ParGeo: A Library for Parallel Computational Geometry

Yiqiu Wang
CSAIL, MIT, Cambridge MA, USA

Rahul Yesantharao
CSAIL, MIT, Cambridge MA, USA

Shangdi Yu
CSAIL, MIT, Cambridge MA, USA

Laxman Dhulipala
University of Maryland, College Park, MD, USA

Yan Gu
University of California, Riverside, CA, USA

Julian Shun
CSAIL, MIT, Cambridge, MA, USA

Abstract

This paper presents ParGeo, a multicore library for computational geometry. ParGeo contains modules for fundamental tasks including kd-tree based spatial search, spatial graph generation, and algorithms in computational geometry.

We focus on three new algorithmic contributions provided in the library. First, we present a new parallel convex hull algorithm based on a reservation technique to enable parallel modifications to the hull. We also provide the first parallel implementations of the randomized incremental convex hull algorithm as well as a divide-and-conquer convex hull algorithm in $\mathbb{R}^3$. Second, for the smallest enclosing ball problem, we propose a new sampling-based algorithm to quickly reduce the size of the data set. We also provide the first parallel implementation of Welzl's classic algorithm for smallest enclosing ball. Third, we present the BDL-tree, a parallel batch-dynamic kd-tree that allows for efficient parallel updates and k-NN queries over dynamically changing point sets. BDL-trees consist of a log-structured set of kd-trees which can be used to efficiently insert, delete, and query batches of points in parallel.

On 36 cores with two-way hyper-threading, our fastest convex hull algorithm achieves up to 44.7x self-relative parallel speedup and up to 559x speedup against the best existing sequential implementation. Our smallest enclosing ball algorithm using our sampling-based algorithm achieves up to 27.1x self-relative parallel speedup and up to 178x speedup against the best existing sequential implementation. Our implementation of the BDL-tree achieves self-relative parallel speedup of up to 46.1x. Across all of the algorithms in ParGeo, we achieve self-relative parallel speedup of 8.1–46.61x.

2012 ACM Subject Classification Computing methodologies → Shared memory algorithms

Keywords and phrases Computational Geometry, Parallel Algorithms, Libraries

Digital Object Identifier 10.4230/LIPIcs.ESA.2022.88

Related Version *Full Version*: https://arxiv.org/abs/2207.01834
Related Paper: https://arxiv.org/abs/2112.06188

Supplementary Material *Software (Source Code)*: https://github.com/ParAlg/ParGeo

Funding This research is supported by DOE Early Career Award #DE-SC0018947, NSF CAREER Award #CCF-1845763, Google Faculty Research Award, Google Research Scholar Award, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

1 Introduction

Computational geometry algorithms have important applications in various domains, including computer graphics, robotics, computer vision, and geographic information systems [29, 43]. It is important to provide users with libraries of efficient computational geometry algorithms that they can easily use in their own higher-level applications. Furthermore, due to the growing sizes of data sets that need to be processed today, and the ubiquity of parallel (multicore) machines, it is beneficial to use parallel algorithms to speed up computations. In this paper, we present the ParGeo library for parallel computational geometry, which includes a rich set of parallel algorithms for geometric problems and data structures, including kd-trees, k-nearest neighbor search, range search, well-separated pair decomposition, Euclidean minimum spanning tree, spatial sorting, and geometric clustering. ParGeo also contains a collection of geometric graph generators, including k-nearest neighbor graphs and various spatial networks. Algorithms from ParGeo can either run sequentially, or run using parallel schedulers such as OpenMP, Cilk, or ParlayLib.

While there exist numerous libraries for computational geometry, most of them are not designed for parallel processing. For example, Libigl [35] is a library that specializes in the construction of discrete differential geometry operators and finite-element matrices. However, only some aspects of Libigl take advantage of parallelism. In comparison, the algorithms and implementations of ParGeo are designed for parallelism, and target a different set of problems. CGAL (Computational Geometry Algorithms Library) [2] is a well-known library of computational geometry algorithms that includes a wide range of algorithms, but most implementations are not parallel. Batista et al. [15] targeted a few important algorithms, including spatial sorting, box intersection, and Delaunay triangulation for shared-memory parallel processing, with code in CGAL. In comparison, ParGeo targets similar classes of problems as CGAL, but *all* of our implementations are highly parallel. PMP [47], Cinolib [39], and Tetwild [34] are libraries for polygonal and polyhedron meshes, tackling different problems from ParGeo. MatGeom [5] is a library for sequential geometric computing with MATLAB. The Problem Based Benchmark Suite [46, 12] is a multicore benchmark suite that has some overlap in algorithms with ParGeo. LEDA [40] is a library of data structures and algorithms for sequential combinatorial and geometric processing. ArborX [38] is a parallel library for spatial search.

In this paper, in addition to providing an overview of work on ParGeo, we describe new parallel algorithms implemented in ParGeo for convex hull, smallest enclosing ball, and batch-dynamic kd-tree that we developed. For convex hull, we develop new parallel algorithms for both $\mathbb{R}^2$ and $\mathbb{R}^3$, where our key algorithmic novelty is a reservation technique to enable parallel modifications to the hull. For smallest enclosing ball, we propose a new sampling-based algorithm based on Larsson et al.'s [37] approach to quickly reduce the size of the data set. We also provide the first parallel implementation of the classic randomized incremental algorithm [27]. For kd-trees, we develop the BDL-tree, a new parallel data structure that supports batch-dynamic operations (construction, insertions, and deletions) as well as exact k-NN queries. BDL-trees consist of a set of exponentially growing kd-trees and perform batched updates in parallel.

To demonstrate the efficiency of our proposed algorithms and library, we perform a comprehensive set of experiments on synthetic and real-world geometric data sets, and compare the performance across our parallel implementations as well as optimized sequential baselines. On 36 cores with two-way hyper-threading, our best convex hull implementations achieve up to 44.7x (42.8x on average) self-relative speedup and up to 559x (325x on average) speedup against the best existing sequential implementation for $\mathbb{R}^2$, and up to 24.9x (11.81x

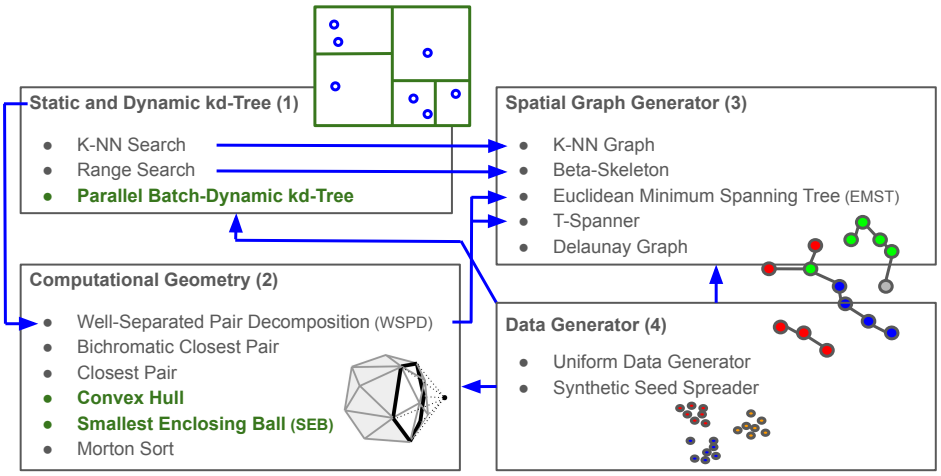


Figure 1 The figure shows an overview of modules in ParGeo. An arrow indicates that a component is used inside another component. In this paper, we present new algorithms and techniques for the modules highlighted in green.

on average) self-relative speedup and up to 124x (61.4x on average) speedup against the best existing sequential implementation for $\mathbb{R}^3$. Our sampling-based smallest enclosing ball algorithm achieves up to 27.1x (20.08x on average) self-relative speedup and up to 178x (109x on average) speedup against the best existing sequential implementation for $\mathbb{R}^2$ and $\mathbb{R}^3$. Our BDL-tree achieves self-relative speedup of up to 35.4× (30.0× on average) for construction, up to 35.0× (28.3× on average) for batch insertion, up to 33.1× (28.5× on average) for batch deletion, and up to 46.1× (40.0× on average) for full k-NN. Finally, across all implementations in ParGeo, we achieve self-relative parallel speedup of 8.1–46.6x (on average 23.2x).

