Randomized Parallel Convex Hull

by

Edward Xiao & Cole Christini

Final course project for 15-418: Parallel Computer Architecture and Programming

April 28th, 2025

[Project Website]

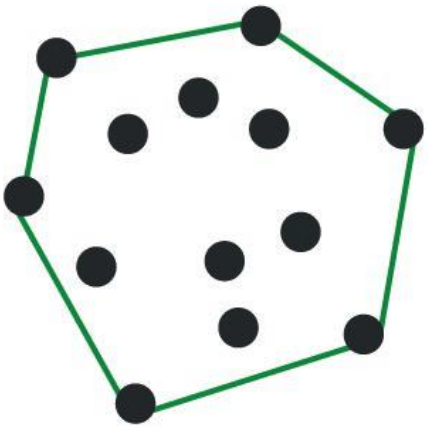
Summary

We implemented Guy Blelloch's randomized incremental parallel convex hull algorithm on the CPU shared address space model using OpenMP and Taskflow. We then evaluated its performance across many different implementations on a M4 Pro Mac mini, the GHC machines, and the PSC nodes. We demonstrated that Blelloch's algorithm can indeed exhibit high degrees of parallelism with a proper implementation strategy, although there are some factors which limit its scalability at very high thread counts.

Background

The Convex Hull Problem

Convex hull is a well-studied problem in computer science concerning geometric optimizations. Formally, given a set of points in space, the convex hull is the smallest convex region which contains all of the points in that space. In 2-D, the convex hull is simply a polygon which encloses all points in the plane, as shown below.



Convex hull in 2 dimensions

The convex hull problem has numerous applications, namely those in economics, image processing, and optimization. Notably, computing the Voronoi diagram and Delaunay triangulation of 2-D points has a straightforward reduction to the 3-D convex hull problem, which itself has many more applications in geometry. There are many known algorithms which solve the convex hull problem in every dimension. For this project, we decided to focus mainly on the 2-D convex hull problem since there are simple and efficient sequential algorithms for it which makes it easy for us to compare our parallel performance to a close-to-optimal sequential algorithm, and also because the parallel solution we chose generalizes easily to any higher dimensionality.

Sequential Algorithms

Perhaps one of the most well-known sequential algorithms for the 2-D convex hull problem is the Graham scan. It also happens to be the simplest work-efficient convex hull algorithm with a time complexity of $O(n \log n)$ stemming from sorting the points[1]. We chose to implement a close variation of Graham scan, formally known as Andrew's monotone chain [1]. The algorithm is

[1] It turns out that $O(n \log n)$ is the best that can be done, as sorting can be reduced to solving convex hull.

based on the observation that the convex hull can be split into two halves, a top hull and a bottom hull, delimited by the left and right-most points in the set. All points in the top hull make clockwise rotations when scanning from left to right, and conversely all points in the bottom hull form counterclockwise rotations. This leads to a direct scan-line algorithm, where we add points to the top hull by default and repeatedly remove points from the hull while there is a counterclockwise rotation. We do the exact opposite for the bottom hull, and simply combining the two produces the entire final convex hull. Our implementation for this approach is straightforward and shown below:

```
std::vector<Point> convex_hull(std::vector<Point> &points) {
  int n = std::size(points);
  std::sort(points.begin(), points.end());
  std::vector<Point> upper, lower;
  for (int i = 0; i < n; i++) {
      while (upper.size() >= 2 && ccw(upper[upper.size() - 2],
upper[upper.size() - 1], points[i]) < 0) {
      upper.pop_back();
      }
      upper.push_back(points[i]);
  }
  for (int i = 0; i < n; i++) {
      while (lower.size() >= 2 && ccw(lower[lower.size() - 2],
lower[lower.size() - 1], points[i]) > 0) {
      lower.pop_back();
      }
      lower.push_back(points[i]);
  }
  std::vector<Point> hull(upper);
  hull.insert(hull.end(), lower.begin() + 1, lower.end() - 1);
  return hull;
}
```

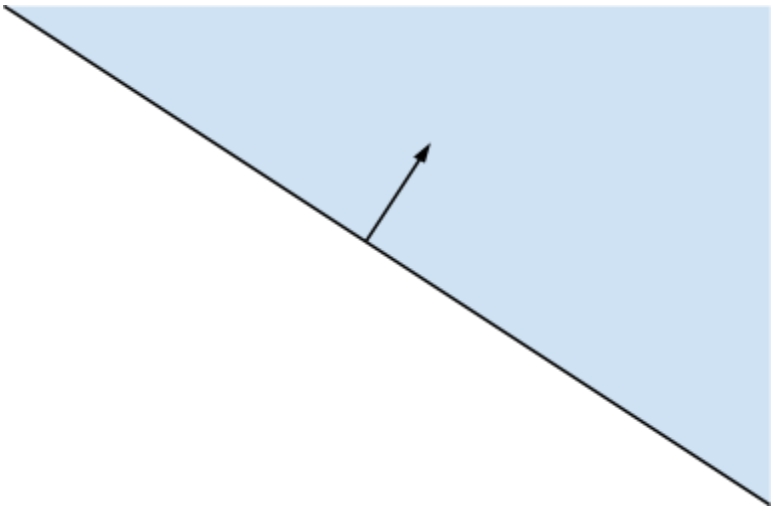
Implementation of sequential convex hull

As can be seen, the algorithm is implemented using a vector as a stack, and the main computation is amortized *O(n)* work. As such, the bottleneck of the algorithm is the sorting part which is simply a call to std::sort, a very well optimized sequential sorting algorithm. We

conclude that this is an efficiently optimized sequential algorithm that we can use to assess the absolute performance gain of our parallel algorithm.

Blelloch's Randomized Parallel Convex Hull Algorithm

In 2020, Guy Blelloch presented an $O(n \log n)$ work and $O((\log n)(\log^* n))$[2] (with high probability) span algorithm that solves the convex hull problem [2]. The key idea of the algorithm is that of a **ridge**, or vertex of a convex hull. Each ridge has two incident facets, for which we can define a visible set. A point is in the visible set of a facet if it is in the outward-facing half-space defined by the line passing through it.



The outwards half space defined by a line

A facet is 'finalized' when there are no points in its visible set. When all facets are finalized, we know that the set of facets is precisely the convex hull. While the goal is to finalize all facets and thus find the convex hull, we instead process ridges in order to find the facets. In order to process a ridge, we look at the two incident facets; if both of them are finalized then we can say that the ridge is finalized and will be in our convex hull as well. Otherwise, we check whether there is a

[2] Here, log* n represents the iterated logarithm function—the number of times one needs to take the log before reaching a number ≤ 1.

point that is visible from both facets incident on the ridge. If there is, then we know that this ridge cannot be in the final convex hull. If there is no such point, we replace one facet (selected arbitrarily) with two facets: one from its first point to a point in its visible set, and the other from that point to the original ridge we were processing. We then recurse on the new ridges. We present pseudocode for this below:

```
H = set of current facets
C = map from facets to visible sets
M = map from ridges to incident facets

function convex_hull(V = {v1,...,vn})
  H = the convex hull of {v1,v2,v3}
  parallel foreach t in H do
    C[t] = {v in V | visible(v, t)}
  parallel foreach (t1,t2) in H sharing ridge r do
    process_ridge(t1,r,t2)
  return H

function process_ridge(t1,r,t2)
  if C(t2) = C(t1) = ∅ then return
  else if min(C(t2)) = min(C(t1)) then H = H \ {t1,t2}
  else if min(C(t2)) < min(C(t1)) then
     process_ridge(t2,r,t1)
  else
    p = min(C(t1))
    t = join r with p
    C(t) = {v' in t1 U t2 | visible(v', t)}
    H = (H \ {t1}) U {t}
    parallel foreach r' in boundary of t do
      if r = r' then process_rudge (t, r, t2)
      else if (not M.insert_and_set(r', t)) then
        T' = M.get_value(r', t)
        process_ridge(t, r', t')
```

Pseudocode for the parallel convex hull algorithm

We repeat this process until all facets are finalized. The depth of recursion is *O(*log *n)* with high probability, which is how we obtain the above bound.

Key Data Structures

There are 3 key data structures in the algorithm mentioned above. **C** is a map from facets to their visible sets. We never delete from this data structure, but there is read-write contention as every thread needs to access this on every recursive step, and some will also write to it. The facet set, **F**, stores the facets of the current convex hull. We only access this when adding or removing facets, but this will happen on every non-terminal recursive call and so it is a primary source of contention. The last key data structure is **M**, which maps ridges to their adjacent facets. We write to this each time we add a new facet—either updating the value attached to an existing ridge or adding a new ridge into the data structure. While we also don't delete from this data structure, we do write to it regularly and so this is our third key point of contention.

Workload & Breakdown

The main source of work in our algorithm is the actual recursive calls. As mentioned above, we expect that the recursion depth is $O(\log n)$ with high probability. This recursion dependence is the primary limiting factor on parallelism in this algorithm, but the recursive structure is what allows the algorithm to be work efficient. There are trivially simple algorithms that are embarrassingly parallel, but these have work up to $O(n^3)$, whereas the best sequential algorithms (i.e. Graham Scan) have a time complexity of $O(n \log n)$.

