A Consortium of Parallel QuickHull

Dhruv Mehta, Riley Wilburn, Andrew Witten, Evan Wolf
University of Maryland, College Park
CMSC 818X

***Abstract*—This paper presents four different options to parallelize QuickHull, a well known algorithm to compute the planar convex hull. For each option, we present the algorithm, implementation, and theoretical analysis of runtime. Our experimental results show that Parallel Search has the best empirical runtime and scaling when compared to the other three parallel implementations.**

I. Introduction

A convex hull is the smallest convex shape or set that contains a given point set. In two dimensions, the convex hull is the smallest convex polygon that contains the entire point set. The two dimentional case is known as the *planar convex hull*.

Computing the convex hull of a point set is one of the oldest problems in the field of computational geometry and theoreticians have published an assortment of different algorithms to compute it. One of these algorithms for solving the planar convex hull is QuickHull. QuickHull was created independently by W. Eddy in 1977 and A. Bykat in 1978. We discuss this algorithm in II-A.

This paper discusses different approaches to parallelizing QuickHull and measures their performance. For each parallel algorithm we implemented, we will discuss how well they perform and scale on three different point distributions. We will examine the tradeoffs between each algorithm to determine if one is outright the best, or if certain algorithms perform better in certain situations.

II. Algorithm Definitions

A. Serial

The Serial algorithm is the same as defined by W. Eddy. The algorithm first determines the leftmost (A) and rightmost (B). The point set is then divided into points above and below $\overline{AB}$. QuickHull is then called on both the upper and lower point sets to compute the upper and lower hull. A recursive call of QuickHull is passed the leftmost (P) and rightmost (Q) points along with a subset of the point set (S_k). It then searches for the point C which is furthest from the line $\overline{PQ}$. This point C is added to the hull. S_k is then divided into points above $\overline{PC}$ and points above $\overline{CQ}$ and two recursive calls to QuickHull are made using each partition [1]. Refer to Algorithm 1 for the psuedocode.

B. Parallel Search

The Parallel Search algorithm we are using is a simple modification to the Serial Algorithm. The primary change is that the loops are parallelized, dividing the search space evenly among the processes. After each loop, relevant values are then reduced to each thread to ensure correctness. Refer to Algorithm 2 for the psuedocode.

C. Partition Space

The Partition Space algorithm was motivated by the high cost of frequent communication in Parallel Search. Instead of communication on each call, the communication is done at the beginning and end of the algorithm. The algorithm partitions the 2D plane into p vertical slices (if there are p processes) and computes the convex hull of each vertical slice in parallel using serial QuickHull. Partition Space then merges the p convex hulls into a single convex hull. Pairs of hulls are merged in parallel in a tournament style merge algorithm. This algorithm requires that p be a power of two.

To merge two convex hulls, first compute the rightmost point of the left convex hull, and the leftmost point of the right convex hull, and draw a line between the two points. Then, while there exists a clockwise right turn on the right convex hull, move the point on the right convex hull clockwise to the right. Likewise, while there exists a counterclockwise left hand turn on the left convex hull, move the point on the left counterclockwise to the left [2]. See the cited paper for more details. Refer to Algorithm 3 for the pseudocode.

D. Fork Join

The Fork Join algorithm was motivated by the basic idea of parallelizing the recursive calls in QuickHull rather than the loops at each level. This approach runs one of QuickHull's recursive calls on another process up to some depth threshold. Fork Join does not require external coordination to determine which process should handle which recursive call. Instead, this is determined by partitioning the process space by progressively decreasing powers of two. Conveniently, the depth of recursion can be used to accomplish this process space partitioning allowing the scheduling of work to be done purely locally. However, this approach does add a soft requirement that the algorithm be run with numbers of processes that are powers of two. Refer to Algorithm 4 for the pseudocode.

E. Process Pool

The Process Pool algorithm is very similar to Fork Join in that it runs individual recursive calls to QuickHull in parallel by running them on separate processes. While Fork Join simply splits at every recursive stage until it runs out of processes, the Process Pool implementation requires that a QuickHull call ask for a new process from a dedicated process

manager before executing two calls in parallel. Additionally, a process will be returned to the process pool if a call to QuickHull has no points to operate on. The intuition behind this approach is that certain calls to QuickHull may have more points than other calls and thus may benefit from more processes, especially when a distribution is asymmetric. With the process pool approach, if one side of the distribution is solved early, the resources can be allocated to the other side instead of idling. Refer to Algorithm 5 for the pseudocode.

III. Implementation

The algorithms have been implemented in C or C++ depending on the preference of the individual developer. However many utilities were shared between the programs and each program used MPI. We chose MPI because it offers us a higher degree of control compared to OpenMP for some of the more complex models. Some programs would likely have been faster with OpenMP or been easier to implement. However, in order to have a fair comparison of the algorithms we decided they must all have the same communication library. Additionally, in order to avoid certain edge cases, we assume each program will be provided a power of two for the number of processes and number of points. This was accepted in the design to simplify development.

A. Serial

The serial program was implemented as an edge case of the Parallel Search program. Parallel Search takes a flag which will have the program avoid all MPI calls so that it can be run in a single process with no communication overhead.

B. Parallel Search

The implementation of Parallel Search matches tightly to Algorithm 2. However, there are a few implementation details worth noting. First, in order to keep all processes in sync, each reduce needed to be a reduce all. This increases the communication overhead compared to a standard reduce but without it some other, equivalent, communication would need to occur in order to share the correct inputs to the QuickHull call.

