A Parallel Batch-Dynamic Data Structure for the Closest Pair Problem

Yiqiu Wang
MIT CSAIL
yiqiuw@mit.edu

Shangdi Yu
MIT CSAIL
shangdiy@mit.edu

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

Abstract

We propose a theoretically-efficient and practical parallel batch-dynamic data structure for the closest pair problem. Our solution is based on a serial dynamic closest pair data structure by Golin et al. [SIAM J. on Computing, 1998], and supports batches of insertions and deletions in parallel. For a data set of size $n$, our data structure supports a batch of insertions or deletions of size $m$ in $O(m \log (1 + (n + m)/m))$ expected work and $O(\log(n+ m) \log^2(n + m))$ depth with high probability, and takes linear space. The key techniques for achieving these bounds are a new work-efficient parallel batch-dynamic binary heap, and careful management of the computation across multiple points to minimize work and depth.

We provide an optimized multicore implementation of our data structure using dynamic hash tables, parallel heaps, and dynamic $k$-d trees. Our experiments on a variety of synthetic and real-world data sets show that it achieves a parallel speedup of up to 38.57x (15.10x on average) on 48 cores with hyper-threading. In addition, we also implement and compare four parallel algorithms for static closest pair problem, for which no practical implementations exist in the literature. On 48 cores with hyper-threading, the algorithms achieve up to 51.45x (29.42x on average) speedup, and Rabin’s algorithm performs the best on average. Comparing our dynamic algorithm to the fastest static algorithm, we find that it is advantageous to use the dynamic algorithm for batch sizes of up to 70% of the data set. As far as we know, our work is the first to experimentally evaluate parallel algorithms for the closest pair problem, in both the static and the dynamic settings.

1 Introduction

The closest pair problem is a fundamental computational geometry problem with applications in robot motion planning [3, 35], computational biology [43], collision detection, hierarchical clustering, traveling salesman heuristics, greedy matching [24]. In a lot of cases, the data involved in these problems can evolve over time. In the case that a subset of the data gets updated, a dynamic algorithm can be superior to a static algorithm that recomputes the result.

There is a rich literature on sequential dynamic closest pair algorithms [2, 9, 17, 30, 39, 40, 45, 47, 48, 51]. However, none of them have been implemented and none of them are parallel. The main contribution of our paper is the design of a theoretically-efficient and practical parallel batch-dynamic data structure for the dynamic closest pair. Our solution is inspired by the sequential solution of Golin et al. [30], which takes $O(n)$ space to maintain $O(n)$ points and supports $O(\log n)$ time updates, and is the fastest existing sequential algorithm. Our parallel solution takes a batch update of size $m$ and maintains the closest pair in $O(m \log (1 + (n + m)/m))$ expected work and $O(\log(n+ m) \log^2(n + m))$ depth with high probability (whp).\(^1\). Compared to the sequential algorithm of Golin et al., our algorithm is work-efficient for single updates, and has a better complexity for multiple updates since we process the updates in batches. Our data structure is based on efficiently maintaining a sparse partition of the points (a data structure used by Golin et al. [30]) in parallel. This requires carefully organizing the computation to minimize the work and depth, as well as using a new parallel batch-dynamic binary heap that we design in this paper. As far as we know, our heap is the first parallel batch-dynamic binary heap in the literature, and may be of independent interest.

We implement our dynamic data structure with optimizations to improve practical performance. In particular, we combine multiple heaps needed in the theoretically-efficient algorithm into a single heap, which reduces overheads. We also implement a parallel batch-dynamic $kd$-tree to speed up neighborhood queries for high-dimensional data sets. We evaluate our parallel batch-dynamic algorithm on a variety of real-world and synthetic data sets, and on 48 cores with hyper-threading we achieve self-relative parallel speedups of up to 38.57x across various batch sizes. Our algorithm achieves a throughput of up to $1.35 \times 10^7$ and $1.06 \times 10^7$ updates per second for insertions and deletions respectively.

In addition, we implement and evaluate four parallel algorithms for the static closest pair problem. There has been a rich literature on sequential [5, 7, 8, 19, 22, 26, 29, 31, 33, 42, 46] and parallel [4, 11, 13, 37, 38] static algorithms for the closest pair. However, none of the existing algorithms have been evaluated empirically. We implement a divide-and-conquer

\(^1\) A bound holds with high probability (whp) on an input of size $n$ if it holds with probability at least $1 - 1/n^c$ for some constant $c > 0$.
algorithm [13] with $O(n \log n)$ work and $O(\log^2 n)$ depth, a variant of Rabin’s randomized algorithm [42] with $O(n)$ expected work and $O(\log n \log^{*} n)$ depth whp, our parallelization of the sequential sieve algorithm [33] with $O(n)$ expected work and $O(\log n \log^{*} n)$ depth whp, and an incremental algorithm [11] with $O(n)$ expected work and $O(\log n \log^{*} n)$ depth whp. We optimize the code and compare their performance. On 48 cores with hyper-threading, our algorithms achieve self-relative parallel speedups of up to 51.45x. Our evaluation of the static algorithms show that Rabin’s algorithm is on average 7.63x faster than the rest of the static algorithms. Finally, we compare our parallel batch-dynamic algorithm with the static algorithms and find that it is advantageous to use the batch-dynamic algorithm for batches containing up to 70% of the data set.

We summarize our contributions below.

1. The first parallel algorithm for batch-dynamic closest pair, which is work efficient, and has polylogarithmic depth.
2. A work-efficient parallel batch-dynamic binary heap, which can be of independent interest.
3. Highly-optimized implementations of our parallel batch-dynamic algorithm, and four existing parallel static algorithms for the closest pair problem.
4. The first experimental evaluation of parallel static and dynamic closest pair algorithms, which shows that our algorithms achieve excellent parallel speedup.

2 Preliminaries

In this section, we overview the concepts, notations, and the computational model used in this paper. We summarize all notations used throughout the paper in Table 1.

**Problem Definition.** We consider a metric space $(S, d)$ where $S$ contains $n$ points in $\mathbb{R}^d$, and $d$ is the $L_p$-metric where $1 \leq p < \infty$. The static closest pair problem computes and returns the closest pair distance $\delta(S) = \min\{d(p, q) \mid p, q \in S, p \neq q\}$, and point pair $p$ and $q$. The dynamic closest pair problem computes the closest pair of $S$, and also maintains the closest pair upon insertions and deletions of points. A parallel batch-dynamic data structure processes batches of insertions and deletions of size $m$ in parallel. In this paper, we propose algorithms for static and parallel batch-dynamic closest pair on $(S, d)$, and our implementations and experiments use the Euclidean metric ($L_2$-norm).

**Computational Model.** We use the classic work-depth model for analyzing parallel shared-memory algorithms [21, 32]. The work $W$ of an algorithm is the number of instructions in the computation, and the depth $D$ is the longest sequential dependence chain length. Using Brent’s scheduling theorem [15], we can execute a parallel computation in $W/p + D$ running time using $p$ processors. In practice, we use the randomized work-stealing scheduler in Cilk, which achieves a running time of $W/p + O(D)$ in expectation [14]. We say that a parallel algorithm is work-efficient if its work asymptotically matches the work of the best sequential algorithm for the same problem. We assume that arbitrary concurrent writes are supported in $O(1)$ work and depth.

Our pseudocode uses the fork and join keywords for fork-join parallelism [21]. A fork creates a task that can be executed in parallel with the current task, and a join waits for all tasks forked by the current task to finish.

**Parallel Primitives.** Prefix sum takes as input a sequence $[a_1, a_2, \ldots, a_n]$, an associative binary operator $\oplus$, and an identity $i$, and returns the sequence $[i, a_1 \oplus a_2, \ldots, (a_1 \oplus a_2 \oplus \ldots \oplus a_{n-1})]$ as well as the overall sum of the elements. Filter takes an array $A$ and a predicate function $f$, and returns a new array containing $a \in A$ for which $f(a)$ is true, in the same order that they appear in $A$. Both prefix sum and filter can be implemented in $O(n)$ work and $O(\log n)$ depth [32]. We use a parallel minimum algorithm, which computes the minimum of $n$ points in $O(n)$ expected work and $O(1)$ depth whp [52].

We use parallel dictionaries, which support $n$ insertions, deletions, or lookups in $O(n)$ expected work and $O(\log^* n)$ depth whp [28]. Finally, we use integer sorting on $n$ keys in the range $[0, \ldots, O(\log n)]$, which takes $O(n)$ work and $O(\log n)$ depth [44, 52].

3 Review of the Sequential Closest Pair Data Structure

In this section, we review the sequential dynamic closest-pair data structure proposed by Golin et al. [30], which implements the serial static closest pair algorithm by Khuller and Matias [33]. Our new parallel algorithm also uses this data structure which is referred to as the sparse partition of an input set.

3.1 Sparse Partition

Given an input set $S$ with $n$ elements, a sparse partition $[30]$ is defined as a sequence of $5$-tuples $(S_i, S'_i, p_i, q_i, d_i)$ with size $L (1 \leq i \leq L)$. The sequence is constructed inductively using the following rules until $S_{i+1} = \emptyset$:

1. $S_1 = S$
2. $S'_i \subseteq S_i \subseteq S_{i+1}$
3. If $|S_i| > 1$, then $p_i$ is uniformly drawn in $S_i$, and $d_i = d(p_i, q_i) = d(p_i, S_i)$, which is the closest neighbor of $p_i$ in $S_i$
4. For all $x \in S_i$:
   - (4.1) If $d(x, S_i) > d_i/3$ then $x \in S'_i$
   - (4.2) If $d(x, S_i) \leq d_i/6k$ then $x \notin S'_i$
   - (4.3) If $x \in S_{i+1}$, then there is a point $y \in S_i$ such that $d(x, y) \leq d_i/3$ and $y \in S_{i+1}$
5. $S_{i+1} = S_i \setminus S'_i$

In expectation, the sparse partition contains $O(\log n)$ levels, and $|S_i|$ decreases geometrically, so the expected sum of all
3.2 A Grid-Based Implementation of Sparse Partition

We now describe Golin et al.’s grid-based implementation of the sparse partition. There are \( L \) levels of the sparse partition, and we refer to each as level \( i \) (\( 1 \leq i \leq L \)). We maintain each level using a grid data structure, which is similar to many closest pair algorithms (e.g., [29, 30, 33, 42]).

To represent \( S_i \), we place the points into a grid \( G_i \) with equally-sized axis-aligned grid boxes with side length \( d_i/6k \), where \( k \) is the dimension, and \( d_i \) is the closest pair distance of the randomly chosen pivot \( p_i \). Denote the neighborhood of a point \( p \) in \( G_i \) relative to \( S \) by \( N_i(p, S) \), which refers to the set of points in \( S \setminus \{ p \} \) contained in the collection of \( 3^k \) boxes bordering the box containing \( p \), including \( p \)’s box. We say that point \( p \) is sparse in \( G_i \) relative to \( S \) if \( N_i(p, S) = \emptyset \). We use this notion of sparsity to define \( S_i' = \{ p \in S_i : p \text{ is sparse in } G_i \text{ relative to } S_i \} \). The points in \( S_i' \) are also stored in a separate grid.

To construct a grid based sparse partition, the algorithm proceeds in rounds, where in each round one of the \( i \)-th levels is constructed. The algorithm starts with \( i = 1 \) where \( S_1 = S \), and we iteratively determine the side length of grid \( G_i \) based on a random pivot, and place \( S_i \) into \( G_i \). Then we compute \( S_i' \) based on point sparsity defined above, and set \( S_{i+1} = S_i \setminus S_i' \). The algorithm proceeds until \( S_i = S_i' \) (i.e., \( S_{i+1} = \emptyset \)). The expected work for construction is \( O(n) \) since \( |S_i| \) decreases geometrically [30]. The correctness of the algorithm is also proved in [30].

We give an example of the grid-based implementation of the sparse partition in Figure 1. We illustrate the grid \( G_i \) for the \( S_i \) of each level, as well as the pivot \( p_i \) and \( p_i \)’s closest neighbor \( q_i \). The grid size is set to \( d_i/6k = d(p_i, q_i)/12 \) for \( k = 2 \). The sparse points, represented by the hollow blue circles, have empty neighborhoods, and they do not have a closest neighbor within a distance of \( d_i/3 \) away. The solid

| Notation | Definition |
|----------|------------|
| \( k \)  | Dimensionality of the data set. |
| \( S \)  | Point data set \( \{p_1, p_2, \ldots, p_n\} \) in \( \mathbb{R}^k \). |
| \( n \)  | Size of \( S \) (|\( S |)). |
| \( m \)  | Size of a batch update. |
| \( d(p, q) \) | Distance between points \( p, q \in S \). |
| \( \delta(S) \) | \( \min\{d(p, q) : p, q \in S, p \neq q\} \), i.e., the distance of the closest pair in set \( S \). |
| \( d(p, S) \) | \( \min\{d(p, q) : q \in S \setminus p\} \), i.e., the distance of \( p \) to its nearest neighbor in set \( S \). |
| \( d_i^*(p) \) | The restricted distance of point \( p, d_i^*(p) := d(p, S_i' \cup S_{i+1}' \cup \ldots \cup S'_{i+L-1}). |
| \( (S, S_i', p_i, q_i, d_i) \) | The 5-tuple representing each level of the sparse partition data structure, where \( S_i \) and \( S_i' \) are point sets, \( p_i \) is the pivot point, \( q_i \) is the closest point of \( p_i \) in \( S_i \), and \( d_i := d(p_i, q_i) \). |
| \( H_i \) | The parallel heap associated with level \( i \) of the sparse partition. |
| \( L \) | The number of levels in the sparse partition. |
| \( b_i(p) \) | The box containing \( p \) on level \( i \). |
| \( b_i^*(p) \) | The box with a offset of \( \sigma \) relative to \( b_i(p) \). \( \sigma \) is a \( k \)-tuple over \( \{-1, 0, 1\} \), where the \( j \)’th component indicates the relative offset in the \( j \)’th dimension. |
| \( N_i(p) \) | The box neighborhood of \( p \), i.e., the collection of the \( 3^k \) boxes bordering and including the box containing \( p \) on level \( i \). |
| \( N_i^*(p) \) | The partial box neighborhood of \( p \). Specifically, the intersection of \( N_i(p) \) with the boxes bordering and including \( b_i^*(p) \). |
| \( N_i(p, S) \) | The neighborhood of \( p \) in set \( S \), i.e., the set of points in \( S \setminus p \) contained in \( N_i(p) \). |

Table 1: Summary of Notation.
As observed by both Khuller and Matias [33] and Golin et al. [30], although the grid data structure rejects far pairs, and becomes more fine-grained with a larger $i$, the grid at the last ($L$-th) level does not necessarily contain the closest pair. For example, illustrated in Figure 1, $S_1$ for the last level does not contain the closest pair $(x, y)$, as $x$ is sparse on level 2 and did not get copied to $S_1$. Therefore, we need to check all levels to find the closest pair.

