



ConcurrentHull: A Fast Parallel Computing Approach to the Convex Hull Problem

Sina Masnadi and Joseph J. LaViola Jr.

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

October 27, 2020

ConcurrentHull: A Fast Parallel Computing Approach to the Convex Hull Problem

Sina Masnadi and Joseph J. LaViola Jr.

University of Central Florida, Orlando, FL, USA
{masnadi, jjl}@cs.ucf.edu

Abstract. The convex hull problem has practical applications in mesh generation, file searching, cluster analysis, collision detection, image processing, statistics, etc. In this paper, we present a novel pruning-based approach for finding the convex hull set for 2D and 3D datasets using parallel algorithms. This approach, which is a combination of pruning, divide and conquer, and parallel computing, is flexible to be employed in a distributed computing environment. We propose the algorithm for both CPU and GPU (CUDA) computation models. The results show that ConcurrentHull has a performance gain as the input data size increases. Providing an independently dividable approach, our algorithm has the benefit of handling huge datasets as opposed to other approaches presented in this paper which failed to manage the same datasets.

Keywords: Convex Hull · Parallel Algorithms · CUDA.

1 Introduction

Computing the convex hull of a set of points is one of the substantial problems in computer graphics. The convex hull of a set of points P is the smallest convex polygon/polyhedron that contains the points and is denoted by $CH(P)$. Imagine using needle pins on a surface to represent the points, the convex hull can be visualized by wrapping a rubber band around these needles [4]. Given a set of n points in a plane, the convex hull of these points is the set of points which are located on the perimeter of the smallest area that contains all the points. The convex hull of a set of 3D points P is the smallest convex polyhedron containing all points of the set P .

The complexity of the algorithms are based on the total number of points (n) and the number of points that create the hull (h). Among these algorithms, the most popular ones are the “Graham scan” algorithm and the “divide-and-conquer” algorithm [9].

The $O(n \log n)$ complexity of Graham Scan algorithm is emanated from the radial sort of the points. The algorithm starts with one of the points that is sure to be in the convex set which is called P . For this purpose it will find the point with the lowest y-coordinate, if there exists more than one point with the minimum y-coordinate then it will find the one with the lowest x-coordinate. This can be done in $O(n)$ time. Then the rest of the nodes will be sorted by

the angle they make with P and x-axis. The next step is figuring out if the next point is a right turn or a left turn regarding its preceding point and P . If it is a right turn (clockwise), then it means that p_2 is not a part of the convex hull and it will be swapped with p_3 and the algorithm will proceed to the next set of points. This process will end when it reaches the starting point P .

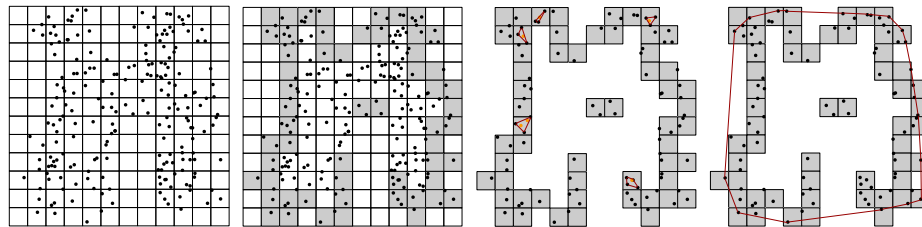
2 Related Work

The research on implementing parallel algorithms is scarce. In 1988, Miller et al. introduced a parallel solution for solving the convex hull problem which given a sorted set of points and a machine with $\theta(n)$ processors can solve it in $\theta(\log n)$ [8]. However the high amount of required processors reduces the practicality of this algorithm. Bleloch et al. introduced a new definition and analysis of the configuration dependence graph for the randomized incremental convex hull algorithm and showed it is inherently parallel with $O(\log n)$ dependence depth [3].

The other approach is using GPU processing power for solving the problem. Jurkiewicz et al. introduced a CUDA algorithm which applies quick sort on the points and solve the problem in $O(k \log g)$. $g = \frac{n}{p}$ input size of the problem per SIMD-processor and $k = \frac{n}{v}$ input size of the problem per scalar processor [6]. GHull is a 3D CUDA-based convex hull algorithm that was proposed by Gao et al. which is up to 10x faster than the fastest CPU convex hull software, QuickHull [5]. CudaHull is another 3D convex hull algorithm which is based on the QuickHull approach [10].

