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

Yiqiu Wang
Massachusetts Institute of Technology, Cambridge, MA, USA

Shangdi Yu
Massachusetts Institute of Technology, Cambridge, MA, USA

Yan Gu
University of California, Riverside, CA, USA

Julian Shun
Massachusetts Institute of Technology, Cambridge, MA, USA

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., 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 $O(m(1+\log((n+m)/m)))$ expected work and $O(\log(n+m)\log^*(n+m))$ 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 sets of 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 we are not aware of any existing practical implementations. On 48 cores with hyper-threading, the static 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 20% of the data set. As far as we know, our work is the first to experimentally evaluate parallel closest pair algorithms, in both the static and the dynamic settings.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms; Computing methodologies → Shared memory algorithms; Theory of computation → Computational geometry

Keywords and phrases Closest Pair, Parallel Algorithms, Dynamic Algorithms, Experimental Algorithms

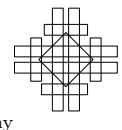
Digital Object Identifier 10.4230/LIPIcs.SoCG.2021.60

Related Version *Full Version*: https://arxiv.org/abs/2010.02379

Supplementary Material *Software (Source Code)*: https://github.com/wangyiqiu/closest-pair

Funding This research was supported by DOE Early Career Award #DESC0018947, NSF CAREER Award #CCF-1845763, Google Faculty Research Award, DARPA SDH Award #HR0011-18-3-0007, and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

Acknowledgements We thank Yihan Sun for help on using the PAM library [45].



1 Introduction

The closest pair problem is a fundamental computational geometry problem with applications in robot motion planning [30, 3], computational biology [37], collision detection, hierarchical clustering, and greedy matching [18]. In many cases, the data involved can quickly change over time. In the case that a subset of the data gets updated, a dynamic algorithm can be much faster than a static algorithm that recomputes the result from scratch.

We consider a metric space (S, d) where S contains n points in $\mathbb{R}^k$, and d is the L_t-metric where $1 \leq t \leq \infty$. 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 the closest pair (p, 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 points of size m in parallel. In this paper, we propose a new parallel batch-dynamic data structure for closest pair on (S, d).

There is a rich literature on sequential dynamic closest pair algorithms [34, 46, 39, 42, 43, 40, 13, 27, 9, 24, 2]. More details on this related work is provided in [44] and the full version of this paper. However, as far as we know, none of the existing dynamic algorithms 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 dynamic closest pair, along with a comprehensive experimental study showing that it performs well in practice. Our solution is inspired by the sequential solution of Golin et al. [24], which takes $O(n)$ space to maintain $O(n)$ points and supports $O(\log n)$ time updates, and is the fastest existing sequential algorithm for the L_t-metric. Our parallel solution takes a batch update of size m and maintains the closest pair in $O(m(1 + \log((n+m)/m)))$ expected work and $O(\log(n+m)\log^*(n+m))$ depth (parallel time) with high probability (*whp*).[1] Compared to the sequential algorithm of Golin et al., our algorithm is work-efficient (i.e., matches the work of the sequential algorithm) for single updates, and has better depth for multiple updates since we process a batch of updates in parallel. Our data structure is based on efficiently maintaining a sparse partition of the points (a data structure used by Golin et al. [24]) 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. This is the first parallel batch-dynamic binary heap in the literature, and may be of independent interest.

We implement our data structure with optimizations to improve performance. In particular, we combine the multiple heaps needed in our theoretically-efficient algorithm into a single heap, which reduces overheads. We also implement a parallel batch-dynamic kd-tree to speed up neighborhood queries. We evaluate our algorithm on both real-world and synthetic data sets. On 48 cores with two-way hyper-threading, we achieve self-relative parallel speedups of up to 38.57x across various batch sizes. Our algorithm achieves throughputs of up to 1.35×10^7 and 1.06×10^7 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 significant work on sequential [41, 36, 8, 20, 7, 25, 23, 28, 16, 14, 5] and parallel [4, 33, 32, 11, 10] static algorithms for the closest pair (more details can be found in [44] and the full version of the paper). As far as we know, none of the existing parallel algorithms have been evaluated and compared empirically. We implement a divide-and-conquer algorithm [11], a variant of Rabin's randomized algorithm [36], our parallelization of the sequential sieve algorithm [28], and a randomized incremental algorithm [10]. On 48

[1] Holds with probability at least $1 - 1/n^c$ for an input of size n and some constant $c > 0$.

cores with two-way hyper-threading, our algorithms achieve self-relative parallel speedups of up to 51.45x.[2] Our evaluation of the static algorithms shows 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 can be advantageous to use the batch-dynamic algorithm for batches containing up to 20% of the data set. Our source code is publicly available at `https://github.com/wangyiqiu/closest-pair`.

2 Review of the Sparse Partition Data Structure

We give an overview of the sequential dynamic closest pair data structure proposed by Golin et al. [24], which is based on the serial static closest pair algorithm by Khuller and Matias [28]. More details are presented in the full version of the paper. Our new parallel algorithm also uses this data structure, which is referred to as the *sparse partition* of an input set.

2.1 Sparse Partition

For a set S with n points, a **sparse partition** [24] is defined as a sequence of 5-tuples $(S_i, S'_i, p_i, q_i, d_i)$ with size L $(1 \leq i \leq L)$, such that **(1)** $S_1 = S$; **(2)** $S'_i \subseteq S_i \subseteq S$; **(3)** If $|S_i| > 1$, p_i is drawn uniformly at random from S_i, and we compute the distance $d_i = d(p_i, q_i)$, to p_i's closest point q_i in S_i; **(4)** For all $x \in S_i$: **a)** if the closest point of x in S_i (denoted as $d(x, S_i)$) is larger than $d_i/3$, then $x \in S'_i$; **b)** if $d(x, S_i) \leq d_i/6k$ then $x \notin S'_i$; and **c)** if $x \in S_{i+1}$, there is a point $y \in S_i$ such that $d(x, y) \leq d_i/3$ and $y \in S_{i+1}$; **(5)** $S_{i+1} = S_i \setminus S'_i$.