The restricted distance $d_i^*(p)$ [30] is the closest pair distance to point $p$ to any point in $\bigcup_{0 \leq j \leq k} S'_{i-j}$, and defined as $d_i^*(p) := \min\{d(p, S'_{i-k} \cup S'_{i-k+1} \cup \ldots \cup S'_{i})\}$, where $p \in S'_{i}$. Golin et al. show that $\delta(S) = \min_{1 \leq k \leq L} \min_{p \in S'_{i}} d_i^*(p)$, meaning that the closest pair can be found by taking the minimum among the restricted distance pairs for all points in last $k + 1$ levels of $S'_{i}$. For completeness, we cite from Golin et al. [30] the correctness proof for $\delta(S) = \min_{1 \leq k \leq L} \min_{p \in S'_{i}} d_i^*(p)$. [30] uses a slightly different definition for the restricted distance, $d_i^*(p) := \min\{d_i, d(p, S'_{i-k} \cup S'_{i-k+1} \cup \ldots \cup S'_{i})\}$, but including $d_i$ is not required for correctness.

**Lemma 3.1.** $d_i^*(p) > d_i/6k$ for $p \in S_1$.

**Proof.** Let $1 \leq j \leq i$ and let $q \in S'_j$. Since $p \in S_j$, it follows from (b.2) of the definition of the sparse partition that $d(p, q) \geq d(q, S_j) > d_i/6k > d_i/6k$.

**Lemma 3.2.** $d_L/6k \leq \delta(S) \leq d_L$.

**Proof.** Let $\delta(S) = d(p, q)$ for some $p \in S'_j$ and $q \in S'_j$, and without loss of generality $j \leq i$. It follows from the definition of the sparse partition that $p, q \in S_i$, and we have $d(p, q) = d(p, S_i) > d_i/6k$, hence $d(p, q) > d_i/6k$.

$\delta(S) \leq d_L$ obviously holds since $d_L$ is the distance between two points.

**Theorem 3.3.** $\delta(S) = \min_{1 \leq k \leq L} \min_{p \in S'_i} d_i^*(p)$

**Proof.** Since the restricted distance is the distance between two points, $\delta(S) \leq \min_{1 \leq k \leq L} \min_{p \in S'_{i}} d_i^*(p)$. Let $\delta(S) = d(p, q)$ for some $p \in S'_j$ and $q \in S'_j$. Assume without loss of generality that $j \leq i$, and it is obvious that $d(p, q) = d(p, \bigcup_{k \leq i} S'_{j}) \geq d_i^*(p)$. Therefore $\delta(S) \geq \min_{1 \leq k \leq L} \min_{p \in S'_{i}} d_i^*(p)$, hence $\delta(S) = \min_{1 \leq k \leq L} \min_{p \in S'_{i}} d_i^*(p)$.

We then restrict the value of $i$ to $L - k, L - k + 1, ..., L$. By Lemma 3.1, we have $\min_{p \in S'_{i}} d_i^*(p) > d_i/6k$. We also know
from Lemma 3.2, and the properties of the sparse partition \((d_{i+1} \leq d_i/3)\), that for \(i < L - k, d_i/6k \geq d_{L-k} \geq (3^{k+1}/6k) \cdot d_L \geq \delta(S)\).

The sequential algorithm [30] computes the restricted distance for each point in \(S'_i\), and stores in min-heaps \(H_i\), for \(1 \leq i \leq L\). To obtain the closest pair, we simply read the minima of \(H_i\) for \(L - k \leq i \leq L\) to obtain \(k + 1\) values, and then take the minimum. This takes \(O(1)\) work.

4 Parallel Batch-Dynamic Data Structure

We first give an overview of our batch-dynamic algorithms from a high level, and how they related to the sparse partition and the heaps.

In Section 4.1, we introduce a parallel algorithm that constructs the data structure given an input point set \(S\). The algorithm constructs the grid structure one level at a time, until all points become sparse. For constructing the heaps, the algorithm constructs each heap asynchronously to improve parallelism. Our construction algorithm takes \(O(n)\) expected work and \(O(\log n \log^* n)\) depth whp.

Next, we present the parallel batch update algorithms. For batch insertions, there are two main tasks: updating the grid (Section 4.2) and updating the heap (Section 4.3). Since this step is complicated, in Figure 2 we show an example for the readers to get a high-level overview of this process. In Figure 2 (left), when points \(\{f, g\}\) are inserted to the grid originally containing \(\{a, b, c, d, e\}\), we first update \(S_1\) to include \(f\) and \(g\), and \(S_2\) to include \(f\) but not \(g\), since \(g\) is sparse in \(S_1\). In addition, the sparse points \(a\) and \(e\) in \(S_1\) become non-sparse due to the new insertion of \(f\), and so we move them to \(S_2\). The insertion and movement of points among the grids constitutes the grid update step. Deletions work similarly, but in the reverse order as shown in Figure 2 (right). Given an update of size \(m\), we can update the grid in \(O(m)\) amortized work in expectation and \(O(\log(n + m) \log^*(n + m))\) depth whp.

After updating the grids, we need to update their corresponding heaps. We denote the restricted distance of point \(x\) as \((x, y)_i = d_i(x, y)\), where \(y \in \bigcup_{0 \leq j \leq k} S'_{i-j}\) is \(x\)'s neighbor that give rise to the distance. As shown in Figure 2 (left), due to the insertion of sparse point \(g\) to \(S_1\), entry \((g, b)\)1 is added to \(H_1\). Some entries in \(H_1\) are moved due to the point movements. For instance, \((a, f)\)1 from \(H_1\) is moved to \((a, f)\)2 in \(H_2\) because \(a\) has moved from \(S_1\) to \(S_2\). Some entries are updated, for instance, \((c, d)\)2 is updated to \((c, f)\)2 in \(H_2\) since the new point \(f\) is closer to \(c\) than \(d\) is. Again, deletions work similarly but in the reverse order. We describe an algorithm to make the heap updates highly parallel in subsection 4.3, where we complete all of the heap updates in \(O(m \log(1 + (n + m)/m))\) amortized work in expectation and \(O(\log(n + m) \log^*(n + m))\) depth whp. In addition, in Section 5 we design a new parallel batch-dynamic binary heap to achieve this work bound and low depth at the same time.

Reading the closest pair from our data structure takes \(O(1)\) work and depth. We call find-min on \(H_i\) for \(L - k \leq i \leq L\) to obtain \(k + 1\) values, and then take the minimum.

4.1 Parallel Construction

Our parallel construction algorithm is shown in Algorithm 1. Given an input point set \(S\), we output the grid structure and the heaps for all levels, i.e., \((S_i, S'_i, p_i, q_i, d_i)\) and \(H_i\) for all \(1 \leq i \leq L\). The BUILD\(_{ij}\) procedure takes in point set \(S_j\) and constructs the level \(i\). Initially, we set \(S_1\) to be \(S\), as shown on Line 2. Line 4 picks a pivot point \(p_i\), and computes its closest pair to determine the side length of the grid boxes. Lines 5–6 construct the level \(i\) grids. In particular, we use a parallel dictionary to store the grid boxes, and check the sparsity of each point \(x\) by looking up neighboring boxes in the dictionary based on box ID of \(x\). We also obtain \(S'_i\) during this process. On Line 7, we compute the restricted distance of each point in \(S'_i\), and then spawn a thread to asynchronously construct the heap for the restricted distances. We recursively call BUILD on Line 9 to construct the next level until all points in \(S_j\) are sparse.

Analysis. For each call to BUILD, given \(O(|S_i|)\) points, insertions to the parallel dictionary take \(O(|S_i|)\) work and \(O(\log^* |S_i|)\) depth whp. Line 4 computes the distance of \(p_i\) to each \(q_j \in S_i\), taking \(O(|S_i|)\) work and \(O(1)\) depth. Then we obtain \(q_i\) via a parallel minimum computation taking constant depth. Checking the sparsity of points takes \(O(|S_i|)\) work and \(O(1)\) depth, since each point checks at most \(3^k\) boxes bordering on its own. Therefore, except for the cost of Line 8 and the recursive call on Line 9, each call to BUILD takes \(O(|S_i|)\) expected work and \(O(\log^* |S_i|)\) depth whp. Line 8 creates a parallel heap of \(O(|S'_i|)\) entries, we cite the bound here and provide more details about it’s design in Section 5. The construction of the heap takes \(O(|S'_i|)\) work and \(O(\log |S'_i|)\) depth. Since \(\sum |S_i| = O(n)\) [30], the total work across all calls to BUILD is hence \(O(n)\) in expectation. Since our heap is of linear size, the total space usage of our data structure is also \(O(n)\) in expectation.

We now prove the algorithm has polylogarithmic depth, by proving the lemma below.

Lemma 4.1. Algorithm 1 makes \(O(\log n)\) calls to BUILD whp, and the sparse partition has \(O(\log n)\) levels whp.

Proof. We show that with at least half of the probability, \(|S_{i+1}| < |S_i|/2\). Consider to relabel the points in \(S_i = \{r_1, r_2, \ldots, r_{|S_i|}\}\) such that \(d(r_1, S_i) \leq d(r_2, S_i) \leq \ldots \leq d(r_{|S_i|}, S_i)\). If we pick the pivot \(p_i = r_j\) then for every \(r_k\) with \(k > j\), we have \(d(r_j, S_i) \leq d(r_k, S_i)\) so \(r_k\) is in \(S_{i+1}\). The pivot is chosen randomly from \(S_i\) and independently across the levels,
Figure 2: This is an illustration of the interaction between our parallel batch-dynamic insertion (left) and deletion (right) algorithms with the data structure. For ease of illustration, we do not show all of the points in the data set that leads to this grid structure. We show our data structure with two levels, and explicitly show $S_i$ and $H_i$ for each level. The grid structure in the upper half of the figures determines the sparsity of points. We represent different types of points as illustrated in the legend in the middle. In the lower half of the figures, we show the heaps with the restricted distances that they store. We show the restricted distance of a point $x$ on level $i$ as $(x, y)$ if another point $y$ is the closest neighbor to $x$ on level $i$. For both insertion and deletion, we annotate the direction of the update between grids using bold arrows (i.e., insertion starts with $S_1$ and deletion starts with $S_2$). We indicate the movement of points and heap entries using dotted arrows.

Algorithm 1: Construction

Input : Point set $S$.
Output : A sparse partition and its associated heaps.

1. **Algorithm MAIN()**
2. \[ \text{BUILD}(S, 1); /* Initially, } S_1 := S. */
3. **Procedure BUILD($S_i$, $i$)**
4. Choose a random point $p_i \in S_i$. Calculate $d_i := d(p_i, S_i)$, set the grid side length to $d_i/6k$, and store $p_i$'s nearest neighbor as $q_i$.
5. Create a parallel dictionary to store $S_i$ from now on. In parallel, compute the box ID of each point in set $S_i$ based on the grid size, and store the point to the box keyed by the box ID in the dictionary.
6. Create a parallel dictionary to represent $S'$. In parallel, determine if each point $x$ in $S_i$ is sparse by checking $N(q, S_i)$.
7. Store the sparse points in set $S'_i$, and the remaining points in a new point set $S_{i+1}$.
8. For each point $x \in S'_i$, compute $d'_i(x)$ by checking it's neighborhoods $N_i(x, S'_j)$ where $i - k \leq j \leq i$.
9. Create a heap for \{$d'_i(x) : x \in S'_i$\}.
10. BUILD($S_{i+1}, i + 1$) if $S_{i+1}$ is not empty.

so we can use a Chernoff bound to upper bound the number of recursion levels to be $O(\log n)$ whp.

Let random variables $X_i$ be 1 if in level $i$ the set size decreases by more than a half, and 0 otherwise. Let $X = \sum X_i$, we use the form of Chernoff bound that:

$$\Pr [X \geq (1 - \delta) \cdot E[X]] \leq e^{-\delta^2 E[X]/2}.$$

We know $\Pr[X_i = 1] \geq 1/2$, and the recursion must have stopped no later than when we have $X_i$ as 1 for $\log_2 n$ times.

We now analyze the probability that the recursion has more than $8c \log_2 n$ levels for $c \geq 1$ but the algorithm does not finish. In this case, $E[X] = 4c \log_2 n$, and for $\delta = 1 - 1/4c$,
we have:

\[ \Pr[X \leq (1 - \delta)\mathbb{E}[X]] = \Pr[X \leq \log_2 n] \leq \exp\left(-\left(1 - \frac{1}{4c}\right)^2 \cdot 4c \log_2 n\right)/2 \leq \exp(-c \log_2 n) < n^{-c}. \]

This means that sparse partition has no more than \(8c \log_2 n = O(\log(n))\) levels whp, so as the number of recursive calls in Algorithm 1 makes to BUILD.

As mentioned earlier, the depth of each call to BUILD is \(O(\log^* n)\) whp, excluding the cost for heap construction and the recursive call. Since the heap insertions are asynchronous, they take a total of \(O(\log n)\) depth. Therefore, the total depth of Algorithm 1 is \(O(\log n \log^* n)\) whp. This gives the following theorem.

**Theorem 4.2.** We can construct a data structure that maintains the closest pair containing \(n\) points in \(O(n)\) expected work, \(O(\log n \log^* n)\) depth whp, and \(O(n)\) expected space.