Hybrid GPU-CPU is another approach to the convex hull problem Tang et al. presented a hybrid CPU-GPU algorithm to compute the convex hull of points in three or higher dimensional spaces [11]. The GPU is used to remove the points that do not lie on the boundary, then the CPU computes the convex hull for the remaining points.

Here we explain our approach for 2D and 3D convex hull and for each of them, we first describe the approach for CPU-based implementation and then the CUDA implementation approach. We will then discuss the benchmarking task and how our algorithm performed against other algorithms.



(a) Partitioning the input points (b) Grey cells are valid partitions. (c) Convex hull of valid partitions. (d) Convex hull of finalized points.

Fig. 1: A demonstration of the 2D ConcurrentHull

3 ConcurrentHull

Our approach is based on highly parallel pruning of the interior points. The data is partitioned into an equal sized grid. The pruning is done by the crawlers. Each crawler can perform individually from other crawlers. A crawler starts from a partition in a given direction and tries to find the first valid partition and then it halts. We will discuss the implementation of the 2D and 3D ConcurrentHull algorithm for CPU and GPU.

4 2D Convex Hull

The following steps are a high-level presentation of how we are going to find the convex hull of 2D points:

- Loading the data and partitioning them using a grid
- Pruning the partitions using *crawlers*
- Finding convex hulls of remaining partitions and removing the points inside their convex hulls
- Calculating the final convex hull using the remaining point

Figure 1 shows a demonstration of 2D ConcurrentHull algorithm.

4.1 CPU 2D Convex Hull

Calculating the convex hull for a set of 2D points starts with loading the points and finding their extreme X and Y values. In the next step, the plane will be divided to a $k \times k$ grid. Each grid cell is called a *partition*. The partition which is located on i and j coordinates of the grid is represented by $\rho_{k \times i+j}$. A *valid partition* is a partition that we assume has at least one point in the convex hull, but there might be no point inside a valid partition that is a part of the final convex hull. In Figure 1 the example set of points has been divided into a 12×12 grid.

In order to prune the partitions, we use crawlers. A *crawler* is a piece of code which crawls from a starting point towards a defined direction with a width of one partition. It checks if any partitions on its path have any points inside them. It will mark the first non-empty partition that it finds as valid partition and stops crawling. Three crawlers start from each partition of each side of the grid, one perpendicular, the other two in the direction of the grid diagonals (Figure 2). The pruning is done using $12 \times (k - 2) + 4$ crawlers which run in parallel. Since each corner of the grid only has one diagonal crawler, we have $3 \times (k - 2)$ crawlers for each side of the grid (excluding the corners) and 4 crawlers for the corners. Using crawlers we will find valid partitions and the rest of the partitions will be pruned. Figure 1b shows the sample pruned partitions, the white partitions will be removed. Algorithm 2 shows how a crawler works.

Convex hull for each partition is computed in parallel and the points inside the convex hull are removed. If a partition has less than four points, it will be

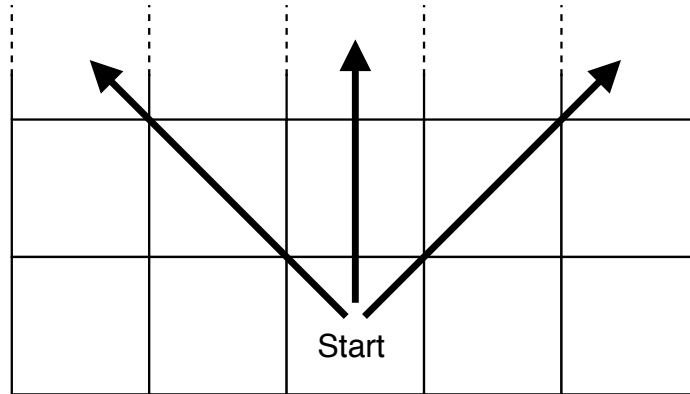


Fig. 2: The figure shows the directions for three 2D crawlers of a starting point. Each arrow represents one crawler's direction.

left untouched as a valid partition (note that at least three points are needed to compute a convex hull). Each partition's points is passed to a thread and each thread finds the $CH(\rho_v)$ using the *Quick Hull* algorithm. Points inside the convex hulls of the partitions can be removed as they are already surrounded by a convex hull inside the final convex hull. Figure 1c shows the calculated convex hull for the valid partitions. Yellow points will be removed from the set. The final convex hull will be computed using the remaining points.

