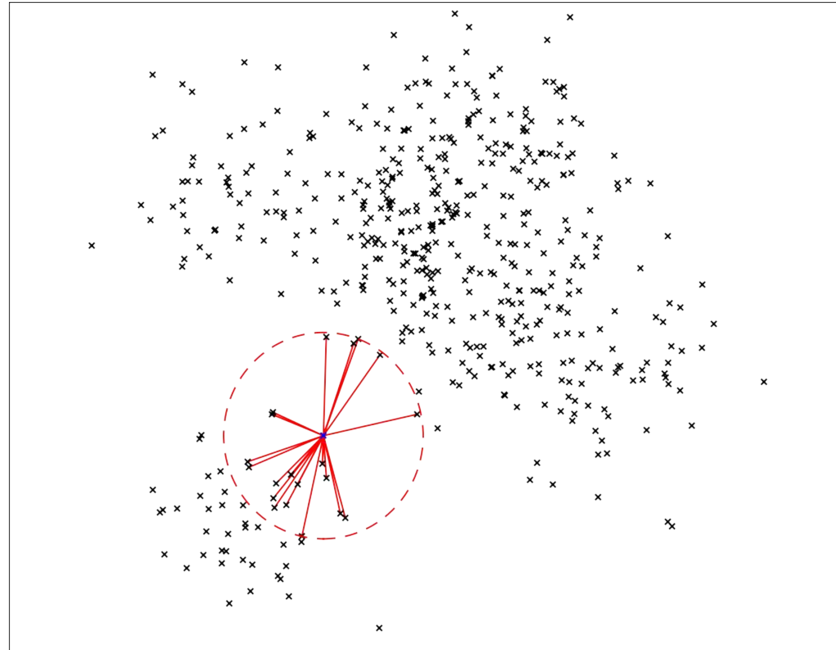
ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms

By MSBDGSS

Problem

- Given a data set P, an integer parameter k, and a query point q: output the k data points in P that are closest (according to some metric) to q



Challenges for Efficient KNN Algorithms

- Curse of Dimensionality - Non-trivial algorithms typically have query times that depend exponentially on the dimension

Approximate Nearest Neighbor Search

- Used as the core subroutine for many modern applications including search recommendations, machine learning, and information retrieval
- Some of the best-performing ANNS algorithms today are graph-based ANNS algorithms
- These algorithms construct a “proximity” graph whose vertices are the point set. Vertices (points) are then connected to “closeby” vertices (points)
- An ANNS search consist of a traversal of the proximity graph from a source point that greedily explores points that are closer to the query until the search converges

Issues

- There is very little work that systematically studies how parallel graph-based ANNS algorithms scale
- Existing parallel implementations of graph-based algorithms rely on locks which introduce non-deterministic outputs
- Existing benchmarks for graph-based ANNS algorithms focus on relatively small input sizes and evaluate sequential performance

Contributions of this Paper

- ParlayANN - a parallel ANNS library that scales to billion-point datasets, scales to more than a hundred threads, and is deterministic
- New general techniques for building ANNS graphs in parallel
- A high-performance implementation of ParlayANN that contains implementations of four state of the art graph-based algorithms: DiskANN, HNSW, HCNNG, and PyNNDescent
- Accurate depiction of performance comparison among ANNS algorithms on billion-scale datasets

Scalability Results

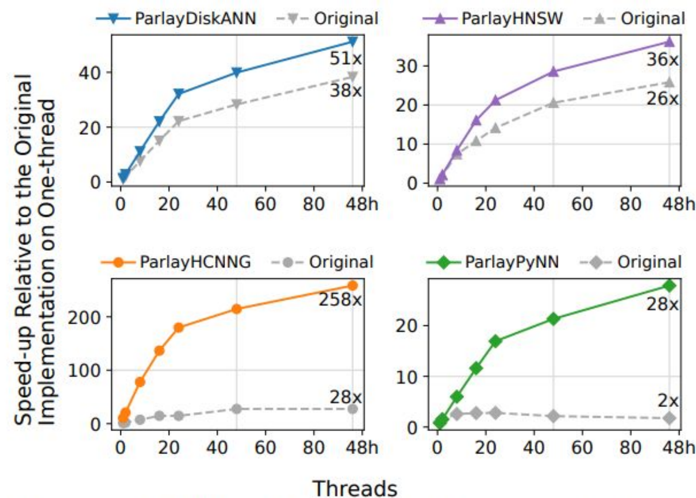


Figure 1. Scalability of original and our new implementations of four ANNS algorithms on various number of threads. Within each subfigure, all numbers are *speedup numbers relative to the original implementation on one thread*. Higher is better. Results were tested on a machine with 48 cores using dataset BIGANN-1M (10^6 points). "48h": 48 cores with hyperthreads. The two implementations in the same subfigure always use the same parameters and give similar query quality (recall-QPS curve).