4.2 **Maintenance with Batch Updates**

In this section, we describe our parallel batch insertion and deletion algorithms. When inserting or deleting a set \(Q\) of \(m\) points to or from the set \(S\) of \(n\) points, the algorithms update \((S_i, S'_i, p_i, q_i, d_i)\) and the parallel heaps \(H_i\) for \(1 \leq i \leq L\) to maintain its properties (possibly changing the value of \(L\)).

**Insertion Algorithm.** We first ensure that the pivot \(p_i\) in each level is randomly chosen after the batch insertion, so we may need to re-select the pivot at each level. We first describe what happens on level \(i = 1\), and then describe subsequent levels. Recall that on level \(i = 1\), the pivot \(p_1\) is a randomly chosen point from \(S_1 = S\). With the insertion of \(Q\), With probability \(|Q|/(|Q| + |S_1|)\), a new pivot \(p'_1\) from \(Q\) replaces the existing pivot \(p_1\). When the pivot changes, we update the grid size and rebuild the data structure by calling BUILD\((S_1, Q, 1)\) in Algorithm 1. Otherwise, the original \(p_1\) remains the pivot, and we update the new pivot distance \(d(p'_1, S_1 \cup Q)\) if there exists \(q'_i \in Q\) such that \(d(p'_1, q'_i) < d(p_1, S_1)\), the original pivot distance of \(S_1\).

We next discuss parallel batch insertion to the sparse partitions if \(p_1, q_1, d_1\) all remain unchanged. We denote the subset of points in \(Q\) that are not sparse in \(S_1 \cup Q\) as \(Q_1\), and they will be passed on to level 2. The subset of points in \(Q\) that are sparse will be inserted into \(S'_1\). In addition, a subset of sparse points in \(S'_1\) can become no longer sparse due to the insertion of \(Q\), and be moved down to level \(i = 2\). We denote these points as the set \(\text{down}_{i-1}\), borrowing notation from Golin et al. [30]. In the case that \(Q_1\) and \(\text{down}_{i-1}\) are empty, no points need to be inserted to \(S_2\).

Maintaining subsequent levels is similar. In Algorithm 2, we present the algorithm of the batch insertion for all of the levels. We let \(Q_i\) be the subset of points in \(Q\) that are inserted at level \(i\), and \(\text{down}_i\) be the set of points that move from level \(i - 1\) to level \(i\) due to the insertion of \(Q_i\). We compute \(\text{down}_i\) by \(\text{down}_i = \{x \mid x \in S_{i-1} \cup (q_i, S'_{i-1} \cup \text{down}_{i-1})\text{ for some }q_i \in (Q_i)_1\}\). Each call to the procedure INSERT\((Q_i, \text{down}_i)\) on Line 3 updates \((S_i, S'_i, p_i, q_i, d_i)\) and \(H_i\). Initially, \(Q_1 = Q\) and \(\text{down}_1\) is empty, as shown on Line 2. We first focus on the GRIDINSERT procedure defined on Line 7 and called on Line 4, which updates the grid structure of the sparse partition \((S_i, S'_i, p_1, q_1, d_1)\). On Line 8, we re-select \(p_i\) and re-compute \(q_i, d_i\) with probability \((|Q_1| + |\text{down}_i|)/(|Q_1| + |\text{down}_i| + |S_i|))\) to ensure a randomly selected pivot. If \(p_i\) is not re-selected, we check if any point in \(Q\) is closer to \(p_i\) than \(q_i\), or if no, \(q_i, d_i\) need to be updated. If the pivot is changed, we call BUILD on Line 9 to rebuild from level \(i\) since the grid size will change.

In addition, we call BUILD if the level \(i\) is greater than \(L\). In the case that BUILD is called, we terminate the insertion algorithm.

Otherwise, on Lines 10–12, we insert the points in both \(\text{down}_i\) and \(Q_i\) into the dictionary representing \(S_i\). Then we check if the points that we inserted are sparse, and insert sparse ones into the dictionary representing \(S'_i\). The points that are not sparse will be added to sets \(\text{down}_{i+1}\) and \(Q_{i+1}\) and passed on to the next level. On Line 13, we determine additional elements of \(\text{down}_{i+1}\) by including the neighbors of \(\text{down}_i\) in \(S'_i\). If \(Q_{i+1}\) and \(\text{down}_{i+1}\) are empty, nothing further needs to be done for subsequent levels, and the tuples \((S_i, S'_i, p_i, q_i, d_i)\) for \(i < L \leq L\) remain unchanged. We delay the description of updating the heap in parallel (Line 5) to Section 4.3.

**Correctness.** Consider a round \(i\) that inserts a non-empty \(Q_i\) \(\cup\) \(\text{down}_i\). After the insertion, the pivot is still chosen uniformly at random, since on Line 8, we choose \(p_i\) such that each point in \(S_i \cup Q_i \cup \text{down}_i\) has the same probability of being chosen.

Lines 10–12 ensure that \(S'_i\) contains exactly all of the sparse points of \(S_i\). Lines 11–12 ensure that all sparse points in \(Q_i\) and \(\text{down}_i\) inserted into \(S_i\) are included in \(S'_i\). Line 13 additionally ensures that all points that were originally sparse in \(S'_i\) but are no longer sparse after the insertion are removed from \(S'_i\). Given that the non-sparse points in the original \(S_i\) will not become sparse due to the batch insertion, \(S'_i\) must contain exactly all of the sparse points of the updated \(S_i\).

**Analysis.** We first show two key lemmas that bounds the total size of \(\text{down}_i\) and \(Q_i\) to be proportional to the batch size \(m\) across all the levels.

**Lemma 4.3.** \(|\bigcup_{1 \leq i \leq L} \text{down}_i| \leq m \cdot 3^k = O(m)\)

**Proof.** We want to prove that the number of points moved across sparse partitions for the insertion of \(Q\) of size \(m\) points is \(O(m)\). We borrow the notation from Golin et al. [30]. For a level \(i\) and a point \(q \in Q\), we let \(\text{down}_i(q)\) denote the subset of points \(\text{down}_i\) that are also in \(N_i(q, S'_i)\). Let \(b_i(q)\) denote the box that contains point \(q\). Let the *box neighborhood* of \(q\) in \(G_i\) denoted by \(N_i(q)\), be the neighborhood of \(b_i(q)\), consisting
Algorithm 2: Batch Insert

\textbf{Input}: \((S_i, S'_i, p_i, q_i, d_i)\) and \(H_i\) for \(1 \leq i \leq L\); a batch \(Q_i\) to be inserted.

1 \hspace{1em} \textbf{Algorithm MAIN(} \)
2 \hspace{2em} \textbf{INSERT}(Q, \emptyset, 1);
3 \hspace{2em} \textbf{Procedure INSERT}(Q_i, \text{down}_i, i)
4 \hspace{3em} (Q_{i+1}, \text{down}_{i+1}) := \text{GRIDINSERT}(Q_i, \text{down}_i, i);
5 \hspace{3em} \text{HEAPUPDATE}(i);
6 \hspace{3em} \textbf{if} \ (Q_{i+1} \cup \text{down}_{i+1}) \neq \emptyset \textbf{ then INSERT}(Q_{i+1}, \text{down}_{i+1}, i + 1)
7 \hspace{2em} \textbf{Procedure GRIDINSERT}(Q_i, \text{down}_i, i)
8 \hspace{3em} \textbf{Determine} if \(p_i, q_i\), and \(d_i\) should change when inserting \(Q_i\) and \(\text{down}_i\), which happens with probability
9 \hspace{4em} (|Q_i| + |\text{down}_i|)/(|Q_i| + |\text{down}_i| + |S_i|), or if a new point is closer to \(p_i\) than the previously closest point \(q_i\).
10 \hspace{3em} If \(p_i, q_i\), or \(d_i\) change on Line 8, or if \(i > L\), call \text{BUILD}(Q_i \cup \text{down}_i \cup S_i, i)\) to build subsequent levels, and terminate
11 \hspace{3em} the batch insertion.
12 \hspace{3em} \textbf{Insert} each point in \(\text{down}_i\) and \(Q_i\) into the dictionary of \(S_i\) in parallel.
13 \hspace{3em} For each point \(x\) in \(Q_i\) in parallel, check if it is sparse in \(S_i\). If so, insert \(x\) into the dictionary of \(S'_i\), and otherwise, insert \(x\) into \(Q_{i+1}\).
14 \hspace{3em} For each point \(x\) in \(\text{down}_i\) in parallel, check if it is sparse in \(S_i\). If so, insert \(x\) into the dictionary of \(S'_i\), and otherwise, insert \(x\) into \(\text{down}_{i+1}\).
15 \hspace{3em} In parallel, for each point \(x\) in \(Q_i\), and for each point \(r\) in the neighborhood \(N(x, S'_i)\), delete \(r\) from \(S'_i\), and insert \(r\) into \(\text{down}_{i+1}\).
16 \hspace{2em} \textbf{return} \((Q_{i+1}, \text{down}_{i+1})\);

of \(b_1(q)\) itself and the collection of \(3^k - 1\) boxes bordering on \(b_1(q)\). We number the \(3^k\) boxes in \(N_i(q)\) as a \(k\)-tuple over values \(\{-1, 0, 1\}\), where the \(j\)th component indicates the relative offset of the box with respect to \(b_1(q)\) in the \(j\)th dimension. We denote the box with a relative offset of \(\sigma\) with respect to \(b_1(q)\) as \(b_1^\sigma(q)\), where \(\sigma\) is the \(k\)-tuple. We further define the \textit{partial box neighborhood} of a point \(q\), denoted by \(N^\sigma(q)\), as the set of boxes in \(N_i(q)\) that intersect with the boxes bordering on and including \(b_1^\sigma(q)\).

Let \(x \in \text{down}_{i+1}(q)\) for some level \(j\). By definition, \(x\) is in a box of \(N_j(q)\) for some \(q \in Q\) and \(N_j(x, S) = \emptyset\). Therefore, the partial neighborhood \(N^\sigma_j(q)\) contains no points other than \(x\). Consider some other point \(y \in b_1^\sigma(q)\) for any \(l > j\). Since \(d_i \leq d_{j+1} \leq d_j/3\) by properties of the sparse partition, \(N^\sigma_j(y)\) is spatially contained in \(N^\sigma_j(y)\), and hence we have \(y \in N^\sigma_j(q)\).

Therefore, for any level \(l > j\), there cannot be any point in \(\text{down}_{i+1}(q)\) with signature \(\sigma\) except for \(x\) (similar to the argument for \(q\)). Therefore, for any \(q\) and \(p\), the intersection \(b_1^\sigma(p) \cap b_1^\sigma(q)\) contains at most one element. This leads to the result, given two points \(q, p \in Q\), \(|\text{down}_i(q)| + |\text{down}_i(p)| \leq 2 \cdot 3^k\). The argument can be directly extended to an arbitrary subset of \(Q\). Let \(j = 0\), and given that \(l \leq L\), the total size of \(\text{down}_i(q)\) for \(q \in Q\) is upper bounded by \(m \cdot 3^k = O(m)\) across all the levels. \(\square\)

\textbf{Lemma 4.4.} \(\sum_{1 \leq i \leq L} E[|Q_i|] = O(m)\)

\textbf{Proof.} The key argument is to show \(|Q_{i+1}|\), the number of points that are not sparse in \(S_i\) decreases by at least a factor of two in expectation compared to \(|Q_i|\). During the batch insertion, we ensure that \(p_i\) is randomly chosen from \(Q_i \cup S_i\), which is done by Line 8 of Algorithm 2. We assume that the user does not know about the random choices made inside the sparse partition data structure. Consider points \(r\) in \(Q_i\) in an increasing order of \(d(r, S_i \cup Q_i)\). There is a 1/2 chance that the pivot \(p_i\) is chosen to be with \(d(p_i, S_i \cup Q_i)\) not larger than that of at least half of the points in \(Q_i\), making them sparse and not in \(Q_{i+1}\). Therefore \(|Q_{i+1}| \leq |Q_i|/2\) in expectation. Given that \(|Q_i| = O(m)\), we have that \(\sum_{1 \leq i \leq L} E[|Q_i|] = O(m)\). \(\square\)

We Lemma 4.3 and 4.4, we now show the work and depth bound of Algorithm 2, summarized in the theorem below.

\textbf{Theorem 4.5.} We can maintain a sparse partition for a batch of \(m\) insertions in \(O(m)\) amortized work in expectation and \(O(\log(n + m) \log^2(n + m))\) depth whp.
PROOF. The expected cost of rebuilding on Line 9 summed across all rounds is proportional to the batch size. First, we re-select the pivot and rebuild with probability $(|Q_i| + \text{down}_i)/(|Q_i| + |\text{down}_i| + |S_i|)$. When the pivot $p_i$ is unchanged, it may update its closest point to $q'_{i}$ from $Q_i \cup \text{down}_i$. It is easy to show that $q'_{i}$ can be the nearest neighbor of at most $3^k - 1$ points in $S_i$. Hence considering all candidates $Q_i \cup \text{down}_i$, it follows that they can be the nearest neighbors to $O(3^k \cdot (|Q_i| + |\text{down}_i|))/|S_i|$, in which case we rebuild the sparse partition. The expected work of rebuilding at level $i$ is $O(|S_i| \cdot (|Q_i| + |\text{down}_i|))/|Q_i| + |\text{down}_i| + |S_i|) + 3^k \cdot (|Q_i| + |\text{down}_i|)/|S_i|) = O(m)$. As we terminate the insertion algorithm when a rebuild occurs, the rebuild can occur at most once for each batch, which contributes $O(m)$ in expectation to the work and $O(\log(n + m) \log^2(n + m))$ whp by the depth to Theorem 4.2.