Parallel Search splits the point set after the initial extrema are found. This reduces the search space of the algorithm and is a known speed up technique for serial quickhull. However, in a parallel context this technique reduces the computational load of each process while maintaining a constant communication cost. Below a certain threshold this becomes a poor trade off. Past this threshold, Parallel Search gathers the relevant points to one process and executes on that process only. This threshold is relative to the number of points per node, therefore we decide to merge when the number of points is less than 2^8 per process.

C. Partition Space

The first step of the Partition Space algorithm is to partition the 2D plane into p vertical slices (where p is the number of processes) and assign each slice to a different process. We thought of two possibilities for doing this: 1) compute the points with smallest and largest x coordinate, and then divide the space into equally spaced slices and 2) randomly sample $p-1$ points and partition all n points on those $p-1$. The first strategy works well for inputs uniformly distributed across the x-axis, but a single point with a very small or very large x-coordinate ruins the algorithm's load balancing. The second strategy is resilient to outliers, but in our experiments we found that it had very high variance for distributions with no outliers. The experiments presented here use the first strategy.

After assigning points to processes, each process computes the convex hull for its points using Serial QuickHull. Any convex hull algorithm could have been used here, but in practice QuickHull seems to be the fastest.

Then all convex hulls are merged. The intuition behind the merge algorithm is to recursively merge convex hulls $0, ..., \frac{p}{2} - 1$ and recursively merge convex hulls $\frac{p}{2}, ..., p-1$. Then merge the output hulls. To do this in parallel, first use $\frac{p}{2}$ processes to merge pairs of hulls such as pair 0 and 1, pair 2 and 3, through hull pair $p-2$ and $p-1$. Then use $\frac{p}{4}$ processes to merge the outputs of those in parallel. Continue this way until there is only one hull. In our implementation merging the convex hulls is iterative and not recursive.

D. Fork Join

The Fork Join implementation had a few unexpected edge cases related to short-circuiting the protocol. In effect, when a recursive call to a child process is submitted, if there are less than 4 points, the algorithm short circuits while only issuing work to some of the child processes. Originally, this meant that while the algorithm itself finished, the unused child processes caused the whole program to block indefinitely. This issue was solved by explicitly propagating a cancellation message to any unused trees of child processes. Aside from this minor detail, the implementation closely follows Algorithm 4. The issue with early cancellation shows a more core issue with the algorithm. Having unused child processes effectively reduces the degree of parallelism, which reduces performance. Initially, a few blocking MPISend and MPIRecv calls were used for each child process. A version with only one MPISend and MPIRecv per call was also written, but this did not have a significant impact on performance, so the original implementation was used for all of the results in Section V.

E. Process Pool

The Process Pool implementation required a lot of planning and debugging to sync up all of the MPI calls correctly since this version of parallel QuickHull had the most communication by far. All of the communication calls are roughly the same as described by Algorithm 5.

It is worth nothing that the Process Pool implementation includes the initial MPISend to the first two calls of QuickHull in the timing measurement. All of the other algorithms either do not time this call or loaded the data points on all of the processes. In hindsight, Process Pool could have also been designed this way to avoid timing this large MPISend. Additionally, we could have potentially cut down on the

communication even more by sending arrays of indices instead of sending entire points themselves as integers are smaller than a custom struct.

It is also likely that Process Pool would have benefited from an optimization similar to the one used for Parallel Search. As shown in Section V, Process Pool does not scale well at all, likely due to high communication overhead. Cutting down this overhead might have helped the algorithm do better.

IV. Theoretical Analysis

In this section we will perform a theoretical analysis of the different algorithms assuming a symmetric distribution of points where each call of QuickHull is passed in n points and makes thw two recursive calls of QuickHull on $\frac{n}{2}$ points. This is important to the analysis as the distribution and number of points eliminated with each call will affect the expected performance.

A. Serial

In the average case for a symmetric distribution, the serial algorithm follows the standard recurrence $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n)$. This simplifies to $\mathcal{O}(n \log n)$ runtime for computation. This is also the best case for the serial algorithm.

The worst case runtime is $\mathcal{O}(n^2)$ however. Given a pathological ordering of the points, one recursive call to Quickhull can be given 0 points and the other can be given $\mathcal{O}(n)$ points. The recurrence $T(n) = T(n-1) + \mathcal{O}(n)$ simplifies to $\mathcal{O}(n^2)$ computation.

As this algorithm is serial, there is no communication cost associated with it.

B. Parallel Search

Parallel Search decreases the amount of work done at each level by utilizing parallelization. In the serial algorithm, the amount of computation done at level i in the average case is $\mathcal{O}(2^i(\frac{n}{2^i})) = \mathcal{O}(n)$. There are $\log_2 n$ levels, which is why Serial QuickHull has $\mathcal{O}(n \log n)$ performance as described in Section IV-A. If we ignore the optimization we implemented, then Parallel Search is only doing $\mathcal{O}(2^i(\frac{n}{2^i p}))$ computation per level. This means the computation time of Parallel Search in the average case can be solved by the summation $\sum_{i=0}^{\log n-1} 2^i(\frac{n}{2^i p}) = \sum_{i=0}^{\log n-1} \frac{n}{p} = \frac{n}{p} \log n$. Thus the average case (and best case) for computation in Parallel Search is $\mathcal{O}(\frac{n}{p} \log n)$.