2 The ParGeo Library

Our main goal in designing ParGeo was to enable reusable and efficient parallel implementations of geometric algorithms and data structures. We present an overview of the modules of ParGeo in Figure 1, highlighting how the modules interact with each other. ParGeo contains efficient multicore implementations of static and batch-dynamic kd-trees (Module (1)). The code supports kd-tree based spatial search, including k-nearest neighbor and range search. Our code is optimized for fast kd-tree construction by performing the split in parallel (either by spatial median or by object median), and performing the queries in a data-parallel fashion. which we will introduce in Section 5.

ParGeo contains a module for parallel computational geometry algorithms (Module (2)). Our kd-tree can be used to generate a well-separated pair decomposition [26] (WSPD), which can in turn be used to compute the hierarchical DBSCAN [52], ParGeo contains parallel implementations for the bichromatic closest pair, closest pair, convex hull, smallest enclosing ball, and Morton sorting.

In addition, ParGeo contains a collection of geometric graph generators (Module (3)) for point data sets. Our kd-tree's k-NN search is used to generate the k-NN graph, and the range search is used to generate the β-skeleton graph [36]. Our WSPD generated from the kd-tree can also be used to compute the Euclidean minimum spanning tree [25, 52], and spanners [26]. ParGeo also generates the Delaunay graph.

Table 1 Runtimes (seconds) and parallel speedups (T_1/T_{36h}) for ParGeo implementations on uniform hypercube data sets of varying dimensions and 10 million points. T_1 and T_{36h} denote the single-threaded and the 36-core hyper-threaded times, respectively. For batch-dynamic kd-tree updates, each batch contains 10% of the data set.

Implementation	T_1	T_{36h}	Speedup
kd-tree Build (2d)	5.51	0.43	12.70x
kd-tree Build (5d)	8.39	0.89	9.40x
kd-tree k-NN (2d)	31.45	0.68	46.34x
kd-tree Range Search (2d)	17.14	0.37	46.61x
Batch-dynamic kd-tree Construction (5d)	6.70	0.60	10.70x
Batch-dynamic kd-tree Insert (5d)	8.80	1.10	8.10x
Batch-dynamic kd-tree Delete (5d)	29.20	1.20	23.90x
WSPD (2d)	6.72	0.24	27.63x
EMST (2d)	33.02	1.58	20.86x
Convex Hull (2d)	0.38	0.0088	43.13x
Convex Hull (3d)	2.36	0.097	24.36x
Smallest Enclosing Ball (2d)	0.053	0.0033	16.30x
Smallest Enclosing Ball (5d)	0.13	0.014	9.54x
Closest Pair (2d)	10.35	0.52	19.90x
Closest Pair (3d)	28.00	2.32	12.07x
k-NN Graph (2d)	37.89	1.46	25.99x
Delaunay Graph (2d)	55.91	2.03	27.53x
Gabriel Graph (2d)	59.61	1.99	29.99x
β-skeleton Graph (2d)	113.27	3.20	35.37x
Spanner (2d)	27.19	2.15	12.67x

ParGeo contains a point data generator module (Module (4)) for which can generate uniformly distributed data sets, and clustered data sets of varying densities [31]. These data sets are used for benchmarking the other modules.

As shown in Table 1, on a machine with 36 cores with two-way hyper-threading, ParGeo achieves self-relative parallel speedups of 8.1–46.61x (23.15x on average) on a uniformly distributed data set, across all of the benchmarks. In the subsequent sections, we present three new algorithmic contributions provided in the library.

3 Convex Hull

The convex hull of a set of points P in $\mathbb{R}^d$ is the smallest convex polyhedron containing P. It is common to represent the convex hull using a set of **facets**. The boundary of two facets is a **ridge**. For example, in $\mathbb{R}^3$, assuming the points are in general position (no four points are on the same plane), each facet is a triangle, and each ridge is a line that borders two facets (see Figure 2(a)).

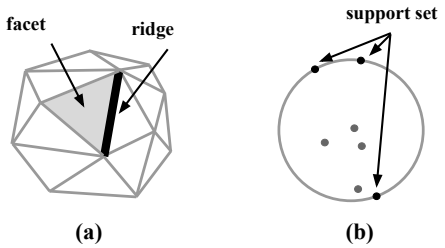


Figure 2 (a) A facet and a ridge of a convex hull in $\mathbb{R}^3$. (b) The support of the smallest enclosing ball in $\mathbb{R}^2$.

The randomized incremental algorithm and the quickhull algorithm are the most widely used algorithms for solving convex hull in practice. The randomized incremental algorithm for $\mathbb{R}^d$ was proposed by Clarkson and Shor [27]. Given a point data set P in $\mathbb{R}^d$, the randomized incremental algorithm first constructs a d-simplex, a generalization of a tetrahedron in d-dimensions as the initial hull. Then, the algorithm adds the points to the polyhedron in a random order, updating the hull if necessary. In practice, the quickhull algorithm [33, 14], another incremental algorithm, is often used. Unlike the randomized algorithm, the quickhull algorithm processes a point that is furthest from a facet, which enables the hull to be expanded more quickly. The quickhull algorithm is by far one of the most common implementations for convex hull due to its simplicity and efficiency [4, 6, 7, 1, 3, 2]. There have also been works that study parallel implementations of quickhull, but they are either limited to $\mathbb{R}^2$ [41, 48], or do not return the exact convex hull for $\mathbb{R}^3$ [49, 51]. Recently, Blelloch et al. [24] proposed a new randomized incremental algorithm that is highly parallel in theory. However, the algorithm does not seem to be practical due to numerous data structures required for bookkeeping.

In this section, we describe our new parallel reservation-based algorithm. Our algorithm is able to express both the randomized incremental convex hull algorithm and the quickhull algorithm. Specifically, unlike a sequential incremental algorithm that adds one point per round, we add multiple points in parallel per round. We resolve conflicts caused by the parallel insertion using a reservation technique. We also apply a general parallelization technique based on divide-and-conquer, which in combination with our parallel incremental algorithm, leads to faster implementations in practice.

Parallel Reservation-Based Algorithm. Our parallel reservation-based algorithm can be implemented as either a randomized incremental algorithm or a quickhull algorithm. We will first introduce the overall structure of the algorithm. Then, we will describe the details about the implementations, and compare with existing approaches. We will base our description in the context of $\mathbb{R}^3$ for the sake of clarity, but the algorithm can be extended to $\mathbb{R}^d$ for any constant integer $d \geq 2$.