For the rest of the algorithm, in terms of work, Line 10–12 does work proportional to $O(\sum_i |Q_i| + |\text{down}_i|) = O(m)$ across all the levels as a corollary of Lemma 4.3 and Lemma 4.4. On Line 13, the number of points in the neighborhood $N_i(x, S'_i)$ of each $x$ is upper bounded by $3^k$ since the points in $S'_i$ are sparse, therefore it takes $O(3^k \cdot m) = O(m)$ expected work. Note that the work is amortized due to resizing the parallel dictionary when necessary. In terms of depth, looking up and inserting points takes $O(\log^*(m + n))$ depth using the parallel dictionary. Therefore, all operations in Lines 10–13 takes $O(\log^*(m + n))$ depth, and across all $O(\log(n + m))$ whp rounds, the total depth is $O(\log(n + m) \log^*(n + m))$ whp. □

Deletion Algorithm. The pseudocode for our batch deletion algorithm is shown in Algorithm 3. It takes as input $(S_i, S'_i, q_i, d_i)$ and $H_i$ for $1 \leq i \leq L$, and a batch of points $Q$ to be deleted. We update the data structure level-by-level similar to the insertion algorithm, but in the opposite direction, starting at the last level $L$. We define $Q_i$ for level $i$ as $Q \cap S_i$. From the property of the sparse partition, $Q_j \subseteq Q_i$ for all $i < j \leq L$. At each level, we delete each point in $Q_i$ from $S_i$, and also from $S'_i$ if it exists.

While the insertion algorithm moves sets of points down to level $i + 1$ to level $i$, the deletion algorithm moves points in the opposite direction, from level $i + 1$ to $i$. We define $u_{p_i}$ to be the set of points that move from level $i + 1$ to level $i$, i.e., $u_{p_i} = \{ x \in S_{i+1} : N_i(x, S'_i) \subseteq Q_i \}$. They are the points $x$ in $S_{i+1}$ that only contain points from $Q_i$ in their neighborhoods $N_i(x, S'_i)$ in level $i$; and when $Q_i$ is deleted, they will become sparse in $S_i$, and will no longer be in $S_{i+1}$ anymore. Eventually, the points in $u_{p_i} \setminus u_{p_{i-1}}$ are added to both $S_i$ and $S'_i$.

Initially, on Line 2, we determine $Q_i$ for all levels. Note that this could be done efficiently via a backward pass starting from level $L$. Given $Q_i \subseteq Q_j$ for $i > j$, when a point is added to $Q_i$, it will be added to all $Q_j$ where $j < i$. We pass an empty $u_{p_i}$ to procedure DELETE, as shown on Line 3 of Algorithm 3. In the procedure DELETE (Line 5), the algorithm performs the deletion from the grid at level $i$ on Line 6, updates the heap on Line 7, and then Line 8 recursively calls DELETE on level $i - 1$ until deletion is complete on level 1. Like in the insertion algorithm, we determine whether to rebuild at each level, but unlike insertion we delay the rebuild until the end of the algorithm (Line 4). We call rebuild just once, on the level with the smallest $i$ that needs a rebuild (as this will also rebuild all levels greater than $i$).

In the procedure call GRIDDELETE($u_{p_i}, i$), On Line 10, we determine if the pivot needs to change based on whether at least one of $p_i$ and $q_i$ are in $Q_i$. If so, we mark level $i$ for rebuilding. On Line 11, we insert $u_{p_i}$ into $S'_i$, and on Line 12, we delete the points in $Q_i$ from $S_i$ and $S'_i$ if they exist.

On Line 13, we determine $u_{p_{i-1}}$ by finding the points that will become sparse in level $i - 1$. Since the movement of $u_{p_{i-1}}$ from level $i$ to $i - 1$ is due to the deletion of $Q_i$, we could enumerate the candidates for $u_{p_{i-1}}$ from $N_i(x, S'_i)$ where $x \in Q_i$. Then, for each candidate $r$, we check if $N_{i-1}(r, S_{i-1})$ only consists of points in $Q_{i-1}$, which are to be deleted in $i - 1$. If so, $r$ will move up to a level less than or equal to $i - 1$, and so we add $r$ to $u_{p_{i-1}}$. A few details need to be noted to make the computation of $u_{p_{i-1}}$ work. First, when checking the neighborhood $N_i(x, S'_i)$ for the candidates $r$, we should only check a neighboring box if it contains at most one point, since otherwise the candidate would not be sparse in $S_{i-1}$. This bounds the work of enumerating candidates to $O(3^k \cdot m)$. Second, when checking a candidate $r$ of whether $N_{i-1}(r, S_{i-1})$ contains only points in $Q_{i-1}$, each check can potentially take $O(m)$ work since $|Q_{i-1}| = O(m)$. This can make the work for checking the neighborhood for all potential candidates be quadratic in $m$. As a remedy, we keep a counter initialized to 0 for each box that contains at least a point from $Q_{i-1}$. Then for each point $q$ in $Q_{i-1}$ in parallel, we use an atomic add to increment the counter of the box that contains $q$. At the end, we compare the counter of each box with its total number of points to determine if it only contains points in $Q_{i-1}$. This process takes $O(m)$ work and $O(1)$ depth. Then for each box in the neighborhood of a candidate, it takes constant work to check if it only contains points from $Q$.

Analysis. Since the probability of a rebuild at each level $i$ is $|Q_i \cap S_i|/|S_i|$ and the work for the rebuild is $O(|S_i|)$, the expected work of rebuilding at level $i$ is $O(|Q_i|) = O(m)$. Since we do at most one rebuild across all levels, it contributes $O(m)$ in expectation to the work and $O(\log(n + m) \log^*(n + m))$ whp to the depth.

The total size of $Q_i$ and $u_{p_i}$ across all the levels is proportional to the batch size. Since the point movement is the exact opposite of that of batch insertion, the proof is very similar. We omit the proof and just show the lemma below.
Algorithm 3: Batch Delete

\begin{algorithm}
\textbf{Input:} \((S_i, S'_i, p_i, q_i, d_i)\) and \(H_i\) for \(1 \leq i \leq L\); a batch \(Q\) to be deleted.

1. \textbf{Algorithm MAIN()}
   
2. \hspace{1em} Determine \(Q_i\) for all \(1 \leq i \leq L\). Specifically, for each level from \(L\) to 1, compute \(Q_i = Q \cap S_i\).
3. \hspace{1em} \textbf{DELETE}(0, L);
4. \hspace{1em} Rebuild from the level with the smallest \(i\) that needed a rebuild. Specifically, call \textbf{BUILD} \((S_i, i)\).

5. \hspace{1em} \textbf{Procedure} \textbf{DELETE}(up, i)
   
6. \hspace{2em} up\(_{i-1} := \text{GRIDDELETE}(up, i)\);
7. \hspace{2em} \textbf{HEAP_UPDATE}(i);
8. \hspace{2em} if \(i - 1 \geq 1\) then \textbf{DELETE}(up\(_{i-1}, i - 1\));

9. \hspace{1em} \textbf{Procedure} \text{GRIDDELETE}(up, i)
   
10. \hspace{2em} Determine if \(p_i, q_i\), or \(d_i\) should change after deleting \(Q_i\), which happens if at least one of \(p_i\) or \(q_i\) is in \(Q_i\). If so, mark level \(i\) for rebuild.
11. \hspace{2em} Insert each point in \(up_i\) into the dictionary of \(S'_i\) in parallel.
12. \hspace{2em} For each point \(x \in \mathcal{Q}_i\) in parallel, delete \(x\) from \(S_i\) and \(S'_i\).
13. \hspace{2em} For each point \(r\) in \(\mathcal{N}_i(x, S_i)\) where \(x \in \mathcal{Q}_i\), check \(\mathcal{N}_{i-1}(r, S_{i-1})\). If \(\mathcal{N}_{i-1}(r, S_{i-1}) \subseteq \mathcal{Q}_{i-1}\), then delete \(r\) from \(S_i\) and \(S'_i\), and insert \(r\) into the set \(up_{i-1}\).
14. \hspace{2em} return \(up_{i-1}\);
\end{algorithm}

Lemma 4.6. \(|\bigcup_{1 \leq i \leq L} up_i| \leq m \cdot 3^k = O(m)\)

Lemma 4.7. \(\sum_{1 \leq i \leq L} E[|Q_i|] = O(m)\)

For the rest of the algorithm not including the rebuild, it follows that Lines 11–13 take \(O(m)\) amortized work in expectation and \(O(\log(n + m) \log^*(n + m))\) depth across all the rounds, similar to the insertion algorithm, therefore we omit the detailed proof.

Theorem 4.8. We can maintain a sparse partition under a batch of \(m\) deletions in \(O(m)\) amortized work in expectation and \(O(\log(n + m) \log^*(n + m))\) depth whp.

4.3 Maintaining the Heaps \(H_i\)

On each level \(i\), we maintain a parallel min-heap \(H_i\) storing the restricted distances for each point in \(S'_i\). In this section, we elaborate on the \textbf{HEAP_UPDATE} procedure on Line 5 of Algorithm 2 and Line 6 of Algorithm 3. Each call to \textbf{HEAP_UPDATE}(i) updates \(H_{isl}\) for \(0 \leq l \leq k\) so that the they contain the updated distances in level \(l\).

Point Movements. We first analyze the point movements between the different levels during a batch insertion. During a batch insertion, we process each level \(i\) with inputs \(Q_i\) and \(down_i\) (Algorithm 2). By definition, \(down_i\) contains the points moved from level \(i - 1\) to levels \(i\) and greater. We say that point \(x\) \textit{starts moving} at level \(i\) if \(x \in down_{i+1}\) \(\backslash\) \(down_i\). We say that point \(x\) \textit{stops moving} at level \(i\) if \(x \in down_i\) \(\backslash\) \(down_{i+1}\), or if point \(x\) \(\notin Q_i\) \(\backslash\) \(Q_{i-1}\), i.e., \(x\) is sparse and stays in \(S'_i\). Finally, point \(x\) moves through level \(i\) if it is in \(down_i\) \(\backslash\) \(down_{i+1}\).

Updating the Heaps. The heap \(H_i\) contains the restricted distance \(d'_i(q)\) for \(q \in S'_i\). By definition, \(d'_i(q)\) is the closest distance of \(q\) to another point in \(S'_{i-1}\) where \(0 \leq l \leq k\) (\(k\) is the dimension of the data set). Therefore, following an update on \(S'_i\), we need to update the \(d'_i(q)\) in \(H_{isl}\) for \(0 \leq l \leq k\), and \(q \in S_{isl}\). Specifically, the update happens when \(d'_i(q) = d(q, p_{i})\), but \(p\) starts moving at level \(i\); or when \(p\) stops moving at level \(i\) and \(d(q, p) < d'_i(q)\). Since the update of \(S'_i\) initiates the update on some heap \(H_{isl}\) for \(0 \leq l \leq k\), we call level \(i\) the \textit{initiator} and each heap \(H_{isl}\) a \textit{receptor} of the initiator.

We first start with a more intuitive but less parallel algorithm in Algorithm 4. It takes in the updated sparse partition \((S_i, S'_i, p_i, q_i, d_i)\), set of points \(M_i\) that start moving at \(i\), and a set of points \(M_0\) that stop moving at \(i\). Lines 2–6 process the set of points \(M_i\). On Line 2, we first batch delete \(d'_i(q)\) from \(H_i\) for all \(q \in M_i\) since they start moving at level \(i\). On Lines 3–6, we update each receptor heap if it stores some \(d'_i(q)\) that is generated by a deleted point \(p \in M_i\). To know each potential point \(q\), we iterate over the neighborhood of each \(p\) in \(M_i\) and then check if \(d'_i(q)\) needs to be updated. Lines 7–10 process \(M_0\). On Line 9, we compute new restricted distances and batch insert the points in \(M_2\) into \(H_i\), since they stop moving at level \(i\). Then on Lines 8–10, we update the receptor heaps when a heap contains the restricted distance of point \(q\), but \(q\) has a smaller distance to a newly inserted \(p \in M_2\) than to its previous closest point.

Using our batch-parallel binary heap which we will describe in Section 5, each batch update of the heap takes \(O(\log(n + m))\) depth. The computation of the new restricted distances takes \(O(1)\) depth. Therefore, the naive heap update algorithm takes \(O(k \log(n + m))\) depth per call. For the batch insertion algorithm in Algorithm 2, \textbf{GRID_INSERT} on level \(i + 1\) is blocked by \textbf{HEAP_UPDATE-NAIVE} of level \(i\), to prevent
A similar argument applies for the batch deletion algorithm. This gives an overall depth bound of $H$ heaps, to make each receptor heap pull the update from the initiator level. We illustrate the details in the context of a batch insertion in Algorithm 5. During a batch insertion, we first perform GRIDINSERT on Line 4, and then on Lines 5–6, we fork off the HEAPUPDATE-PULL task, and start the insertion task for the next level in parallel with HEAPUPDATE-PULL.

In the HEAPUPDATE-PULL procedure, on Lines 9–10, we delete the restricted distances of points that start moving, and insert points that stop moving at level $i$. Then, we let $H_i$ pull additional updates from the initiator levels $i - l$ where $0 \leq l \leq k$. Specifically, we delete the restricted distances in $H_i$ that are affected by the initiator updates on Lines 11–12. At this point, the GRIDINSERT of the initiator levels will have completed. Note that in order to compute the restricted distances, the algorithm needs to access the moved points of the initiator levels. We can use parallel dictionaries to store the points that started and stopped moving during the grid updates of each initiator level. On Line 13, we re-compute the new restricted distances, and batch insert them into $H_i$. Figure 3b illustrates this algorithm. While the order of heap operations in Algorithm 5 may differ from that of Algorithm 2, it is still
Algorithm 5: Batch Insert with Parallel Heap Updates

Input : \((S_i, S_i', p_i, q_i, d_i)\) and \(H_i\) for \(1 \leq i \leq L\); a batch \(Q\) to be inserted; point set \(M_1\) that start moving at level \(i\) and point set \(M_2\) that stop moving at level \(i\).