Within the recursive calls, there are two key operations. The first is finding the 'minimum' element in the visible set to check whether both facets have a shared visible point. In the paper, they define minimum in terms of insertion step, but we found that all that's needed is some ordering such that if there is a shared point in both sets, it'll be the minimum for both of them on

some recursive call. This step can take anywhere from $O(n)$ work in a naive implementation to $O(1)$ work by using different data structures.

The other major component of the work done in a recursive call is the construction of the new visible set. This requires checking many points for visibility (which is itself a simple operation involving the cross product). For the initial visible sets (for the starting facets), we need to check every single point. For later sets, it turns out that it suffices to check just the visible sets of the two original facets for a ridge to construct the visible set of the replacement facet. In general, this operation reduces to a merge and a filter, which both have $O(\log n)$ span. While these are theoretically very fast, they still take up a large fraction of the overall work done and so need to be carefully implemented in order to achieve maximum speedup.

Our Approach

Baseline Parallel Implementation

For ease of implementation and performance, we chose to limit our algorithm to integer points only. This also eliminates the risk of any errors due to floating-point arithmetic. Points are then just a pair of integers, and facets are a pair of points. This simplified representation is highly memory-efficient, and allows us to easily implement various comparisons, equality checks, etc. as simple applications of the integer operations. We chose to represent **C**, the map of visible sets, using a hashmap mapping facets to vectors of points. This allows us to have amortized $O(1)$ access and insertion, which is important because these are such common operations in our code. The choice of a vector rather than a set to store the visible points is due to the cost of inserting

and accessing a set. We found that when implementing the algorithm using a set (something which fits the pseudocode given in the paper better), a large fraction of the time in the algorithm was being spent just on insertion calls. As such, we decided that using vectors would provide significantly better performance even if it did necessitate the addition of a merge step to the construction of the new visible sets. We use another hashmap for **M** for similar reasons, while **F** is stored as a hashset to allow for fast access and writing.

The code is broken down into several key functions implementing (almost directly) the algorithm from the paper. The `main` function handles reading in the input data, shuffling the points, and timing the execution. The `convex_hull` function is responsible for constructing the initial convex hull and making the initial calls to `process_ridge`, as well as handling the conversion of facets into the point format we use to store the convex hull. The main bulk of the work happens in the `process_ridge` function, which has the same casing and operations as the function of the same name in the pseudocode. This baseline translation achieved single-threaded performance ~5x slower than the optimized sequential version using Graham scan. We found that this factor didn't change with problem size, which makes sense as it's a work-efficient algorithm.

Parallel Hashmaps

Our initial implementation used **std::unordered_map** for **C** and **M**. While this works well for single-threaded applications, in multi-threaded cases it requires a lock for the whole data structure on insertion, potentially limiting our scaling with the map acting as a point of serialization. It also rescales to double in size whenever it's full, leading to high memory usage and cost. In order to combat these problems, we switched to a faster hashmap implementation

written by Gregory Popovitch. This uses open addressing to make accesses faster, as well as SSE2 SIMD instructions to increase the lookup speed.

The key benefit for our use case is the fact that the hashmap is implemented with internal sub-tables. This is primarily intended to help combat the memory usage issue, but also helps reduce the synchronization overhead. By setting the number of sub-tables to be larger than the number of available threads, we dramatically reduce the probability of two threads trying to write to the same sub-table at the same time, therefore reducing the chance of any stalling while waiting to acquire a mutex. This enables different threads to write to different sub-tables of the map in parallel, thus removing a massive point of synchronization. Internally, the parallel hashmap protects its inner sub-tables from race conditions by assigning a mutex to each one.

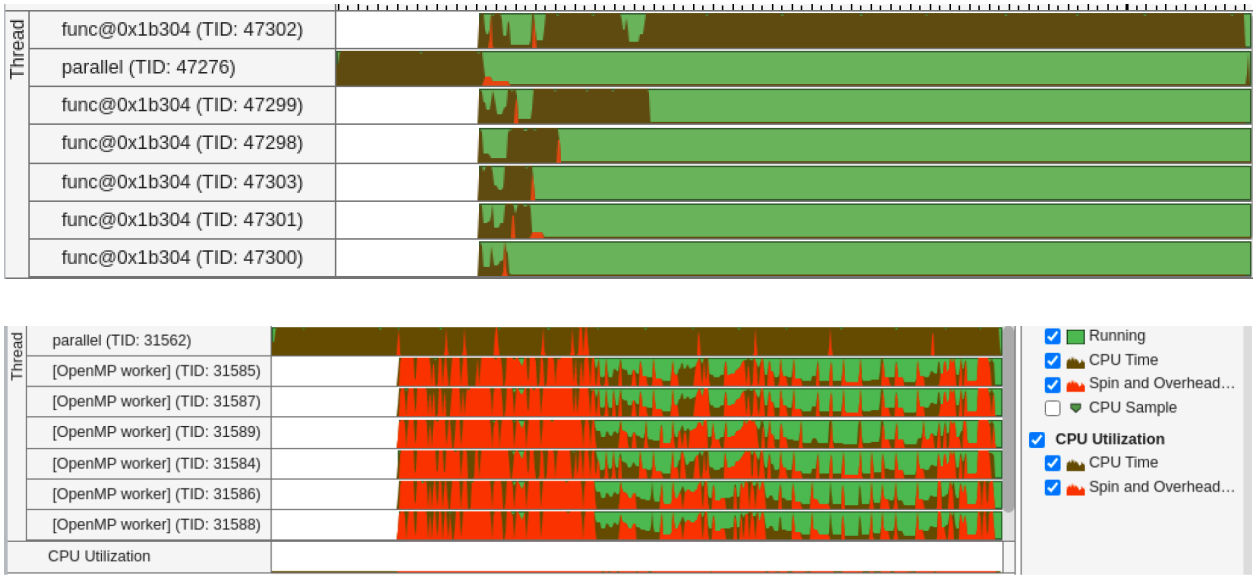
OpenMP Implementation

Our initial parallel implementation used OpenMP due to how simple it makes parallelizing code, as well as the many data structures that are shared between threads and need to be kept updated.

Tasking

Our first approach was to use OpenMP's **tasks API** to parallelize the recursive calls, with each recursive call spawning a new task. This approach most closely resembles the pseudocode, as well as how we'd intuitively expect to parallelize this algorithm. While this approach intuitively makes sense, it ends up being relatively difficult to implement due to the many possible race conditions on the data structures. This is, admittedly, a problem with any parallel implementation but the recursive parallelism makes it both more likely to happen **and** harder to debug when it

does. We initially wrote this with tasks waiting for the recursive calls to finish before returning, before finding a way to remove the waits and thus lower the overhead caused by using those synchronization primitives. There is still synchronization required for updating the shared data structures, which we handled by using OpenMP's critical pragma in order to ensure only one thread could access each of the relevant blocks of code at any given time.



Vtune with waits, Vtune without waits

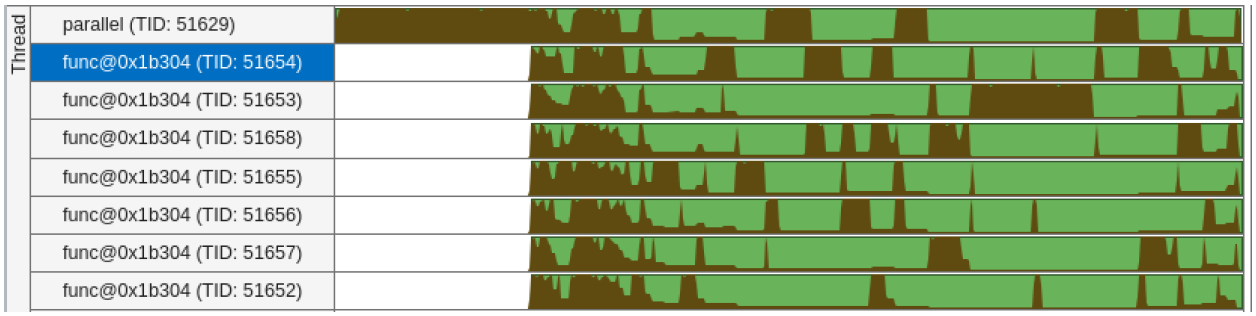
While this approach made intuitive sense to us, and so was the way we chose to parallelize our code first, we found that it actually performed very poorly. In particular, there was very bad work imbalance across the processors. In our original version, we found that there was a high degree of overhead, which we suspected was due to the synchronization primitives needed for the waits. However, even when we removed these we found that tasks were not properly being scheduled across the available threads. In fact, without the waits there was almost no parallelism and most of the threads sat doing nothing for almost the entire lifetime of the programming. We suspect that this may have to do with the scheduling algorithm used by GCC. While Intel's ICC compiler is known to have a much better scheduler, we were not able to use it due to the need to support

AMD in order to be able to run on Bridges 2. We thought that perhaps the scheduler might perform better using different parallelism constructs, and so we shifted our approach to use some of those.