This faster computation time comes at the tradeoff of some communication time. In every call of QuickHull, Parallel Search makes calls to MPIAllReduce which runs in time $\mathcal{O}(\log p)$. This can be represented as the summation $\sum_{i=0}^{\log n-1} 2^i \log p = \log p(\frac{1-2^{\log n-1}}{-1}) \approx n \log p$. Thus the cost of communication in Parallel Search is $\mathcal{O}(n \log p)$. This brings the overall average (and best) case time complexity to $\mathcal{O}(\frac{n}{p} \log n + n \log p)$.

In the worst case where one of the recursive calls to QuickHull gets all of the remaining points, there are essentially n levels with one call each that does $\mathcal{O}(\frac{n}{p})$ computation and $\mathcal{O}(\log_2 p)$ communication. Thus the overall worst case complexity would be $\mathcal{O}(\frac{n^2}{p} + n \log p)$.

C. Partition Space

The performance of the Partition Space algorithm depends heavily on how well the load balancing works. If the load balancing is perfect, then each process receives n/p points. Computing the convex hulls of these takes $\mathcal{O}\left(\frac{n}{p} \log\left(\frac{n}{p}\right)\right)$ time.

Merging two convex hulls of size m where one convex hull is entirely to the left of the other, takes time $\mathcal{O}(m)$ [2]. The Partition Space algorithm requires $\mathcal{O}(\log p)$ rounds of merging. Therefore, the merge step takes time $\mathcal{O}(n \log p)$.

This gives the final runtime of $\mathcal{O}\left(\frac{n}{p} \log\left(\frac{n}{p}\right) + n \log p\right)$. When $p = \mathcal{O}(1)$ or when $p = \mathcal{O}(n)$ this gives no asymptotic speedup. There is asymptotic speedup between these two extremes.

Under poor load balancing, Partition Space turns into serial QuickHull. It even performs worse than serial QuickHull sometimes because there is an extra linear pass to assign points to processes (see experimental section).

D. Fork Join

The best case for Fork Join is the case where every "level" of recursive calls will run in parallel and there will never be a time where a process will need to degrade to serial QuickHull. In this case, the amount of computation and the amount of communication at every level is $\mathcal{O}(n)$. Thus the runtime can be represented by the sum $\sum_{i=0}^{\log_2(n)-1} \frac{n}{2^i} = 2n(1-\frac{1}{2}^{\log_2(n)-1}) = \mathcal{O}(n)$.

In the average case, each process will eventually end up with $\frac{n}{p}$ points, so each individual process run in parallel will take $\mathcal{O}(\frac{n}{p} \log \frac{n}{p})$ computation time in addition to the $\mathcal{O}(n)$ time when all the processes could run their next recursive call in parallel. This gives an overall average case of $\mathcal{O}(n + \frac{n}{p} \log \frac{n}{p})$ for computation. The communication factor does not change and thus does not change from above.

In the worst case, we get the same pathological ordering as in the serial case. In this case, the algorithm will degrade to serial QuickHull early because the other processes will get 0 points to process and have nothing to do. There would be n levels and each level will have $\mathcal{O}(n)$ computation and communication. Thus the worst case for Fork Join is $\mathcal{O}(n^2)$

E. Process Pool

The analysis for Process Pool is largely the same as Fork Join as they have very similar parallelization strategies. The best case and average case will stay exactly the same. Even the pathological worst case will be exactly the same. The one benefit Process Pool theoretically has over Fork Join is for the distributions that fall somewhere between the worst and average case.

You never "lose" a process when you run out of points in Process Pool since the process can be returned to the process manager and reallocated to the other side of the problem. This is the one benefit this implementation has in exchange for the massive increase in communication. Thus, while the absolute worst case is the same as Fork Join, the "average" worst case would theoretically run faster with Process Pool rather than

Fork Join. It is worth noting that while there is a "massive" increase in communication, it is asymptotically the same as Fork Join which is why the asymptotic best, average, and worst cases are also the same.

V. Experimental Results

In this section we will overview our experimental setup and analysis of our algorithms. We ran each algorithm over all combinations of three sets of variables. The first variable set, process count, consisted of running on 1, 2, 4, 8, 16, and 32 worker processes. This distinction of worker processes is made because Process Pool has 2 controller processes which do not contribute to scaling. The second variable, problem size, consists of data inputs of 2^{12}, 2^{16}, 2^{20}, and 2^{24}. Finally, the last variable set, problem distribution, consisted of a uniform disk of radius = 100 units, a uniform rectangle 100x100 units, and an exponential distribution of λ = .01. These distributions are pictured in figures 1, 2, and 3.

Each program was timed for the computational portion of the code only. This choice was made to simplify design choices, but it does have some implications. First, the actual run time of some of the programs running on 2^{24} points is much higher than shown in the results. For example in Parallel Search, all of the data is loaded in by one process and scattered to the others. In contrast, other implementations had each process load in the relevant data straight from the file. Because these two designs are logically identical and can be interchanged we did not include them in our timing. This does however hurt the Process Pool model as it communicates every point from the root process to others during the timed section. This is a necessity of the model and we therefore believe it is valid to include it in the timing data.

The results from our experiment are pictured in figures 4, 5, and 6 on pages 9, 10, and 11 respectively. The first thing which is apparent in our results is that no solution scales well for smaller inputs. Inputs of size 2^{12} and 2^{16} show that there can be some, but not much scaling happening with input of this size. This is to be expected, as the cost of communication is high even if smaller portions of data are being exchanged. It is shown in our theoretical analysis that most of our algorithms are computationally bound, however this analysis undercuts the higher cost of communication. Therefore the extra benefits to computational complexity can be negated by increased communication loads if the computation cost is not sufficiently high.