We first give a high-level overview of the algorithm. Given an ordered set of points $P = \{p_1, p_2, \ldots, p_n\}$, we let $P_r = \{p_1, p_2, \ldots, p_r\}$ be the prefix of P of size r, and $CH(P_r)$ be the convex hull on P_r. We start the construction by first arbitrarily selecting four points from P that do not lie on the same plane and putting them at the beginning of P, forming a tetrahedron $CH(P_4)$. Then, the algorithm proceeds iteratively, but on each round, rather than inserting just p_r to form $CH(P_r)$, we process a batch of points in parallel. On each round, let each point outside of $CH(P_{r-1})$ be called a **visible point**. We first select a batch of visible points, and try to add them to $CH(P_{r-1})$ in parallel in the same round.

The key challenge of this approach is that some of these points cannot be processed in parallel due to concurrent modifications on the shared structures of the convex polyhedron. We use a reservation algorithm to resolve these conflicts, such that we only process the points that modify disjoint facets of the polyhedron. Specifically, each point will perform a priority write [45] with its ID to reserve all of its visible facets. Points that have their ID written to all of its visible facets are *successful*. We then process the successful points in parallel by enabling them to make modifications to $CH(P_{r-1})$. At the end of the round, in parallel, we filter out points that are no longer visible. The algorithm will terminate when there are no more visible points.

We now describe the algorithm in more detail. Figure 3 illustrates the processing of a visible point p_r. We denote a facet as a **visible facet** of p_r if point p_r is in the half space away from the center of the convex hull. We first retrieve the set of visible facets of p_r via

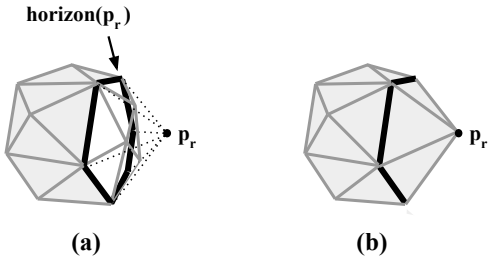


Figure 3 Illustration of adding a visible point p_r to the convex hull. (a) shows the convex hull prior to the addition of p_r. The visible facets are in white, while the non-visible facets are in gray. The thicker line segments correspond to the horizon. (b) shows the convex hull after adding p_r with newly created facets.

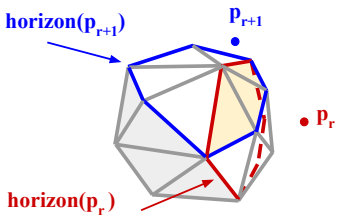


Figure 4 This figure illustrates the attempt to add p_r and p_{r+1} in parallel. The visible points and horizons of p_r and p_{r+1} are in red and blue, respectively. The visible facets to either visible points are in white/yellow, while the other facets are in gray. The overlap of the three visible facets between the p_r and p_{r+1} is in yellow.

facets stored in it. The visible facets of p_r form a closed region, whose boundary is a set of ridges, known as the **horizon**. We delete the visible facets from $CH(P_{r-1})$, and replace them with new facets, where each new facet is formed by adding two ridges from a horizon ridge to p_r.

Because of the structural changes to the convex hull that occur when adding a visible point, concurrent structural changes can cause data races, which need to be avoided. We show an example of the conflict in Figure 4, where we are attempting to add two visible points p_r and p_{r+1} in parallel. As shown in the figure, the closed regions formed by the visible facets of each visible point overlap with each other in three facets, which are highlighted in yellow. Should the two visible points be processed in parallel, the resulting polyhedron may not be well-defined due to data races. When processed sequentially, p_{r+1}'s visible facets would have been different, involving newly created facets by p_r.

Our reservation algorithm allows only a subset of the visible points that update disjoint facets of the convex hull to be processed in parallel on each round. At a high level, we use the lexicographical order of the visible points to determine the priority in processing a facet (a smaller ID has higher priority). In the example shown in Figure 4, since p_r has a smaller ID than p_{r+1}, the three conflicting facets can only be processed by p_r in that round. The pseudocode for the algorithm is shown in Figure 5. P is processed iteratively until it is empty (Line 4). We allocate an extra data field in each facet for performing reservations (Lines 6–8). For each visible point in parallel, we iterate through its visible facets and use a parallel priority write (**WriteMin**) to write its ID to the facets' "reservation" fields. Then on Lines 9–11, we determine which visible points successfully reserved all of its facets. Again, in

```
1  Input: 3-dimensional points P, batch size r
2  Output: 3-dimensional convex hull
3  CH := initialize with 4 points
4  while (P is not empty):
5    Q := a batch of size r of visible points in P
6    par_for (q in Q): /* reservation */
7      for (f in q.visibleFacets):
8        WriteMin(&f.reservation, q.id)
9    par_for (q in Q): /* check reservation */
10     for (f in q.visibleFacets):
11       q.success &&= (f.reservation == q.id)
12   par_for (q in Q): /* process successful points */
13     if (q.success):
14       delete q's visible facets
15       create new facets of q
16       update CH
17   P := ParallelPack(P, visible)
```

Figure 5 Pseudocode for the parallel reservation-based convex hull algorithm (which includes the randomized incremental algorithm and the quickhull algorithm).

parallel for each visible point, we check each of its visible facets for a successful reservation by comparing the value of the reservation field with its token. The reservation of a visible point is only successful if its ID is stored in all of its visible facets. Then, on Lines 12–16, we process the visible points whose reservations are successful, adding them to the hull and updating the appropriate data structures. Finally, on Line 17, we process the points in P such that those remaining as visible points are packed to replace the original P, and the non-visible points are discarded. Note that the visible points that succeeded in the reservation are no longer visible points because they are now part of the convex hull. Some of the remaining points will also no longer be visible points due to the growth of the convex hull.

We use a simple and fast data structure to keep track of the visibility relationship between the visible points and the facets. At each step of the algorithm, when a visible point is processed, it needs to identify the set of visible facets. On the other hand, for the facets undergoing structural changes, they need to identify and redistribute their visible points to new facets. To find the set of visible facets of p_r, it is inefficient to iterate through all of the facets of $CH(P_{r-1})$. While existing approaches [29] keep track of the visibility between visible points and *all* of their visible facets, we found such an approach to be slow becaseu each vertex is associated with multiple facets, making the cost of storing and updating the data structure high. We only store the reference of an arbitrary visible facet to each visible point, from which we use a local breadth-first search to retrieve all of the visible facets only when needed. For storing the visible points in the facets, we assign each point to one of its visible facets. During point redistribution, we gather the points stored in each visible facet into an array, and in parallel distribute each point to a new visible facet replacing the original visible facet. Each such point also stores a reference to this visible facet.

Our reservation-based algorithm can be used to implement the parallel randomized incremental algorithm or the quickhull algorithm for convex hull. For the randomized incremental algorithm, we randomly permute the input points at the beginning, and on each round attempt to add a prefix of the permuted points to the convex hull. For the quickhull algorithm, on each round, we instead select a set of points furthest from a subset of facets.

We describe the implementation of the two algorithms in greater detail in the full version of our paper. Our reservation-based algorithm is inspired by the idea of "deterministic reservations" from Blelloch et al. [22], who introduce this approach to implement parallel algorithms for other problems. In the full version of our paper, we show the work overhead of doing reservations compared to the sequential algorithm is small.