The sparse partition is constructed using these rules until $S_{L+1} = \emptyset$. It contains $O(\log n)$ levels in expectation, as $|S_i|$ decreases geometrically. The expected sum of all $|S_i|$ is linear [24]. We call p_i the **pivot** for partition i. At a high level, S'_i contains points that are far enough from each other, and the threshold d_i that defines whether points are "far enough" decreases for increasing i. In particular, for any $1 \leq i < L$, $d_{i+1} \leq d_i/3$ as shown in Golin et al. [24]. Hence, the closest pair will likely show up in deeper levels that do not contain many points. Based on the definition, each S'_i is non-empty, and $\{S'_1, \ldots, S'_L\}$ is a partition of S.

2.2 A Grid-Based Implementation of Sparse Partition

We now describe Golin et al.'s implementation of the sparse partition. There are L levels of the sparse partition, and we refer to each as **level** i for $1 \leq i \leq L$. We maintain each level using a grid data structure, similar to many closest pair algorithms (e.g., [36, 23, 28, 24]).

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. This can be done using hashing. 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 $3^k - 1$ boxes bordering the box containing p, as well as 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 compute $S'_i = \{p \in S_i : p \text{ is sparse in } G_i \text{ relative to } S_i\}$, which satisfies definition (4) of the sparse partition. The points in S'_i are stored in a separate grid.

An example of the grid structure in two dimensions is shown in Figure 1. We illustrate the grid G_i for the S_i of each level, as well as the pivot p_i and its closest neighbor q_i. The grid size is set to $d_i/6k = d(p_i, q_i)/12$. The sparse points, represented by the hollow blue

[2] With hyper-threading, the parallel speedup can be more than the total core count.

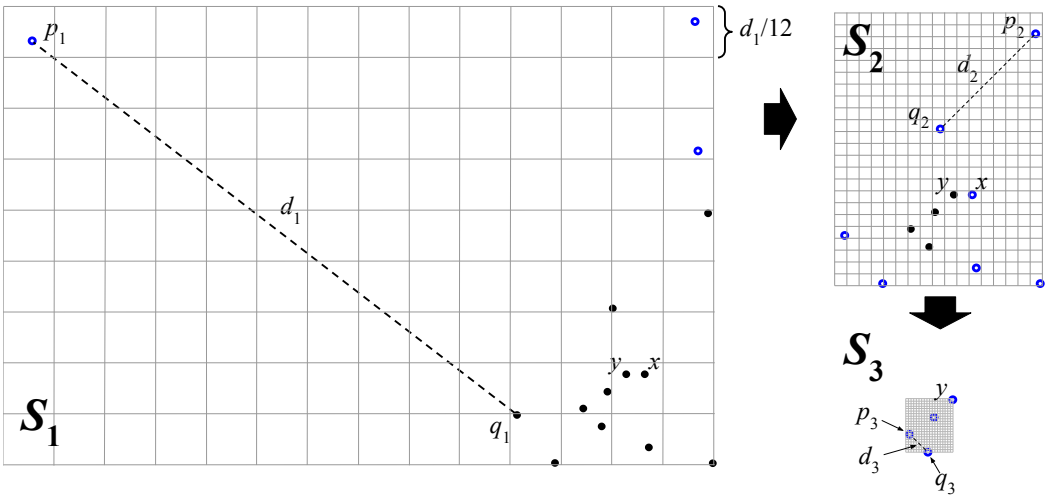


Figure 1 This figure contains an example of 14 points in $\mathbb{R}^2$, for which a grid-based sparse partition $(S_i, S'_i, p_i, q_i, d_i)$ for $1 \leq i \leq 3$ is constructed. On each level, we use a dotted line to indicate d_i, the Euclidean distance between the pivot p_i and its closest neighbor q_i, and we set the grid size to be $d_i/6k = d_i/12$. We denote non-sparse points as solid black circles and sparse points as hollow blue circles. The S'_i sets are represented implicitly by the set of hollow blue circles in each S_i. We denote the true closest pair by letters x and y.

circles, have empty neighborhoods, and do not have another point within a distance of $d_i/3$. The solid black circles, representing the non-sparse points, are copied to the grid G_{i+1} for S_{i+1}. In S_3, all points are sparse.

To construct a sparse partition, the sequential algorithm proceeds in rounds. On round i, the i'th level is constructed. We start with $i = 1$ where $S_1 = S$, and 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 the definition of sparsity 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 [24]. A single insertion of point q starts from S_1, and proceeds level by level. When q is non-sparse in S_i, it will be added to S_{i+1}, and can promote points from S'_i to S'_{i+1} if q falls within their neighborhood. The insertion of q will stop if it becomes sparse at some level, at which point the insertion algorithm finishes. A deletion works in the opposite direction, starting from the last level where the deleted point exists, and working its way back to level 1. Each insertion or deletion takes $O(\log n)$ expected work.

2.3 Obtaining the Closest Pair

As observed by both Khuller and Matias [28] and Golin et al. [24], 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, as illustrated in Figure 1, S_3 for the last level does not contain the closest pair (x, y), as x is sparse on level 2 and not included in S_3. Therefore, we need to check more than just the last level.

The **restricted distance** $d^*_i(p)$ [24] is the closest pair distance from point p to any point in $\bigcup_{0 \leq j \leq k} S'_{i-j}$, and defined as $d^*_i(p) := \min\{d_i, d(p, S'_{i-k} \cup S'_{i-k+1} \cup \ldots \cup S'_i)\}$, where $p \in S'_i$. Golin et al. show that $\delta(S) = \min_{L-k \leq i \leq L} \min_{p \in S'_i} d^*_i(p)$, meaning that the closest pair can be found by taking the minimum among the restricted distance pairs for all points in the last $k+1$ levels of S'_i. The sequential algorithm [24] computes the restricted distance for each

Algorithm 1 Construction.