A second result that is quite clear is that scaling past 32 processes was difficult for us. Due to the fact that the server we tested on had 20 cores per node, a 32 process program requires inter-node communication. This is much slower than intra-node communication and makes scaling past 32 nodes a challenge not overcome at the data-sizes we tested. We had considered testing on larger data sets, however the file sizes were becoming very large and the multiple combinations of testing were becoming intensive.

Between the different distributions, we saw a few differences. First, the disk was on average the slowest experiment. This made sense because the run time of QuickHull is always dependent on the number of points in the hull and a disk should have more points on the hull than a rectangle or the exponential distribution. Parallel Search, Process Pool, and Fork Join all perform consistently across the three distribution types. In contrast, Partition Space performs well on a disk and rectangle, but poorly on the exponential distribution. This is due to how Partition Space divides points. As discussed in Section III-C, the points are divided into even buckets by x-coordinate. Therefore the asymmetrical exponential distribution behaves very poorly.

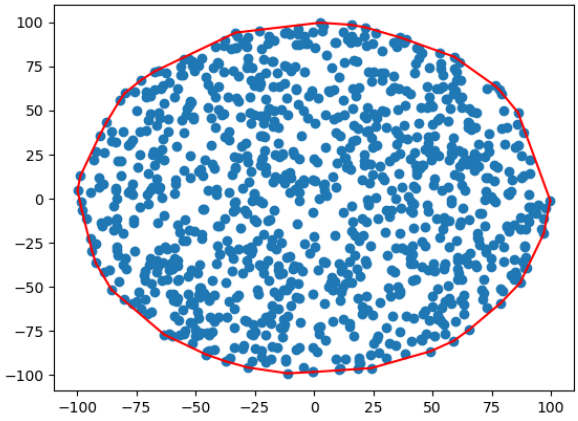

Fig. 1. 1000 points from a uniform disk distribution of radius 100 units. The points forming the convex hull are connected via the red line

Partition Space was designed as an improvement to Parallel Search. However there is a $\mathcal{O}(n)$ delta between the time performance of the two . This is due to the $\mathcal{O}(n)$ cost of splitting the nodes for partitioning. This cost does not outweigh the costs of continuous communication during the processing of QuickHull. This may be improvable in the future by taking arbitrary $\frac{n}{p}$ bucketed points for each process instead of bucketing them by x-coordinate. This would remove the ability to run an elegant hull merge algorithm, however the added cost of running QuickHull on the combined points of the hull should not be too expensive and may turn out to be the best option.

Process Pool was designed to handle cases where Fork Join may have unbalanced datasets resulting in poor CPU optimization. While Process Pool has too much overhead to outperform Fork Join, this benefit can be seen in the exponential data set. With 2^{24} nodes Process Pool performs closer to Fork Join on the exponential distribution than it does with the other two distributions. This shows that our intuition in that respect was correct.

VI. Future Work

In implementing these models many lesson were learned that can be used for future iterations on this work. First, the time to load data should be included in the measurements. This requires a strict definition for implementations ahead of time, so that the designs can be optimal for all of the algorithms.

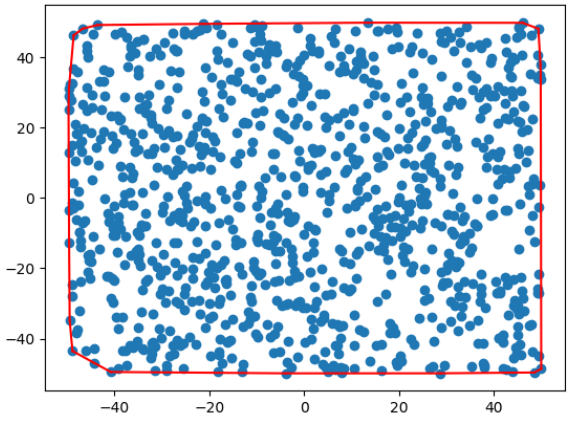

Fig. 2. 1000 points from a uniform rectangle distribution of 100 units by 100 units. The points forming the convex hull are connected via the red line

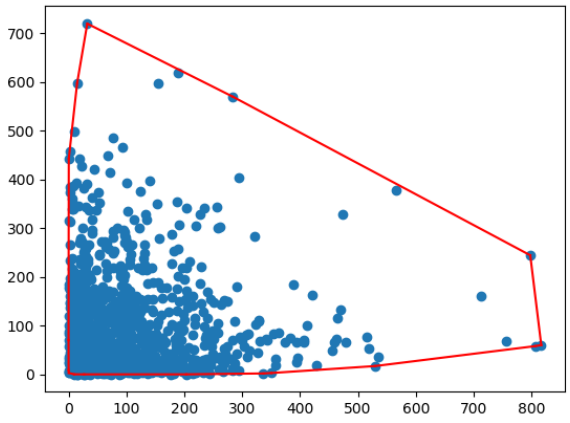

Fig. 3. 1000 points from an exponential distribution of λ = .01. The points forming the convex hull are connected via the red line

Second, we suspect that code complexity is limiting our performance of some of the more complex models. The computation portion of Parallel Search and the Serial model were able to be implemented in a fairly bare-bones C program. In contrast, more complex models such as Fork Join utilized C++ and have more intricate programs. This has meant that optimizing the code is more difficult. As a bench mark for this theory, all models when run on a single process should approximate a serial implementation. This means optimal implementations should have similar run times. This is not the case, as the difference in the one process run times increases as the data set increases.

