Parallel Quickhull - Computing the convex hull of points on the plane

Overview

Finding the convex hull of points is a fundamental problem in Computational Geometry with a wide range of applications. Quickhull[1] is a well-known convex hull algorithm and is widely used in practice to compute convex hulls. Although it has a worst-case time complexity of $O(n^2)$, in the average case(assuming a uniform distribution of points), it is expected to run in O(n), which is why it works so well in practice.

The goal of this project is to implement a parallelized version of the Quickhull algorithm. The algorithm will be implemented in the shared-memory model, where processes read and write from a single global address space. We want the parallel algorithm's running time on a single processor(i.e., the *work* of the parallel algorithm) to be the same as that of the Quickhull algorithm. We also want a theoretical near-linear speedup with the number of processors, although, in practice, we may not achieve this due to issues of data locality.

Background

Parallel algorithms analysis

The running time of parallel algorithms is represented using the terms *work* and *span*. Both *work* and *span* will be represented in asymptotic notation.

Work is defined as the number of steps of computation it takes to run the algorithm on a single processor.

Span is defined as the number of steps of computation it takes to run the algorithm, assuming we have an unbounded number of processors that can run in parallel.

We note that the *work* of a parallel algorithm is the same as the running time in classic sequential algorithms analysis.

Parallel programming primitives

There are a few primitive parallel operations used in this algorithm which are described here.

Parallel for-loops

Parallel for-loops are used to perform a set of operations in parallel. For example, given a list of *n* points, we can compute the distance of each point from a given line in parallel, by spawning a separate process to do the same for each point. This operation would take *O(n)* work and *O(n log(n))* span. The *O(log(n))* span is due to the time taken to launch *n* processes. If we assume a binary forking model, where a single process spawns 2 new processes, and this goes on recursively until we have *n* processes, it would take *O(log(n))* span to spawn all *n* processes.

Parallel reduction

Reductions are used to compute a single value by processing a set of data in parallel. For example, given a list of *n* points, we can compute the point that is farthest away from a given line using parallel reduction. We use a parallel for-loop to compute the distance of all points from the given line in parallel. After doing this, the maximum distances computed by each process are aggregated together to obtain the final result. This operation also costs *O(n)* work and *O(log(n))* span, similar to the cost of parallel for-loops.

Parallel prefix sum

Given an array of numbers, the prefix sum is an array of the sum of all of its prefixes. For example, given an array:

1 2 3 4 5

Its prefix sum would be:

1 3 6 10 15

Computing the prefix sum in parallel is a widely studied problem[2] and can be done in *O(n)* work and *O(log(n))* span.

Filtering elements using prefix sum

Given a set of elements, we can use the parallel prefix sum algorithm to efficiently obtain the subset of elements that satisfy a given condition. For example, consider the problem where we are given a list *P* of *n* input points, and we want to obtain a list of all of the points that lie on the left of a given line *L.*

1. Using a parallel for-loop, initialize an n-sized array *A* of zeros
2. Using a parallel for-loop, for each point *P[i]*, set *A[i]* = 0 if *P[i]* is on the left of *L*
3. Set A to be its prefix sum using the parallel prefix sum algorithm
4. Allocate a new array *PL* of size *A[n]*
5. Using a parallel for-loop, for each point *P[i]*, set *PL[A[i]] = P[i]* if *P[i]* is on the left of *L*

We can see that the above algorithm costs us *O(n)* work and *O(log(n))* span.

Algorithm

This section contains the pseudo-code for the Parallel Quickhull algorithm. Assume we take as input *P* - a list of *n* points on the plane, and output *C* - the list of points on the convex hull of *P*.

1. Allocate *CH,* an array of numbers of size *n.* Use a parallel for-loop to fill it with zeros.
2. Find *L*, the point in *P* with the smallest x-coordinate using parallel reduction.
3. Find *R*, the point in *P* with the largest x-coordinate using parallel reduction.
4. Set *CH[L] = 1* and *CH[R] = 1.*
5. Compute *U*, the list of points on the left(i.e, the points above) of line *LR* using the parallel filtering primitive.
6. Compute *B*, the list of points on the right(i.e, the points below) of line *LR* using the parallel filtering primitive.
7. Execute upper_hull(*U, LR)* and upper_hull(*B, RL)* in parallel.
8. Compute *C,* the list of points in *P* such that *CH[P[i]] = 1* using the parallel filtering primitive.
9. Sort the points in *C* by angle
10. Return *C*

Upper hull

The pseudo-code for upper_hull(*P, LR),* where *LR* is a line and *P* is a set of points above the line is given below.

1. Find *F*, the point in *P* that is farthest away from *LR* using parallel reduction.
2. Set *CH[F] = 1.*
3. Compute *LU*, the list of points on the left(i.e., above) line *LF* using the parallel filtering primitive.
4. Compute *RU*, the list of points on the left(i.e., above) line *FR* using the parallel filtering primitive.
5. Execute upper_hull(*LU, LF)* and upper_hull(*RU, FR)* in parallel.

