The Graham scan algorithm seemed to offer little or no benefit across all of the test cases, whereas the Jarvis march algorithm was only able to offer an advantage for uniform. For other cases, Jarvis March's speedup drastically dropped as the number of points increased.

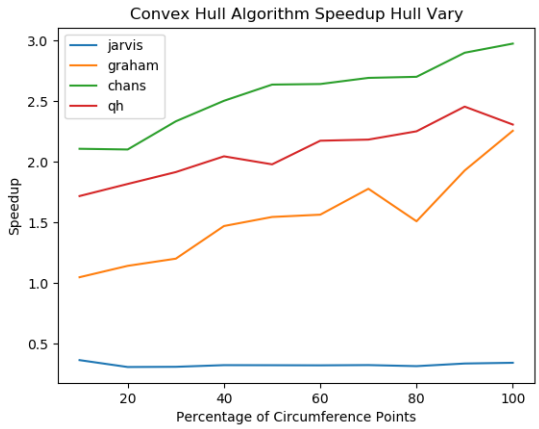
In contrast, Quickhull and Chan's Algorithm both had a speedup that was consistently at least 2.0. For practical analysis in later sections, we will as a result only be looking at Chan's Algorithm and Quickhull.

Interestingly, in all four scenarios Chan's algorithm experiences a sharp increase in speedup as the points initially increase up to about 500,000. This is likely because of the cost associated with splitting up the subgroups and the group sizes not being very close to the number of actual hull points when estimating (starting off a little large).

3.2 Hull Variation

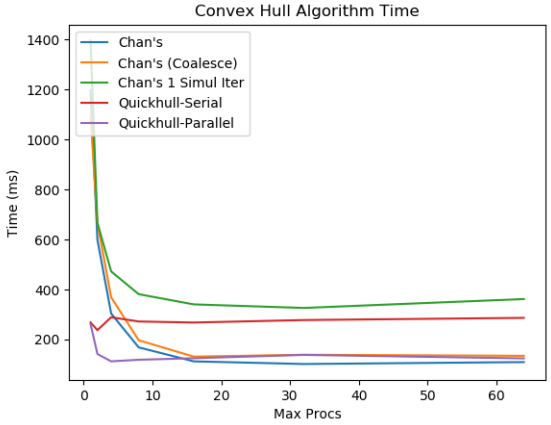
Another feature we wanted to examine was how much increasing the amount of points on the hull impacted performance. This was done using the circular test case described in our testing section where we controlled the percentage of points that were able to be generated on the circumference of the circle, which are far more likely to end up in a convex hull. To test this, we used a constant test case size of 1,600,000 points but gradually increased the number of circumference points from 10% to 100% in increments of 10%.

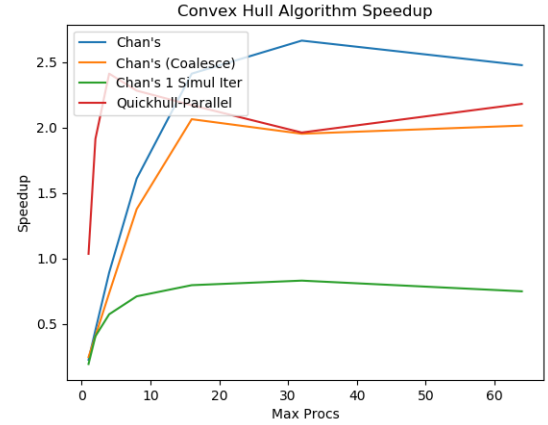




One of the most noticeable results from this testing was that Jarvis March, both the parallel and serial version, performed considerably worse as the percentage of circumference points increased, reflecting the fact that its algorithmic complexity of $O(nh)$ is the most severely impacted by the number of points on the hull. When examining only the algorithms deemed practically viable in section 4.1, we noticed that their performance, especially the parallel implementations, was only marginally impacted by increasing the percentage of hull points, further supporting the generalizability of our implementations of Quickhull and Chan's Algorithm on various workloads.

3.3 Scaling Performance





To evaluate how well our parallel implementations scaled, we ran trials on 1.6 million points with 60% of points on the circumference of a circle, and varied `runtime.GOMAXPROCS`, which limits the number of OS threads that will execute user-level code simultaneously. Technically threads blocked by system calls can lead to the runtime creating more OS threads, but should not occur during our tests. Note that our tests were run on a 20-core machine and we incremented `GOMAXPROCS` in powers of 2, so this lies between points 16 and 32. We tested our parallel Quickhull alongside a few variations of our parallel Chan's Algorithm implementations to understand in depth what was happening.

Quickhull was able to achieve its peak speedup, 2.4, with only 4 OS threads as opposed to the Chan's Algorithm implementations which all reached their peaks around the number of actual logical cores, 20. This was fairly surprising and we speculate this may have to do with cache locality and the fact that Quickhull requires a slightly more expensive computation performed many times per point. With fewer OS threads it may be more likely that a subsequent recursive call is placed back on the same CPU as its parent, who was operating on the same section of points anyways. When profiling the Quickhull execution with pprof we saw that it was the function that retrieves points from memory and performs a point-to-line computation that

took up a majority of the run time.