Parallel BFS

After the poor performance of the task-based approach, we decided to fully rethink what parallel constructs we would use as well as the order of scheduling. Rather than using tasking, we decided to implement parallelism using the standard **parallel for** constructs. Doing so necessitated a restructuring of the code: rather than a single `process_ridges` function that makes many recursive calls, we instead broke it up into two functions: a `parallel_process_ridges` function that uses OpenMP parallel pragmas to process the current set of ridges in parallel, and a new `process_ridge` function that processes a single ridge but does not make any recursive calls. We chose to use dynamic scheduling here due to the highly variable cost of processing a ridge, depending on which case it falls into and the size of the visible sets of the two initial incident facets. In order to minimize the amount of synchronization needed, each call constructs its own `local_ridges` vector. After all calls in a round are done, we consolidate all of these using a reduction into the new ridges vector. After processing all ridges and consolidating the new facets into the updated data structure, we make a recursive call to `parallel_process_ridges` with the new ridges vector as the input. This scheduling order was much clearer and easier to analyze. The multiple parallel rounds are equivalent to doing a breadth-first traversal of the call graph—all the same calls still occur, but rather than traversing down a branch of recursive calls and spawning tasks as we go, we instead do an entire layer in parallel and then do all the recursive calls that would have resulted from that layer after. The reason we chose not to parallelize with recursive calls in the parallel for loop version is primarily

because of the high overhead associated with nested parallelism in OpenMP. We expected that this scheduling would provide slightly better performance, primarily due to the lowered overhead of parallel for loops as compared to tasking. However, we again saw issues with the scheduler as in the tasked version.



Vtune results of parallel BFS

In particular, we found that the OpenMP scheduling algorithm would almost always only have two threads doing any work, while the others sat almost entirely unused. We were not able to identify any synchronization construct or other piece of code that would cause this behavior. Profiling with VTune's threading mode didn't show a significant amount of time spent on synchronization. At the time, we thought this was an issue with the scheduler but looking back it was more likely an issue with the initial tasks being very large—leading to a poor work distribution as each task would only use one thread no matter the size.

Parallel Filtering with Scan

Our last thought for parallelizing parts of the code with OpenMP was improving the speed of finding the new visible set. This is an **embarrassingly parallel** task—evaluativing visibility can be done for each point in parallel. After doing so, the remainder of the problem is simply a filter, another task that can be implemented efficiently in parallel using scan. While we tried this approach as the primary source of parallelism, we found that it did not provide a major parallel

speedup. After measuring the average size of the joined visible sets that we were filtering, we found that most of them were relatively small. On a test case with 2^{20} points, the average size was only 64k. Because evaluating the visibility of an individual point is already quite cheap, the overhead of launching the parallel region outweighed the speedup in calculating the size of the visible set.

Switching to Taskflow

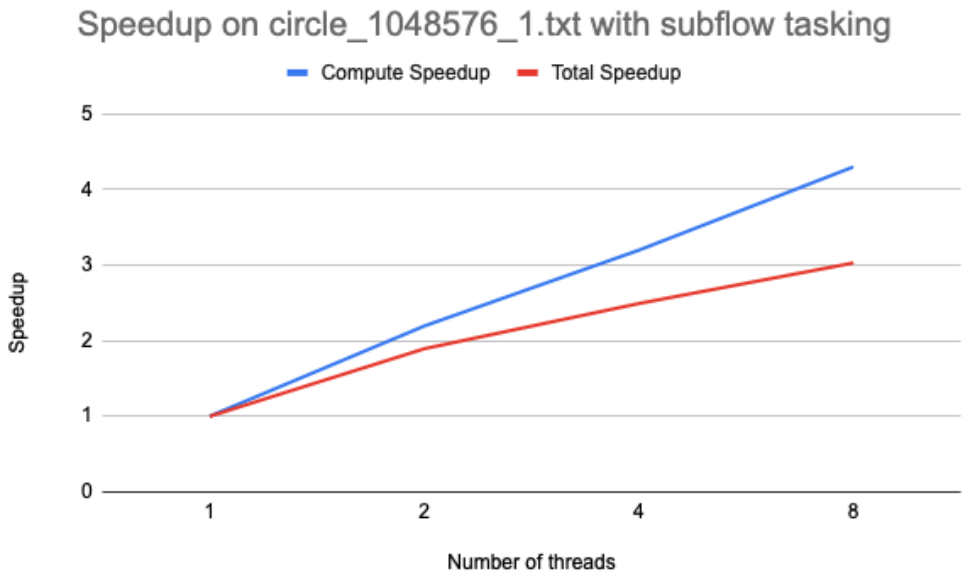
Due to the issues we experienced with the OpenMP task scheduler that limited the amount of parallelism we observed when executing the code, we decided it would be worthwhile to also experiment with other parallel programming frameworks in C++ and see how our performance compares. This is when we came across Taskflow, a versatile C++ library to express and execute high performance parallel code on multiple cores [3]. Thus, we learned all of the parallel programming tools provided by Taskflow and iteratively worked on numerous different approaches to maximize the achieved parallelism.

Taskflow Tasking

Our first attempt at migrating our OpenMP code to Taskflow was by translating the task-based approach from before. In Taskflow, OpenMP tasks most directly correspond to synchronous tasks whose dynamic recursive dependencies are best expressed using recursive subflow tasking. We specified the recursive calls to process_ridge like so,

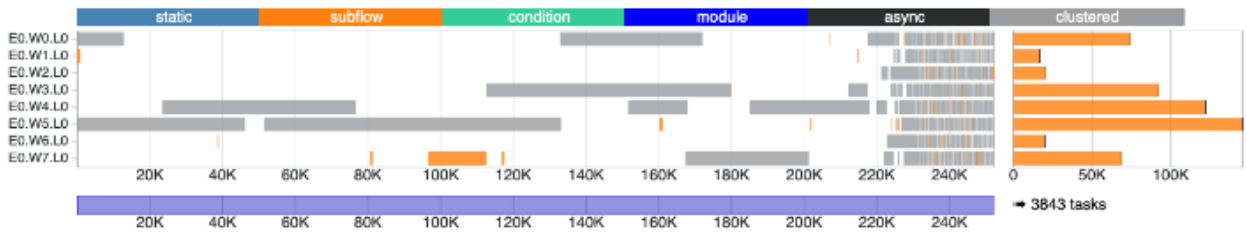
```
sbf.emplace([f_prime, p, f_new](tf::Subflow &sbf) {
      process_ridge(f_prime, p, f_new, sbf);
      });
```

dynamically creating a subtask dependency graph that can then be scheduled to the parallel workers using the builtin work stealing scheduling algorithm found in the Taskflow executor. We immediately observed a considerable boost in performance, due to the fact that the Taskflow scheduler was actually distributing the small parallel tasks very evenly. Our speedup graph for this approach is shown below,



Speedup of subflow tasking

from which we can see the maximum speedup on 8 threads is up to 4.3. Although this is better than the speedup we achieved using OpenMP tasks, it is still not quite an ideal speedup yet. To analyze why this is the case, we profiled our code with 8 threads running on the case above and saw an interesting utilization pattern.



Profiling results of subflow tasking

As can be seen from the profiler output, our code actually achieves very good parallelism towards the end of the execution with almost perfect utilization of all 8 workers to complete the small recursive tasks where the visible set of each remaining ridge is tiny. However, we see that the utilization of the workers is very poor (almost sequential) for a large portion of the execution in the beginning. This is because there simply aren't enough tasks to schedule in the opening phases of the algorithm, and instead the workers are occupied with long sequential tasks where the visibility sets are massive. This is in contrast to the tail end of the execution, where tasks are small and abundant. To improve our scalability, we are going to have to deal with the issue of low parallelism in the beginning of the algorithm.

Async Tasking

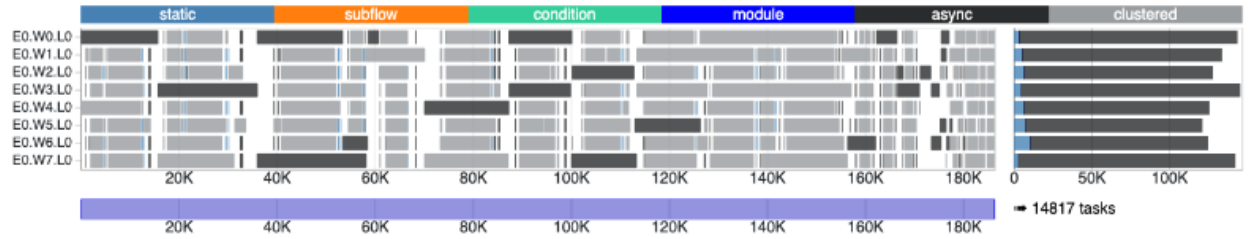
To prepare for further optimization, we first needed to choose a better Taskflow model to represent the parallelism in our problem. After more investigation, we found that the asynchronous tasking model actually better suits our needs in the parallel convex hull algorithm. This is because the tasks themselves don't actually have dependencies on the other tasks directly, but instead depend on the visibility set computation that is done within the parent task. Therefore, as long as we insert synchronization points to ensure the visibility set computation is completed before the child tasks are executed, the correctness of the algorithm is preserved. These synchronization points correspond to the executor corun method for asynchronous tasks, and so we migrated our code to use asynchronous tasks instead. The performance and scalability of the code remained largely the same, with the issue still being that there were insufficient tasks to execute in the beginning of the algorithm. However, we now had a better idea of what needed to be done to use this to our advantage.