Parallel Divide-and-Conquer. We adopt a common parallelization strategy using divide-and-conquer, which calls our reservation-based algorithm as a subroutine. Some early convex hull algorithms are based on divide-and-conquer, notably, the algorithm by Preparata and Hong [42]. The algorithm splits the input into two spatially disjoint subsets by a mid-point along one of the axis, recursively computes the convex hull on each subset, and then merges the results together. Later work [10, 28, 11] extended this approach to the parallel setting. However, most of these approaches rely on complicated subroutines to merge convex hulls, which are not practical and have not been implemented, to the best of our knowledge.

We implement a practical divide-and-conquer algorithm by partitioning the input into $c \cdot numProc$ equal subsets, where c is a small constant and $numProc$ is the number of processors. For each subset, the convex hull of the subset is computed by a single processor using the sequential quickhull algorithm, but run in parallel across the different subsets. Then, the vertices of the outputs of the subproblems are collected to form a new input, from which the final convex hull is computed using our reservation-based parallel algorithm described earlier.

Point Culling via Pseudohull Computation. We also implement a multicore variant of Tang et al.'s pseudohull heuristic [50], originally proposed for the GPU. Starting from an initial tetrahedra, we recursively grow each facet into three new facets, using the furthest point from the facet, similar to the quickhull algorithm. The visible points associated with the facet are redistributed to the new facets. This results in a polyhedron, and the points in the interior of the polyhedron will not be part of the convex hull. Therefore, we can prune away the points inside the polyhedron and compute the convex hull on the rest of the points.

There are several differences in our implementation from Tang et al.'s algorithm. Our implementation executes the recursive calls on different facets asynchronously in parallel, whereas Tang et al.'s implementation maps the algorithm to the GPU architecture by pre-allocating space for the facets and visible points, and runs the algorithm in an iterative manner in lock-step. Specifically, successively generated facets and points points associated them are updated by multiple threads in parallel in each iteration. We use a parallel maximum-finding routine to find the furthest point of each facet in each call. Rather than growing the pseudohull until there are no more visible points as done by Tang et al., we set a threshold on the number of points associated with a facet, below which we stop growing the pseudohull. This prevents stack overflow on large and skewed data sets due to too many recursive calls, and the extra unpruned points do not contribute significantly to the work of the final computation of the convex hull. At the end of pruning, we use our parallel reservation-based quickhull algorithm to compute the final hull on the remaining points, whereas Tang et al. uses a sequential implementation.

4 Smallest Enclosing Ball

The smallest enclosing ball of P in $\mathbb{R}^d$ is the smallest d-sphere containing P. It is well known that the smallest enclosing ball is unique and defined by a **support set** of $d+1$ points on the surface of the ball (see Figure 2(b)).

Welzl [53] showed that by using a randomized incremental algorithm, the smallest enclosing ball can be computed in $O(n)$ time in expectation for constant d. The algorithm iteratively expands the support set of the ball by adding points in a random order until the ball contains all of the points. The algorithm was later improved by Gartner [32] with practical optimizations for speed and robustness. Larsson et al. [37] proposed practical parallel algorithms that use a new method for expanding the support set, and their implementations work on both CPUs and GPUs. Later, Blelloch et al. [23] proposed a parallel algorithm based on Welzl's algorithm, but without any implementations.

In this section, we describe our new algorithms for the smallest enclosing ball problem based on Larsson et al.'s approach [37]. We propose a sampling-based algorithm to quickly reduce the size of the data set. We also provide the first parallel implementation of Welzl's classic algorithm.

Given a ball B, we define **visible points** to be points that lie outside of B. Existing approaches for computing the smallest enclosing ball focus on expanding the support set in an iterative manner, and output the enclosing ball when there are no more visible points. Welzl's algorithm expands the support set by adding points in a random order [53]. In comparison, Larsson et al.'s approach scans the input to search for good support sets in a round-based manner. In $\mathbb{R}^3$, Larsson's algorithm divides the space into eight orthants centered at the center of B. On each round, the input is scanned to find the furthest visible points in each orthant. B is then updated to the next intermediate solution using the existing support set of B and the new visible points found during the scan. The algorithm iterates until there are no more visible points. It is parallelized within each round by performing the scan on the input in parallel.

Sampling-Based Algorithm. We find each iteration in Larsson et al.'s algorithm to be unnecessarily expensive due to having to scan the entire data set on every round. Our approach is to use a sampling heuristic to first obtain a good initial ball, inspired by Welzl's randomized algorithm. Specifically, we use small random samples to obtain good estimates of the support set at a negligible cost.

We show the pseudocode of our algorithm in Figure 6. Our sampling-based algorithm consists of two phases: the sampling phase (Line 5–13) and the final compute phase (Line 15–20). First, we initialize the ball using a few arbitrary points (Line 3). Then, we iterate through a random permutation of the input to take multiple samples (Line 5–13). On each iteration, we scan through a constant-sized segment of the unseen part of the input, which is equivalent to a random sample. We perform an orthant scan similar to Larsson's approach. Our implementation of orthant scan will return a new estimate of the support set based on the sample, and a boolean *hasOutlier* indicating whether the sample contains visible points with respect to the current smallest enclosing ball B (Line 7). We recompute B using the new support set. If there are visible points in the current sample, we will continue the sampling process with our new B. If there are no visible points in the sample, the support set likely contains most of the points, and so we terminate sampling and move on to the next phase. Now, with a good estimate of the optimal smallest enclosing ball, we run Larsson's orthant scan to compute the final smallest enclosing ball (Line 15–20). The sampling phase allows us to generate good support sets without having to scan the entire input.

We parallelize the orthant scan, which is the most expensive operation of the algorithm. Specifically, we divide the input array to orthant scan into blocks, and process each block sequentially, but in parallel across different blocks. Afterward, the extrema for the orthants obtained from the blocks are merged, and a new support set is computed on these points and the existing support set of B.

```
Input: d-dimensional points P, batch size c
Output: d-dimensional smallest enclosing ball
B = ball()
/* Sampling phase */
scanned = 0
while (scanned < n):
  hasOutlier, support =
    orthantScan(P[scanned:min(scanned+c,n)-1],B)
  scanned += c
  if (!hasOutlier):
    break /* current sample does not violate B */
  else
    B = constructBall(support)
/* Final computation phase */
while (hasOutlier):
  hasOutlier, support = orthantScan(P, B)
  if (!hasOutlier):
    return B
  else
    B = constructBall(support)
```

Figure 6 Pseudocode for the parallel sampling-based algorithm for smallest enclosing ball.

We parallelize the orthant scan, which is the most expensive operation of the algorithm. Specifically, we divide the input array to orthant scan into blocks, and process each block sequentially, but in parallel across different blocks. Afterward, the extrema for the orthants obtained from the blocks are merged, and a new support set is computed on these points and the existing support set of B.

Parallel Welzl's Algorithm and Optimizations. We also implemented and optimized the parallel version of Welzl's algorithm described by Blelloch et al. [23]. Welzl's sequential algorithm uses a random permutation of the input P and processes the points one by one. If the algorithm encounters a visible point p_i with respect to the current bounding ball B, B is recomputed on P_i, the prefix of points up until p_i, using recursive calls to the algorithm. Blelloch et al.'s parallel algorithm also uses a random permutation of P. Across iterations, the algorithm processes prefixes of P of exponentially increasing size. If the prefix contains at least one visible point, the earliest visible point p_i is identified, and B is recomputed on prefix P_i by recursively calling the parallel algorithm. Each prefix is processed in parallel.