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

Algorithm MAIN()
  BUILD(S, 1); /* Initially, $S_i := S$. */
Procedure BUILD(S_i, i)
  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.
  Create a parallel dictionary S_i^{dict} to store points in S_i keyed by box ID. In parallel, compute the box ID of each point in S_i based on the grid size, and store the point in the box keyed by the box ID in S_i^{dict}.
  Create a parallel dictionary $S_i'^{\mathsf{dict}}$ to store points in S_i' keyed by box ID. In parallel, determine if each point x in S_i is sparse by checking $N_i(q, S_i)$ (using S_i^{dict}). Store the sparse points in S_i' and $S_i'^{\mathsf{dict}}$, and the remaining points in a new point set represented by an array S_{i+1}.
  In parallel for each point $x \in S_i'$, compute $d_i^*(x)$ by checking its neighborhoods (using $S_j'^{\mathsf{dict}}$) in S_j' where $i - k \le j \le i$.
  fork Create a heap for $\{d_i^*(x) : x \in S_i'\}$.
    BUILD(S_{i+1}, $i+1$) if S_{i+1} is not empty.
  join

point in S_i', and stores it in a min-heap H_i for level i. To obtain the closest pair, we read the minimum of H_i for $L-k \le i \le L$, and then take the overall minimum. This takes $O(1)$ work. In the full version of the paper, we provide more background on the restricted distance.

3 Computational Model

We use the classic **work-depth model** for analyzing parallel algorithms [15, 26]. The **work** W of an algorithm is the number of instructions in the computation, and the **depth** D is the length of the longest sequential dependence. Using Brent's scheduling theorem [12], we can execute a parallel computation in $W/p + D$ running time using p processors. 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 [15]. 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. In the full version of the paper, we present details on the parallel primitives that we use.

For our batch-dynamic data structure, we assume that the updates are independent of the random choices made in our data structure.

4 Parallel Batch-Dynamic Data Structure

In this section, we introduce our parallel batch-dynamic algorithm, including the construction of the sparse partition (defined in Section 2) and how to handle batch updates.

As shown in Algorithm 1, we start with an initial point set S, on which we construct a grid structure recursively level by level, until all points become sparse. Starting with $S_i = S$, the algorithm works on point set S_i for level i. We first pick a pivot point $p_i \in S_i$

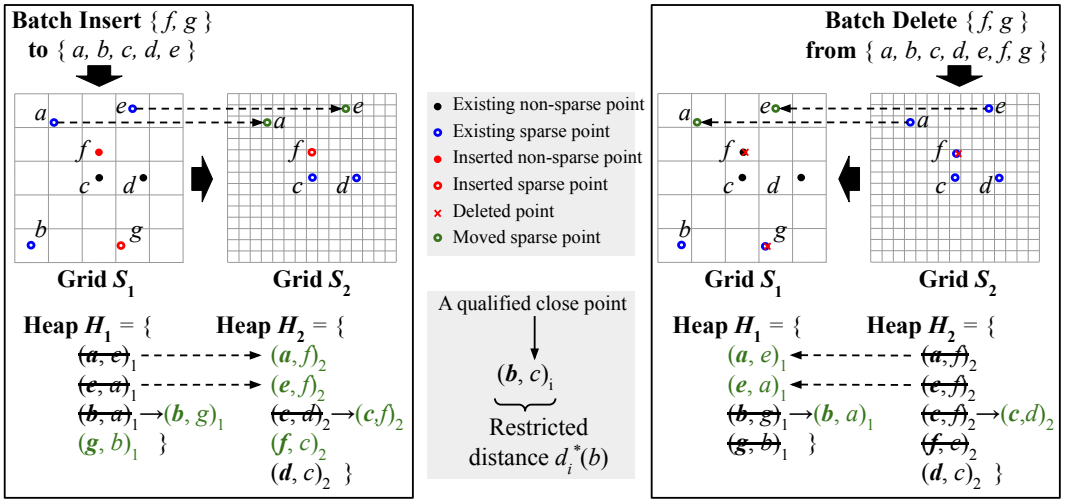


Figure 2 The figure illustrates the interaction between our parallel batch-dynamic insertion (left) and deletion (right) algorithms with the data structure. For ease of illustration, we do not show all the points in the data set. We show the data structure with two levels, and explicitly show S_i and 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 defined in the middle legend. In the lower half of the figures, we show the heaps with the restricted distances that they store. We show the pair defining the restricted distance of a sparse point x on level i as $(\boldsymbol{x}, y)$ if another point y is the closest sparse point to x in levels $i - j$ where $0 \leq j \leq 2$. For both insertion and deletion, we annotate the direction of the update between grids using large 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.

and obtain its closest pair by computing distances to all other points in parallel, followed by computing the minimum to determine the side length of the grid boxes, which we use a parallel dictionary [22] to store (Lines 4–5). We check the sparsity of each point x in parallel by looking up neighboring boxes using the dictionary, and store the sparse points S'_i in a new parallel dictionary and the remaining points in a new point set array S_{i+1} (Line 6). Then, we compute the restricted distances of all points in S'_i in parallel (Line 7), and spawn a thread to asynchronously construct the heap H_i to store the restricted distances (Line 8). We recursively call the construction procedure on S_{i+1} to construct the next level until all points in a level are sparse (Line 9). In the full version of the paper, we include more details about the algorithm, prove a high probability bound on the number of levels, and explain how to achieve optimal linear work and space. We summarize our bounds in the following theorem.

▶ **Theorem 1.** *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.*

Next, we present our parallel algorithm that processes a batch Q of m insertions or deletions. For $m \geq n$, we can simply rebuild the data structure on all of the points using Theorem 1 to obtain the desired bounds. We now describe the case for $m < n$. For batch updates, there are two main tasks: updating the grid and updating the heap. We first describe updating the grid. We let Q_i be the subset of points in Q that are inserted at level i, and $down_i$ be the set of points that move from level $i - 1$ to level i due to the insertion of Q_i. We start with a simple example of an insertion in Figure 2 (left), which originally contains five points $\{a, b, c, d, e\}$. For simplicity, we assume that the pivot remains unchanged

Algorithm 2 Batch Insert.

Input : $(S_i, S'_i, p_i, q_i, d_i)$ and H_i for $1 \le i \le L$; a batch Q to be inserted.