1 Algorithm Main()
2 \hspace{1em} INSERT(Q, ∅, 1);
3 Procedure INSERT(Q_i, down_i, i)
4 \hspace{1em} \((Q_{i+1}, \text{down}_{i+1}) := \text{GRIDINSERT}(Q_i, \text{down}_i, i)\);
5 \hspace{1em} fork HEAPUPDATE-PULL(i);
6 \hspace{1em} INSERT(Q_{i+1}, \text{down}_{i+1}, i+1);
7 \hspace{1em} join

8 Procedure HEAPUPDATE-PULL(i)
9 \hspace{1em} Batch delete \(d'_i(p)\) ∀\(p \in M_1\) from \(H_i\).
10 \hspace{1em} Compute and batch insert \(d'_i(q)\) ∀\(q \in M_2\) into \(H_i\).
11 \hspace{1em} Batch delete \(d'_i(q)\) from \(H_i\) if \(d'_i(q) = d(q, p)\) and \(p\) started moving at initiator level \(i - l\) where \(0 \leq l \leq k\).
12 \hspace{1em} Batch delete \(d'_i(q)\) from \(H_i\) if \(d(q, p) < d'(q)\) and \(p\) stopped moving at initiator level \(i - l\) where \(0 \leq l \leq k\).
13 \hspace{1em} Re-compute \(d'(q)\) for each \(q\) deleted on Lines 11–12, and batch insert then into \(H_i\).

Work Analysis. We have shown the depth including the heap updates across all levels is \(O((n + m) \log^2 (n + m)) \text{whp}\). Here we show the work performed by the heap updates.

Theorem 4.9. In addition to maintaining a sparse partition, we can update \(H_i\) for \(1 \leq i \leq L\) under a batch insertion/deletion of size \(m\) in amortized \(O(m \log(1 + (n + m)/m))\) work and \(O((n + m) \log^2 (n + m)) \text{depth whp}\).

Proof. All updates on the heaps in our data structure are a result of points that start or stop moving at some level. First, we are concerned with those added to or deleted from \(S_i'\) and hence \(H_i\). Since \(S_i'\) for \(1 \leq i \leq L\) are disjoint sets, \(O(m)\) points from \(Q\) are inserted or deleted from \(H_i\) across all \(1 \leq i \leq L\).

In Section 5, we show that the total work for a batch of \(r\) updates to our parallel heap is \(O(r \log(1 + (n + m)/m))\). Therefore, the total work for heap operations is hence \(O(m \log(1 + (n + m)/m))\). \(\square\)

5 Parallel Batch-Dynamic Binary Heap

One of the key components in parallelizing our closest pair algorithm is a parallel heap that supports batch updates (inserts and deletes) and finding the minimum element (find-min) efficiently. We could implement a parallel heap using a parallel binary search tree, which supports a batch of \(m\) updates to a
Algorithm 6: Batch Delete with Parallel Heap Update

Input : \((S_i, S_i', p_i, q_i, d_i)\) and \(H_i\) for \(1 \leq i \leq L\); a batch \(Q\) to be deleted.

1. Algorithm \text{MAIN}()
2. \quad \text{DELETE}(\emptyset, 1);
3. \quad Perform \text{HEAPUPDATE-PULL}(i) for \(1 \leq i < k\) in parallel.
4. Procedure \text{DELETE}(up_i, i)
5. \quad up_{i-1} := \text{GRIDDELETE}(up_i, i);
6. \quad fork \text{HEAPUPDATE-PULL}(i+k);
7. \quad \text{DELETE}(up_{i-1}, i-1);
8. join

A simple \text{HEAPIFY} algorithm. We start with a simple version of the \text{HEAPIFY} algorithm (Algorithm 7) that achieves the work bound in Theorem 5.1. The \text{HEAPIFY} algorithm takes \(m\) updates from a valid heap and returns another valid heap, which can be used to implement batch insertions and deletions. The algorithm runs in two phases. The first phase works on increase-key updates (Line 1–5), and the second phase on decrease-key updates (Line 6–10). In both phases, we first use integer sort to categorize all updates based on the level of the nodes in the heap. Then for the first phase, we work bottom-up on the heap level-by-level. On each level, we run in parallel the sequential \text{DOWN-HEAP} procedure for all nodes that have their keys increased. The second phase is slightly more sophisticated since in the \text{UP-HEAP} procedure, it is possible that both subtrees of an interior node have nodes updated. Hence, the \text{UP-HEAP} is run in a synchronous manner—for all updates in level \(l\), we synchronously run \text{UP-HEAP} for one level, then for another level, until the root. We do so for work-efficiency, and once two sibling nodes finish the \text{UP-HEAP}, only one of them can swap to the parent, and the other \text{UP-HEAP} just quits.

Correctness and Work Bound. The correctness can be shown inductively on subtrees of increasing height. For the base case, all leaf nodes are valid binary heap subtrees, each containing one node. Then on the first iteration, we run \text{DOWN-HEAP} for updated keys on the second to last level. If the increased keys violate the heap property, then \text{DOWN-HEAP} will heapify this subtree, which has two levels. Similarly, for each node \(\nu\) with increased keys on level \(l\), \(\nu\)’s both children’s subtrees are valid binary heap subtrees, so after \text{DOWN-HEAP}, the
subtree rooted at $v$ is a valid binary heap subtree. The correctness for Up-Heap can be shown symmetrically. The only difference is that in Up-Heap, the update paths can overlap, and the correctness is guaranteed since it is implemented in a round-synchronous manner.

We now consider the work of this algorithm. Let $h$ be the height of the binary heap. For the worst case analysis, we always assume that Down-Heap pushes a node to the leaf and that Up-Heap pushes a node to the root. The case for Down-Heap is simple—for $m = 2^r - 1$ increase-keys, the worst case is when they are in the top $r$ levels. Each Down-Heap is independent and the total work is

$$\sum_{i=0}^{r} 2^i(h - i) = m(h - r) + O(m) = O\left(m \log \left(\frac{n}{m} + 1\right)\right).$$

The work for Up-Heap is more involved. Let $m_i$ be the number of increase-keys on level $i$. We know that $m_i \leq 2^i$ and $\sum m_i \leq m$. For level $i$, the work for all calls to Up-Heap is upper bounded by the number of nodes on the path from the root to all updated nodes in level $i$. It can be shown that the number of such nodes is $O\left(m_i \log \frac{n}{2^h m_i}\right)$ (Theorem 6 in [10]). Hence, the overall work for all levels is $W = O\left(\sum_{i=0}^{\log_2 n} m_i \log \frac{n}{2^h m_i}\right)$. Let $m' = \sum_i m_i$, and we know $m' \leq m$. To bound the work, we can use the method of Lagrange multipliers, and compute the partial derivative of $m_i$ (without the big-$O$), which solves to

$$\frac{\partial}{\partial m_i} W = \frac{\partial}{\partial m_i} \left(-m_i \log \frac{m_i}{n/2^h} \right) = \log \frac{n/2^h}{m_i} - 1.$$

Since the constraint for $\sum m_i$ is linear, $W$ is maximized when $\frac{\partial}{\partial m_i} W = \frac{\partial}{\partial m_j} W$ for all levels $0 \leq i, j \leq h$, which solves to $m_i = m'/2^{h-i+1}$. Plugging in it gives

$$W = O\left(\sum_{i=0}^{\log_2 n} \frac{m'}{2^{h-i+1}} \log \frac{n}{2^h (m'/2^{h-i+1})}\right)$$

$$= O\left(\frac{\log_2 n}{\log 2} \log \left(\frac{n}{m} + 1\right)\right) = O\left(m \log \left(\frac{n}{m} + 1\right)\right).$$

In addition to Down-Heap and Up-Heap, we also need to integer sort the updates in Line 2 and 7, which takes $O(m)$ work and $O(\log n)$ span. Hence, the total work for Algorithm 7 is $O\left(m \log \left(\frac{n}{m} + 1\right)\right)$, as stated in Theorem 5.1.

**Parallelism.** Directly running Algorithm 7 gives $O(\log^2 n)$ depth—there are $O(\log n)$ tree levels, and on each level, Up-Heap or Down-Heap requires $O(\log n)$ depth. We can improve the depth bound to $O(\log n)$ using the Async-Heapify algorithm.

If we assume free global synchronization after each instruction (like on a PRAM), the Up-Heap or Down-Heap in different levels can be pipelined. More specifically, in the first phase for Down-Heap, once the first swap in level $i$ is finished, we can immediately start the Down-Heap on level $i - 1$, instead of waiting the Down-Heap in level $i$ to finish first. It is easy to check that the swaps in the Down-Heap from level $i - 1$ will never catch the swaps from level $i$. Therefore, the span of this algorithm can be improved to $O(\log n)$.

Unfortunately, it is unrealistic to map this algorithm on real machines using any tools (programming language or libraries) with the same work and span bounds since none of them support such global synchronization in practice. We can manually synchronize, but then that will either be not work-efficient, or we need to add a packing phase after each synchronization, which will increase the span.
Algorithm 8: The Async-Heapify algorithm

Input: A binary min-heap of size \(n\) with \(m\) updates to change key \(k_i\) to \(k'_i\) on node \(n_i\).
Output: An updated binary heap.

1. \(\text{foreach } v \text{ in the heap do}\)
2. \(\quad v\)'s flag is 1 if \(v\)'s key is increased, 0 otherwise
3. \(\text{foreach } v \text{ with flag } 1 \text{ do}\)
4. \(\quad \text{Down-Heap}(v)\)
5. \(\text{foreach } v \text{ in the heap do}\)
6. \(\quad v\)'s flag is 1 if \(v\)'s key is decreased, 0 otherwise
7. Mark the wait-flag to 1 for nodes with both subtrees having increased keys
8. \(\text{foreach } v \text{ with flag } 1 \text{ do}\)
9. \(\quad \text{Up-Heap}(v)\)

Procedure Down-Heap\((v)\)

10. if \(v\) is leaf then
11. if CAS\(\left(v.\text{flag}, 1, 0\right)\) is failed then
12. \(v.\text{flag} \leftarrow 0\)
13. \(\text{Down-Heap}(v\text{'s parent})\)
14. Let \(lC\) be \(v\)'s left child and \(rC\) be right child
15. if CAS\(\left(lC.\text{flag}, 1, 2\right)\) then Quit;
16. if CAS\(\left(rC.\text{flag}, 1, 2\right)\) then Quit;
17. Let \(c\) be \(lC\) or \(rC\) with smaller key
18. if \(c.\text{key} < v.\text{key}\) then
19. Swap \(v\)'s and \(c\)'s keys
20. \(c.\text{flag} \leftarrow 1\)
21. if CAS\(\left(v.\text{flag}, 1, 0\right)\) then \(\text{Down-Heap}(c)\) ;
22. else
23. \(v.\text{flag} \leftarrow 0\)
24. Run \(\text{Down-Heap}(v\text{'s parent})\) and \(\text{Down-Heap}(c)\) in parallel
25. else
26. if CAS\(\left(v.\text{flag}, 1, 0\right)\) is failed then
27. \(v.\text{flag} \leftarrow 0\)
28. \(\text{Down-Heap}(v\text{'s parent})\)

Procedure Up-Heap\((v)\)

29. if \(v\) is root then
30. if CAS\(\left(v.\text{flag}, 1, 0\right)\) failed then
31. \(v.\text{flag} \leftarrow 0\)
32. \(\text{Up-Heap}(v.\text{leftchild})\)
33. Quit
34. if CAS\(\left(v.\text{parent.wait-flag}, 1, 0\right)\) then Quit;
35. if CAS\(\left(v.\text{parent.flag}, 1, 2\right)\) then Quit;
36. if \(v\text{'s sibling has smaller key} \text{ then } v \leftarrow v\text{'s sibling};
37. if \(v.\text{key} < v\text{'s parent.key} \text{ then } \text{Swap } v\text{'s and } v\text{'s parent's keys ;}
38. if CAS\(\left(v.\text{flag}, 1, 0\right)\) then \(\text{Up-Heap}(v\text{'s parent})\) ;
39. else
40. \(v.\text{flag} \leftarrow 0\)
41. Run \(\text{Up-Heap}(v\text{'s parent})\) and \(\text{Up-Heap}(v\text{'s leftchild})\) in parallel

15
We now discuss how the Async-Heapify algorithm without synchronizing the operations but only using compare-and-swap. The Async-Heapify algorithm also has two phases for increase-keys and decrease-keys. Within each phase, however, we apply all Up-Heap or Down-Heap simultaneously. This algorithm is discussed in Algorithm 8.

We first discuss the first phase for Down-Heap. As mentioned above, we cannot process two nodes that one is the other’s parent at the same time due to data race. Hence, for each node, we give a flag to detect such conflicts, with three possible states: 0 means no thread is working on this node, 1 means a thread is working on this thread, and 2 means two threads are working on this node and the parent node. The transition of the states is shown in Algorithm 8. Then we start the parallel-for-loop for all nodes with increased keys, and we can guarantee that all swaps in Down-Heap behave the same as in Algorithm 7. The second phase for Up-Heap can be addressed similarly, but in addition, since not only a thread can chase up another Up-Heap on the ancestor node, it also needs to wait the sibling node to be finalized, before the Up-Heap can be applied. Hence, in the Async-Heapify algorithm, each node has an additional flag (wait-flag) that is initially 0, and set to be 1 if this node has both subtrees with increase-key updates. This step can be computed by marking all ancestor nodes for all nodes with increased keys, from each node to the root. Once such traversing reach a node that is already marked, it changes the wait-flag of this node to 1 and quits. The total number of traversed nodes is \( O(m \log(n/m + 1)) \) (cite BFS), and the span is \( O(\log n) \). Then in Up-Heap, we first check the wait-flag and CAS it to 0, and quit immediately if succeeded. Otherwise, we apply the update similarly to Down-Heap that is discussed above, with the difference that Up-Heap cannot quit and has to process all the way to the root or deactivated by the wait-flag, because it needs to trigger the operation for the join point corresponding to this Up-Heap that is initially deactivated by the wait-flag.