We implement the algorithm with some practical optimizations. When there are numerous visible points in the prefix, the work of the parallel algorithm will increase significantly, because each time a visible point is discovered, the points after the visible point in the same prefix will have to be reprocessed in the next round. Therefore, given that there will be more visible points in the initial rounds when the prefix size is small (< 500000), we process these prefixes sequentially by calling Welzl's sequential algorithm. This also reduces the amount of overhead from parallel primitives, since there is limited parallelism for small prefixes.

In addition, we extend existing optimizations of Welzl's sequential algorithm to the parallel setting. We implement the move-to-front heuristic [53], which upon encountering a visible point, moves the visible point to the front of P, so that it will be processed earlier in recursive calls, reducing the number of subsequent visible points. We also parallelize the pivoting heuristic proposed by Gartner [32]. In this heuristic, upon encountering a visible

point, rather than processing the visible point directly, we search P for a *pivot point* furthest away from the center of the current B, and use the pivot point to compute the new B instead of the visible point. We use a parallel maximum-finding algorithm to identify the pivot point.

5 Parallel Batch-Dynamic kd-tree

The kd-tree, first proposed by Bentley [16], is a binary tree data structure that arranges and holds spatial data to speed up spatial queries. At each node, the data set is split into two using an axis-aligned hyperplane along a dimension, until the node holds a small constant number of points. kd-trees are used in a wide range of applications, such as in databases, machine learning, data compression, and cluster analysis.

In this section, we introduce the BDL-tree, a parallel batch-dynamic kd-tree implemented using the logarithmic method [17, 18]. Our BDL-trees build on ideas from the Bkd-Tree by Procopiuc et al. [44] and the cache-oblivious kd-tree by Agarwal et al. [9]. The logarithmic method [17, 18] for converting static data structures into dynamic ones is a very general idea. At a high level, the idea is to partition the static data structure into multiple structures with exponentially growing sizes (powers of 2). Then, inserts are performed by only rebuilding the smallest structure necessary to account for the new points. In the specific case of the kd-tree, a set of N_s static kd-trees is allocated, with capacities $[2^0, 2^1, \ldots, 2^{N_s-1}]$, as well as an extra buffer tree with size 2^0. Then, when an insert is performed, the insert cascades up from the buffer tree, rebuilding into the first empty tree with all the points from the lower trees. If desired, the sizes of all of the trees can be multiplied by a buffer size X, which is a constant that is tuned for performance.

We implement the underlying static kd-trees in an BDL-tree using the van Emde Boas (vEB) [13, 30, 9] recursive layout. Agarwal et al. [9] show that this memory layout can be used with kd-trees to make traversal cache-oblivious. We provide more details of the static tree structure, and parallel algorithms for the construction, deletion, and k-NN search in the full version of our paper.

Parallel Batch Insertion. Batch insertions are performed in the style of the logarithmic method [17, 18], with the goal of maintaining the minimum number of full trees within BDL-tree. Thus, upon inserting a batch P of points, we rebuild larger trees if it is possible using the existing points and the newly inserted batch. We use a bitmask to determine which static trees in the structure to destroy and reconstruct after each insertion. Specifically, we build a bitmask F of the current set of full static trees. Given the buffer kd-tree size X, we add the value $\lfloor |P|/X \rfloor$ to F when a point set P is inserted, after which the bitwise difference with the previous F indicates which trees need to be changed. We gather the points in the trees to be destroyed, and with P, we construct a subset of new trees in parallel. As an implementation detail, note that we first add $|P| \mod X$ points to the buffer kd-tree – if we fill up the buffer kd-tree, then we gather the X points from it and treat them as part of P, effectively increasing the size of P by X. Refer to Figure 7 for an example of this insertion method ($X > 2$ in this example). In Figure 7a, the BDL-tree contains X points, giving a bitmask of $F = 1$ (because only the smallest tree is in use). If we insert $X + 1$ points, then we put one point in the buffer tree and compute $F_{new} = 1 + \lfloor \frac{X}{X} \rfloor = 2$, and so we have to deconstruct static tree 0 and build static tree 1, as shown in Figure 7b. Then, if we insert $X + 1$ points again, then we again put one point in the buffer tree and compute $F_{new} = 2 + \lfloor \frac{X}{X} \rfloor = 3$, and so we simply construct tree 0 on the X new points (leaving tree 1 intact), as seen in Figure 7c. Finally, if we then insert $X - 1$ points, this would fill the buffer

(a) Static tree 0 is full.

(b) Static tree 1 is full and buffer tree has 1 point.

(c) Static trees 0 and 1 are full and buffer tree has 2 points.

(d) Static tree 2 is full and buffer tree has 1 point.

Figure 7 A BDL-tree in various configurations with $X > 2$; starting from (a), inserting $X + 1$ points gives (b), then inserting $X + 1$ points gives (c), and then inserting $X - 1$ points gives (d).

up, and so we take 1 point from the buffer and insert X points; then, $F_{new} = 3 + \lfloor \frac{X}{X} \rfloor = 4$, and so we deconstruct trees 0 and 1, and construct tree 2, as seen in Figure 7d. We include a more detailed explanation of the algorithm, and explain the batch deletion algorithm in the full version of our paper.

Data-Parallel k-NN. In the data-parallel k-NN implementation, we parallelize over S, the set of points to search for nearest neighbors for. First, we allocate a k-NN buffer for each of the points in S. Then, iterating over each of the non-empty trees in the BDL-tree sequentially, we call the data-parallel k-NN subroutine on the tree, passing in the set S of points and the k-NN buffers. Because we reuse the same set of k-NN buffers for each k-NN call (note that each k-NN call is internally parallel), we end up with the k-nearest neighbors of the entire pointset for each point in S. We include a more detailed explanation in the full version of our paper.

6 Experimental Evaluation

Data Sets. We use several types of synthetic data sets. The first is **Uniform** (**U**), consisting of points distributed uniformly at random inside a hypercube with side length $\sqrt{n}$, where n is the number of points. The second type **InSphere** (**IS**) is similar to the first, but the points are distributed in a hypersphere instead. We also use **OnSphere** (**OS**) and **OnCube** (**OC**) data sets, where points are uniformly distributed on the surface of a hypersphere and a hypercube, respectively. The surfaces have a thickness equal to 0.1 times the diameter or side length of the sphere or cube. We name the data sets in the format of **Dimension-Name-Size**.

We also use the following real-world data sets from the Stanford 3D Scanning Repository [8]: **3D-Thai-5M** is a 3-dimensional point data set of size 4999996 from a scanned thai-statue; and **3D-Dragon-3.6M** is a 3-dimensional point data set of size 3609600 from a scanned statue of a dragon.

Testing Environment. The experiments are run on an AWS c5.18xlarge instance with 2 Intel Xeon Platinum 8124M CPUs (3.00 GHz), for a total of 36 two-way hyper-threaded cores and 144 GB RAM. We compile our benchmarks with the `g++` compiler (version 9.3.0) with the `-O3` flag, and use ParlayLib [20] for parallelism.

6.1 Convex Hull

We test the following implementations for convex hull (our new implementations are underlined). All implementations are for both $\mathbb{R}^2$ and $\mathbb{R}^3$.

- **CGAL**: sequential C++ implementation of quickhull in CGAL [2].
- **Qhull**: sequential C++ implementation of quickhull [6] by Barber et al. [14].