The final consideration for future work is how multiple libraries should be handled. We found that restricting ourselves to OpenMPI was not perfect for any model. Parallel Search would have been easier to implement with OpenMP. Other models, such as Process Pool, would have benefit greatly from using Charm++. However, in this paper we restricted the library usage to enable comparisons of the theoretical algorithms. It remains an open philosophical question if a fair comparison of algorithms should allow unique library choices.

VII. Conclusion

QuickHull has proven to have many interesting options for parallelization. We have found that in our experiment the simplest parallelization performed the best, in part because it was highly optimizable. This does not discount the merits of our other models. We believe we have shown there is potential for optimizations to each algorithm if they are performed carefully. We have the most hope for the discussed modification to Partition Space which uses arbitrary bucketing. We have also shown that it is best practice to include the entire run time of a program when comparing efficiency, as some programs may have heavier requirements that would not be otherwise accounted for.

References

[1] W. Eddy, "A new convex hull algorithm for planar sets," *ACM Trans. Math. Softw.*, vol. 3, pp. 398–403, 1977.

[2] F. Preparata and S. J. Hong, "Convex hulls of finite sets of points in two and three dimensions," *Commun. ACM*, vol. 20, pp. 87–93, 1977.

VIII. Appendix

A. Algorithms

Algorithm 1 Serial QuickHull

Require: $points \neq \emptyset$
Require: $hull = \emptyset$

Main
  Find the leftmost and rightmost points A and B
  $hull \leftarrow \{A, B\}$
  $S_1 = \{p \in points |$ FindSide(A, B, p) $> 0\}$
  $S_2 = \{p \in points |$ FindSide(A, B, p) $< 0\}$
  QuickHull(A, B, S_1)
  QuickHull(B, A, S_2)
  print $hull$

QuickHull(P, Q, S_k)
  if $S_k = \emptyset$ **then**
    return
  end if
  Find point C that is furthest from $\overline{PQ}$
  Add C to $hull$ between points P and Q
  $S_1 = \{p \in S_k |$ FindSide(P, C, p) $> 0\}$
  $S_2 = \{p \in S_k |$ FindSide(C, Q, p) $> 0\}$
  QuickHull(P, C, S_1)
  QuickHull(C, Q, S_2)

Algorithm 2 Parallel Search

Require: $points \neq \emptyset$
Require: $hull = \emptyset$

Main
  MPIScatter($points$)
  Find the leftmost and rightmost points A and B within scattered points
  MPIAllReduce($\{A, B\}$)
  $hull \leftarrow \{A, B\}$
  $S_1 = \{p \in points |$ FindSide(A, B, p) $> 0\}$
  $S_2 = \{p \in points |$ FindSide(A, B, p) $< 0\}$
  QuickHull(A, B, S_1)
  QuickHull(B, A, S_2)
  print $hull$

QuickHull(P, Q, S_k)
  if $S_k = \emptyset$ **then**
    return
  end if
  Find point C that is furthest from $\overline{PQ}$
  MPIAllReduce(C)
  Add C to $hull$ between points P and Q
  if the size of S_k across all points is less than 2^8 per process **then**
    Gather all S_k in process 0
  end if
  Add C to $hull$ between points P and Q
  $S_1 = \{p \in S_k |$ FindSide(P, C, p) $> 0\}$
  $S_2 = \{p \in S_k |$ FindSide(C, Q, p) $> 0\}$
  QuickHull(P, C, S_1)
  QuickHull(C, Q, S_2)

Algorithm 3 Partition Space

Require: $points \neq \emptyset$
Require: p is a power of 2
Require: r is rank of current process

PartitionSpace
  smallest ← points[0]
  largest ← points[0]
  mypoints ← ∅
  for each point t in points **do**
    if t.x > largest.x **then**
      largest ← t
    end if
    if t.x < smallest.x **then**
      smallest ← t
    end if
  end for
  for each point t in points **do**
    idx $\left\lfloor \leftarrow n \frac{t.x - smallest.x}{largest.x - smallest.x} \right\rfloor$
    if idx == r **then**
      add t to mypoints
    end if
  end for
  Compute convex hull on mypoints
  $round \leftarrow 1$
  while true **do**
    if p = 1 (we have done $\log(p)$ rounds) **then**
      break out of while loop
    end if
    if r is odd **then**
      MPISend myHull to r - round
      break from while loop
    else
      neighborHull ← MPIRecv hull from r + round
      MERGE neighborHull into myHull
      $r \leftarrow \left\lfloor \frac{r}{2} \right\rfloor$
      $p \leftarrow \left\lfloor \frac{p}{2} \right\rfloor$
      $round \leftarrow 2 * round$
    end if
  end while

Algorithm 4 Fork Join

Require: $points \neq \emptyset$
Require: $hull = \emptyset$
Require: p is a power of 2

Master (Proc 0)
  $d = \log_2(p)$
  Find the leftmost and rightmost points A and B
  $hull \leftarrow$ Delegate(A, B, A, d)
  print $hull$

Worker (Proc 1-$p-1$)
  $S_k, P, Q, d \leftarrow$ MPIRecv()
  $hull \leftarrow$ QuickHull(P, C, S_k, $d-1$)
  MPISend($hull$) $\rightarrow$ Proc $(rank - (rank\& - rank))$