Although the Async-Heapify algorithm is more complicated, the analysis is straightforward. The Async-Heapify algorithm makes the same number of Up-Heap and Down-Heap calls as compared to the simple Heapify algorithm, and each Up-Heap or Down-Heap in the Async-Heapify algorithm has constant work. The depth bound seems to be more complicated, but it can also analyzed easily. For a specific Up-Heap or Down-Heap, it can be hanged if it catches up another operation, or revoked once the conflict is resolved. However, once such a hang-and-revoke is incur, it can advance for at least one level. Now consider the worst case that we have \( O(\log n) \) updates all on a root-to-leaf chain and they block each other, and WLOG let’s assume Up-Heap. For the root node, it will finish using constant work and call its child if it blocks it. Then after a constant number of operations, the next Up-Heap will finish, and the process goes on until all \( O(\log n) \) Up-Heap finish. The overall depth is therefore \( O(\log n) \) for each Up-Heap, and \( O(\log n) \) for the maximum number of additional cost due to the hang-and-revoke overhead. Here we assume the entire tree path is full of Up-Heap operations, and in practice there can be fewer, but that can only help because that reduce the total number of hang-and-revoke to the number of Up-Heap on this path, which can only be smaller. Putting all pieces together, we prove Theorem 5.1.

### 5.2 Using Heapify for Batch Update

We have described how to perform batches of increase-keys and decrease-keys, and now we explain how to perform batch insertions and deletions. A batch of \( m \) insertions to a binary heap of size \( n \) can be implemented using decrease-keys. We can first add the \( m \) elements to the \( n \) elements in the heap with keys of \( \infty \), and the new heap with \( n + m \) elements is valid. Then we decrease the keys of these \( m \) elements to their true values and run the Heapify algorithm. It is easy to check that the final heap is correct after inserting these \( m \) elements, and in this case we only use the Up-Heap part.

A batch of \( m \) deletions can be processed similarly, but the deletions will generate “holes” in the tree structure, and so we need an additional step to fill these holes first. We can first pack the last \( m \) elements in the heap based on whether they are deleted. Then we use them to fill the rest of the empty slots by deletions, and run the Heapify algorithm. Namely, we modify the deleted keys to the filled keys, and the Heapify algorithm will return a heap with \( n - m \) elements. Hence, it takes \( O(m \log(1 + (n + m)/m)) \) work and \( O(\log(n + m)) \) depth for batch insertions with size \( m \), and \( O(m \log(1 + n/m)) \) work and \( O(\log n) \) depth for batch deletions with size \( m \).

### 6 Implementation

In this section, we describe techniques that make the implementation of our dynamic algorithms efficient in practice.

**Simplified Data Structure.** While the sparse partition maintains \( (S_i, S'_i, p_i, q_i, d_i) \) and \( H_j \) for \( 1 \leq i \leq L \), we found implementing \( S'_i \) and it’s associated heap \( H_j \) on every level not fast in practice. We found it is more efficient to compute \( (S_i, p_i, q_i, d_i) \) for \( 1 \leq i \leq L \), and just one heap \( H^* \) that stores the closest neighbor distance for all \( q \) in \( S_j \), where \( j \leq L - \lceil \log_3 2\sqrt{k} \rceil \). When \( L \) changes due to insertion or deletion, we recompute \( j \) and rebuild \( H^* \) if necessary.

For correctness, we prove that any point pair \( (a, b) \) where \( a, b \in S \setminus S_j \) cannot be the closest pair.

**Lemma 6.1.** \( \delta(S) < d(a, b) \) for \( a, b \in S \setminus S_j \).

**Proof.** We denote the size of the grid \( G_i \) at level \( i \) as \( g_i = d_i/6k \) as defined earlier. Without loss of generality, assume \( a, b \in S_{j-1} \setminus S_j \), and we have \( d(a, b) > g_{j-1} \geq 3 \cdot g_j \) by properties of the sparse partition. On the other hand, it is
obvious that $\delta(S) \leq 2\sqrt{k} \cdot g_{k-1}$. Given the property of the sparse partition $g_{i+1} \leq g_i / 3$, we have $3 \cdot g_j = 3 \cdot g_{L - \lfloor \log_2 2\sqrt{k} \rfloor} > 2\sqrt{k} \cdot g_{k-1}$ for all $k$. Therefore, $d(a, b) > \delta(S)$. 

Since $a, b$ are both sparse in some $S_h$ where $h < j$, it follows that $d(a, q) > d(q, S_j)$ and $d(b, q) > d(q, S_j)$ for some $q \in S_j$. Therefore, the closest pair distance $\delta(S) = d(p, q)$ for some $p, q \in S_j$.

The parallel heap uses the implementation from the PAM library [49, 50]. We do not explicitly store $S'_i$, and we obtain it by checking if each point in $S_i$ is sparse on-the-fly.

**Spatial Tree.** Our analysis earlier assumes a constant dimensionality $k$, and some of the work bounds are exponential in $k$, e.g., a grid’s box neighborhood is of size $3^k$. For $k \geq 5$, the straightforward implementation is inefficient due to large constant overhead in the work. Hence, we implement a parallel batch-dynamic $kd$-tree for $k \geq 5$. This is because performing a range query on the tree works better in practice compared to traversing all of the box neighborhoods. Our dynamic $kd$-tree is a standard spatial median $kd$-tree [6] augmented with the capability for parallel batch updates. Each internal node maintains metadata on the points in this subtree, which are partitioned by a spatial median along the widest dimension. The points are only stored at leaf nodes. We flatten subtree as a single leaf node when it maintains no more than 16 points.

The tree supports batch insertion by first adding the batch to the root, and then traversing down multiple branches of the tree in parallel. At each internal node, we partition the inserted batch by the spatial median stored at the node, and modify its metadata, such as the point count and the coordinates of its bounding box. At each leaf node, we directly modify the metadata and store the points. The tree supports batch deletions by modifying the metadata, and marking the deleted points as invalid at the leaves. We manage the memory periodically to free up the invalid entries.

### 7 Static Algorithms and Implementations

In addition to our batch-dynamic closest pair algorithm, we implement several parallel algorithms for the static closest pair problem, which we describe in this section. We evaluate all of them against each other, and compare them to our parallel batch-dynamic algorithm in Section 8. As far as we know, this paper presents the first experimental study of parallel algorithms for static closest pairs.

**Divide-and-Conquer Algorithm.** The first divide-and-conquer algorithm for closest-pair is by Bentley [6], which has $O(n \log n)$ work and is optimal in the algebraic decision tree model. Blel-loc and Maggs [12] parallelized this algorithm and it takes $O(n \log n)$ work and $O(\log^2 n)$ depth. There is an earlier parallel algorithm based on multi-way divide-and-conquer by Atallah and Goodrich [4], which takes $O(n \log n \log \log n)$ work and $O(\log n \log \log n)$ depth.

We implement the divide-and-conquer algorithm by Blelloc and Maggs [12]. The main idea of the algorithm is to divide the space containing all the points $S$ along an axis-aligned hyperplane by the median point along a dimension fixed throughout the algorithm, to form left and right subproblems. We then recursively find the closest pair in each of the two subproblems in parallel to obtain results $\delta_L$ and $\delta_R$. Then, we merge the two subproblems, and consider the points near the median point, which are the points within a distance of $\min\{\delta_L, \delta_R\}$ from the median point. We call the set of such points a central slab and then recursively solve the problem on it using an efficient “boundary merging” technique to obtain $\delta_M$. The closest pair will be $\delta(S) = \min\{\delta_L, \delta_R, \delta_M\}$. Finding the median and perform the merge can be done using standard parallel primitives.

The algorithm requires the central slab to be sorted in a dimension $d$ different from the dimension of the divide-and-conquer. The algorithm sorts the points along $d$ by performing recursive partitioning and merging at each level of the divide-and-conquer. Since the central slab can be linear in size, sorting can be expensive in theory. However, we find that the central slab is very small for inputs that arise in practice. Therefore, we simply sort the central slab when needed without using partitioning and merging, which results in better performance in practice. We also coarsen the base case, and switch to a quadratic-work brute-force algorithm when the subproblem size is sufficiently small.

**Rabin’s Algorithm.** Rabin’s algorithm [42] is the first randomized sequential algorithm for the problem. Assuming a unit-cost floor function, Rabin’s algorithm has $O(n)$ expected work. MacKenzie and Stout [38] design a parallel algorithm based on Rabin’s algorithm, and achieve $O(n)$ expected work and $O(1)$ expected depth. However, their algorithm has large constant factor overheads.

We design a simpler parallel version of Rabin’s algorithm. The algorithm takes a sample of $n^c$ points where $c < 1$, and recursively compute the closest distance $\delta'$ of the sample. Then, we construct a grid structure on all the points $S$ using a parallel dictionary, where the box size is set to $\delta'$. For each point $x \in S$, we find its closest neighbor by exploring its neighborhood, and then take the minimum among all $x$ to obtain $\delta(S)$. In terms of work, MacKenzie and Stout [38] showed by recursively finding the closest pair on a sample of size $n^c$, the total work is $O(n)$ in expectation. We find $c = 0.8$ to work well in practice. In terms of depth, our implementation has $O(\log n)$ levels of recursion, each taking $O(\log^* n)$ depth whp, which includes parallel dictionary operations and finding the minimum in parallel. The total depth is $O(\log n \log^* n)$ whp. In the recursion, we coarsen the base case by switching to a brute-force algorithm when the problem is sufficiently small.
Sieve Algorithm. Khuller and Matias [33] propose a simple sequential algorithm called the sieve algorithm that also takes $O(n)$ expected work. The dynamic algorithm by Golin et al. [30] is based on the sieve algorithm. The algorithm proceeds in rounds, where in round $i$, it chooses a random point $x$ from the point set $S_i$ (where $S_1 = S$) and computes $d_i(x)$, the distance to its closest neighbor. Then, the algorithm constructs a grid structure on $S_i$, where each box has a side length of $d_i(x)$. It then moves the points that are sparse in $S_i$ into a new set $S_{i+1}$, and proceeds to the next round, until $S_{t+1}$ is empty. Finally, the algorithm constructs a grid structure on $S$ with boxes of size equal to the smallest box computed during the algorithm. For each point $x \in S$, we compute its closest neighbor using the grid, and then take the minimum to obtain $\delta(S)$.

The sequential algorithm takes $O(n)$ expected work as the number of points decreases geometrically from one level to the next. We obtain a parallel sieve algorithm by using our parallel construction for the sparse partition in Algorithm 1, but without the heap. Our parallel sieve algorithm takes $O(n)$ expected work and $O(\log n \log^* n)$ depth whp.

Incremental Algorithm. Golin and Raman [29] present a sequential incremental algorithm for closest pair with $O(n)$ expected work. Blelloch et al. [11] present a parallel version of this incremental algorithm, which we implement. The parallel algorithm works by maintaining a grid using a dictionary, and inserting the points in a randomized order in batches of exponentially increasing size. The side length of the grid box is the current closest pair distance, which is initialized to the size of the entire data set. Our parallel batch-dynamic algorithm works by maintaining a grid using a dictionary, of this incremental algorithm, which we implement. The parallel algorithm achieves a throughput of up to $35 \times 10^7$ points per second for deletion across all batch sizes.

8 Experiments

Algorithms Evaluated. We evaluate our parallel batch-dynamic algorithm by benchmarking its performance on batch insertions (dynamic-insert) and batch deletions (dynamic-delete). We also evaluate the four static implementations described in Section 7, which we refer to as divide-conquer, rabin, sieve, and incremental. We also implement sequential versions of all of our algorithms that do not have the overheads of parallelism. The running times of our algorithms are shown in Tables 2 and 3.

Data Sets. We use the synthetic seed spreader (SS) data sets produced by Gan and Tao’s generator [27]. The generator produces points generated by a random walk in a local neighborhood, but jumping to a random location with some probability. SS-varden refer to the data sets with variable-density clusters. We also use a synthetic data set called Uniform that contains points distributed uniformly at random inside a bounding hyper-grid with side length $\sqrt{n}$, where $n$ is the total number of points. The points have double-precision floating point values. We generated the synthetic data sets with 10 million points for dimensions $k = 2, 3, 5, 7$. We name the data sets in the format of Dimension-Name-Size.

In addition, we use the following real-world data sets, containing points with double-precision floating point values.

(1) 7D-Household-2M [23] is a 7-dimensional data set containing household sensor data with 2,049,280 points excluding the date-time information.

(2) 16D-Chem-4M [1, 25] is a 16-dimensional data set with 4,208,261 data points containing chemical sensor data.

(3) 3D-Cosmo-298M [34] is a 3-dimensional astronomy data set with 298,246,465 valid data points. We extracted the $x, y,$ and $z$ coordinate information to construct the 3-dimensional data set.

Testing Environment. We perform all of our experiments on an r5.24xlarge machine on Amazon EC2. The machine has 2 $\times$ Intel Xeon Platinum 8259CL CPU (2.50 GHz) CPUs for a total of 48 hyper-threaded cores, and 768 GB of RAM. By default, we use all cores with hyper-threading. We use the g++ compiler (version 7.5) with the -O3 flag, and use Cilk Plus, which is supported in g++, for parallelism in our code [36]. We use the -48h and -1t suffixes in our algorithm names to denote the 48-core with hyper-threading and single-threaded times, respectively. We allocate a maximum of 2 hours for each test, and do not report the times for tests that exceed this limit.