1 **Algorithm** MAIN()
2 | INSERT(Q, $\emptyset$, 1);
3 **Procedure** INSERT(Q_i, $down_i$, i)
4 | $(Q_{i+1}, down_{i+1})$:= GRIDINSERT(Q_i, $down_i$, i);
5 | HEAPUPDATE(i);
6 | **if** $(Q_{i+1} \cup down_{i+1}) \neq \emptyset$ **then** INSERT(Q_{i+1}, $down_{i+1}$, $i+1$) ;
7 **Procedure** GRIDINSERT(Q_i, $down_i$, i)
8 | Determine if p_i, q_i, and d_i should change when inserting Q_i and $down_i$, which happens with probability $(|Q_i| + |down_i|)/(|Q_i| + |down_i| + |S_i|)$, or if a new point is closer to p_i than the previously closest point q_i.
9 | If p_i, q_i, or d_i change on Line 8, or if $i > L$, call BUILD($Q_i \cup down_i \cup S_i, i$) to rebuild subsequent levels, and terminate the batch insertion.
10 | Insert each point in $down_i$ and Q_i into the dictionary of S_i in parallel.
11 | 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}.
12 | For each point x in $down_i$ 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 $down_{i+1}$.
13 | 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 $down_{i+1}$.
14 | **return** $(Q_{i+1}, down_{i+1})$;

and also omits S'_i. $Q_1 = \{f, g\}$ is the set of points inserted into the grid at level 1. We first update S_1 to include f and g, and then update S_2 to include $Q_2 = \{f\}$ but not g, since g is already sparse in S_1. In the example, the insertion of Q_1 triggers further point movements of $\{a, e\}$ from level 1 to level 2, as the sparse points a and e in S_1 become non-sparse due to the insertion of f.

We explain the insertion algorithm in detail, and defer most details of the deletion algorithm to the full version of the paper. As shown in Algorithm 2, the update proceeds recursively level by level (Lines 3–6). Each call to the procedure INSERT(Q_i, $down_i$, i) updates $(S_i, S'_i, p_i, q_i, d_i)$ and H_i. Initially, $Q_1 = Q$ and $down_1$ is empty, as shown on Line 2. For each level i, we update the pivot and rebuild the level with probability $(|Q_i| + |down_i|)/(|Q_i| + |down_i| + |S_i|)$ to ensure that the pivot is still selected uniformly at random among the points in S_i, and we also update the pivot if a new point is closer to p_i than the previous closest point q_i (Lines 8–9). Otherwise, we insert the points in both $down_i$ and Q_i into the dictionary representing S_i. We then check if the points that we inserted are sparse, and insert the sparse ones into the dictionary representing S'_i. The points that are not sparse will be added to sets $down_{i+1}$ and Q_{i+1} and passed on to the next level (Lines 10–12). We then determine additional elements of $down_{i+1}$ by including the points in the neighborhood of Q_i in S'_i (Line 13). In general, $down_{i+1}$ is computed by $down_{i+1} = \{x \mid x \in N_i(q, S'_i \cup down_i)$ for some $q \in Q_i\}$. If Q_{i+1} and $down_{i+1}$ are empty, nothing further needs to be done for subsequent levels, and the tuples $(S_l, S'_l, p_l, q_l, d_l)$ for $i < l \le L$ remain unchanged.

We now argue that the algorithm is correct. Consider a round i that inserts a non-empty $Q_i \cup down_i$. After the insertion, the pivot is still chosen uniformly at random, since on Line 8, we choose p_i such that each point in $S_i \cup Q_i \cup down_i$ has the same probability of being chosen. All sparse points in Q_i and $down_i$ inserted into S_i are included in S'_i (Lines 11–12). 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.

In our algorithm, each point can be moved across multiple levels as a result of a batch insertion. In the full version of the paper, we prove the following lemmas, which are key to maintaining work-efficiency.

▶ **Lemma 2.** $|\bigcup_{1\le i\le L} down_i| \le m \cdot 3^k = O(m)$

▶ **Lemma 3.** $\sum_{1\le i\le L} E[|Q_i|] = O(m)$

Deletions work similarly in the reverse direction as shown in Figure 2 (right), and we provide more details in the full version of the paper. We obtain Lemmas 4 and 5.

▶ **Lemma 4.** *We can maintain a sparse partition for a batch of m insertions in $O(m)$ amortized work in expectation and $O(\log(n+m)\log^*(n+m))$ 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 $(|Q_i|+|down_i|)/(|S_i|+|Q_i|+|down_i|)$. When the pivot p_i is unchanged, it may update its closest point to q_i^* from $Q_i \cup down_i$. 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 $Q_i \cup down_i$, it follows that they can be the nearest neighbors to $O(3^k \cdot (|Q_i|+|down_i|))$ points in S_i. Therefore, the pivot distance changes with probability at most $3^k \cdot (|Q_i|+|down_i|)/|S_i|$, in which case we rebuild the sparse partition. The expected work of rebuilding at level i is $O((|S_i|+|Q_i|+|down_i|)\cdot((|Q_i|+|down_i|)/(|S_i|+|Q_i|+|down_i|)+3^k\cdot(|Q_i|+|down_i|)/|S_i|)) = O(m)$. 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 $O(\log(n+m)\log^*(n+m))$ *whp* to the depth by Theorem 1.