The justification for using partitions and crawlers for pruning instead of using the Quick Hull algorithm in the first step is that the Quick Hull algorithm finds the points that lie inside the triangle[2] by comparing all of the points with the three lines that are created by the extreme points and although the operation is of $O(n)$, it is still a time-consuming task. On the other hand, to prune the points using partitions and crawlers, we only do one comparison in each crawler's step ($\text{partition.points.size()} > 0$) which is faster and in the worst case is of $O(12k^2)$ in which k is a constant.

Lemma 1. *The final convex hull points are inside the valid partitions and none of the points from the final convex hull can be inside the pruned partitions.*

Proof. We prove this lemma by contradiction. Consider a set of 2D points P divided to $k \times k$ partitions. We denote $\bigcup_v CH(\rho_v)$ as CH' . If there exists a pruned point $p_p \in CH(P)$ and $p_p \notin CH'$. This means p_p is outside CH' . If it is outside the CH' , crawlers have missed the partition (ρ_p). If there is a valid partition on the right of ρ_p , because we assumed p_p is outside the CH' there has been a crawler crawling from left to right that picked the partition on the right of ρ_p as a valid partition, but since crawlers stop when they find the first partition with at least one point inside, it should have stopped when it reached ρ_p and it is a contradiction to the crawler definition. The same case applies to the other directions around ρ_p .

Algorithm 1 Crawler2D

```

1: procedure CRAWL
2:    $K \leftarrow \text{sidesize}$ 
3:    $\text{start}_i \leftarrow \text{startpoint}_x\text{coordinate}$ 
4:    $\text{start}_j \leftarrow \text{startpoint}_y\text{coordinate}$ 
5:    $\text{direction}_i \leftarrow \text{direction}_x\text{coordinate}$ 
6:    $\text{direction}_j \leftarrow \text{direction}_y\text{coordinate}$ 
7:    $\text{current}_i \leftarrow \text{start}_i$ 
8:    $\text{current}_j \leftarrow \text{start}_j$ 
9:   loop:
10:  if  $\text{size}(\text{partition}(\text{current}_i, \text{current}_j)) > 0$  then
11:     $\text{validPartitions.append}(\text{partition}(\text{current}_i, \text{current}_j))$ 
12:    return
13:  else
14:     $\text{current}_i \leftarrow \text{current}_i + \text{direction}_i$ 
15:     $\text{current}_j \leftarrow \text{current}_j + \text{direction}_j$ 
16:    if  $\text{current}_i < K$  and  $\text{current}_j < K$  then
17:      goto loop.

```

For the points inside the partitions convex hulls, they cannot be outside the $CH(P)$ because if they are outside $CH(P)$ they should also be outside of the convex hull of a partition, which is a contradiction to the assumption. \square

4.2 GPU 2D Convex Hull

We apply the CPU approach for CUDA with some minor modifications. A CUDA kernel is responsible for partitioning the points. After copying the points data to the GPU memory, n threads assign the points to their partitions. Based on the thread ID, each thread picks one point and assign it to its appropriate partition using side size and extreme points values.

GPU crawlers use the same approach as the CPU crawlers. $12 \times k$ threads run the crawler kernel, each thread crawls in its direction and marks its appropriate valid partition.

To compute the convex hull on GPU, we modified CudaChain algorithm [7] to fit our needs. The points data in GPU memory are shared among CudaChain instances which helps algorithm to run faster without the need to make multiple copies of the data. After computing the convex hull of the partitions and removing the points inside them, the remaining points will be used to calculate the final convex hull.

5 3D Convex Hull

Applying the ConcurrentHull idea to 3D points can be done using the same steps as the 2D problem but with minor modifications to meet the 3D problem requirements. We will first describe the CPU algorithm and then the GPU one.