Graph-Based ANNS Algorithms - High Level Approach

- ANNS graph - directed graph with vertices representing points in P (point set)
- For each point p in P, we connect p to points that are “nearby”
- Additionally we connect p to a “small” number of points that are “far” away

Greedy (Beam) Search

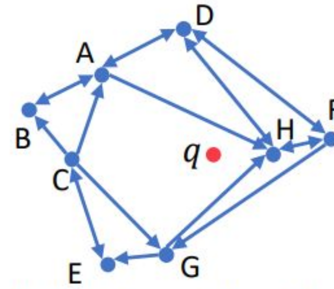
- Used by most ANNS graph algorithms
- Given a query point q, such a search maintains a list (referred to as a beam) of some bounded size which represents a set of nearest neighbor candidates of q
- Initially the beam contains a given source point s. In each step, the algorithm pops the closest point to q from the beam and processes it by adding all its out neighbors to the beam
- The algorithm also keeps track of all points that have been processed in the form of a visited set

Algorithm and Example

Algorithm 1: greedySearch(p, s, L, k).

Input: Point q, starting point s, beam width L, integer k.
Output: Set $\mathcal{V}$ of visited points and set $\mathcal{K}$ of k-nearest neighbors to point q.

1. $\mathcal{V} \leftarrow \emptyset$
2. $\mathcal{L} \leftarrow \{s\}$
3. **while** $\mathcal{L} \setminus \mathcal{V} \neq \emptyset$ **do**
4. $\quad p^* \leftarrow \arg\min_{(p \in \mathcal{L} \setminus \mathcal{V})} \|p, q\|$
5. $\quad \mathcal{V} \leftarrow \mathcal{V} \cup \{p^*\}$
6. $\quad \mathcal{L} \leftarrow \mathcal{L} \cup N_{\text{out}}(p^*)$
7. $\quad$ **if** $|\mathcal{L}| > L$ **then** retain only L closest points to q in $\mathcal{L}$
8. $\mathcal{K} \leftarrow k$ closest points to q in $\mathcal{V}$
9. **return** $\mathcal{V}, \mathcal{K}$



Starting from A, $L = 3$

q ● query point

A Points in $\mathcal{V}$ (set of processed vertices)

A A Points in $\mathcal{L}$ (the beam set)

A The processed point $p^* \in \mathcal{L}$, i.e., the closest point to q in $\mathcal{L} \setminus \mathcal{V}$.

A Removed from $\mathcal{L}$ because $|\mathcal{L}| > L$. Only the L closest points in $\mathcal{L}$ are kept.

Finish here: all points in $\mathcal{L}$ are in $\mathcal{V}$. Nearest neighbor found is **H**.

$\mathcal{L}$: A, $\mathcal{V}$: $\emptyset$ $\xrightarrow{\text{+BDH}}$ $\mathcal{L}$: H A D B, $\mathcal{V}$: A $\xrightarrow{\text{+FD}}$ $\mathcal{L}$: H F A D, $\mathcal{V}$: A H $\xrightarrow{\text{+DHG}}$ $\mathcal{L}$: H F A G D, $\mathcal{V}$: A H F

Better Example

https://pynndescent.readthedocs.io/en/latest/how_pynndescent_works.html

Existing Techniques

- Incremental Algorithms - DiskANN and HNSW
- Clustering-Based Algorithms - HCNNG and PyNNDescent