For the rest of the algorithm, in terms of work, Line 10–12 does work proportional to $O(\sum_i(|Q_i|+|down_i|)) = O(m)$ across all the levels due to Lemmas 2 and 3. 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 $O(\log^*(n+m))$ depth using the parallel dictionary. Therefore, all operations in Lines 10–13 takes $O(\log^*(n+m))$ depth, and across all $O(\log(n+m))$ *whp* rounds, the total depth is $O(\log(n+m)\log^*(n+m))$ *whp*. ◀

▶ **Lemma 5.** *We can maintain a sparse partition for a batch of m deletions in $O(m)$ amortized work in expectation and $O(\log(n+m)\log^*(n+m))$ depth whp.*

Now we describe the parallel updates of min-heaps H_i associated with each level i of the sparse partition. Recall that H_i contains the restricted distances $d_i^*(q)$ for $q \in S'_i$. By definition, $d_i^*(q)$ is the closest distance of q to another point in S'_{i-l} where $0 \le l \le k$ (k is the dimensionality). Therefore, following an update on S'_i, we need to update the $d_i^*(q)$'s in H_{i+l} for $0 \le l \le k$, and $q \in S'_{i+l}$. We use same example in Figure 2 (left), where we denote the restricted distance of point x as $(\boldsymbol{x}, y)_i = d_i^*(x) = d(x,y)$, where $y \in \bigcup_{0\le j\le k} S'_{i-j}$ is another point that defines x's closest distance. As shown in Figure 2 (left), due to the insertion of the sparse point g to S_1, entry $(\boldsymbol{g}, b)_1$ is added to H_1. Some entries in H_1 are moved due to

the point movements, e.g., $(\boldsymbol{a}, e)_1$ from H_1 is moved and updated to $(\boldsymbol{a}, f)_2$ in H_2 because a has moved from S_1 to S_2, and f is now closer. Some entries are updated, e.g., $(\boldsymbol{c}, d)_2$ is updated to $(\boldsymbol{c}, f)_2$ in H_2 since the new point f is closer to c than d.

Our algorithm uses a new parallel batch-dynamic binary heap that we introduce in Section 5. For a batch-parallel binary heap of size n and a batch update (a mix of inserts, deletes, and increase/decrease-keys) of size m, updating the heap takes $O(m \log((n+m)/m))$ work and $O(\log(n+m))$ depth, and find-min takes $O(1)$ work. A simple implementation of our algorithm would execute all min-heap updates for an updated S'_i before processing the updates for S'_j where $j > i$ for insertions and $j < i$ for deletions. This level-by-level dependence leads to $O(\log^2(n+m))$ depth overall for the heap updates. In the full version of the paper, we present an improved algorithm that breaks the level dependencies and enables updates to be performed to a min-heap as early as possible, leading to $O(\log(n+m))$ depth overall for the heap updates. Together with Lemma 4 and Lemma 5, we obtain the following theorem, whose analysis we present in the full version of the paper.

▶ **Theorem 6.** *Updating our data structure for a batch of m insertions/deletions takes amortized $O(m(1+\log((n+m)/m)))$ expected work and $O(\log(n+m)\log^*(n+m))$ depth whp.*

Obtaining the closest pair from our data structure takes $O(1)$ work and depth. We simply call find-min on H_i for $L-k \leq i \leq L$, and then take the overall minimum.

5 Parallel Batch-Dynamic Binary Heap

One of the key components in parallelizing our closest pair algorithm is a parallel binary heap that supports batch updates (inserts and deletes) and find-min efficiently. This heap allows us to perform the parallel construction in linear work and perform updates with low depth. Our data structure may be of independent interest, since to the best of our knowledge, the only existing work on parallelizing a binary heap is on individual inserts or deletes [35].

A binary heap is a complete binary tree, where each node contains a key that is smaller than or equal to the keys of its children. Sequentially, the construction of a binary heap takes linear work, and each insert and delete takes $O(\log n)$ work [15]. The heap is represented as an array, and uses relative positions within the array to represent child-parent relationships. Sequentially, each insertion adds a new node at the end of the heap and runs Up-Heap to propagate the node up to the correct position in the heap. A deletion first swaps the node to delete with the node to the end of the heap, reduces the heap size by one, and then runs Up-Heap followed by Down-Heap (to propagate a node down to its correct position) for the node swapped to the middle of the heap.

Central to our parallel batch-dynamic binary heap is a new parallel Heapify algorithm, that takes m updates from a valid heap of n elements, and returns another valid heap. It runs in two phases: the first phase works on increase-key updates, and the second phase on decrease-key updates. In both phases, we first use parallel integer sorting [38, 47] to categorize all updates based on the level where the update belongs. Simply running the Up-Heap and Down-Heap calls for the different updates in parallel does not achieve work-efficiency and low depth, and also leads to potential data races. Therefore, we *pipeline* each level of the Up-Heap and Down-Heap procedure. Specifically, in the first phase, once the first swap for the Down-Heap in level i is finished, we can immediately start the Down-Heap on level $i-1$, instead of waiting for the Down-Heap in level i to completely finish (the root is at level 0, and level numbers increase going down). The swaps in the Down-Heap calls from

level $i-1$ will never catch up with the swaps from level i. Pipelining the second phase with Up-Heap is more complicated. Our parallel Up-Heap is run in a level-synchronous manner from the top level down to the bottom level. For each node on each level, both of its children may want to swap with the parent for having a larger value. In the parallel algorithm, we only make the child with the smaller value swap with the parent and continue its update to the upper levels, while the update for the other child terminates. We prove in the full paper that our parallel Heapify algorithm takes $O(m(1+\log(n/m)))$ work and $O(\log n)$ depth.

We now 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 first add the m elements to end of the heap with keys of ∞. Then, we decrease the keys of these m elements to their true values and run the parallel Heapify algorithm. 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 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 parallel Heapify algorithm. Hence, batch insertions and deletions take $O(m(1+\log((n+m)/m)))$ work and $O(\log(n+m))$ depth.

We provide more details about our data structure and prove the following theorem in the full version of the paper.

▶ **Theorem 7.** *For a batch-parallel binary heap of size n and a batch update (a mix of inserts, deletes, and increase/decrease-keys) of size m, updating the heap takes $O(m(1+\log((n+m)/m)))$ work and $O(\log(n+m))$ depth, and find-min takes $O(1)$ work.*

6 Implementations

Simplified Data Structure. While the sparse partition maintains $(S_i, S'_i, p_i, q_i, d_i)$ and H_i for each level $1 \le i \le L$, we found that implementing S'_i and its associated heap H_i on every level was inefficient in practice. We found it more efficient to only maintain (S_i, p_i, q_i, d_i) for $1 \le i \le L$, and one heap H^* that stores the closest neighbor distances for all q in S_j, where $j = L - \lceil \log_3 2\sqrt{k} \rceil$. When L changes due to insertion or deletion, we recompute j and rebuild H^* if necessary. We prove in the full version of the paper that H^* contains the closest pair. Our implementation uses the parallel heap from [45]. Additionally, we compute S'_i from S_i on the fly when needed.