Parallel Filter

We implemented the same parallel filtering approach in Taskflow just to see if the scheduler would provide any better results. In general, we found that there was little-to-no speedup using parallel filtering on the same test case with 2^{20} points. After benchmarking the filter implementation, we found that there was only any speedup for filtering on inputs larger than 75k elements due to the scheduler overhead outweighing the time saved on small visibility sets. Given that the mean input size to filtering with over 1M points was less than that, it's not unexpected that this would not provide any major speedup.

Hybrid Filtering

Returning back to our previous asynchronous tasking approach, we now had a great idea to improve the parallelism of the solution. Effectively, the strengths and weaknesses of the parallel tasks and the parallel filter approach are exact opposites of each other. Parallelizing over tasks works well when tasks are abundant and each task is relatively small so that load balancing can excel. On the other hand, using parallel filter within a single task is great when the task is large since the computational speedup outweighs the overhead of launching the parallel scheduler and communication between workers. Instead of choosing one approach over the other, we can have a hybrid approach where we use parallel filter within tasks when there are only a few large tasks in the beginning phase of the algorithm, and we exploit the performance of task-based parallelism when the tasks are small but abundant! As it turns out, this happened to be the key insight that allowed us to unlock the full parallel potential of the algorithm.

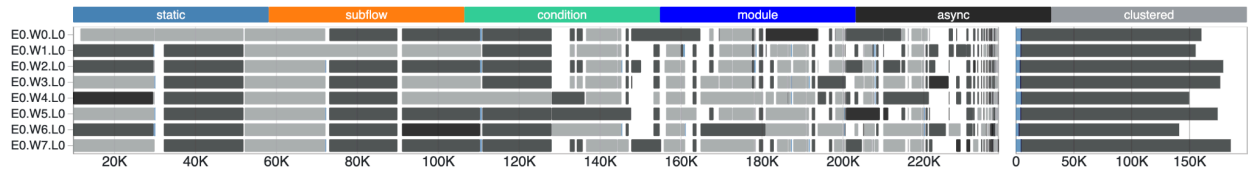


Profiling results of asynchronous hybrid tasking

Profiling the new hybrid algorithm, we see that the utilization of each worker is close to ideal, and the load imbalance between different workers is minimal. Indeed, leveraging the hybrid parallelism present in this algorithm was the most effective way to achieve our desired speedup. To decide when exactly to use parallel filter over sequential filter, we added an adjustable parallel cutoff to our program. Tasks with more total points than the parallel cutoff are executed using the parallel filter, while any smaller tasks are simply done on a single thread.

Parallel BFS (Again)

We also implemented the parallel BFS-based approach in Taskflow. By applying some of the same optimizations as we used in the task-based version (particularly using asynchronous tasks rather than synchronous structures, but using the for-loop based code and scheduling order), as well as hybridization, we were able to achieve a 4x speedup in the parallel version. In order to try and improve this, we profiled this code using the Taskflow profiler.



Profiling results of asynchronous hybrid BFS

While the work balance is okay, it's not quite as good as the task-based version, which also leads to worse overall speedup. This is in large part due to the additional dependencies the BFS solution introduces. Because we process all current ridges before processing any of the new ridges, we end up with enforced rounds that synchronize in between. This drives some of the later gaps in utilization that have hurt our scaling. At those points, the size of the sets to filter is small enough that they are being handled sequentially, but the other threads have far less work to do. An example of this occurring can be seen around 150k µs. In theory, the utilization and balance issues of this solution could be combated by using a RW queue, which would remove the additional synchronization and improve the performance of this solution. In practice, we chose not to do this due to time constraints but it is a potential future direction for this project.

Results

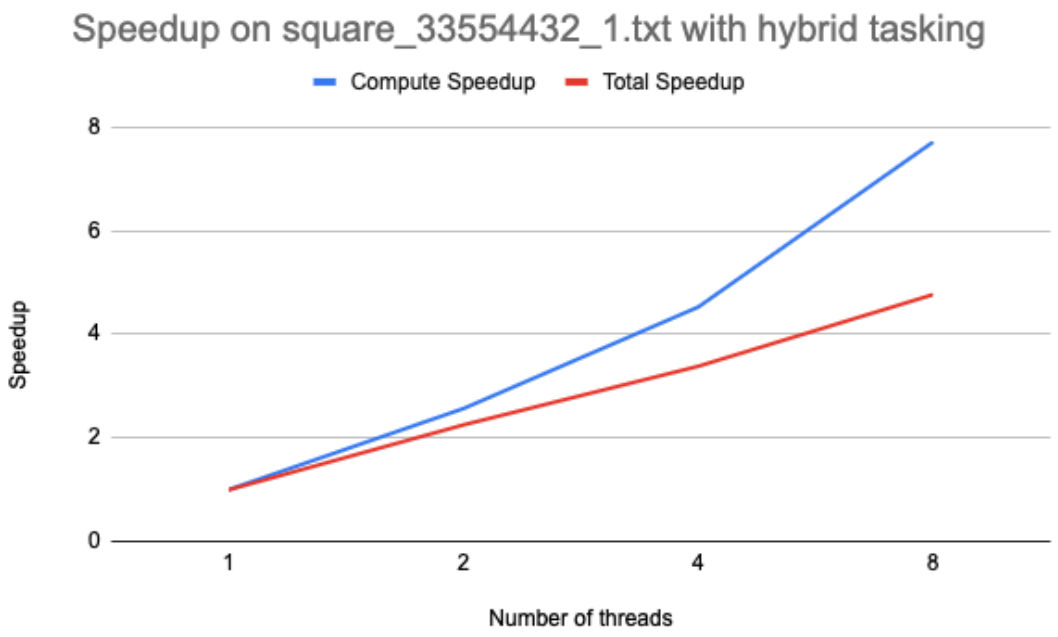
Experimental Setup

We conducted rigorous performance tests on our parallel implementation of the convex hull algorithm. These were executed on 3 different architectures:

1. Mac mini with the latest Apple M4 Pro chip. This chip has 8 performance cores and is amenable to some decent parallelism.
2. GHC cluster machines. These contain 8 Intel Core i7 processors which is similar to the M4 chip, so it will be interesting to see how they compare.
3. PSC Bridges-2 Regular Memory (RM) machines. Each RM machine contains 128 cores, which will allow us to test the scalability of our solution at large thread counts.

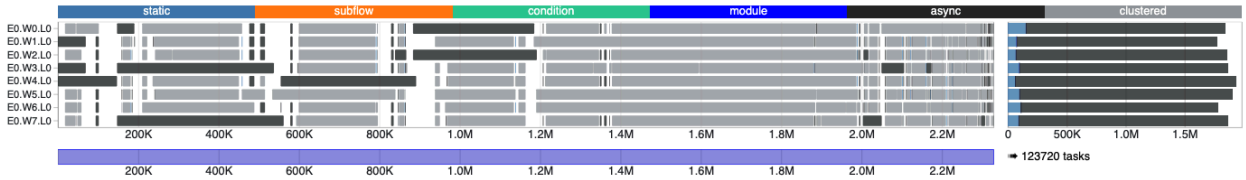
To facilitate the benchmarking of our code, we created a comprehensive testing suite with many different types of test cases and varying number of points in order to measure the sensitivity to problem size. Throughout the results section, unless explicitly stated otherwise, data was collected using our hybrid implementation with asynchronous tasks described in the Hybrid Filtering section. In most speedup plots, we measured both total and compute speedup. Total speedup includes loading the test, shuffling the points, and computing the convex hull while computation speedup only includes that last step. Those pre-processing steps are purely sequential, and so due to Amdahl's law we expect the speedup when including them to be lower than the speedup without them. This gap increases at larger test sizes due to the increased cost of this initialization, as well as the increased opportunity for parallelism.