Delegate(P, C, Q, S_k, d)
  $S_1 = \{p \in S_k |$ FindSide(P, C, p) $> 0\}$
  $S_2 = \{p \in S_k |$ FindSide(C, Q, p) $> 0\}$
  if $d \leq \emptyset$ **then**
    $hull_1 \leftarrow$ QuickHull(P, C, S_1, $d-1$)
    $hull_2 \leftarrow$ QuickHull(C, Q, S_2)
  else
    MPISend(S_2, C, Q, $d-1$) $\rightarrow$ Proc $rank + 2^{(d-1)}$
    $hull_1 \leftarrow$ QuickHull(P, C, S_1, $d-1$)
    $hull_2 \leftarrow$ MPIRecv()
  end if
  stitch $hull_1$ and $hull_2$ into $hull$
  return $hull$

QuickHull(P, Q, S_k, d)
  if $S_k = \emptyset$ **then**
    return
  end if
  Find point C that is furthest from $\overline{PQ}$
  return Delegate(S_k, P, C, Q, d)

Algorithm 5 Process Pool

Require: $points \neq \emptyset$
Require: $hull = \emptyset$
Require: $p \geq 4$

Master (Proc 0)
  Find the leftmost and rightmost points A and B
  $hull \leftarrow \{A, B\}$
  $S_1 = \{p \in points |$ FindSide(A, B, p) $> 0\}$
  $S_2 = \{p \in points |$ FindSide(A, B, p) $< 0\}$
  MPISend (A, B, S_1) $\rightarrow$ Proc 2
  MPISend (B, A, S_2) $\rightarrow$ Proc 3
  while (P, C, Q) $\leftarrow$ MPIRecv **do**
    Add C to $hull$ between points P and Q
  end while
  print $hull$

Proc Manager (Proc 1)
  Create $procStack$ with $p-4$ processes pushed
  while $req \leftarrow$ MPIRecv from Proc $i \in [2, p]$ **do**
    if req = ProcessRequest **then**
      if $procStack \neq \emptyset$ **then**
        MPISend ($procStack$.pop()) $\rightarrow$ Proc i
      else
        MPISend ($\emptyset$) $\rightarrow$ Proc i
      end if
    else
      $procStack$.push(req.process)
    end if
  end while

Worker Proc (Procs 2-p)
  QuickHull($\emptyset$, $\emptyset$, $\emptyset$, Message)

QuickHull(P, Q, S_k, $mode$)
  if $mode$ = Message **then**
    (P, Q, S_k) $\leftarrow$ MPIRecv
  end if
  if $S_k \neq \emptyset$ **then**
    Find point C that is furthest from $\overline{PQ}$
    MPISend (P, C, Q) $\rightarrow$ Proc 0
    $S_1 = \{p \in S_k |$ FindSide(P, C, p) $> 0\}$
    $S_2 = \{p \in S_k |$ FindSide(C, Q, p) $> 0\}$
    MPISend (ProcessRequest) $\rightarrow$ Proc 1
    $newProc \leftarrow$ MPIRecv from Proc 1
    if $newProc \neq \emptyset$ **then**
      MPISend (P, C, S_1) $\rightarrow$ Proc $newProc$
      QuickHull(C, Q, S_2, Recursive)
    else
      QuickHull(P, C, S_1, Recursive)
      QuickHull(C, Q, S_2, Recursive)
    end if
  end if
  if mode = Message **then**
    MPISend (Return, $thisProc$) $\rightarrow$ Proc 1
    Quickhull($\emptyset$, $\emptyset$, $\emptyset$, Message)
  end if

Algorithm 6 Shared Functions

FindSide($p1, p2, p3$)

$v1 \leftarrow (p.y - p1.y) * (p2.x - p1.x)$
$v2 \leftarrow (p2.y - p1.y) * (p.x - p1.x)$
$v \leftarrow v1 - v2$
if $v < 0$ **then**
  return -1
else if $v = 0$ **then**
  return 0
else if $v > 0$ **then**
  return 1
end if

LineDistance($p1, p2, p3$)

$v1 \leftarrow (p.y - p1.y) * (p2.x - p1.x)$
$v2 \leftarrow (p2.y - p1.y) * (p.x - p1.x)$
$v \leftarrow v1 - v2$
return v