Neighborhood Search. Some of the work bounds are exponential in the dimensionality k, e.g., a grid's box neighborhood is of size 3^k. For $k \ge 5$, the straightforward implementation is inefficient due to a large constant overhead in the work. Hence, we implement a parallel batch-dynamic kd-tree for $k \ge 5$. This is because performing a range query on the tree works better in practice, as it only needs to traverse the non-empty boxes in the neighborhood instead of all boxes. 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 its subtree, which are partitioned by a spatial median along the widest dimension. The points are only stored at leaf nodes. We flatten a subtree to a single leaf node when it contains at most 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 at the leaves as invalid. We manage the memory periodically to free up the invalid entries.

Static Algorithms. In addition to our batch-dynamic closest pair algorithm, we implement several sequential and parallel algorithms for the static closest pair problem. As far as we know, this paper presents the first experimental study of parallel algorithms for static closest pair. We implement a parallel divide-and-conquer algorithm by Blelloch and Maggs [11], a simplified and parallel version of Rabin's algorithm [36] that we designed, a parallel version of Khuller and Matias's [28] sieve algorithm that we designed, and a parallel randomized incremental algorithm by Blelloch et al. [10]. We explain more details about these static algorithms and their implementations in the full paper.

7 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 6, which we refer to as **divide-conquer**, **rabin**, **sieve**, and **incremental**. In addition, we implement and evaluate sequential versions of all of our algorithms that do not have the overheads of parallelism. Our implementations use the Euclidean metric (L_2-metric).

Data Sets. We use the synthetic seed spreader (SS) data sets produced by the generator in [21]. It produces points generated by a random walk in a local neighborhood, but jumping to a random location with some probability. **SS-varden** refers to the data sets with variable-density clusters. We also use a synthetic data set called **Uniform**, in which points are distributed uniformly at random inside a bounding hyper-cube 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**. We also use the following real-world data sets: **7D-Household-2M** [17] is a 7-dimensional data set containing household sensor data with $2,049,280$ points excluding the date-time information; **16D-Chem-4M** [19, 1] is a 16-dimensional data set with $4,208,261$ points containing chemical sensor data; and **3D-Cosmo-298M** [29] is a 3-dimensional astronomy data set with $298,246,465$ points.

Testing Environment. Our experiments are run on an `r5.24xlarge` instance on Amazon EC2. The machine has 2 × Intel Xeon Platinum 8259CL CPU (2.50 GHz) CPUs for a total of 48 cores with two-way hyper-threading, 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 for parallelism [31]. 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 times for tests that exceed this limit.

Influence of Batch Size on Throughput. In this experiment, we evaluate our batch-dynamic algorithm by measuring their 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

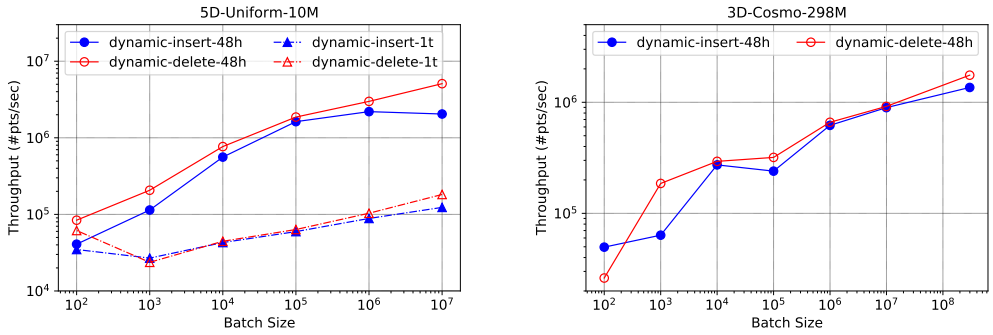


Figure 3 Plots of throughput vs. batch size in log-log scale for our parallel batch-dynamic algorithm on 5D-Uniform-10M and 3D-Cosmo-298M. The algorithm on 48-cores with hyper-threading and 1 thread has a suffix of "48h" and "1t", respectively. For 3D-Cosmo-298M, we omit the 1-thread times as the experiments exceeded our time limit.

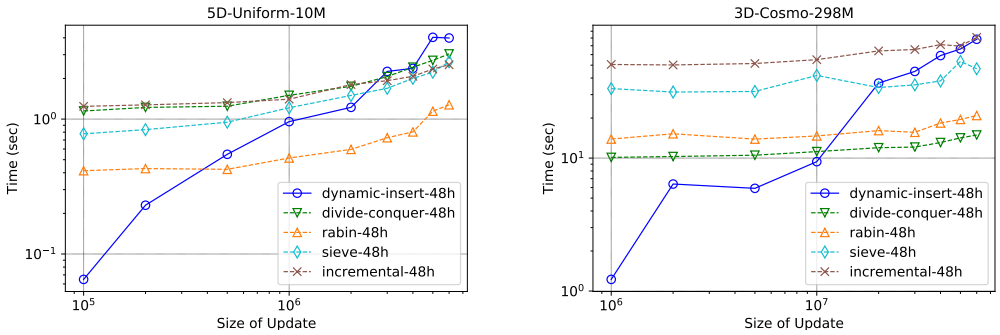


Figure 4 Plots of running time (in seconds) vs. insertion batch size for the dynamic and static methods using 48 cores with hyper-threading on 5D-Uniform-10M and 3D-Cosmo-298M. The plots are in log-log scale.

size from 100 points to the size of the entire data set. Our parallel batch-dynamic algorithm achieves a throughput of up to 1.35×10^7 points per second for insertion, and up to 1.06×10^7 for deletion, under the largest batch size. On average, it achieves 1.75×10^6 for insertion and 1.94×10^6 for deletion across all batch sizes. We show plots of throughput vs. batch size for 5D-Uniform-10M and 3D-Cosmo-298M in Figure 3. We see that the throughput increases with larger batch sizes because of a lower relative overhead of traversing the sparse partition data structure, and the availability of more parallelism.