Results on M4 Pro



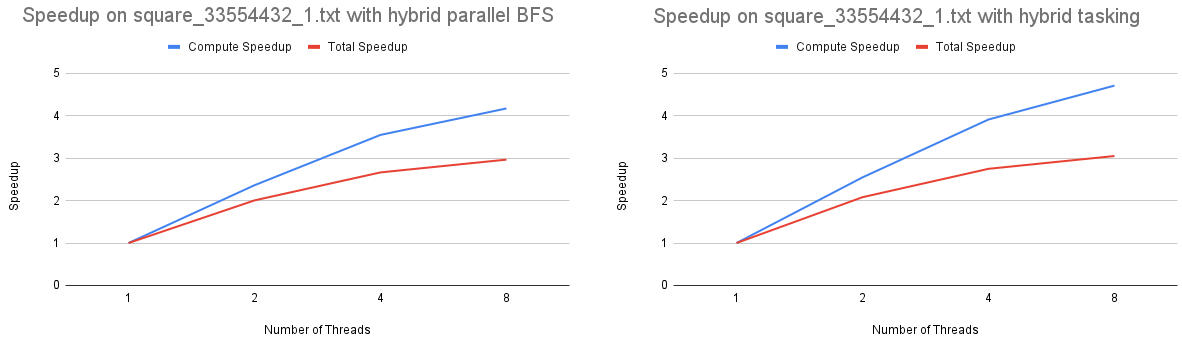
Speedup graph with parallel cutoff set to 500,000

From the figure above, we observe nearly perfect speedup when running on 8 threads and in fact some minor super-linear speedup executing on both 2 and 4 threads. This super-linear speedup can be attributed to increased cache locality, which is explored more in-depth in a subsequent section. Using the profiler, we can observe nearly perfect work distribution when running with 8 threads, and all workers are almost always performing useful computation which explains the speedup of 7.7. The only noticeable gaps towards the beginning of the runtime are caused by allocating large vectors in the parallel filter function, which is an inherently sequential operation.



Profiling results of 8 threads running on square_33554432_1.txt

Results on GHC

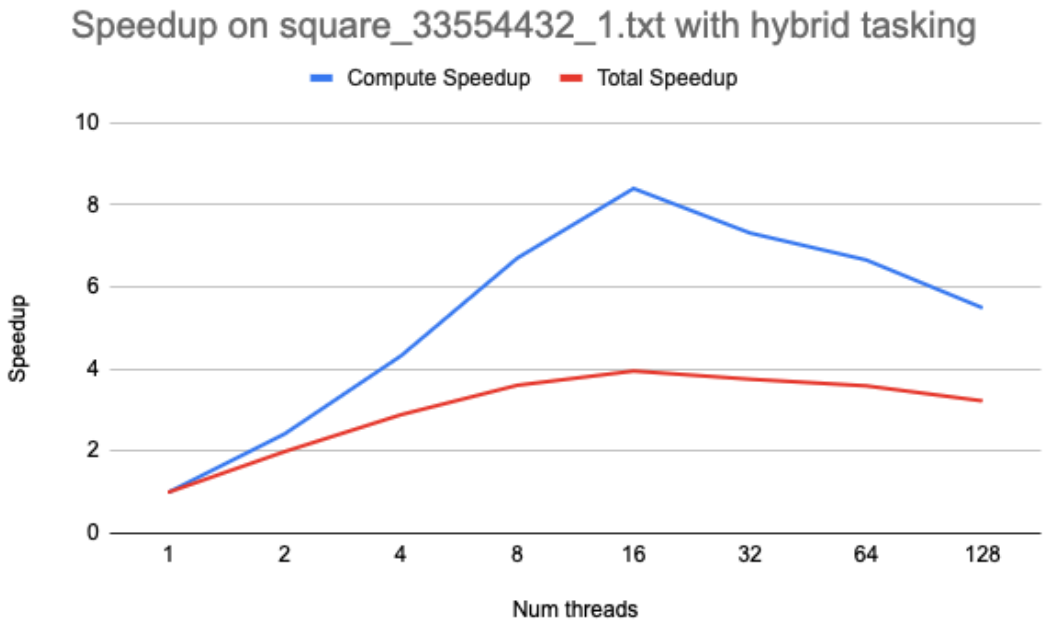


Speedup graphs on GHC with parallel cutoff set to 500,000

On low thread counts, our scaling is very good. At 2 threads, we again saw super-linear scaling (2.36x compute scaling for BFS, 2.54x for tasking). On 4 threads, our tasking-based approach achieves near-perfect speedup, while the BFS approach achieves a competitive but slightly worse

speedup. In general, as the thread count increases we've found that the parallel BFS approach becomes less efficient relative to the task-based approach, with a speedup that is approximately 11.5% worse at 8 threads. This is primarily attributable to the additional dependencies mentioned in the prior section on the BFS approach. Unfortunately, the performance of both approaches fell off heavily at 8 threads, with speedup improving by less than 1x for both approaches (with the BFS approach seeing a smaller improvement). This runs counter to what we observed on the Apple M4 Pro for the tasking approach, where speedup scaled very well through 8 threads. This is due to differences in cache performance, which we detail more in the Cache Locality section.

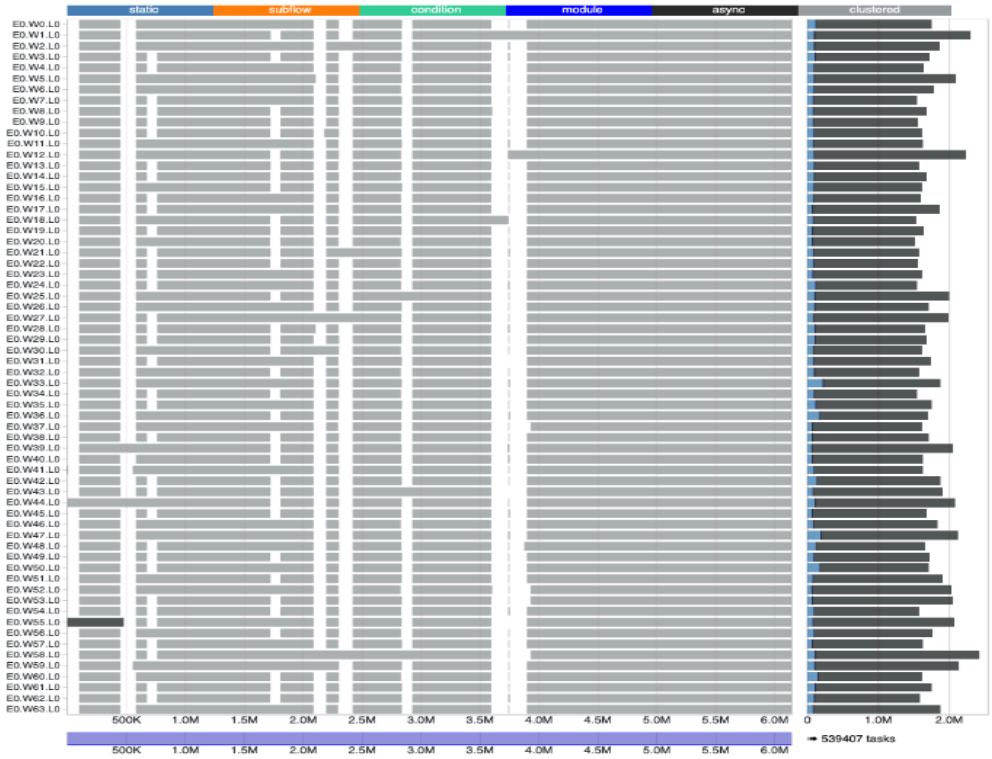
Results on PSC



Speedup graph with parallel cutoff set to 500,000 using hybrid tasking

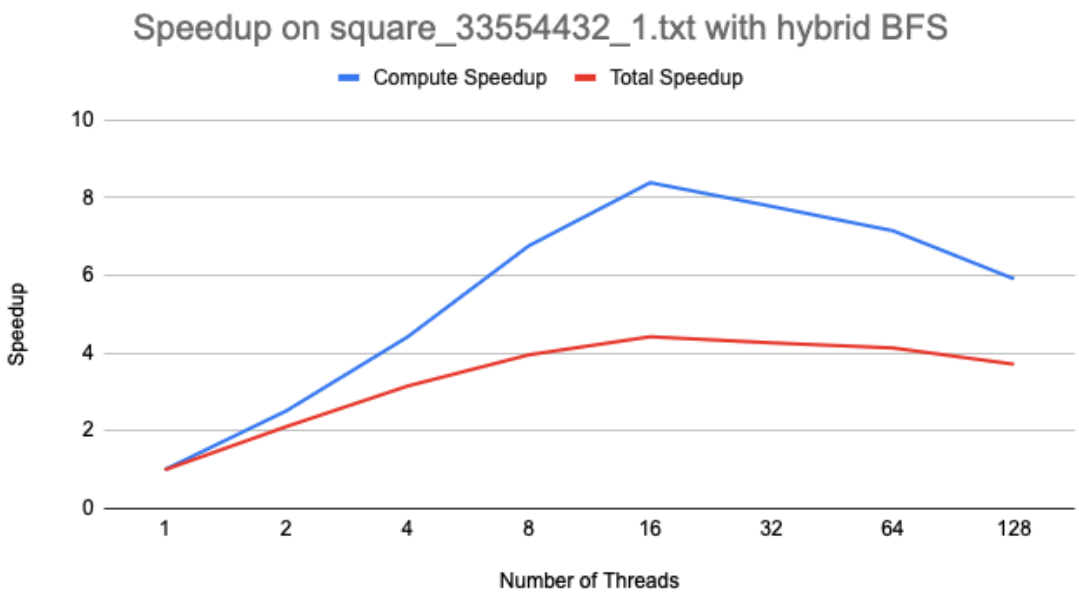
The speedup on PSC scales well up to 8 threads, as expected. However, we notice that the speedup actually starts to diminish once the number of threads increases further, peaking at 8.4

on 16 threads. One reason for this plateauing behavior, which we explore further in the next section of the report, is that the problem size is simply too small. At larger thread counts, the overhead of the dynamic task scheduler and its synchronization becomes much larger relative to the miniscule amount of work that is being done by each thread. Indeed, this is supported by the profiler output below, which shows that overall utilization is still quite ideal even at 64 threads.