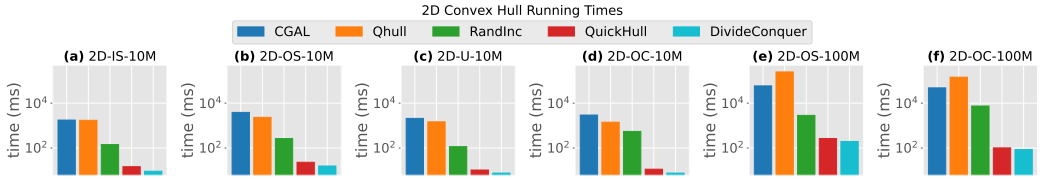


Figure 8 Running times of implementations across different data sets for 2-dimensional convex hull on 36 cores with 2-way hyper-threading.

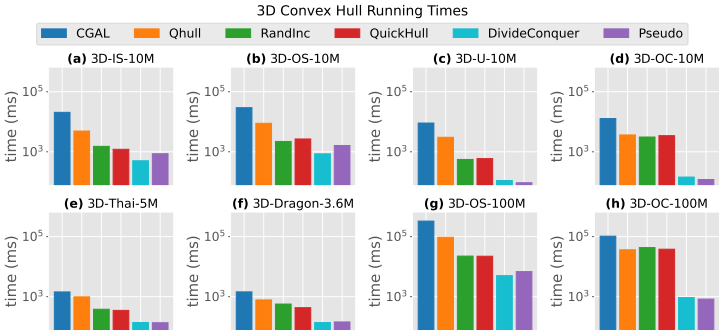


Figure 9 Running times of implementations across different data sets for 3-dimensional convex hull on 36 cores with 2-way hyper-threading.

- **RandInc**: our implementation of the parallel randomized incremental algorithm described in Section 3.
- **QuickHull**: for $\mathbb{R}^2$, it is a simple recursive parallel algorithm [19], and we use the implementation in PBBS [46]; for $\mathbb{R}^3$, we use our parallel quickhull algorithm described in Section 3.
- **Pseudo**: our implementation of the pseudoHull heuristic proposed by Tang et al. [50] for 3-dimensional convex hull described in Section 3. The final stage of the computation uses our *quickHull* algorithm for $\mathbb{R}^3$.
- **DivideConquer**: our divide-and-conquer algorithm described in Section 3.

In Figures 8 and 9, We show a comparison of running times across different methods using 36 cores with two-way hyper-threading. Our implementations achieve significant speedup compared to existing sequential implementations. Our fastest parallel implementations achieve speedups of 190–559x (325x on average) over *CGAL* for 2-dimensional convex hull, and speedups of 10.5–124x (61.4x on average) over *CGAL* for 3-dimensional convex hull. Our fastest parallel implementations have speedups of 147–1673x (605x on average) over 2-dimensional *Qhull*, and speedups of 5.68–43.8x (19.9x on average) over 3-dimensional *Qhull*. When run using a single thread, our parallel implementations achieve speedups of 3.26–12.4x and 1.31–5.05x over *CGAL* for 2 and 3 dimensions, respectively; and 3.39–47.6x and 0.99–2.06x speedups over *Qhull* for 2 and 3 dimensions, respectively.

For $\mathbb{R}^2$, *DivideConquer* is always the fastest method due to having high scalability from processing many independent subproblems in parallel. For $\mathbb{R}^3$, the fastest two methods are *DivideConquer* and *Pseudo*. We observe that on data sets with a larger output size, *Pseudo* is slower than *DivideConquer* (Figures 9(a), (b), and (g)). This is because the final computation after pruning takes longer given that there are a higher number of remaining points after pruning. For instance, the number of remaining points for *3D-IS-10M* and

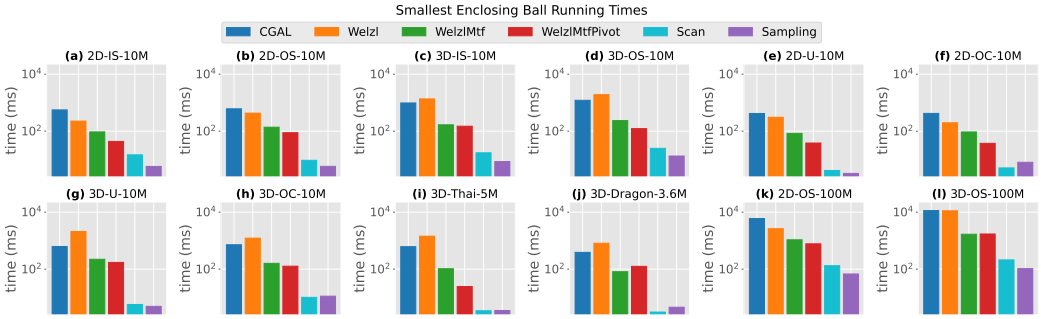


Figure 10 Running times of implementations across different data sets for smallest enclosing ball on 36 cores with 2-way hyper-threading.

3D-U-10M after pruning are 83669 and 2316, respectively, and *Pseudo* is relatively slower on the former. We observe that *RandInc* and *QuickHull* take relative longer compared with the fastest methods for data sets with a smaller output size (Figures 9(c)–(e) and (h)). This is caused by higher contention during the reservation of facets, since there are fewer facets on the intermediate hull. For example, for *3D-IS-10M* and *3D-U-10M*, the output sizes are 14163 and 423, respectively. During the computation, *3D-U-10M* exposes fewer facets for reservation, leading to a lower success rate during the reservations.

DivideConquer achieves the best parallel speedup (42.78x and 16.55x on average for $\mathbb{R}^2$ and $\mathbb{R}^3$, respectively). This is because the bulk of the time is spent in computing independent convex hulls across different threads. On the other hand, the incremental algorithms, *RandInc* and *QuickHull*, demonstrate lower scalability because of load imbalance caused by the different amounts of work for each conflict point being processed in parallel.

6.2 Smallest Enclosing Ball

We test the following implementations for smallest enclosing ball (our new implementations are underlined). All implementations work for both $\mathbb{R}^2$ and $\mathbb{R}^3$.

- **CGAL**: sequential C++ implementation of Welzl's algorithm in CGAL [2].
- **Orthant-scan**: our implementation of Larsson et al.'s orthant-scan algorithm [37].
- **Sampling**: our parallel sampling algorithm described in Section 4.
- **Welzl**: our parallel implementation of Welzl's algorithm described in Section 4.
- **WelzlMtf**: the same as *Welzl*, but with the move-to-front heuristic [10].
- **WelzlMtfPivot**: the same as *Welzl*, but with both the move-to-front and the pivoting heuristic [32].

For smallest enclosing ball, we show the comparison across implementations using 36 cores with two-way hyper-threading in Figure 10. Our fastest parallel implementations have speedups of 70–178x (109x on average) over *CGAL*. On one thread, our fastest implementations achieve speedup of 2.81–7.05x (4.96x on average) over *CGAL*.

Our sampling-based method is the fastest for eight out of the twelve data sets, whereas *Orthant-scan* without sampling is the fastest for the other four. We observe that the sampling phase on average scans only about 5% of the data set, and results in up to 2.55x (1.47x on average) speedup compared to just running *Orthant-scan*. Comparing across different implementations of Welzl's algorithms, we see that the move-to-front, and the pivoting heuristic implemented in parallel consistently improve the running times. Specifically, *WelzlMtf* is 2.09–13.9x faster than *Welzl*, and *WelzlMtfPivot* is 3.4–58.6x faster than *Welzl*. We also see that *Sampling* and *Orthant-scan* are 4.63–34.8x and 2.96–40.3x faster than *WelzlMtfPivot*, respectively.