Efficiency of Batch Insertions. We evaluate the performance of dynamic batch insertion vs. 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, vs. that of a 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. Figure 4 shows the running time as a function of b for 5D-Uniform-10M and 3D-Cosmo-298M. For 5D-Uniform-10M, we see that our batch-dynamic algorithm outperforms the fastest among the static algorithms when the insertion batch size is smaller than 500,000. For 3D-Cosmo-298M, we see that the dynamic method outperforms the fastest static algorithm when the insertion

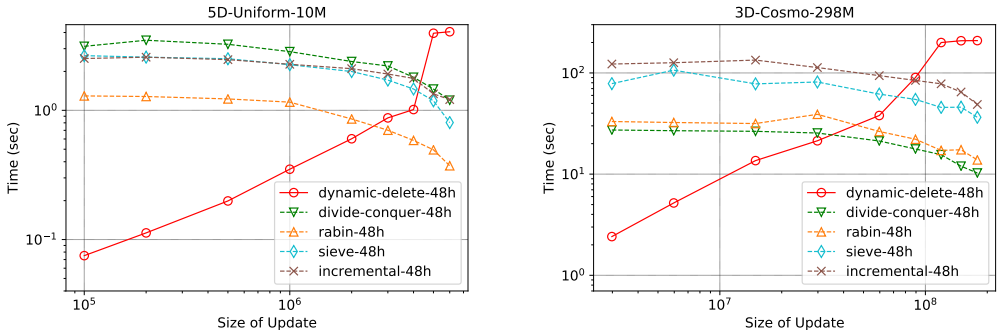


Figure 5 Plots of running time (in seconds) vs. deletion batch size for the dynamic and static methods using 48 cores with hyper-threading on 5D-Uniform-10M and 3D-Cosmo-298M. The plots are in log-log scale.

batch is smaller than 10 million. In general, both the static and dynamic algorithms require more time to process the updates when the batch size is larger. The dynamic algorithm is much more advantageous for small to moderate batch sizes.

Efficiency of Batch Deletions. We evaluate the performance of dynamic batch deletion vs. using a static algorithm to recompute the closest pair. In this experiment, we are given the closest pair of all n points in the data set, and perform a deletion of b points. We compare the time taken for the dynamic algorithm to process one batch deletion of size b, vs. that of a static algorithm for recomputing the closest pair for the $n - b$ remaining points. Figure 5 shows the running time vs. deletion batch size 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. In general, our dynamic algorithm requires more time to process the update when the batch size is larger, while the converse is true for the static algorithms.

Compared to the fastest static algorithms on our data sets, we find that it is faster to use our dynamic algorithm for batch sizes of up to 20% of the data set.

Static Methods. We evaluate and compare the static algorithms and present all detailed running times in Table 1. 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, sieve, and the incremental algorithms 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 7 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, have higher constant factor overheads.

Parallel Speedup and Work-Efficiency. We measure the parallel speedups 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 self-relative 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

Table 1 Running times (in seconds) of static algorithms. "Seq" denotes the sequential implementation. "1t" and "48h" denote the parallel implementation run on 1 thread and 48 cores with hyper-threading, respectively.

	Divide-Conquer			Rabin			Sieve			Incremental		
	Seq	1t	48h	Seq	1t	48h	Seq	1t	48h	Seq	1t	48h
2D-Uniform-10M	9.54	9.62	0.24	11.2	11.6	0.28	23.3	24.5	0.81	22.1	17.7	1.02
3D-Uniform-10M	24.9	25.2	0.66	28.4	30.5	0.78	60.3	60.6	1.82	50.5	46.2	2.50
5D-Uniform-10M	101	136	3.04	25.3	28.4	1.28	56.7	60.6	2.63	49.2	50.3	2.40
7D-Uniform-10M	561	618	14.7	81.7	82.8	1.70	124	135	4.24	93.7	106	4.58
2D-SS-varden-10M	7.58	8.95	0.23	10.5	11.2	0.26	22.2	22.8	0.94	23.4	17.5	1.11
3D-SS-varden-10M	17.3	19.1	0.51	28.4	29.1	0.77	58.4	58.3	1.68	48.7	43.1	1.97
5D-SS-varden-10M	24.9	33.4	0.82	22.6	26.1	1.43	47.2	49.3	2.58	40.4	41.7	2.44
7D-SS-varden-10M	43.1	50.3	1.33	33.1	34.0	1.61	64.4	70.9	3.00	43.4	48.0	2.53
7D-Household-2M	342	392	13.4	7.23	7.70	0.40	15.9	18.1	0.73	13.8	15.7	0.94
16D-Chem-4M	315	499	202	38.3	39.8	1.38	88.2	96.7	2.68	59.1	70.8	3.91
3D-Cosmo-298M	750	747	20.7	1243	1625	31.6	3383	2819	70.6	3456	2629	104

algorithm, Rabin's algorithm, the sieve algorithm, and the incremental algorithm achieve average self-relative speedups of 35.17x, 33.84x, 29.22x, and 19.45x, respectively; in addition, they achieve an average speedup of 19.10x, 23.56x, 10.56x, and 9.23x, respectively, over the fastest serial algorithm for each data set.

Our parallel implementations when run on one thread demonstrate modest overheads over their sequential counterparts. Our parallel batch-dynamic algorithm running on 1 thread has only 1.13x lower throughput on average over our sequential implementation of the algorithm. For the static algorithms, the parallel divide-and-conquer, Rabin's, sieve, and incremental algorithms running on 1 thread are only 1.18x, 1.08x, 1.04x, and 1.00x slower on average, respectively, than their corresponding sequential algorithms.

References

1 Chem dataset. URL: `https://archive.ics.uci.edu/ml/datasets/Gas+sensor+array+under+dynamic+gas+mixtures`.

2 Pankaj K. Agarwal, Haim Kaplan, and Micha Sharir. Kinetic and dynamic data structures for closest pair and all nearest neighbors. *ACM Transactions on Algorithms (TALG)*, 5(1):1–37, 2008.