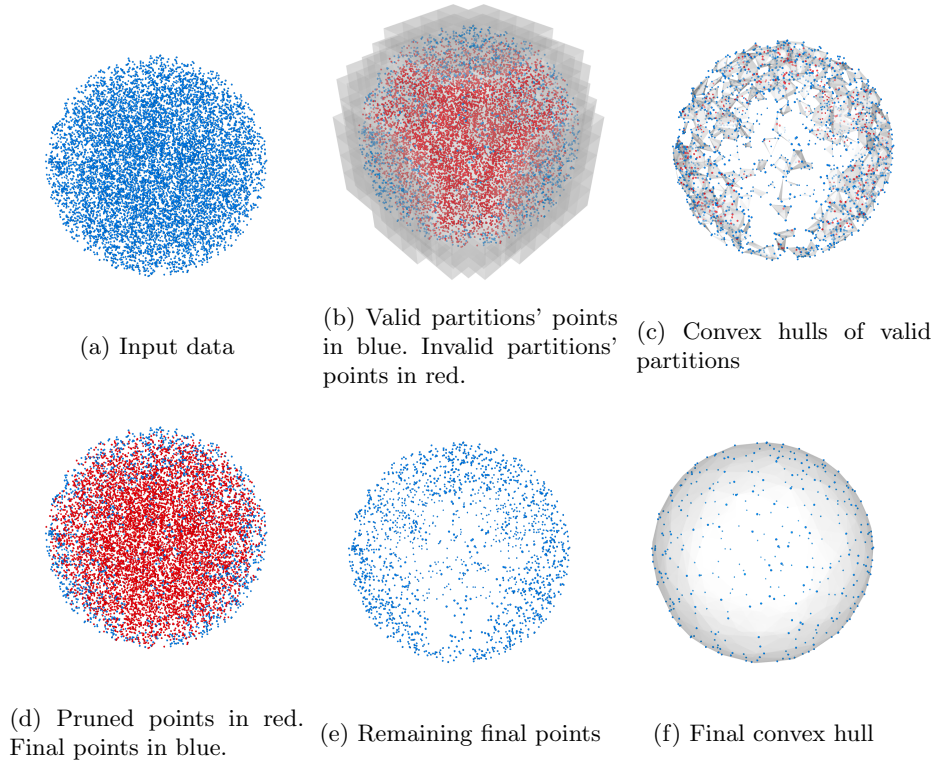


Fig. 3: ConcurrentHull 3D process demonstration.

5.1 CPU 3D Convex Hull

To apply the partitioning, we project a cubic 3D grid on the input data which is a set of 3D points. The grid bounding B is defined by the maximum of furthest points based on X, Y, and Z coordinates. Once we found B , the grid will have $k \times k \times k$ cells each of size $\frac{B}{k}$ and starts from (x_s, y_s, z_s) where $x_s = \min_{p \in P} p.x$, $y_s = \min_{p \in P} p.y$, and $z_s = \min_{p \in P} p.z$.

Similar to 2D crawlers, 3D crawlers also have a starting partition and a direction to crawl. Each of the six sides of the cubic grid has k^2 partition which are the starting partitions of the crawlers. From each starting partition, nine crawlers will be initiated. The adjacent layer of the starting partition has nine partitions that are neighbors of the starting partition which define the directions from starting partitions center towards these partitions centers. In other words we define the direction using a vector $\mathbf{d} = (u, v, z)$, u, v and z are corresponding to X, Y, and Z directions. Depending on the side of the grid which the starting partition is located on, one of the elements of \mathbf{d} is a fixed value (1 or -1) and the other 2 elements can be -1, 0, or 1, which means we have 3^2 different directions to cover. Figure 4 shows the directions for a sample starting partition which is

Algorithm 2 Crawler3D

```

1: procedure CRAWL
2:    $K \leftarrow \text{sidesize}$ 
3:    $\text{start}_i \leftarrow \text{startpoint}_x\text{coordinate}$ 
4:    $\text{start}_j \leftarrow \text{startpoint}_y\text{coordinate}$ 
5:    $\text{start}_k \leftarrow \text{startpoint}_z\text{coordinate}$ 
6:    $\text{direction}_i \leftarrow \text{direction}_x\text{coordinate}$ 
7:    $\text{direction}_j \leftarrow \text{direction}_y\text{coordinate}$ 
8:    $\text{direction}_k \leftarrow \text{direction}_z\text{coordinate}$ 
9:    $\text{current}_i \leftarrow \text{start}_i$ 
10:   $\text{current}_j \leftarrow \text{start}_j$ 
11:   $\text{current}_k \leftarrow \text{start}_k$ 
12:  loop:
13:  if  $\text{size}(\text{partition}(\text{current}_i, \text{current}_j, \text{current}_k)) > 0$  then
14:     $\text{validPartitions.append}(\text{partition}(\text{current}_i, \text{current}_j, \text{current}_k))$ 
15:    return
16:  else
17:     $\text{current}_i \leftarrow \text{current}_i + \text{direction}_i$ 
18:     $\text{current}_j \leftarrow \text{current}_j + \text{direction}_j$ 
19:     $\text{current}_k \leftarrow \text{current}_k + \text{direction}_k$ 
20:    if  $\text{current}_i < K$  and  $\text{current}_j < K$  and  $\text{current}_k < K$  then
21:      goto loop.