Incremental Algorithms

- Work by inserting points (in some order) into the graph
- To insert point p we query the existing graph using a greedy search
- The visited set of this search is then “pruned” and edges are added from/to p to/from each point in the pruned visited set
- The pruning routine attempts to select a subset of neighbors for p that cover a diverse range of edge lengths and directions
- Pruning also ensures that the size of p’s neighborhood has at most a given degree bound
- A smaller degree bound typically results in faster but less accurate searches compared to a larger bound

prune

- Pruning procedure used in DiskANN
- If V is {P} / p and R is n-1 then using this procedure to produce the out-neighbors of every point p, ensures that the distance to any query decreases by a multiplicative factor of α > 1 at every node along the search path
- Essentially ensures logarithmic query time at the cost of quadratic construction time
- DiskANN uses greedy searching to carefully choose V

Algorithm 2: RobustPrune$(p, \mathcal{V}, \alpha, R)$

Data: Graph G, point $p \in P$, candidate set $\mathcal{V}$, distance threshold $\alpha \geq 1$, degree bound R

Result: G is modified by setting at most R new out-neighbors for p

begin

- $\mathcal{V} \leftarrow (\mathcal{V} \cup N_{\text{out}}(p)) \setminus \{p\}$
- $N_{\text{out}}(p) \leftarrow \emptyset$
- **while** $\mathcal{V} \neq \emptyset$ **do**
  - $p^* \leftarrow \arg\min_{p' \in \mathcal{V}} d(p, p')$
  - $N_{\text{out}}(p) \leftarrow N_{\text{out}}(p) \cup \{p^*\}$
  - **if** $|N_{\text{out}}(p)| = R$ **then**
    - `break`
  - **for** $p' \in \mathcal{V}$ **do**
    - **if** $\alpha \cdot d(p^*, p') \leq d(p, p')$ **then**
      - `remove` p' `from` $\mathcal{V}$

insert

Algorithm 2: insert(p, s, R, L).

Input: Point p, starting point s, beam width L, degree bound R.

Output: Point p is inserted into the nearest neighbor graph.

1 $\mathcal{V}, \mathcal{K} \leftarrow \text{greedySearch}(p, s, L, 1)$

2 $N_{\text{out}}(p) \leftarrow \text{prune}(p, \mathcal{V}, R)$

3 **for** $q \in N_{out}(p)$ **do**

4     $N_{\text{out}}(q) \leftarrow N_{\text{out}}(q) \cup \{p\}$

5   **if** $|N_{out}(q)| > R$ **then** $N_{\text{out}}(q) \leftarrow \text{prune}(q, N_{\text{out}}(q), R)$

Challenges for Incremental Algorithms

- Existing parallel implementations of incremental ANN algorithms insert all points in a single parallel loop over all points
- Such implementations need to use locks to sequentialize conflicts as the graph is initially empty
- This results in performance issues and non-determinism

Prefix Doubling

- Insert points in batches of exponentially increasing size
- Each point will add itself based on the snapshot at the end of the last batch and will therefore not conflict with other points in the same batch
- Provides a balance between parallelism (large batches can utilize a large number of threads), progress (no conflicts within batches), and accuracy (each point sees a “reasonably” accurate snapshot of the graph)

Batch Insertion and Pruning

- Using our prefix doubling scheme we insert points in batches
- In lines 7-9, all points in the batch construct their own neighborhood independently on an immutable snapshot (thus no locks needed)
- Lines 11-14 make each p visible to the existing graph by reversing added edges
- We use parallel semi-sort to avoid having to use locks

Algorithm 3: batchBuild($\mathcal{P}, s, R, L$).

Input: Point set $\mathcal{P}$, starting point s, beam width L, degree bound R.

Output: An ANN graph consisting of all points in $\mathcal{P}$.

```
1  start ← 1
2  while start ≤ |P| do                                   // Prefix-doubling
3     end ← min(start × 2, start + θ, |P|)   // θ: batch size upper bound
4     BatchInsert(P[start..end])
5     start ← end + 1
6  Function BatchInsert(P')   // Insert a batch P' to the current index
7     parallel for p ∈ P' do
8        V, K ← greedySearch(p, s, L, 1)
9        N_out(p) ← prune(p, V, R)
10    B ← ⋃_{p∈P'} N_out(p)                    // All (existing) affected points
11    parallel for b ∈ B do
         // N: all points in P' that added b as their neighbors
12       N ← {p | p ∈ P' ∧ b ∈ N_out(p)}
13       N_out(b) ← N_out(b) ∪ N
14       if |N_out(b)| > R then N_out(b) ← prune(b, N_out(b), R)
```