Influence of Batch Size on Throughput. In this experiment, we evaluate our batch-dynamic algorithm by measuring its throughput as a function of the batch size. For insertions, we insert batches of the same size until the entire data set is inserted. For deletions, we start with the entire data set and delete batches of the same size until the entire data set is deleted. We compute throughput by the number of points processed per second. We vary the batch size from $1 \times 10^5$ to the size of the entire data set. Our parallel batch-dynamic algorithm achieves a throughput of up to $1.35 \times 10^7$ points per second for insertion, and $1.06 \times 10^7$ for deletion, under the largest batch size. On average, it achieves $1.75 \times 10^6$ for insertion and $1.94 \times 10^6$ for deletion across all batch sizes.
| Batch Sizes       | Seq | 1 × 10^2 | 1 t | 48h  | Seq | 1 × 10^3 | 1 t | 48h  | Seq | 1 × 10^4 | 1 t | 48h  |
|-------------------|-----|----------|-----|------|-----|----------|-----|------|-----|----------|-----|------|
| 2D-Uniform-10M    | Ins | 2.75 × 10^{-5} | 2.55 × 10^{-5} | 1.40 × 10^{-5} | 3.32 × 10^{-5} | 3.18 × 10^{-5} | 5.66 × 10^{-5} | 2.44 × 10^{-5} | 2.35 × 10^{-5} | 1.66 × 10^{-5} |
|                   | Del | 1.52 × 10^{-5} | 1.52 × 10^{-5} | 9.35 × 10^{-6} | 1.56 × 10^{-5} | 1.67 × 10^{-5} | 3.09 × 10^{-5} | 1.45 × 10^{-5} | 1.55 × 10^{-5} | 9.18 × 10^{-5} |
| 3D-Uniform-10M    | Ins | 9.00 × 10^{-5} | 7.42 × 10^{-5} | 6.13 × 10^{-5} | 8.52 × 10^{-5} | 7.24 × 10^{-5} | 2.99 × 10^{-5} | 7.53 × 10^{-5} | 6.37 × 10^{-5} | 9.52 × 10^{-5} |
|                   | Del | 5.40 × 10^{-5} | 4.77 × 10^{-5} | 3.81 × 10^{-5} | 5.54 × 10^{-5} | 5.06 × 10^{-5} | 1.55 × 10^{-5} | 5.02 × 10^{-5} | 4.59 × 10^{-5} | 6.69 × 10^{-5} |
| 5D-Uniform-10M    | Ins | 4.83 × 10^{-5} | 3.48 × 10^{-5} | 4.08 × 10^{-5} | 2.03 × 10^{-5} | 2.69 × 10^{-5} | 1.14 × 10^{-5} | 5.00 × 10^{-5} | 4.29 × 10^{-5} | 5.60 × 10^{-5} |
|                   | Del | 6.27 × 10^{-5} | 6.16 × 10^{-5} | 8.40 × 10^{-5} | 2.98 × 10^{-5} | 2.37 × 10^{-5} | 2.07 × 10^{-5} | 5.55 × 10^{-5} | 4.46 × 10^{-5} | 7.68 × 10^{-5} |
| 7D-Uniform-10M    | Ins | 4.38 × 10^{-5} | 1.84 × 10^{-5} | 3.67 × 10^{-5} | 2.14 × 10^{-5} | 3.01 × 10^{-5} | 1.23 × 10^{-5} | 4.37 × 10^{-5} | 4.04 × 10^{-5} | 4.65 × 10^{-5} |
|                   | Del | 2.77 × 10^{-5} | 2.48 × 10^{-5} | 2.78 × 10^{-5} | 4.78 × 10^{-5} | 4.51 × 10^{-5} | 3.73 × 10^{-5} | 7.60 × 10^{-5} | 7.26 × 10^{-5} | 1.12 × 10^{-5} |
| 2D-SS-varden-10M  | Ins | 2.44 × 10^{-5} | 2.25 × 10^{-5} | 1.68 × 10^{-5} | 3.99 × 10^{-5} | 3.74 × 10^{-5} | 6.17 × 10^{-5} | 3.67 × 10^{-5} | 3.53 × 10^{-5} | 1.90 × 10^{-5} |
|                   | Del | 9.88 × 10^{-6} | 9.96 × 10^{-6} | 1.02 × 10^{-5} | 1.63 × 10^{-5} | 1.69 × 10^{-5} | 2.82 × 10^{-5} | 1.83 × 10^{-5} | 1.92 × 10^{-5} | 1.21 × 10^{-5} |
| 3D-SS-varden-10M  | Ins | 1.33 × 10^{-5} | 1.12 × 10^{-5} | 9.55 × 10^{-6} | 1.11 × 10^{-5} | 9.56 × 10^{-6} | 3.74 × 10^{-5} | 1.05 × 10^{-5} | 8.89 × 10^{-5} | 1.14 × 10^{-5} |
|                   | Del | 1.00 × 10^{-5} | 9.20 × 10^{-6} | 6.32 × 10^{-6} | 8.40 × 10^{-6} | 8.45 × 10^{-6} | 4.10 × 10^{-5} | 8.33 × 10^{-5} | 8.20 × 10^{-5} | 1.04 × 10^{-5} |
| 5D-SS-varden-10M  | Ins | 1.16 × 10^{-5} | 7.51 × 10^{-6} | 4.85 × 10^{-6} | 1.25 × 10^{-5} | 1.04 × 10^{-5} | 2.42 × 10^{-5} | 1.20 × 10^{-5} | 1.03 × 10^{-5} | 7.19 × 10^{-5} |
|                   | Del | 1.84 × 10^{-5} | 1.74 × 10^{-5} | 1.05 × 10^{-5} | 2.04 × 10^{-5} | 1.94 × 10^{-5} | 3.92 × 10^{-5} | 2.05 × 10^{-5} | 1.94 × 10^{-5} | 1.15 × 10^{-5} |
| 7D-SS-varden-10M  | Ins | 1.01 × 10^{-5} | 6.78 × 10^{-6} | 4.61 × 10^{-6} | 1.11 × 10^{-5} | 8.63 × 10^{-6} | 2.43 × 10^{-5} | 1.09 × 10^{-5} | 8.58 × 10^{-5} | 5.87 × 10^{-5} |
|                   | Del | 1.86 × 10^{-5} | 1.77 × 10^{-5} | 1.14 × 10^{-5} | 2.12 × 10^{-5} | 1.83 × 10^{-5} | 4.36 × 10^{-5} | 2.14 × 10^{-5} | 1.82 × 10^{-5} | 1.30 × 10^{-5} |
| 7D-Household-2M   | Ins | ––            | ––            | 8.15 × 10^{-2} | 7.94 × 10^{-2} | 5.18 × 10^{-2} | 8.94 × 10^{-3} | 2.44 × 10^{-3} | 1.34 × 10^{-3} | 2.66 × 10^{-3} |
|                   | Del | ––            | ––            | 3.66 × 10^{-3} | 2.80 × 10^{-3} | 1.36 × 10^{-3} | 3.16 × 10^{-4} | 1.42 × 10^{-3} | 1.88 × 10^{-3} | 4.19 × 10^{-3} |
| 16D-Chem-4M       | Ins | ––            | ––            | 4.20 × 10^{-2} | 2.62 × 10^{-2} | 2.86 × 10^{-2} | 6.12 × 10^{-3} | 4.88 × 10^{-4} | 6.20 × 10^{-4} | 5.74 × 10^{-4} |
|                   | Del | ––            | ––            | 6.95 × 10^{-2} | 7.00 × 10^{-2} | 7.93 × 10^{-2} | 3.15 × 10^{-2} | 1.48 × 10^{-2} | 1.69 × 10^{-2} | 1.23 × 10^{-2} |
| 3D-Cosmo-298M     | Ins | ––            | ––            | 4.96 × 10^{-4} | ––            | 6.36 × 10^{-4} | ––            | ––            | 2.73 × 10^{-5} | ––            |
|                   | Del | ––            | ––            | 2.62 × 10^{-4} | ––            | 1.86 × 10^{-5} | ––            | ––            | 2.94 × 10^{-4} | ––            |

Table 2: Throughput (number of points processed per second) of the dynamic algorithm with varying batch sizes. "Seq" denotes the sequential implementation, "1t" denotes the parallel implementation run on 1 thread, and "48h" denotes the parallel implementation run on 48 cores with hyper-threading. "Ins" and "Del" denote the throughput for batch-insertion and batch-deletion respectively.
We list the average update throughput for both insertions and deletions under varying batch sizes in Table 2. We show plots of throughput versus batch size for 5D-Uniform-10M and 3D-Cosmo-298M in Figure 4a. For both data sets, we observe that the throughput increases with larger batch sizes because of lower overhead of traversing the sparse partition data structure, and the availability of more parallelism (for the parallel numbers).

**Efficiency of Batch Insertion.** In this experiment, we evaluate the performance of dynamic batch insertion versus using a static algorithm to recompute the closest pair. Specifically, we simulate a scenario where given the data structure storing the closest pair among $c$ data points, we perform an insertion of $b$ additional points. We compare the time taken by the dynamic algorithm to process one batch insertion of size $b$, versus that of the static algorithm for recomputing the closest pair for all $c+b$ points. We set $c$ to contain 40% of the data set and vary $b$.

The running time as a function of $b$ for 5D-Uniform-10M and 3D-Cosmo-298M is shown in Figure 4b. For 5D-Uniform-10M, we see that our batch-dynamic algorithm outperforms the fastest static algorithm when the insertion batch size is smaller than 60 million. For 3D-Cosmo-298M, the dynamic method outperforms the static algorithm when the batch size is less than 3 million. For 5D-Uniform-10M and 3D-Cosmo-298M. For 5D-Uniform-10M, the dynamic algorithm outperforms the fastest static algorithm when the batch size is less than 3 million. For 3D-Cosmo-298M, the dynamic algorithm outperforms the static algorithm when the batch size is less than 60 million.

**Static Methods.** We evaluate and compare the static algorithms and present all detailed running times in Table 3. Among the four parallel static algorithms, Rabin’s algorithm is on average 7.63x faster than the rest of the algorithms across all data sets. The divide-and-conquer, the sieve algorithm, and the incremental algorithm are on average 17.86x, 2.29x, and 2.73x slower than Rabin’s algorithm, respectively. The divide-and-conquer algorithm actually achieves the fastest parallel running time on 6 out of the 11 data sets. However, it is significantly slower for most of the higher dimensional data sets, due to its higher complexity with increased dimensionality. The sieve algorithm and the incremental algorithm, though doing the same amount of work in theory as Rabin’s algorithm, are more complicated, and have higher constant factor overheads, making them slower than Rabin’s algorithm.

**Parallel Speedup and Work-Efficiency.** All of our implementations achieve excellent self-relative parallel speedups. We measure parallel speedup of our implementations by dividing the 1-thread time by the 48-core with hyper-threading time. Our parallel batch-dynamic algorithm achieves up to 38.57x speedup (15.10x on average across all batch sizes), averaging over both insertions and deletions. Our static implementations achieve up to 51.45x speedup (29.42x on average). Specifically, the divide-and-conquer algorithm, Rabin’s algorithm, the sieve algorithm, and the incremental algorithm achieve an average speedup of 35.17x, 33.84x, 29.22x, and 19.45x, respectively.

---

### Table 3: Running time (in seconds) of static algorithms.

| Algorithm          | Seq 1t 48h | Rabin Seq 1t 48h | Sieve Seq 1t 48h | Incremental Seq 1t 48h |
|--------------------|------------|-----------------|------------------|-------------------------|
| 2D-Uniform-10M     | 9.54 9.62 0.24 | 11.17 11.63 0.28 | 23.27 24.53 0.81 | 22.09 17.71 1.02 |
| 3D-Uniform-10M     | 24.86 25.20 0.66 | 28.39 30.46 0.78 | 60.27 60.57 1.82 | 50.46 46.16 2.50 |
| 5D-Uniform-10M     | 101.04 136.12 3.04 | 25.32 28.38 1.28 | 56.74 60.57 2.63 | 49.22 50.29 2.40 |
| 7D-Uniform-10M     | 561.03 618.40 14.74 | 81.65 82.83 1.70 | 124.81 135.74 4.24 | 93.72 106.48 4.58 |
| 2D-SS-varden-10M   | 7.58 8.95 0.23 | 10.52 11.15 0.26 | 22.18 22.78 0.94 | 23.35 17.54 1.11 |
| 3D-SS-varden-10M   | 17.34 19.05 0.51 | 28.36 29.14 0.77 | 58.39 58.28 1.68 | 48.66 43.11 1.97 |
| 5D-SS-varden-10M   | 24.86 33.44 0.82 | 22.59 26.11 1.43 | 47.20 49.33 2.53 | 40.38 41.72 2.44 |
| 7D-SS-varden-10M   | 43.13 50.29 1.33 | 33.10 34.01 1.61 | 64.35 70.90 3.00 | 43.36 48.04 2.53 |
| 7D-Household-2M    | 342.97 498.95 202.13 | 38.27 39.82 1.38 | 88.20 96.66 2.68 | 59.05 70.80 3.91 |
| 16D-Chem-4M        | 315.15 498.95 202.13 | 38.27 39.82 1.38 | 88.20 96.66 2.68 | 59.05 70.80 3.91 |
| 3D-Cosmo-298M      | 720.00 747.47 20.70 | 1242.56 1624.84 31.58 | 3382.83 2818.60 70.63 | 3455.96 2628.45 104.02 |

"Seq" denotes the sequential implementation, "1t" denotes the parallel implementation run on 1 thread, and "48h" denotes the parallel implementation run on 48 cores with hyper-threading.
(a) Plot of throughput vs. batch size in log-log scale for our parallel batch-dynamic algorithm on 5D-Uniform-10M and 3D-Cosmo-298M. The throughput is computed as the number of points processed per second. The algorithm run on 48-cores with hyper-threading and 1 thread has a suffix of "48h" and "1t", respectively. For 3D-Cosmo-298M, we omit the 1-thread time for dynamic as the experiments exceeded our time limit.

(b) Plot of running time (in seconds) vs. insertion batch size for the dynamic and static methods using 48 cores with hyper-threading on 5D-Uniform-10M and 3D-Cosmo-298M. The plot is in log-log scale.

(c) Plot of running time (in seconds) vs. deletion batch size for the dynamic and static methods using 48 cores with hyper-threading on 5D-Uniform-10M and 3D-Cosmo-298M. The plot is in log-log scale.

Figure 4: Parallel performance of the batch-dynamic algorithm on varying batch sizes.
Our parallel implementations are also work-efficient. Comparing with the sequential counterparts, our parallel batch-dynamic algorithm running on 1 thread has a 1.13x higher throughput on average than the sequential algorithm. For the static algorithms, the parallel divide-and-conquer algorithm, Rabin’s algorithm, the sieve algorithm, and the incremental algorithm running on 1 thread are only 1.18x, 1.08x, 1.04x, and 0.98x slower on average, respectively than their corresponding sequential algorithm.