```

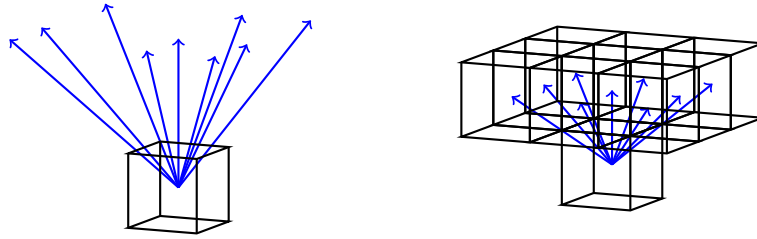


Fig. 4: 3D crawler directions for a sample starting point

located on the bottom of the grid. Overall, $54 \times k^2$ crawlers will be working on the grid to find the valid partitions.

After finding the partitions, 3D quick hull will be used to find the partitions' convex hulls and the same as the 2D algorithm, the points inside the convex hulls will be removed. Note that the partitions with less than four points will also be considered valid partitions. Using the remaining points, the final convex hull for the input problem will be calculated.

Lemma 2. *The final convex hull points are inside the valid partitions and none of the points from the final convex hull can be among the pruned points.*

Proof. The proof is the same as 2D proof. □

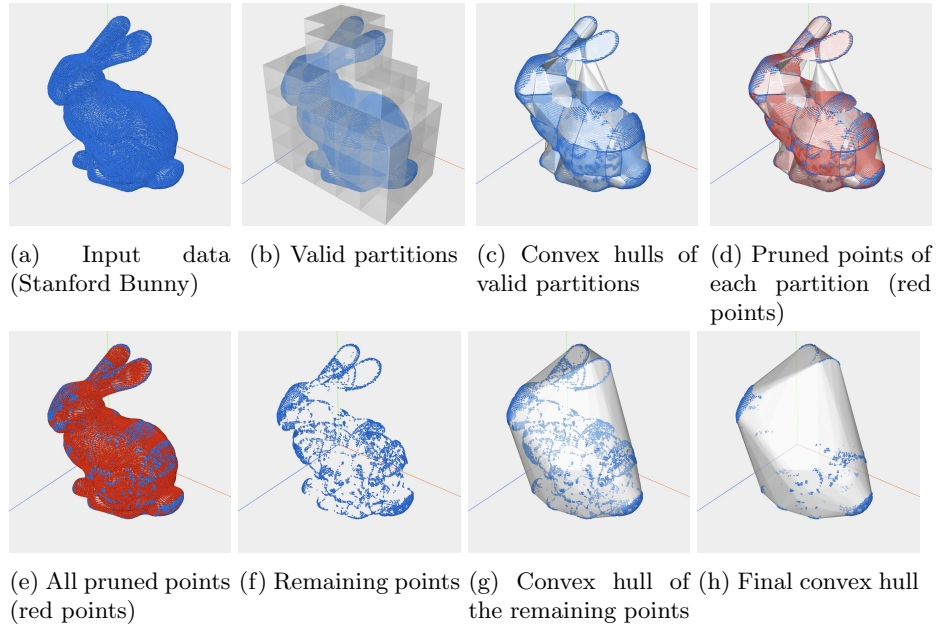


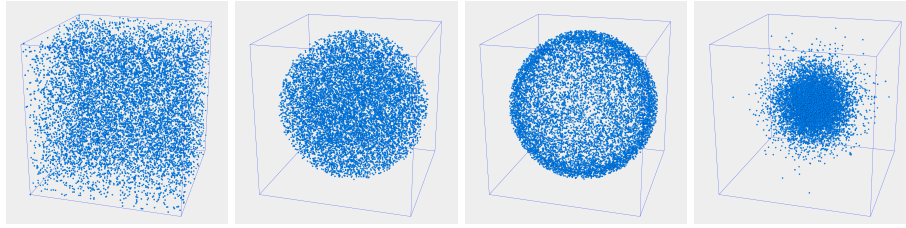
Fig. 5: Finding the convex hull of the Stanford Bunny using ConcurrentHull