Analysis

In this section, we derive asymptotic bounds on the *work* and *span* of the Parallel Quickhull algorithm. We see that apart from the cost of the computing upper hull and sorting the points of the convex hull at the end, the algorithm costs *O(n)* work and *O(log(n))* span. We will now analyze the *work* and *span* of the upper hull subroutine.

The cost of sorting the points on the convex hull is discussed at the end.

Work

We see that each level of recursion costs *O(n)* work. We convince ourselves that the algorithm functions in an identical way to that of the Quickhull algorithm[1].

In the worst case, the recurrence relation would be

$T(n) = T(n - 1) + n$

Which translates to a worst-case upper bound of $O(n^2)$ work.

In the average case, about a quarter of the points will lie on the left of the line *LF* and a quarter of the points on the right of the line *LF*. The average case recurrence relation will be

$T(n) = 2T(n/4) + n$

Which translates to an average-case upper bound of O(n) work.

Span

We can see that each level of recursion costs O(log(n)) span.

In the worst case, the recurrence relation for the span is

$T(n) = T(n - 1) + \log(n)$

Which translates to a worst-case upper bound of O(n log(n)) span.

In the average case, we obtain a recurrence relation in the same way as we did for work.

$T(n) = 2T(n/4) + \log(n)$

Which translates to an average-case upper bound of $O(\log^2(n))$ span.

Cost of sorting

We claim that the cost of sorting the list of points on the convex hull at the end of the algorithm is asymptotically upper-bounded by the cost of the upper hull algorithm.

In the worst case, there can be *n* points on the convex hull. The cost to sort these points is *O(n log(n))* work and *$O(\log^2(n))$* span using an optimal parallel sorting algorithm[3].

Since the span of parallel sorting in the worst case is already as good as our average-case upper hull bounds, let us look at work.

We can think of this worst-case scenario happening when none of the input points falls inside the triangle *LFR* during the upper hull algorithm. The lower bound for work to find the upper hull, in that case, is:

T(n) = 2T(n/2) + n

Which translates to *O(n log(n))* work.

Therefore, both the work and span of parallel sorting are upper-bounded by the cost of computing the upper hull.

In practice, the number of points on the convex hull of a uniform distribution of *n* points is often much smaller. On running the Parallel Quickhull algorithm on 100 million points, the number of points on the convex hull was found to be around 60.

Running time with *p* processors

When we have a fixed number of *p* processors, the running time will be:

$$\frac{T_0}{p} + T_\infty$$

Where T_0 is the time to compute on 1 processor(work) and T_∞ is the time to compute on an unbounded number of processors(span).

In the worst case, this solves to

$$\frac{n^2}{p} + n\,log(n)$$

In the average case,

$$\frac{n}{p} + log^2(n)$$

Assuming $\frac{n}{p}$ is much larger than $log(n)$ and $log^2(n)$,

The worst-case running time is:

$$\frac{n^2}{p}$$

The average-case running time is:

$$\frac{n}{p}$$

Since $O(n^2)$ and $O(n)$ are respectively the worst-case and average-case running times of sequential Quickhull, this gives us a linear speedup with respect to the number of processors within our assumptions.

Experimental results

The parallel quickhull algorithm was implemented using C++ and the OpenMP library. A sequential quickhull algorithm was also implemented for comparison. They were executed on the Stampede 2 supercomputer on a single machine having 64 CPU cores.

Number of points	Sequential Quickhull running time	Parallel Quickhull running time with 1 CPU core	Parallel Quickhull running time with 64 CPU cores
1 million	790 ms	920 ms	172 ms
10 million	8672 ms	9523 ms	2126 ms
100 million	94334 ms	108132 ms	21254 ms

We observe about a 5x increase in running time when using all 64 CPU cores for 100 million points. This is not quite the linear speedup we were hoping for. This can partly be attributed to cache data locality. Since each processor core maintains its own cache, a lot of overhead is incurred due to the processor cores needing to re-read data onto its cache at every level of the recursion. Optimizing the code for cache data locality to improve running time can be taken as future work on this project.

The code is available at https://github.com/balaji97/parallel-quickhull

Conclusion

We have seen a parallel implementation of the Quickhull algorithm that provides a near-linear theoretical speedup and performs well in practice. Further work can be done to optimize the algorithm better for cache data locality, and to identify other tweaks to make the algorithm run faster.

References

1. The Quickhull Algorithm - https://www.cise.ufl.edu/~ungor/courses/fall06/papers/QuickHull.pdf
2. NVIDIA CUDA documentation on Parallel Prefix Sum - https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

3. A parallel sorting algorithm - https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.464.7118&rep=rep1&type=pdf