Profiling results of 64 threads running on square_33554432_1.txt

Note that the gap running between 3.5M and 4.0M microseconds shows the transition from parallel filter to sequential filter, after which tasks are small enough that there is no noticeable idle time per thread. The interspersed gaps before the cutoff transition can still be attributed to the cost of allocating large result vectors in the parallel filter function.

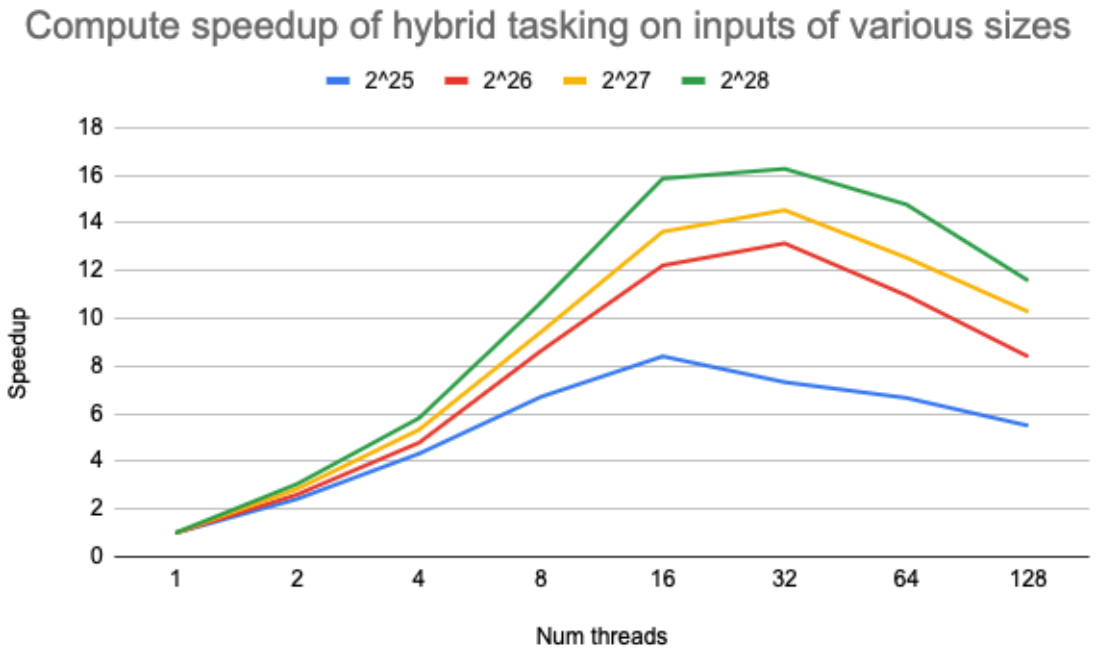


Speedup graph with parallel cutoff set to 500,000 using hybrid BFS

We observed very similar scaling trends with the BFS-based approach as compared to the task-based approach with the caveat that, just like on the GHC machines, it was slightly worse. We again peak in both total speedup and compute speedup at 16 threads. We tested this approach on a test case of size 2^{26} points as well, and found that we scaled better, reaching peak speedup at 32 rather than 16 threads. The overhead incurred by Taskflow means that this test size is too small to maximize speedup, no matter the approach taken.

Problem Size Scaling

To investigate the sensitivity to problem size, we decided to also measure the performance of our implementation on similar test cases with more points.



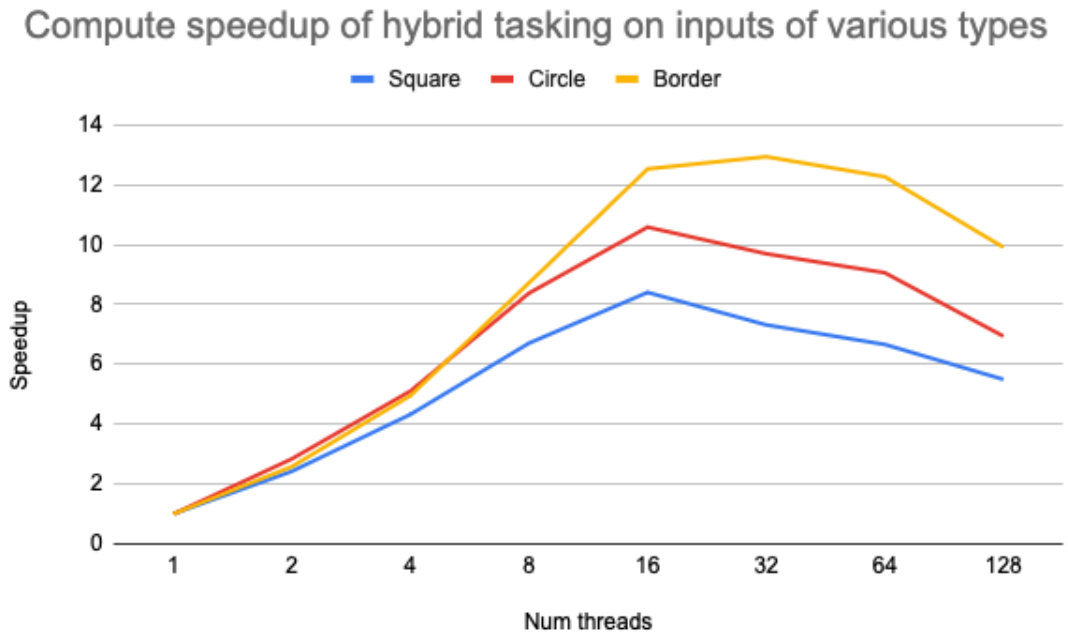
Speedup graph of hybrid tasking with varying input sizes

From these graphs, it is clear that our implementation scales better on larger test cases, reaching a maximum speedup of 16.26 on 2^{28} points with 32 threads. It is also interesting to note that the optimal thread count for a given input increases as the input size increases. Based on this trend, we can expect the scalability to keep improving with even larger inputs, but we decided to stop here given how long it takes to generate even a single test case at this scale.

Scaling by Test Case Type

The test cases in our testing suite can be divided into three different types. The square test cases were generated by repeatedly sampling points within the $[-10^6, 10^6] \times [-10^6, 10^6]$ square, regenerating a given point if it is the same as an earlier one. The circle test cases were generated similarly to the square test cases, but instead sampling points from the circle centered at the

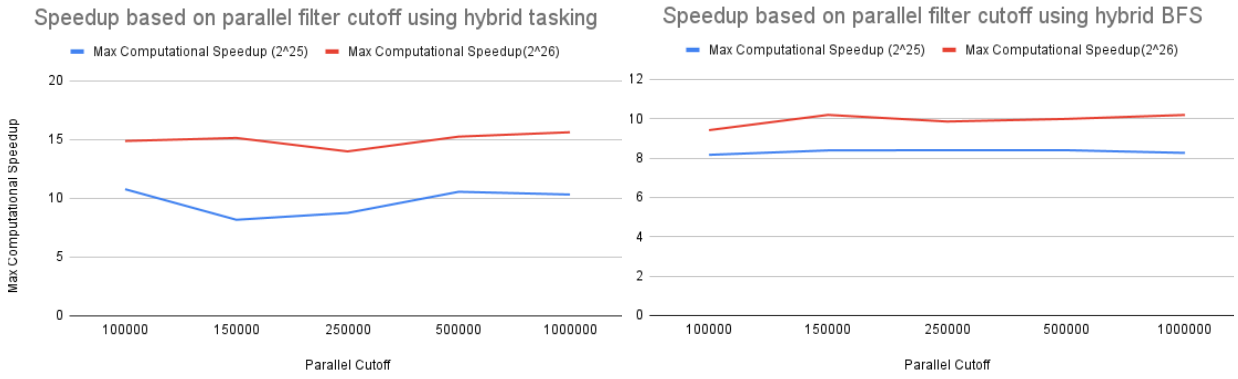
origin with radius 10^6. This causes there to be more points on the convex hull on average. Finally, the border test cases are generated by picking random points from the ring consisting of all points between 995,000 and 10^6 distance away from the origin. This significantly increases the expected number of points on the convex hull. Interestingly, the scaling of our implementation varies considerably depending on the test case type.