5.2 GPU 3D Convex Hull

To partition the input data on the GPU, CPU 3D convex hull partitioning logic is used with the same approach as the 2D partitioning kernel. Each thread is responsible for partitioning one point. Each crawler uses one thread to find its valid partition. The points data is shared among the threads. After finding the valid partitions, gHull [5] will be used to calculate the convex hull of the partitions. Figure 3 shows an example of the 3D algorithm process on a ball dataset and Figure 5 shows the process for the Stanford bunny.

6 Benchmark

Both Synthetic data and real-world data were used to perform the benchmark tests. We performed the benchmarks for CPU 2D, CPU 3D, GPU 2D, and GPU 3D algorithms. The real-world benchmarks are performed for the 3D data as well as the synthetic data. Synthetic benchmarks are performed for the 2D algorithms. Further, the benchmarks for different values of k are plotted. The average values for the synthetic benchmarks are available in Table 1 and Figure 8 shows plots of the average synthetic benchmarks. For each dataset or k value the benchmarks are performed three times and are averaged to minimize the errors in measurements.



(a) Cube with uni- (b) Ball with uni- (c) Sphere with (d) Gaussian distri-
 form distribution form distribution thickness of 1% bution

Fig. 6: Four different synthetic shapes used for 3D benchmark testings.

6.1 2D Benchmark

Synthetic. Four distribution of synthetic data were generated: Square with a thickness of 1%, Square uniform distribution, Circle with a thickness of 1%, Disk uniform distribution.

6.2 3D Benchmark

Synthetic. Four distribution of synthetic data were generated (Figure 6): Cube with uniform distribution, Ball with uniform distribution, Sphere with thickness of 1%, Gaussian distribution.

Real-world. The data for the 3D real-world benchmarks were gathered from LiDAR scans of the Robotic 3D Scan Repository [1].

Table 1: Benchmark numeric results for synthetic data (time in seconds)

# Points	2D CPU			2D GPU			3D CPU			3D GPU		
	2D CPU	QHull	Speedup	2D GPU	CudaChain	Speedup	3D CPU	Qhull	Speedup	3D GPU	gHull	Speedup
10^3	0.00863	0.00713	0.83	0.90207	0.08673	0.1	0.17111	0.0192	0.11	0.41251	0.32855	0.8
10^6	0.01897	0.09982	5.26	0.95632	0.1122	0.12	0.62744	0.25563	0.41	0.56696	1.55492	2.74
10^7	0.06121	0.99196	16.21	1.21163	0.44502	0.37	2.93851	1.95294	0.66	1.19895	9.94595	8.3
10^8	0.31915	13.647	42.76	2.20155	3.38867	1.54	12.91387	26.94304	2.09	6.85567	OOM	
10^9	6.65961	OOM		12.03245	OOM		39.34611	OOM		11.99615	OOM	

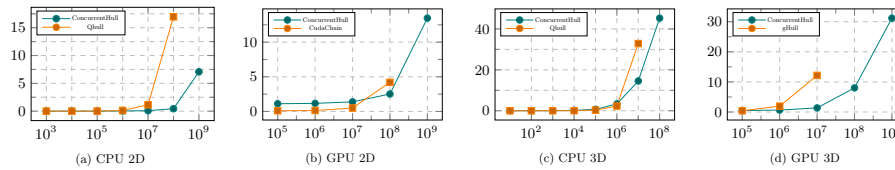


Fig. 8: Average benchmark times (seconds) for different input sizes.