9 Related Work

Static Closest Pair. The problem of finding the closest pair given \( n \) points has been a long-studied problem in computational geometry. There have been several deterministic sequential algorithms \([7, 8, 31, 46]\) that solve the problem optimally in \( O(n \log n) \) time under the standard algebraic decision tree model. Under a different model where the floor function is allowed at unit-cost, Rabin \([42]\) solves the problem in \( O(n) \) expected time. Fortune and Hopcroft \([26]\) present a deterministic algorithm with \( O(n \log n) \) running time under the same model. Later, Khuller and Matias \([33]\) come up with a simple randomized algorithm that takes \( O(n) \) expected time using a sieve data structure. Golin et al. \([29]\) describe a randomized incremental algorithm for the problem that takes \( O(n) \) expected time.

Dietzfelbinger et al. \([22]\) fill in the details for Rabin’s algorithm, in particular concerning hashing and duplicate grouping. Banyassady and Mulzer \([5]\) give a simpler analysis for Rabin’s algorithm. Chan \([19]\) gives an algorithm that takes \( O(n) \) expected time in a randomized optimization framework.

For parallel algorithms, Atallah and Goodrich \([4]\) come up with the first parallel algorithm for geometric closest pair using multi-way divide-and-conquer. The algorithm takes \( O(n \log n \log \log n) \) work and \( O(\log n \log \log n) \) depth. MacKenzie and Stout \([38]\) design a parallel algorithm inspired by Rabin \([42]\) that takes \( O(n) \) work and \( O(1) \) depth in expectation. Blelloch and Maggs \([13]\) parallelize the divide-and-conquer approach in \([7, 8]\), taking \( O(n \log n) \) work and \( O(\log^2 n) \) depth. Blelloch et al. \([11]\) design a randomized incremental algorithm for the problem that takes \( O(n) \) expected work and \( O(\log n \log^* n) \) depth \( \text{whp}. \) Lenhof and Smid \([37]\) solve a close variant, the \( K \)-closest pair problem in \( O(n \log n \log \log n + K) \) work and \( O(\log^2 n \log \log n) \) depth, where \( K \) is the number of closest pairs to return.

Dynamic Closest Pair. There have been semi-dynamic algorithms that focus on only insertions or only deletions. For only deletions, Supowit \([51]\) give an algorithm that maintains the minimal distance for points in \( k \)-space in \( O(\log^k n) \) amortized time per deletion. The method uses \( O(n \log^{k-1} n) \) space. For only insertions, Schwarz et al. \([45]\) design data structures taking \( O(n) \) space and \( O(\log n) \) time per insertion.

For fully-dynamic closest pair algorithms supporting both insertions and deletions, Overmars \([39, 40]\) gives an \( O(n) \) time update algorithm that takes \( O(n \log \log n) \) space. Aggarwal et al. \([2]\) showed that in a 2-dimensional Voronoi diagram, points can be inserted and deleted in \( O(n) \) time, which leads to an update time of \( O(n) \) for the closest pair using only \( O(n) \) space. Smid \([47]\) gives a dynamic data structure of size \( O(n) \), that maintains closest pair of points in \( k \)-space, where distances are measured in the \( L_k \) metric, in \( O(n^{2/3} / \log n) \) time per update.

Later work improve the running time to polylogarithmic time per update. Smid \([48]\) uses a data structure of size \( O(n \log^k n) \) and maintains the closest pair in \( O(\log^k n \log \log n) \) amortized time per update. Callahan and Kosaraju \([16]\) present general technique for dynamizing problems in Euclidean-space that make use of the well-separated pair decomposition \([17]\). For dynamic closest pair, their proposed algorithm requires \( O(n) \) space and \( O(\log^k n) \) time for updates. Bespamyatnikh \([9]\) describes a data structure that takes \( O(n) \) space, and has \( O(\log n) \) deterministic update time for the closest pair in \( L_4 \) metric. The main idea is to dynamically maintain a fair-split tree and a heap of neighbor pairs. The algorithm incurs large constant overheads, and does not currently seem to be practical. Golin et al. \([30]\) describe a randomized data structure for the problem in \( L_4 \) metric. For fixed dimensionality, the data structure supports insertions to and deletions in \( O(\log n) \) expected time and requires expected \( O(n) \) space.

Eppstein \([24]\) solves a stronger version of the problem by supporting arbitrary distance functions. His algorithm maintains the closest pair in \( O(n \log n) \) time per insertion and \( O(n \log^2 n) \) amortized time per deletion using \( O(n) \) space. Cardinal and Eppstein \([18]\) later design a more practical version of this algorithm. Chan \([20]\) presents a modification of Eppstein’s algorithm \([24]\), which improves the amortized update time to \( O(n) \).

10 Conclusion

We have presented a parallel batch-dynamic data structure for the closest pair problem. For inserting or deleting \( m \) points from a set of \( n \) points, the data structure takes \( O(m \log((n + m) / (m + 1))) \) expected work and \( O(\log(n + m) \log^*(n + m)) \) depth \( \text{whp}. \) In addition, we have shown experimentally that it achieves good parallel speedup and high throughput across varying batch sizes. We have also implemented four parallel static closest pair algorithms, which also achieve good parallel speedup. We find that it is more efficient to use our dynamic algorithm than the fastest static algorithm for batch sizes of up to 70% of the data set. For future work, we are interested in designing parallel closest pair algorithms for metrics other than the \( L_p \)-metric.
Acknowledgements

This research was supported by DOE Early Career Award #DESC0018947, NSF CAREER Award #CCF-1845763, Google Faculty Research Award, and DARPA SDH Award #HR0011-18-3-0007.

References

[1] [n.d.]. CHEM Dataset. https://archive.ics.uci.edu/ml/datasets/Gas+sensor+array+under+dynam+s+gas+mixtures.
[2] Alok Aggarwal, Leonidas J. Guibas, James Saxe, and Peter W. Shor. 1989. A Linear-time Algorithm for Computing the Voronoi Diagram of a Convex Polygon. *Discrete & Computational Geometry* 4, 6 (1989), 591–604.
[3] Suguru Arimoto and Hiroshi Noborio. 1988. A 3D Closest Pair Algorithm and its Applications to Robot Motion Planning. *IFAC Proceedings Volumes* 21, 16 (1988), 471 – 480.
[4] Mikhail J. Atallah and Michael T. Goodrich. 1986. Efficient Parallel Solutions to Some Geometric Problems. *J. Parallel Distrib. Comput.* 3, 4 (1986), 492–507.
[5] Bahareh Banyassady and Wolfgang Mulzer. 2007. A Simple Analysis of Rabin’s Algorithm for Finding Closest Pairs. *European Workshop on Computational Geometry (EuroCG)* (2007).
[6] Jon L. Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 508–517.
[7] Jon L. Bentley. 1980. Multidimensional Divide-and-conquer. *Commun. ACM* 23, 4 (1980), 214–229.
[8] Jon L. Bentley and Michael J. Shamos. 1976. Divide-and-conquer in Multidimensional Space. In *ACM Symposium on Theory of Computing (STOC)*. 220–230.
[9] Sergei N. Bespamyatnikh. 1998. An Optimal Algorithm for Closest-pair Maintenance. *Discrete & Computational Geometry* 19, 2 (1998), 175–195.
[10] Guy E. Blelloch, Daniel Fierzovic, and Yihan Sun. 2016. Just Join for Parallel Ordered Sets. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
[11] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2016. Parallelism in Randomized Incremental Algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.
[12] Guy E. Blelloch, Yan Gu, and Yihan Sun. 2017. Efficient Construction of Probabilistic Tree Embeddings. In *Intl. Colloq. on Automata, Languages and Programming (ICALP)*.
[13] Guy E. Blelloch and Bruce M. Maggs. 2010. Parallel Algorithms. In *Algorithms and Theory of Computation Handbook: Special Topics and Techniques*. 25–25.
[14] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (1999), 720–748.
[15] Richard P. Brent. 1974. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM* 21, 2 (April 1974), 201–206.
[16] Paul B. Callahan and S. Rao Kosaraju. 1995. Algorithms for Dynamic Closest Pair and n-Body Potential Fields. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 263–272.
[17] Paul B. Callahan and S. Rao Kosaraju. 1995. A Decomposition of Multidimensional Point Sets with Applications to k-nearest-neighbors and n-body Potential Fields. *J. ACM* 42, 1 (1995), 67–90.
[18] Jean Cardinal and David Eppstein. 2004. Lazy Algorithms for Dynamic Closest Pair with Arbitrary Distance Measures. In *Algorithm Engineering and Experiments (ALENEX)*.
[19] Timothy M. Chan. 1999. Geometric Applications of a Randomized Optimization Technique. *Discrete & Computational Geometry* 22, 4 (1999), 547–567.
[20] Timothy M. Chan. 2020. Dynamic Generalized Closest Pair: Revisiting Eppstein’s Technique. In *Symposium on Simplicity in Algorithms*. 33–37.
[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd edition). MIT Press.
[22] Martin Dietzfelbinger, Torben Hagerup, Jarki Katajainen, and Martti Penttonen. 1997. A Reliable Randomized Algorithm for the Closest-pair Problem. *J. Algorithms* 25, 1 (1997), 19–51.
[23] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml
[24] David Eppstein. 2000. Fast Hierarchical Clustering and Other Applications of Dynamic Closest Pairs. *J. Experimental Algorithmics* 5 (2000), 1–es.
[25] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. 2015. Reservoir Computing Compensates Slow Response of Chemosensor Arrays Exposed to Fast Varying Gas Concentrations in Continuous Monitoring. *Sensors and Actuators B: Chemical* 215 (2015), 618–629.
[26] Steve Fortune and John Hopcroft. 1979. A Note on Rabin’s Nearest-neighbor Algorithm. *Inform. Process. Lett.* 8, 1 (1979), 20–23.
[27] Junhao Gan and Yuifei Tao. 2017. On the Hardness and Approximation of Euclidean DBSCAN. *ACM Trans. Database Syst.* 42, 3 (2017), 14:1–14:45.
[28] Joseph Gil, Yossi Matias, and Uzi Vishkin. 1991. Towards a Theory of Nearly Constant Time Parallel Algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*.
[29] Mordecai Golin, Rajeev Raman, Christian Schwarz, and Michiel Smid. 1995. Simple Randomized Algorithms for Closest Pair Problems. *Nordic J. of Computing* 2, 1 (March 1995), 3–27.
[30] Mordecai Golin, Rajeev Raman, Christian Schwarz, and Michiel Smid. 1998. Randomized Data Structures for the Dynamic Closest-pair Problem. *SIAM J. Scientific Computing* 27, 4 (1998), 1036–1072.
[31] Klaus Hinrichs, Jurg Nievergelt, and Peter Schorn. 1988. Plane-sweep Solves the Closest Pair Problem Elegantly. *Inform. Process. Lett.* 26, 5 (1988), 255–261.
[32] Joseph Jala. 1992. *Introduction to Parallel Algorithms*. Addison-Wesley Professional.
[33] Samir Khuller and Yossi Matias. 1995. A Simple Randomized Sieve Algorithm for the Closest-Pair Problem. *Information and Computation* 118, 1 (April 1995), 34–37.
[34] Philip D. MacKenzie and Quentin F. Stout. 1998. Ultrafast Expected Optimization Technique. *Sensors and Actuators B: Chemical* 4, 2 (1998), 245–260.
[35] Md. Nasir Uddin Laskar and TaeChoong Chung. 2012. Mobile Robot Path Planning : an Efficient Distance Computation between Obstacles using Discrete Boundary Model (DBM).
[36] Charles E. Leiserson. 2010. The Cilk++ Concurrency Platform. *J. Supercomputing* 51, 3 (2010).
[37] Hans-Peter Lenhof and Michiel Smid. 1995. Sequential and Parallel Algorithms for the k Closest Pairs Problem. *International J. of Computational Geometry & Applications* 5, 03 (1995), 273–288.
[38] Philip D. MacKenzie and Quintin F. Stout. 1998. Ultrafast Expected Time Parallel Algorithms. *J. Algorithms* 26, 1 (1998), 1–33.
[39] Mark H. Overmars. 1981. Dynamization of Order Decomposable Set Problems. *J. Algorithms* 2, 3 (1981), 245–260.
[40] Mark H. Overmars. 1983. The Design of Dynamic Data Structures. Vol. 156.
[41] Maria Cristina Pinotti and Geppino Pucci. 1995. Parallel Algorithms for Priority Queue Operations. *Theoretical Computer Science (TCS)* 148, 1 (1995), 171–180.
[42] Michael O. Rabin. 1976. Probabilistic Algorithms. (1976).
Sanguthevar Rajasekaran and Sudipta Pathak. 2014. Efficient Algorithms for the Closest Pair Problem and Applications. (07 2014).

Sanguthevar Rajasekaran and John H. Reif. 1989. Optimal and Sublogarithmic Time Randomized Parallel Sorting algorithms. *SIAM J. Scientific Computing* 18, 3 (1989).

Christian Schwarz, Michiel Smid, and Jack Snoeyink. 1994. An Optimal Algorithm for the On-line Closest-pair Problem. *Algorithmica* 12, 1 (1994), 18–29.

Michael I. Shamos and Dan Hoey. 1975. Closest-point Problems. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. 151–162.

Michiel Smid. 1991. Maintaining the Minimal Distance of a Point Set in Polylogarithmic Time. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1–6.

Michiel Smid. 1992. Maintaining the Minimal Distance of a Point Set in Polylogarithmic Time. *Discrete & Computational Geometry* 7, 4 (1992), 415–431.

Yihan Sun and Guy E. Blelloch. 2018. Parallel Range and Segment Queries with Augmented Maps. *arXiv preprint:1803.08621* (2018).

Yihan Sun and Guy E. Blelloch. 2019. Parallel Range, Segment and Rectangle Queries with Augmented Maps. In *Algorithm Engineering and Experiments (ALENEX)*. 159–173.

Kenneth J. Supowit. 1990. New Techniques for Some Dynamic Closest-point and Farthest-point Problems. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 84–90.

Uzi Vishkin. 2010. Thinking in Parallel: Some Basic Data-Parallel Algorithms. (2010). University of Maryland.