Speedup graph of hybrid tasking on various test cases types with 2^{25} points

Specifically, the code has better speedup on circle tests than square tests and better speedup on border tests than circle tests. Based on our results from the problem size scaling section, we believe that this improvement in speedup can be attributed to the fact that there is more overall work to do in the circle cases than the square cases, which effectively makes it similar to a bigger test case. The same can be said for why the implementation performs better on border tests than circle tests, with the added benefit that border cases don't have many ridges with large visibility sets so we generate many small tasks that can be executed in parallel earlier on in the execution.

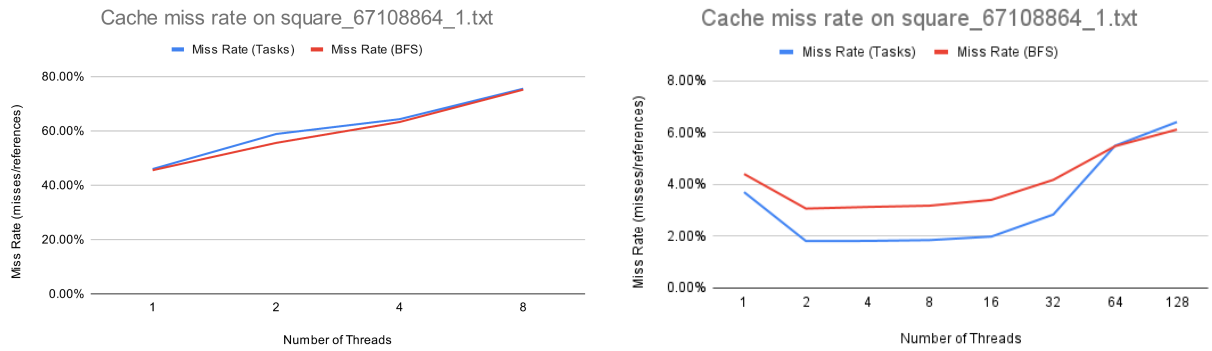
Sensitivity to Parallel Cutoff



Sensitivity to parallel cutoff

We found that changing the cutoff at which we switched from purely sequential to parallel filtering generally had very little effect on the computational (and thus total as well, omitted for brevity) speedup. In general, there was no real trend observable—it stayed approximately the same no matter the parallel filter threshold selected. This holds for both implementations (BFS and task-based) with minor variations. In general, the 1M threshold seemed to have better relative performance on the task-based approach (when compared to other thresholds) vs. on the BFS approach. This matches our intuition, since the lack of additional dependencies in the task-based approach should allow the scheduler to take advantage of the fact that less processors are being used for filtering and allocate them towards processing more ridges sooner. In contrast, those enforced dependencies in the BFS-scheduled version mean that in the early rounds, when the number of tasks is few and the size of sets to be filtered is relatively large, we see a greater work imbalance. Because of this, we would expect to see an improvement in speedup on the BFS version as we shrink the parallel cutoff. The fact that we don't is likely attributable to the overhead that every parallel filter incurs. Having to do more parallel region launches balances out the slightly better work balance.

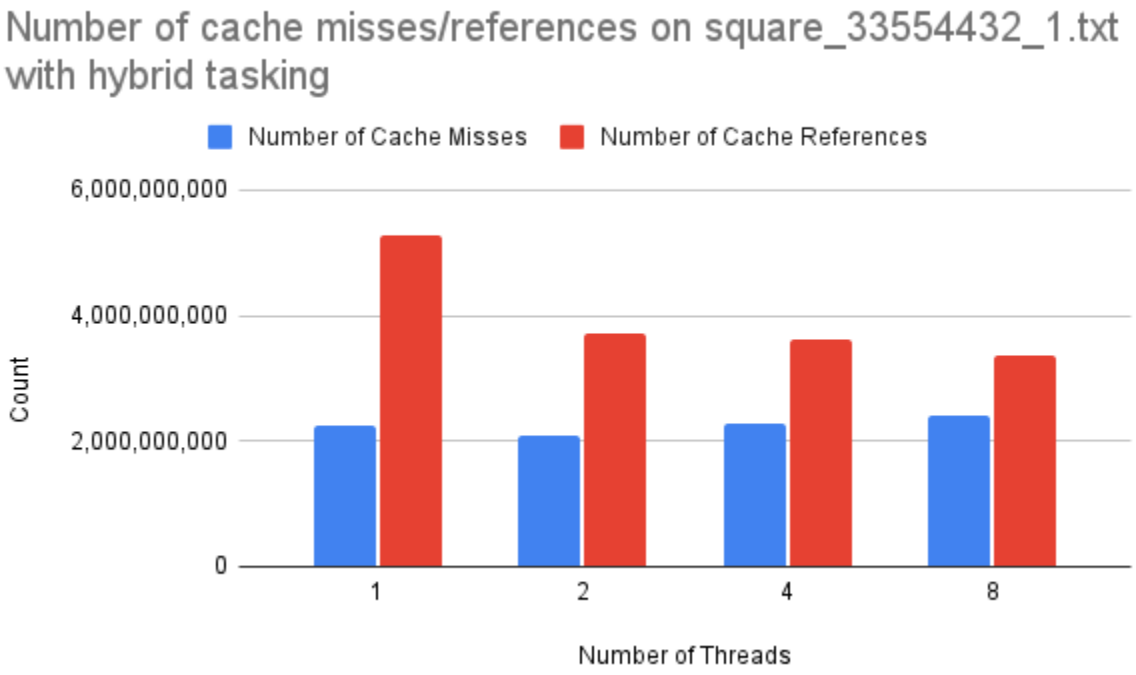
Cache Locality



Cache miss rate on GHC (left) & PSC (right)

We observed highly different cache behavior between systems. On the Apple M4 Pro (not presented here) and the Epyc 7742 used in Bridges 2, we found that the miss rate was relatively low and dropped when increasing from 1 thread to 2 threads. We believe that this drop is due to the increased cache locality as each processor handles a smaller fraction of the work and total cache size increases, which helps to explain the super-linear scaling that we observed at low thread counts on all systems. At high thread counts, we see that it begins to increase again. This can again be attributed to locality behavior—at very high thread counts, individual processors begin to get ridges that are more and more separated, leading to each processor needing to access more unique visible sets and so having more misses. We also saw that the tasked approach is significantly more cache-efficient than the BFS approach. This intuitively makes sense—particularly for small tasks, if we process the adjacent ridges immediately after the current one on a particular processor (which is a possibility), we would have the visible set, etc. still in cache. On the other hand, the BFS approach will never have this occur due to the enforced segregation between layers.

In contrast to the aforementioned platforms, we saw very different cache behavior on the GHC machines. The percentage of cache misses starts significantly higher, and climbs as the number of threads increases.



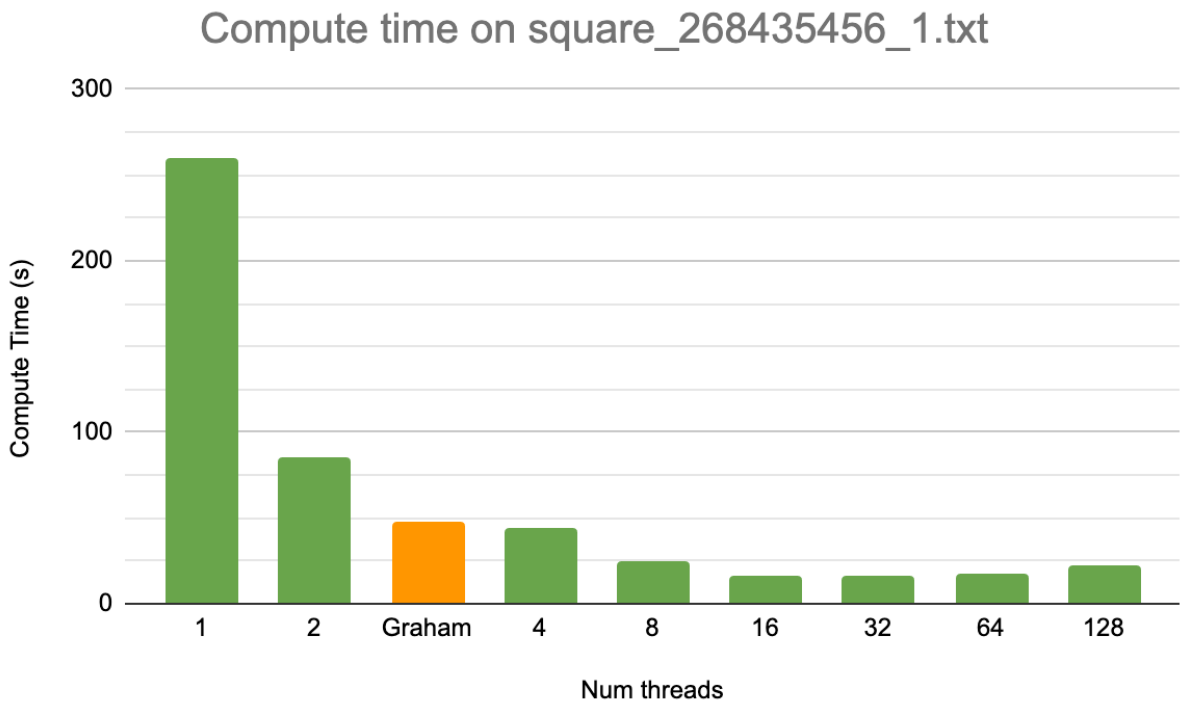
Cache misses/references on GHC

Interestingly, the absolute number of cache misses stays almost identical between iterations—the total number of cache references just drops significantly as the thread count increases. There are a couple of reasons for the difference in cache performance on GHC. For one, the GHC machines' cache size is simply much smaller. GHC's Intel i7-9700 processors have 12 MB of shared L3 cache, while the two AMD EPYC Rome 7742 processors on the PSC nodes each have 256 MB of shared L3 cache. The M4 Pro chip has a largely different architecture without an L3 cache, but instead has a total of 36 MB of shared L2 cache. The drop in the number of cache references on all platforms can be attributed to the difference in sorting behavior. Our filtering algorithm (when operating in its two-vector form to find the new visible set) sorts the points to