To get a clearer picture of Chan's Algorithm we ran these tests on three implementation which all differed in how they handled the multiple iterations that Chan's Algorithm requires for some inputs. The line labeled "Chan's" plots Chan's Algorithm running two iterations concurrently without any communication between the iterations. The line labeled "Chan's (Coalesce)" plots Chan's Algorithm running two iterations concurrently while communicating subhull points between the two iterations to reduce repeated work. Lastly, the line labeled "Chan's 1 Simul Iter" runs Chan's Algorithm with only a single iteration at a time. For all three versions the other subroutines in Chan's (subhull computation and subhull wrapping) were performed concurrently.

Parallel Chan's without coalescing turned out to have the best speedup and appeared to scale better than the other implementations. We were surprised that the coalescing implementation turned out a fair amount slower, although this does make sense. When trying to coalesce subhulls between concurrent iterations, there is much additional overhead incurred by needing to communicate to the global map and iterating over your point partitions to check for existing work. Additionally, whether or not work gets shared with our implementation is quite dependent on the scheduling of the goroutines for the jobs, since jobs across the range of points all get a separate goroutine. Thus the number of points saved by coalescing varied quite a bit between different trials. In practice we found that on this test case for `MAXPROCS=20` anywhere from 100,000 to 500,000 points were able to be shared between the two simultaneous iterations, but it appears that the extra overhead with getting these points outweighs the saved computation time.

Additionally, these results clearly demonstrate that running multiple iterations of Chan's Algorithm concurrently enables much faster convergence, despite each iteration having to share compute with other ones. We can see that when using any more than 2 or 4 threads that the concurrent iteration implementation begins performing much better than the single iteration parallel implementation, and scales much better.

To evaluate if these speedup results are consistent with theoretical expectations, we measured the times for independent stages of a sequential Chan's on the same input data. The results are shown in the table below. Note that the time for subhull computation and subhull wrapping is summed over multiple iterations.

Chan's Stage	Time Taken (s)	Fraction of Total Time
Subhull Computation	0.857	0.676
Subhull Wrapping	0.402	0.317
Total	1.267	1

Using Amdahl's law we can get a rough estimate what the speedup should be. Subhull computation is able to take full advantage of the number of threads we have, while subhull wrapping is limited to roughly 4 threads due to the limitation of 2D constraining where the Jarvis March phase can start from (discussed Section 2.2.2). Thus we can estimate that an ideal parallel implementation with 16 threads would take

$$T_{//} = \frac{0.676}{16} + \frac{0.317}{4} + (1 - 0.676 - 0.317)$$

$$T_{//} = 0.1285$$

of the serial amount of time, giving a speedup of roughly 7.78. The graph above shows our Chan's implementation achieving about 2.4 times speedup, but versus the fastest serial implementation, which was Quickhull, not Chan's. If we directly compare our serial Chan's time versus the parallel, we the results appear to more closes resemble this prediction. Parallel Chan's running just 1 iteration at a time achieves a speedup of 5.18, which is to be expected given the extra overhead required to manage more Goroutines, copy addition memory, and synchronize steps. If we look at Parallel Chan's running multiple iterations at a time, we actually see that we achieve a speedup of roughly 9.2 on the same test case. This is likely because as described in our earlier hypothesis, executing the multiple iterations concurrently is faster overall than waiting for previous iterations to complete first. Thus this sometimes enables the concurrent iteration implementation to converge after doing less work than the serial implementation.

3.4 Extra Observations

One interesting thing we saw when profiling the Graham Scan was that the sorting performance of Go's default `sort.Slice()` was much worse than expected. After trying out our own implementation of sequential quicksort instead, we found we could achieve a roughly 10% faster sorting phase. After further research into the topic we found that Go's `sort.Slice()` is really sorting `interface{}` behind the scenes so it relies on functions from the Go reflect package to perform swaps and get slice lengths at runtime. Thus when we use our own implementation we can use the slice built-in's and don't have to incur this slight overhead, which apparently builds up a noticeable amount in large sorts.

Another quirk we caught when profiling (pproc was cool) was that the `pow` operation was stupidly expensive. In particular our implementation was using `float32` to represent coordinates, and using `pow` in the computation of point distances had a dramatic effect on the overall run time. For example, changing the call to `pow` with just a multiplication reduced the time for our parallel Quickhull implementation from approximately 120ms to 40ms on an input size of 100,000 points. We speculate that a part of this is because the Go `pow` function operates on `float64` rather than `float32` as we are using, but this was still a surprisingly significant difference.

After going back to Go from the Rust lab it makes one appreciate the static checking that Rust does to avoid accidentally mutating shared state. In particular Go's slice is super useful for passing along partitions of data, but it was very easy to forget about side-effects in the underlying array that could be shared with others. This lead to quite a few strange races when composing functions that otherwise worked perfectly separately.

4 Conclusion

We implemented parallel versions of Quickhull and Chan's Algorithm in Golang to compute convex hulls and analyzed the performance advantages and disadvantages of various implementation techniques. We note that despite Chan's Algorithm's good theoretical asymptotic runtime, in practice a sequential implementation does not perform well, with Quickhull being the best performer. However, we show that a parallel implementation of Chan's Algorithm can outperform a parallel implementation of Quickhull, even across various workloads.