6.3 BDL-tree

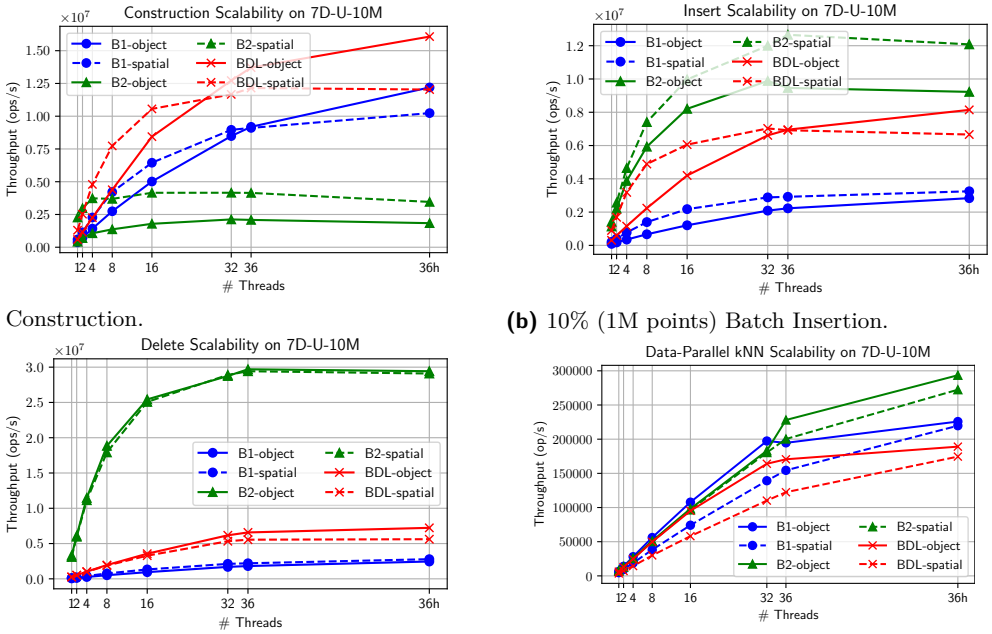
We designed a set of experiments to investigate the performance and scalability of BDL-tree and compare it to two baselines that we also implemented. **B1** is a baseline where the kd-tree is rebuilt on each batch insertion and deletion in order to maintain balance. This allows for improved query performance (as the tree is always perfectly balanced) at the cost of slowing down updates. **B2** is another baseline that inserts points directly into the existing tree structure without recalculating the splits. This results in very fast updates at the cost of potentially skewed trees (which slows down query performance). **BDL** is our BDL-tree described in Section 5. We consider splitting the points based on either using the object median (median among the points along a dimension) or the spatial median (splitting the space along a dimension in half).

Construction. Figure 11a shows the scalability of the throughput on the 7D-U-10M data set. As we can see from the results, **BDL** achieves similar or better performance both serially and in parallel than both **B1** and **B2**, and has similar or better scalability than both. With the object median splitting, it achieves up to 34.8× self-relative speedup, with an average self-relative speedup of 28.4×. We also note that the single-threaded runtimes are faster with spatial median splitting than with object median splitting. This is because spatial median only involves splitting points at each level compared with finding the median for object median, hence it is less expensive to compute; however, we also note that the scalability for spatial median is lower because there is less work to distribute among parallel threads. The construction of B2 is significantly slower than that of B1, because a separate memory buffer is allocated at each leaf node in B2 to allow for future insertions. The construction of the BDL-tree is faster than both B1 and B2 because splitting the construction across multiple trees while keeping the number of elements the same reduces the total work, and provides ample parallelism when running on multiple threads.

Batch Insertion. In this benchmark, we measure the performance of our batch insertion implementation as compared to the baselines. We measure the time required to insert 10 batches each containing 10% of the points in the data set into an initially empty tree for each of our two baselines as well as our BDL-tree.

From Figure 11b, we see that **B2** achieves the best performance on batched insertions – this is due to the fact that it does not perform any extra work to maintain balance and simply directly inserts points into the existing spatial structure. **BDL** achieves the second-best performance – this is due to the fact that it does not have to rebuild the entire tree on every insert, but amortizes the rebuilding work across the batches. Finally, **B1** has the worst performance, as it must fully rebuild on every insertion. Similar to construction, we note that spatial median splitting performs better in the serial case but has lower scalability. With object median splitting, **BDL** achieves parallel self-relative speedup of up to 35.5×, with an average self-relative speedup of 27.2×.

Batch Deletion. We measure the time required to delete 10 batches each containing 10% of the points in the data set from an initially full tree for each of our two baselines as well as the BDL-tree. From Figure 11c, we observe that **B2** has vastly superior performance – it does almost no work other than tombstoning the deleted points so it is extremely efficient. Next, we see that **BDL** has the second-best performance, as it amortizes the rebuilding across the batches, rather than having to rebuild across the entire point set for every delete. Finally, **B1** has the worst performance as it rebuilds on every delete. With object median splitting, **BDL** achieves parallel speedup of up to 33.1×, with an average speedup of 28.5×.



(a) Construction.

(b) 10% (1M points) Batch Insertion.

(c) 10% (1M points) Batch Deletion.

(d) Full (10M points) k-NN for $k = 5$.

Figure 11 Plot of throughput (operations per second) of batch operations over thread count for both object and spatial median implementations for the 7D-U-10M data set. The prefix of the implementation name refers to the median splitting heuristic. "36h" corresponds to 36 cores with two-way hyper-threading.

Data-Parallel k-NN. We measure the performance and scalability of our k-NN implementation as compared to the baselines. As shown in in Figure 11d, the results show that **B1** and **B2** have similar performance. Furthermore, they are both faster than BDL-tree. This is to be expected, because the k-NN operation is performed directly over the tree after it is constructed over the entire data set in a single batch. Thus, both baselines will consist of fully balanced trees and will be able to perform very efficient k-NN queries. On the other hand, **BDL** consists of a set of multiple trees, which adds overhead to the k-NN operation, as it must be performed separately on each of these individual trees. In the full version of our paper, we show that when the trees are constructed via a set of batch insertions rather than all at once, the performance of **B2** suffers significantly due to the tree being unbalanced.

Comparison with Zd-tree. We compared with the Zd-tree recently proposed by Blelloch and Dobson [21]. The Zd-tree data structure combines the approach of a kd-tree and Morton ordering of the data set, and supports parallel batch-dynamic insertions and deletions, and k-NN. The implementation currently only supports 2 and 3 dimensional data sets, whereas our implementation is not restricted to 2 and 3 dimensions. We tested their implementation on 3D-U-10M. Using all threads, their implementation takes 0.12 seconds to construct, and an average of 0.026 and 0.024 seconds for insertion and deletion of 10% of the data points, and takes 1.65 seconds for k-NN. Our BDL-tree implementation is 3.3×, 23.1×, and 45.83× slower, for construction, insertion, and deletion, respectively, but achieves roughly the same speed for k-NN search. The reason is that the Morton sort used in their implementation is fast and highly optimized for 2 and 3 dimensions; however, extending this technique to higher dimensions would result in overheads due to more bits needed for the Morton ordering.