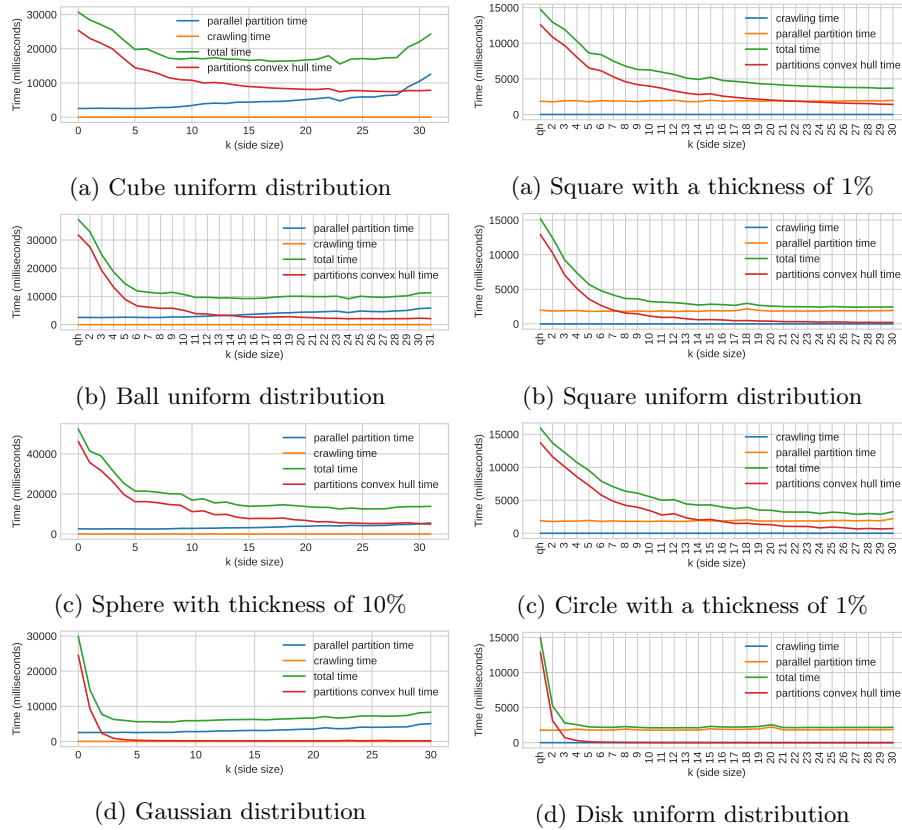


Fig. 9: Effect of k value for 10^8 3D points with different distributions using ConcurrentHull CPU 3D. The X axis shows side size for partitioning. qh stands for qhull times.

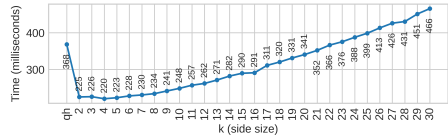
Fig. 10: Effect of k value for 10^8 2D points with different distributions using ConcurrentHull CPU 2D. The X axis shows side size for partitioning. qh stands for qhull times.

7 Results

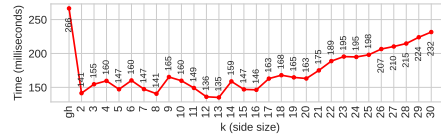
We used a computer with a Core i7 (i7-8700K) CPU, 32GB of RAM, and a NVIDIA GTX 1080 Ti GPU with 11 GB GDDR5X memory to run the benchmarks.

For 2D and 3D comparisons, the input points are randomly distributed point sets generated on a square and a cube respectively. To evaluate the performance of ConcurrentHull for 2D and 3D CPU algorithms, inputs are benchmarked against Qhull. For GPU tests, 2D inputs are benchmarked against CudaChain, while 3D inputs are benchmarked against gHull. Convex hull execution time was used as the benchmark measure. Figure 8 shows the benchmark results.

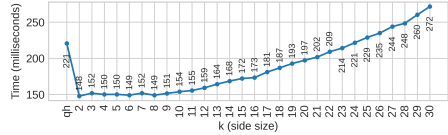
As the results show, on large inputs our approach performs much faster than other approaches, while for a small number of inputs, our approach does not show



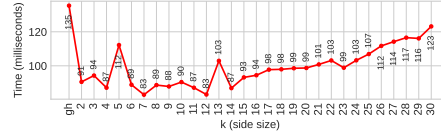
(a) Average times for thermobremen dataset.



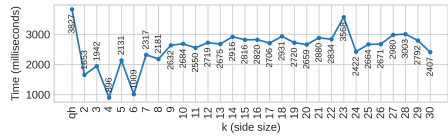
(a) Average times for thermobremen dataset.



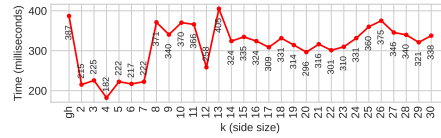
(b) Average times for andreasgart dataset.