be filtered as a first step. When only one thread is available (program wide), Taskflow defaults to `std::sort` to sort those points. Otherwise, it uses a parallel version of pattern-defeating quicksort. It turns out that `std::sort` has significantly more memory accesses, which explains the drop in reference count. On all other platforms, misses also drop. On GHC, however, the cache size is so small that we are constantly getting capacity misses in our cache, harming scaling.

Comparison to Graham Scan

To demonstrate the scalability of our parallel implementation of Blelloch's algorithm, we plotted its compute times on various thread counts against our baseline sequential Graham scan implementation.



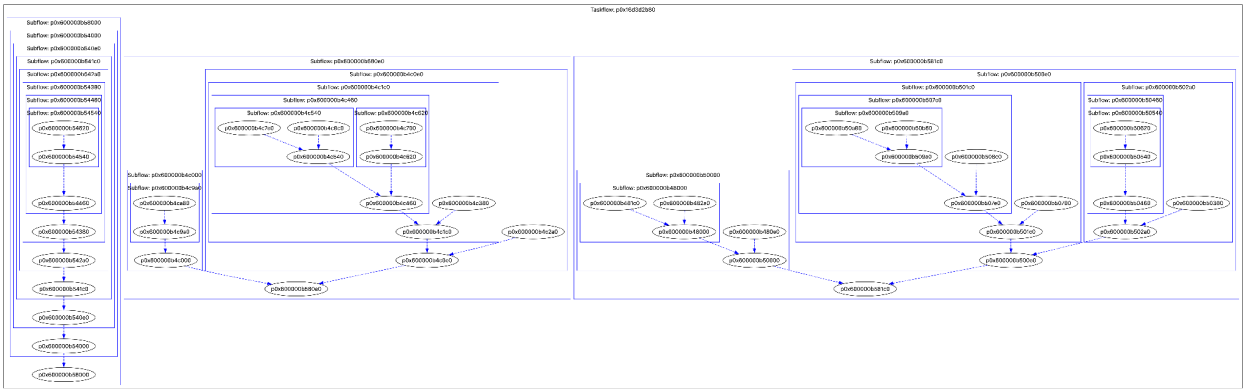
Compute time of Graham scan vs. parallel code with varying thread counts

As we can observe from the figure above, there is a roughly 5x constant factor overhead in the parallel algorithm compared to the sequential algorithm, due to the inherent cost of recursion as well as the frequent accesses to hashmap data structures in the implementation of the parallel algorithm. However, when the parallel algorithm is executed on 4 or more threads, the speedup from parallelism is enough to overcome the constant factor overhead and beat the efficient sequential algorithm. The performance continues to improve up until the optimal point at 32 threads, where the parallel algorithm is over 3 times faster than the sequential algorithm. Thus, we have shown that it is indeed beneficial to use our parallel implementation over an efficiently implemented sequential algorithm on multicore computer architectures.

Discussion

Limits on Parallelism

There were 3 key limits on the parallelism of our algorithm, and thus the speedup we were able to achieve. Namely, there is a fundamental algorithmic limit (which we did not run into but is worth mentioning), there is the overhead of parallel constructs, and finally there is the contention on our data structures.



Dependency graph of tasks (corresponding to calls to process_ridge) in the tasked approach

As mentioned in our summary, the length of the longest chain of dependencies in the call graph is $O(\log n)$ w.h.p. In theory, this (times the work per call) should be a lower bound on our speedup. In practice, this bound can only be achieved with a number of processors significantly larger than what we had available. Thus, while this is a hard lower bound on the best possible parallelism that could be achieved, in practical terms this did not limit our parallelism due to the limited number of processors we had available.

The second (and much more relevant) limitation on our parallelism is the overhead incurred by the parallel constructs we employed. We measured the time taken to launch both synchronous and asynchronous tasks/taskflows. Synchronously launching a flow from a worker thread (what we do with filter) takes ~30µs, while launching an asynchronous task (i.e. a call to `process_ridge`) takes ~6µs at best, and up to ~30µs at worst. Given that we launch >500,000 tasks for some test cases, this adds up very quickly and becomes a limit on our scaling. We believe that this overhead is in large part attributable to the cost of the dynamic scheduler employed by Taskflow, and particularly impacts us towards the end of the runtime when tasks take a very small amount of time but there is a large number of them. Unfortunately, we were not able to directly measure this, but it seems like a likely source.

The third limitation is contention for our core data structures. As mentioned in our summary, there are 3 key data structures (**C**, **M**, and **F**) that all threads regularly contend for. By carefully selecting which hash table implementation was used for **C** and **M**, we were able to minimize the impact of contention. In particular, we set the number of internal tables in both **C** and **M** to 128.

Because each internal table has its own individual mutex rather than there being a single global one, this minimizes the synchronization stall incurred by threads waiting for access. With 128 internal tables, we expect that the amount of synchronization stall should be minimal, since our choice of hash function means that keys are distributed approximately evenly over the internal tables. Thus, we expect that the number of threads trying to concurrently access is going to be limited. A basic probabilistic analysis (this is just the birthday problem) allows one to see that if 16 threads are all simultaneously trying to write to one of these tables, the expected number of collisions is less than 2. We use a hash set for **F** for similar reasons, allowing us to minimize the impacts of contention. By making these optimizations, we are able to limit the extent to which data structure contention hurts our performance.

Overall, the core limit on the parallel speedup we are able to achieve is the overhead. This is not something we are able to influence to any significant extent, since it is an inherent constraint of whatever parallel library is chosen.

Comparison Between Task vs. BFS approach

As mentioned in prior sections, we tried two scheduling approaches. Namely, the task based approach, which allows the scheduler a great deal of flexibility on the order in which recursive calls are scheduled, and the BFS approach. Fundamentally, these two approaches should be comparable in performance. In the limiting case of infinite or near-infinite processors (where our speedup is limited by the span bound), they reduce to the exact same thing. The exact same tasks get executed, it is only just the order in which they occur that varies.

The performance differences we observed can be attributed to several factors. The key driver of the performance difference is the additional ‘layer’ dependencies mentioned above. These are possible to avoid. Using a RW queue to store ridges as they occur and adding a task for each one would allow removing this and should (less minor impacts due to cache locality) minimize the performance difference. Unfortunately, we were not able to implement this in the time available but it is a potential direction for future improvement. The other primary driver is the difference in cache efficiency (as can be seen in the Cache Locality section of our results). Due to the difference in scheduling order, the tasked approach seems to be significantly more cache efficient, needing to read from main memory much less often.

Reflection

Overall, we were satisfied with the performance we were able to achieve. Our code scales very well at low thread counts, achieving **super-linear scaling** early on due to large gains in cache locality. While we did run into some setbacks (including issues with the OpenMP scheduler leading to us having to switch parallelism libraries), we were able to reach all of our core goals and performance targets. Overall, we learned a great deal from this project. We were able to improve our performance debugging skills, learned more about the OpenMP library and its constraints, and learned a new, cutting edge parallel library that allowed us to achieve much better performance.

References

[1] A. M. Andrew, ‘Another efficient algorithm for convex hulls in two dimensions’, Information Processing Letters, vol. 9, no. 5, pp. 216–219, 1979.

[2] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun, ‘Randomized Incremental Convex Hull is Highly Parallel’, in Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 2020, pp. 103–115.

[3] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, ‘Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System’, IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 6, pp. 1303–1320, Jun. 2022.

Work Distribution

Overall, both teammates contributed equally to the trajectory of the project. Edward initially set up the testing suite and implemented the sequential Graham scan algorithm. Cole then wrote a single-threaded implementation of the parallel algorithm. The process of parallelizing the baseline single-threaded implementation was collaborative. Edward wrote the initial OpenMP implementations, while Cole focused on in-depth profiling of these to try and improve performance. Edward also wrote the tasked version of the Taskflow implementation and came up with the hybrid parallelism idea and implementation, while Cole implemented the BFS version and the initial parallel filtering code. Both partners then wrote the report and created the poster together.

Since we feel that both partners had equal contribution to the project, we would like to proceed with a 50/50 split for the available credit.