7 Conclusion

In this paper, we presented ParGeo, a multicore library for computational geometry containing modules for fundamental tasks including kd-tree based spatial search, spatial graph generation, and algorithms in computational geometry. We also presented new parallel algorithms, implementations, and optimizations for convex hull, smallest enclosing ball, and parallel batch-dynamic kd-tree. We performed a comprehensive experimental study showing that our new implementations achieve significant speedups over prior work and obtain high parallel scalability.

References

1. C++ implementation of the 3d quickhull algorithm. `https://github.com/akuukka/quickhull`.
2. The computational geometry algorithms library. `https://www.cgal.org/`.
3. Header only 3d quickhull in c99. `https://github.com/karimnaaji/3d-quickhull`.
4. A header-only C implementation of the quickhull algorithm for building n-dimensional convex hulls and Delaunay meshes. `https://github.com/leomccormack/convhull_3d`.
5. Matlab geometry toolbox for 2d/3d geometric computing. `https://github.com/mattools/matGeom`.
6. Qhull. `http://www.qhull.org/`.
7. Quickhull3d: A robust 3d convex hull algorithm in Java. `https://www.cs.ubc.ca/~lloyd/java/quickhull3d.html`.
8. The Stanford 3d scanning repository. `http://graphics.stanford.edu/data/3Dscanrep/`.
9. Pankaj K. Agarwal, Lars Arge, Andrew Danner, and Bryan Holland-Minkley. Cache-oblivious data structures for orthogonal range searching. In *Proceedings of the Symposium on Computational Geometry*, pages 237–245, 2003.
10. A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3(1):293–327, March 1988.
11. Nancy M. Amato and Franco P. Preparata. The parallel 3d convex hull problem revisited. *International Journal of Computational Geometry & Applications*, 2(02):163–173, 1992.
12. Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. The problem-based benchmark suite (PBBS), v2. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 445–447, 2022.
13. Lars Arge, Gerth Stølting Brodal, and Rolf Fagerberg. Cache-oblivious data structures. In *Handbook of Data Structures and Applications*, pages 545–565. Chapman and Hall/CRC, 2018.
14. C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, December 1996.
15. Vicente H.F. Batista, David L. Millman, Sylvain Pion, and Johannes Singler. Parallel geometric algorithms for multi-core computers. *Computational Geometry*, 43(8):663–677, 2010.
16. Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.
17. Jon Louis Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244–251, 1979.
18. Jon Louis Bentley and James B Saxe. Decomposable searching problems I. static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.
19. Guy E Blelloch. *Vector models for data-parallel computing*, volume 2. MIT Press Cambridge, 1990.
20. Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. ParlayLib - a toolkit for parallel algorithms on shared-memory multicore machines. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, pages 507–509, 2020.

21 Guy E. Blelloch and Magdalen Dobson. Parallel nearest neighbors in low dimensions with batch updates. In *Proceedings of the Symposium on Algorithm Engineering and Experiments*, pages 195–208, 2022.

22 Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 181–192, 2012.

23 Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. *Journal of the ACM*, 2020.

24 Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Randomized incremental convex hull is highly parallel. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, pages 103–115, 2020.

25 Paul B. Callahan and S. Rao Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 291–300, 1993.

26 Paul B. Callahan and S. Rao Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42(1):67–90, 1995.

27 Kenneth L. Clarkson and Peter W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom*, 4:387–421, 1989.

28 N. Dadoun and D.G. Kirkpatrick. Parallel construction of subdivision hierarchies. *Journal of Computer and System Sciences*, 39(2):153–165, 1989.

29 Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000.

30 Erik D Demaine. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, 8(4):1–249, 2002.

31 Junhao Gan and Yufei Tao. DBSCAN revisited: Mis-claim, un-fixability, and approximation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 519–530, 2015.

32 B. Gärtner. Fast and robust smallest enclosing balls. In *European Symposium on Algorithms*, 1999.

33 Jonathan S. Greenfield. A proof for a quickhull algorithm, 1990.

34 Yixin Hu, Qingnan Zhou, Xifeng Gao, Alec Jacobson, Denis Zorin, and Daniele Panozzo. Tetrahedral meshing in the wild. *ACM Trans. Graph.*, 37(4):60:1–60:14, July 2018.

35 Alec Jacobson, Daniele Panozzo, et al. libigl: A simple C++ geometry processing library, 2018. https://libigl.github.io/.

36 David G. Kirkpatrick and John D. Radke. A framework for computational morphology. In *Computational Geometry*, volume 2 of *Machine Intelligence and Pattern Recognition*, pages 217–248. Springer, 1985.

37 Thomas Larsson, Gabriele Capannini, and Linus Källberg. Parallel computation of optimal enclosing balls by iterative orthant scan. *Computers & Graphics*, 56:1–10, 2016.

38 D. Lebrun-Grandié, A. Prokopenko, B. Turcksin, and S. R. Slattery. ArborX: A performance portable geometric search library. *ACM Trans. Math. Softw.*, 47(1), December 2020.

39 Marco Livesu. cinolib: a generic programming header only C++ library for processing polygonal and polyhedral meshes. *Transactions on Computational Science XXXIV*, 2019. https://github.com/mlivesu/cinolib/.

40 Kurt Mehlhorn and Stefan Näher. Leda: A platform for combinatorial and geometric computing. *Commun. ACM*, 38(1):96–102, January 1995.

41 S. Näher and Daniel Schmitt. A framework for multi-core implementations of divide and conquer algorithms and its application to the convex hull problem. In *Canadian Conference on Computational Geometry (CCCG)*, 2008.

42 F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Commun. ACM*, 20(2):87–93, February 1977.

43 Franco P. Preparata and Michael I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

44 Octavian Procopiuc, Pankaj K Agarwal, Lars Arge, and Jeffrey Scott Vitter. Bkd-tree: A dynamic scalable kd-tree. In *International Symposium on Spatial and Temporal Databases*, pages 46–65, 2003.

45 Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Reducing contention through priority updates. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, pages 152–163, 2013.

46 Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: The problem based benchmark suite. In *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures*, pages 68–70, 2012.

47 Daniel Sieger and Mario Botsch. The polygon mesh processing library, 2020. http://www.pmp-library.org.

48 D Srikanth, Kishore Kothapalli, R Govindarajulu, and P Narayanan. Parallelizing two dimensional convex hull on NVIDIA GPU and Cell BE. In *International Conference on High Performance Computing (HiPC)*, pages 1–5, 2009.

49 Ayal Stein, Eran Geva, and Jihad El-Sana. Cudahull: Fast parallel 3d convex hull on the gpu. *Computers & Graphics*, 36(4):265–271, 2012. Applications of Geometry Processing.

50 Min Tang, Jie yi Zhao, Ruo feng Tong, and Dinesh Manocha. GPU accelerated convex hull computation. *Shape Modeling International (SMI) Conference*, 36(5):498–506, 2012.

51 Stanley Tzeng and John D Owens. Finding convex hulls using quickhull on the GPU. *arXiv preprint arXiv:1201.2936*, 2012.

52 Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. Fast parallel algorithms for Euclidean minimum spanning tree and hierarchical spatial clustering. In *Proceedings of the International Conference on Management of Data*, pages 1982–1995, 2021.

53 Emo Welzl. Smallest enclosing disks (balls and ellipsoids). In *New Results and New Trends in Computer Science*, pages 359–370, 1991.