B. Results

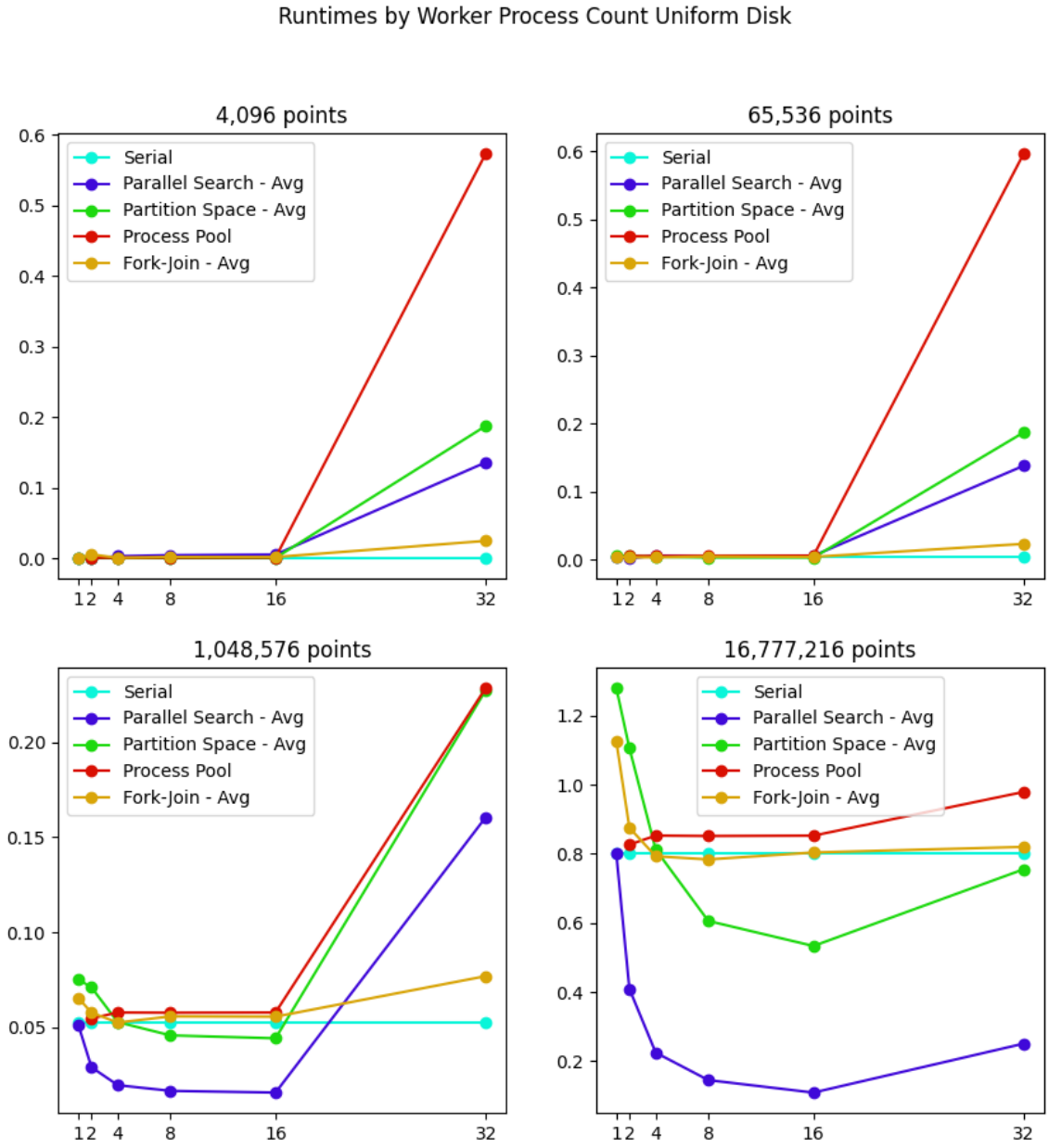

Fig. 4. Results from the trials for a Uniform Disk

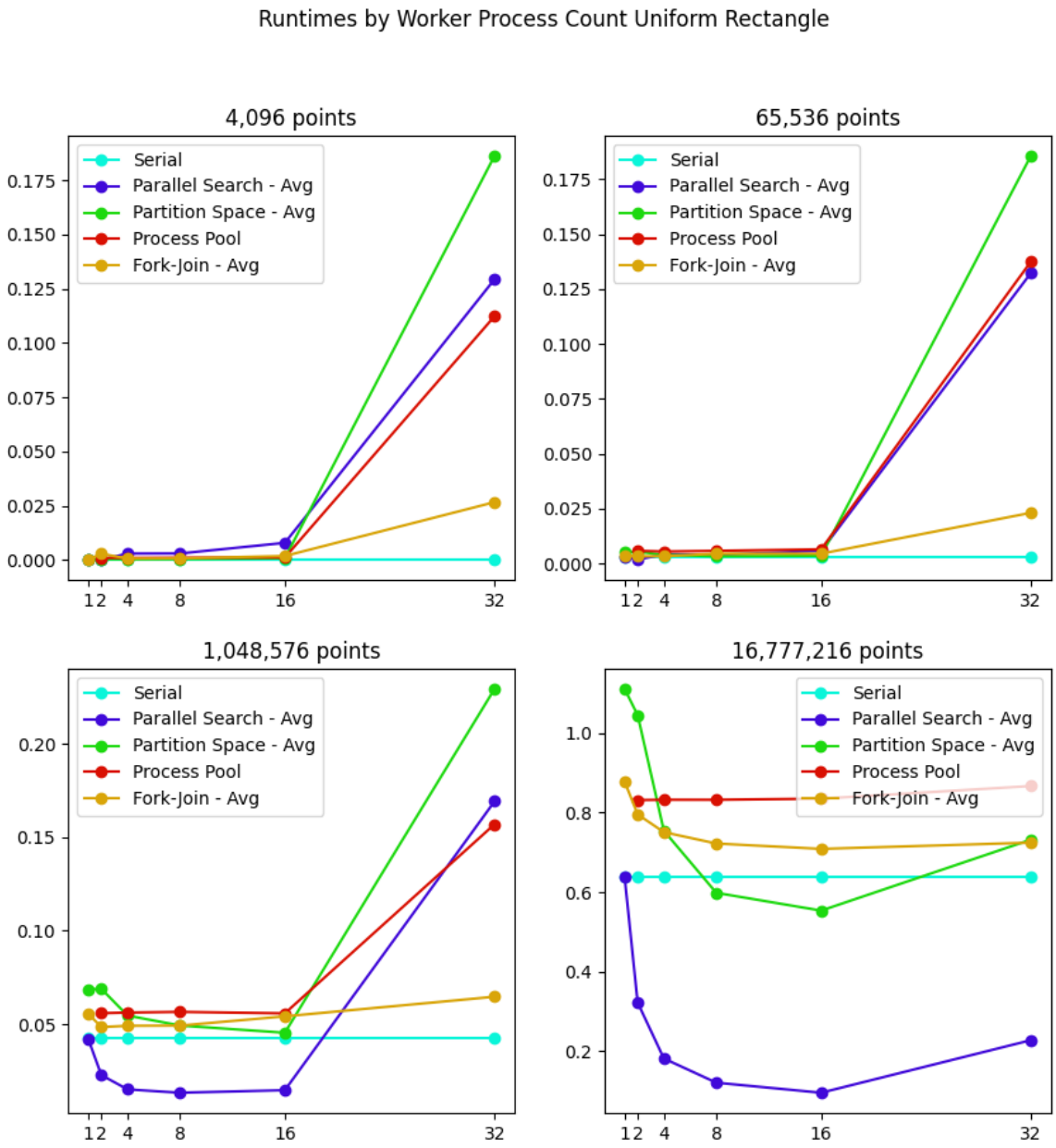