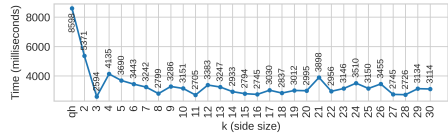
(b) Average times for andreasgart dataset.



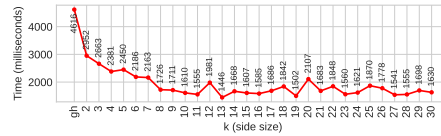
(c) Average times for zagreb dataset.



(c) Average times for zagreb dataset.



(d) Average times for wue city dataset.



(d) Average times for wue city dataset.

Fig. 11: Real world 3D data CPU algo-

Fig. 12: Real world 3D data GPU algo-

an improvement. This behavior is the result of partitioning overhead for a small number of points, whereas for a large number of points the overhead is negligible and the partitioning and pruning significantly improves the performance.

The other noteworthy outcome is that for huge input sizes ($> 10^8$), all three other algorithms could not produce results due to memory management issues. Since our method first removes the unnecessary points, it can handle much larger input point sets.

The effect of changing the value of k is demonstrated in Figures 9-12 for multiple distributions of 2D and 3D datasets. In the charts, $k=0$ means using gHull directly. As results the suggest, increasing k will reduce the execution time up to a point (optimal k), after that the performance will deteriorate.

7.1 LIMITATIONS AND FUTURE WORK

One of the disadvantages of partitioning is that we do not know the optimum size for the grid (k). If we choose a small grid size, it can be ineffective because all of the points will end up in the valid partitions if we choose a large grid size

we can end up with too many partitions which can compromise the performance. Partition pruning can underperform if invalid partitions are all empty or have a small number of points in them.

For future work, we would like to investigate the application of partitioning and crawlers on the dynamic convex hull problem. Moreover, utilizing convex hull merging algorithms might help ConcurrentHull to achieve a better performance. In this case, instead of calculating the convex hull for the aggregated valid partitions pruned points, the already calculated convex hull of each partition can be merged with the others. Finding optimal k requires further investigation of the problem which is an opportunity for future work.

8 Conclusions

This paper introduces ConcurrentHull, a fast convex hull computation technique based on pruning. The results show that the ConcurrentHull algorithm can outperform other algorithms on large enough set of points. For instance, on an input of 10^8 2D points, our approach outperformed Qhull by a speedup of 42 times on synthetic data. This approach benefits from discarding of the points which certainly are not a part of the convex hull using a highly parallel algorithm. Further, the inherent attributes of crawlers help ConcurrentHull achieving a better memory management on large sets of points ($> 10^8$).

References

1. Robotic 3D Scan Repository (2020), <http://kos.informatik.uni-osnabrueck.de/3Dscans/>
2. Barber, C.B., Dobkin, D.P., Huhdanpaa, H.: The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)* **22**(4), 469–483 (1996)
3. Blemloch, G.E., Gu, Y., Shun, J., Sun, Y.: Randomized incremental convex hull is highly parallel. In: *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*. pp. 103–115 (2020)
4. De Berg, M., Van Kreveld, M., Overmars, M., Schwarzkopf, O.C.: *Computational geometry*. In: *Computational geometry*, pp. 1–17. Springer (2000)
5. Gao, M., Cao, T.T., Nanjappa, A., Tan, T.S., Huang, Z.: ghull: A gpu algorithm for 3d convex hull. *ACM Transactions on Mathematical Software (TOMS)* **40**(1), 1–19 (2013)
6. Jurkiewicz, T., Danilewski, P.: Efficient quicksort and 2d convex hull for cuda, and msimd as a realistic model of massively parallel computations (2011)
7. Mei, G.: Cudachain: an alternative algorithm for finding 2d convex hulls on the gpu. *SpringerPlus* **5**(1), 696 (2016)
8. Miller, R., Stout, Q.F.: Efficient parallel convex hull algorithms. *IEEE Transactions on Computers* **37**(12), 1605–1618 (1988)
9. o’Rourke, J.: *Computational geometry in C*. Cambridge university press (1998)
10. Stein, A., Geva, E., El-Sana, J.: Cudahull: Fast parallel 3d convex hull on the gpu. *Computers & Graphics* **36**(4), 265–271 (2012)
11. Tang, M., Zhao, J.y., Tong, R.f., Manocha, D.: Gpu accelerated convex hull computation. *Computers & Graphics* **36**(5), 498–506 (2012)