Batch Size Truncation

- For large batch sizes graph accuracy can suffer
- To account for this we upper bound the batch size by some theta (empirically set to 0.02n)
- In practice this relaxation doesn't affect scalability or parallelism since 2% of the input is enough to utilize all threads

Clustering-Based Algorithms

- These algorithms construct clustering trees
- The algorithm splits the input into two pieces and keeps recursively splitting until the number of points drops below a given threshold, reaching a “leaf” cluster
- The recursive structure of the splitting produces the cluster tree
- Within each leaf cluster, a local ANN graph with stronger conditions is build
- We use different random seeds to generate different cluster trees and hence multiple local ANN graphs
- The final ANN graph is taken as the union of these local ANN graphs (modulo some additional post-processing)

Challenges for Clustering-Based Algorithms

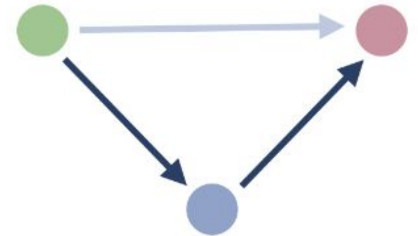
- Lock based merging of local ANN graphs
- Local ANN graph construction generates costly intermediate structures (maintaining all the local ANN graphs is costly in terms of space and time)

Parallelizing Clustering-Based Algorithms

- Construct multiple cluster trees in parallel
- Parallelize the construction of each tree using a parallel divide and conquer combined with a parallel partitioning primitive to assign points to different branches in parallel
- To avoid per-point locks when combining local ANN graphs use parallel semi-sort

DiskANN

- Essentially Algorithm 2 using Robust Pruning
- This can be thought of as streamlining navigation by pruning out long edges of triangles
- We can optimize DiskANN by using prefix doubling



PyNNDescent

- Construct an initial ANN graph where each point is connected to k “random” other points
- This is achieved using random projection trees
- The local ANN graphs connects each point to the exact k nearest neighbors within each leaf
- In an interactive manner we refine the initial graph by first undirecting the graph
- Then for for each point p compute its two hop neighborhood and retain the k closest candidates
- https://pynndescent.readthedocs.io/en/latest/how_pynndescent_works.html
- Optimize using random edge sampling and batch computation for hop neighborhoods
- Despite optimizations the paper was unable to scale PyNNDescent to work on billion scale data sets
- The main issue is the computation of each points two hop neighborhood