3 Suguru Arimoto and Hiroshi Noborio. A 3d closest pair algorithm and its applications to robot motion planning. *IFAC Proceedings Volumes*, 21(16):471–480, 1988.

4 Mikhail J. Atallah and Michael T. Goodrich. Efficient parallel solutions to some geometric problems. *J. Parallel Distrib. Comput.*, 3(4):492–507, 1986.

5 Bahareh Banyassady and Wolfgang Mulzer. A simple analysis of Rabin's algorithm for finding closest pairs. *European Workshop on Computational Geometry (EuroCG)*, 2007.

6 Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

7 Jon L. Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4):214–229, 1980.

8 Jon L. Bentley and Michael I. Shamos. Divide-and-conquer in multidimensional space. In *ACM Symposium on Theory of Computing (STOC)*, pages 220–230, 1976.

9 Sergei N. Bespamyatnikh. An optimal algorithm for closest-pair maintenance. *Discrete & Computational Geometry*, 19(2):175–195, 1998.

10 Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, page 467–478, 2016.

11 Guy E. Blelloch and Bruce M. Maggs. Parallel algorithms. In *Algorithms and Theory of Computation Handbook: Special Topics and Techniques*, pages 25–25. Chapman & Hall/CRC, 2010.

12 Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.

13 Paul B. Callahan and S. Rao Kosaraju. Algorithms for dynamic closest pair and n-body potential fields. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 263–272, 1995.

14 Timothy M. Chan. Geometric applications of a randomized optimization technique. *Discrete & Computational Geometry*, 22(4):547–567, 1999.

15 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3rd edition)*. MIT Press, 2009.

16 Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997.

17 Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL: `http://archive.ics.uci.edu/ml`.

18 David Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. *J. Experimental Algorithmics*, 5:1–es, 2000.

19 Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical*, 215:618–629, 2015.

20 Steve Fortune and John Hopcroft. A note on rabin's nearest-neighbor algorithm. *Information Processing Letters*, 8(1):20–23, 1979.

21 Junhao Gan and Yufei Tao. On the hardness and approximation of euclidean DBSCAN. *ACM Transactions Database Systems*, 42(3):14:1–14:45, 2017.

22 Joseph Gil, Yossi Matias, and Uzi Vishkin. Towards a theory of nearly constant time parallel algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 698–710, 1991.

23 Mordecai Golin, Rajeev Raman, Christian Schwarz, and Michiel Smid. Simple randomized algorithms for closest pair problems. *Nordic J. of Computing*, 2(1):3–27, 1995.

24 Mordecai Golin, Rajeev Raman, Christian Schwarz, and Michiel Smid. Randomized data structures for the dynamic closest-pair problem. *SIAM J. Scientific Computing*, 27(4):1036–1072, 1998.

25 Klaus Hinrichs, Jurg Nievergelt, and Peter Schorn. Plane-sweep solves the closest pair problem elegantly. *Information Processing Letters*, 26(5):255–261, 1988.

26 Joseph JaJa. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

27 Sanjiv Kapoor and Michiel Smid. New techniques for exact and approximate dynamic closest-point problems. *SIAM J. Scientific Computing*, 25(4):775–796, 1996.

28 Samir Khuller and Yossi Matias. A simple randomized sieve algorithm for the closest-pair problem. *Information and Computation*, 118(1):34–37, April 1995.

29 YongChul Kwon, Dylan Nunley, Jeffrey P. Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *Scientific and Statistical Database Management*, pages 132–150, 2010.

30 Md. Nasir Uddin Laskar and TaeChoong Chung. Mobile robot path planning : an efficient distance computation between obstacles using discrete boundary model (dbm), 2012.

31 Charles E. Leiserson. The Cilk++ concurrency platform. *J. Supercomputing*, 51(3), 2010.

32 Hans-Peter Lenhof and Michiel Smid. Sequential and parallel algorithms for the k closest pairs problem. *International J. of Computational Geometry & Applications*, 5(03):273–288, 1995.

33 Philip D. MacKenzie and Quentin F. Stout. Ultrafast expected time parallel algorithms. *J. Algorithms*, 26(1):1–33, 1998.

34 Mark H. Overmars. Dynamization of order decomposable set problems. *J. Algorithms*, 2(3):245–260, 1981.

35 Maria Cristina Pinotti and Geppino Pucci. Parallel algorithms for priority queue operations. *Theoretical Computer Science (TCS)*, 148(1):171–180, 1995.

36 Michael O. Rabin. Probabilistic algorithms, 1976.

37 Sanguthevar Rajasekaran and Sudipta Pathak. Efficient algorithms for the closest pair problem and applications. *arXiv preprint*, 2014. `arXiv:1407.5609`.

38 Sanguthevar Rajasekaran and John H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Scientific Computing*, 18(3):594–607, 1989.

39 Christian Schwarz and Michiel Smid. An $O(n \log n \log \log n)$ algorithm for the on-line closest pair problem. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, SODA '92, page 280–285, USA, 1992. Society for Industrial and Applied Mathematics.

40 Christian Schwarz, Michiel Smid, and Jack Snoeyink. An optimal algorithm for the on-line closest-pair problem. *Algorithmica*, 12(1):18–29, 1994.

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

42 Michiel Smid. Maintaining the minimal distance of a point set in less than linear time. In *Algorithms Rev.*, pages 33–44, 1991.

43 Michiel Smid. Maintaining the minimal distance of a point set in polylogarithmic time. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1–6, 1991.

44 Michiel Smid. Closest-point problems in computational geometry. In *Handbook of Computational Geometry*, pages 877–935. North Holland / Elsevier, 2000.

45 Yihan Sun and Guy E. Blelloch. Parallel range, segment and rectangle queries with augmented maps. In *Algorithm Engineering and Experiments (ALENEX)*, pages 159–173, 2019.

46 Kenneth J. Supowit. New techniques for some dynamic closest-point and farthest-point problems. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 84–90, 1990.

47 Uzi Vishkin. Thinking in parallel: Some basic data-parallel algorithms. University of Maryland, 2010.