Fig. 5. Results from the trials for a Uniform Rectangle

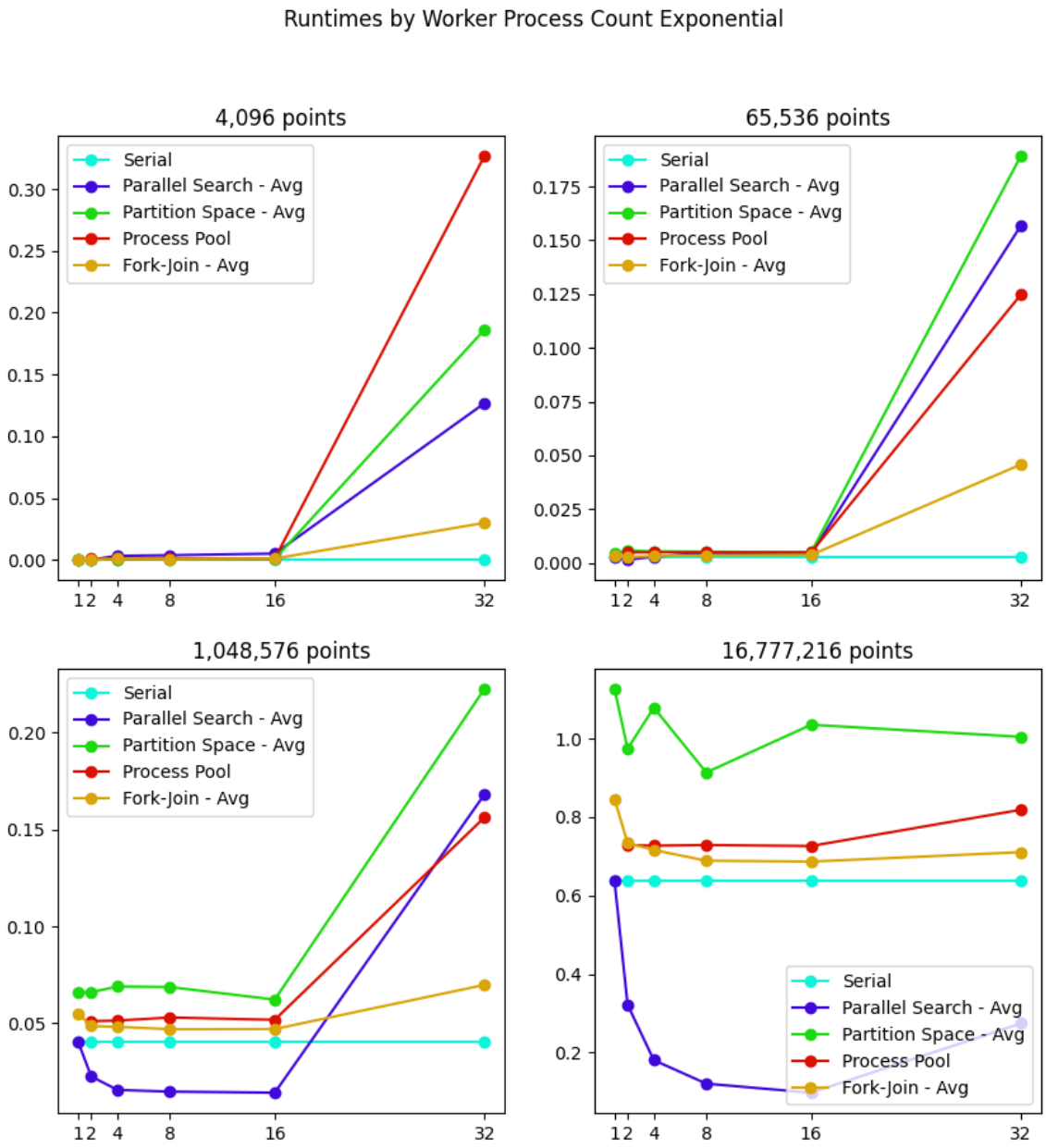

Fig. 6. Results from the trials for a Exponential Distribution