Results

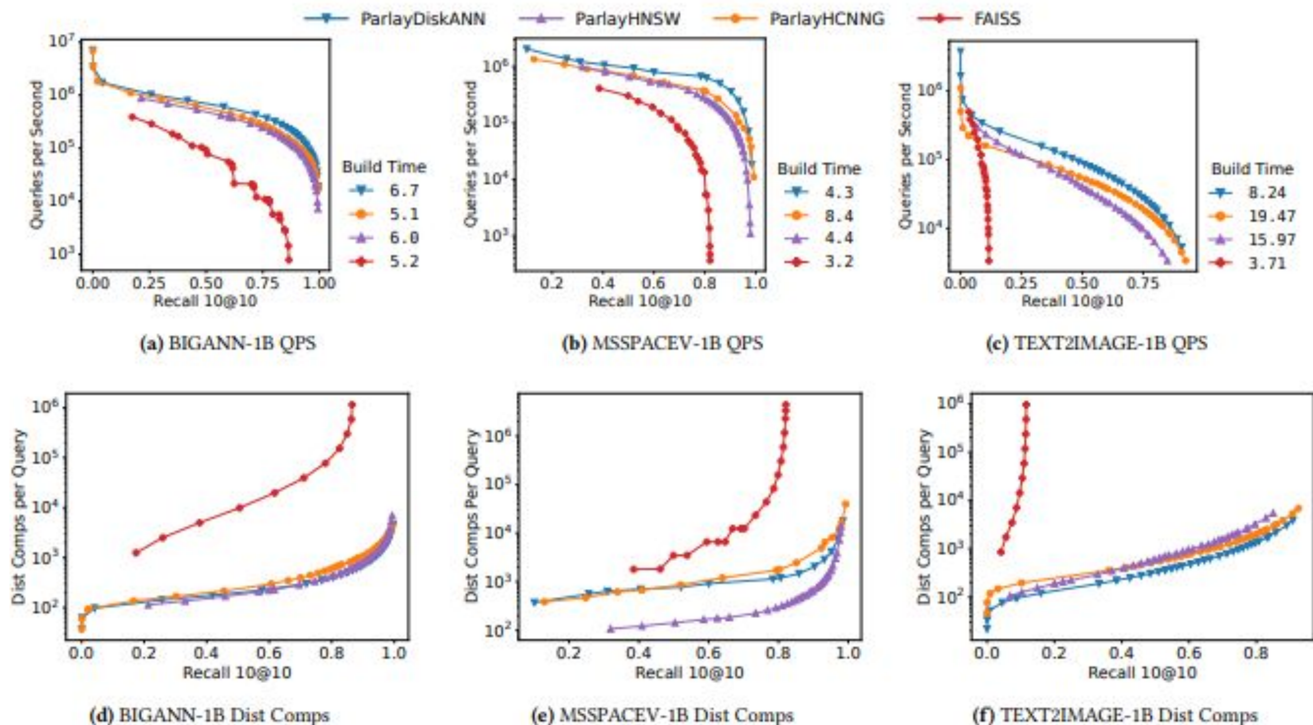


Figure 3. Build time (hours), QPS, recall, and distance comparisons for all algorithms on billion-size datasets.

Results - Continued

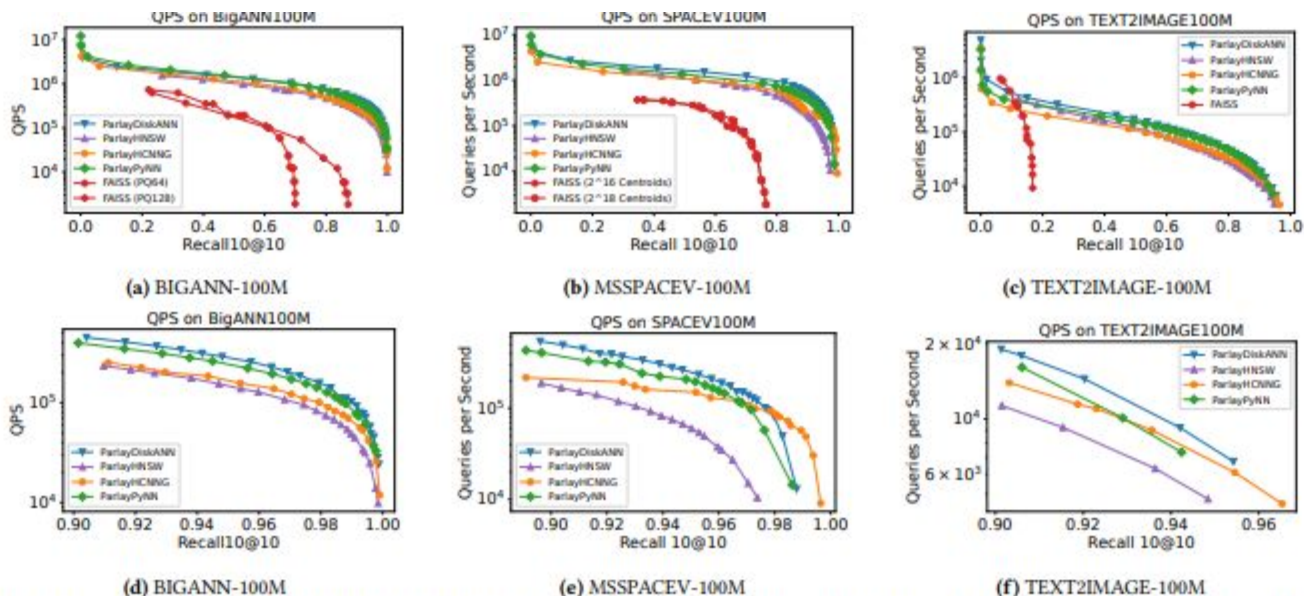


Figure 4. QPS-recall curves on all 100-million size datasets. The first row shows the overall QPS/recall curve, while the second row zooms into a higher-recall regime. The build times are given in Tab. 1