ANN Retrieval with Random Hyperplane LSH

A Mini-Benchmark on an IntelCore i5-9600K CPU with sequential and parallel query paths

Mondia Wu
University of Florence
mondia.wu@edu.unifi.it

Abstract

Approximate Nearest Neighbor (ANN) search is a core primitive in large-scale vision, recommendation, and robotics systems. We revisit the classic random hyperplane *variant of* ***Locality-Sensitive Hashing*** *(LSH) and instrument a compact* `C++17` *implementation that offers both sequential and OpenMP-parallel query paths. Using the SIFT1M dataset, we benchmark the code on an IntelCore i5-9600K (6 cores, 6 threads).*

1. Introduction

1.1. Approximate Nearest-Neighbor (ANN) Retrieval Problem Definition

Let $\mathcal{B} = \{\mathbf{x}_i \in \mathbb{R}^d\}_{i=1}^N$ be a database of high-dimensional feature vectors and $\mathbf{q} \in \mathbb{R}^d$ a query vector. The *k-nearest-neighbour* (kNN) problem asks for

$$\mathcal{N}_k(\mathbf{q}) = \underset{\substack{\mathcal{S}\subset\mathcal{B} \\ |\mathcal{S}|=k}}{\arg\min} \sum_{\mathbf{x}\in\mathcal{S}} d(\mathbf{q}, \mathbf{x}),$$

where $d(\cdot,\cdot)$ is usually the Euclidean or cosine distance. Brute-force search costs $\mathcal{O}(Nd)$ operations per query, which becomes prohibitive once N or d reaches millions.

Approximate-nearest-neighbour (ANN) algorithms relax the guarantee: they return a set whose members are within a factor $(1+\varepsilon)$ of the true distance with high probability [2]. In practice this reduces query time to sub-linear in N while keeping recall $\gtrsim 90\%$.

1.2. Application Domains

ANN search underpins a wide spectrum of modern systems:

- **Computer vision**: large-scale image retrieval, loop-closure detection in SLAM, copy-move forgery spotting.
- **Recommendation**: matching user/item embeddings in collaborative-filtering pipelines.
- **Natural language processing**: sentence-embedding and semantic search engines.
- **Audio & video**: fingerprinting, near-duplicate detection.
- **Bioinformatics**: protein or DNA sequence embedding search.
- **Robotics**: real-time location recognition and place recognition.

1.3. Locality-Sensitive Hashing (LSH)

Locality-Sensitive Hashing is a family of techniques that map vectors to *hash keys* such that

$$\Pr[h(\mathbf{x}) = h(\mathbf{y})] \quad \text{is higher when} \quad d(\mathbf{x}, \mathbf{y}) \text{ is small.}$$

The random-hyperplane variant [?] draws $\mathbf{a} \sim \mathcal{N}(\mathbf{0}, I_d)$ and emits one bit

$$h_{\mathbf{a}}(\mathbf{x}) = \operatorname{sign}(\mathbf{a}^\top \mathbf{x}),$$

which equals 1 if $\mathbf{x}$ lies on the positive side of the hyperplane defined by $\mathbf{a}$ and 0 otherwise. Concatenating k such bits yields a k-bit key with 2^k buckets; repeating the procedure across L independently sampled tables increases recall.

1.4. Why LSH

1. **Sub-linear query cost**: only vectors in buckets matching the query's key are re-ranked exactly, typically a few hundred out of millions.
2. **Simplicity and portability**: dot-products and bit manipulations—no complex data structures or training.
3. **Provable guarantees**: collision probabilities can be tuned analytically via (L, k) to meet a target recall R.
4. **Parallel-friendly**: once built, hash tables are read-only; queries distribute trivially across CPU cores or GPUs.
5. **Metric flexibility**: variants exist for ℓ_p norms, angular/cosine distance, and Jaccard similarity.

Together, these properties make LSH a robust baseline and a practical choice for medium-scale ANN tasks on commodity hardware.

2. What did we do

1. a simple code based on LSH theory,
2. provide a C++17 implementation with a sequential and an OpenMP query path of the problem, with $d = 128$,
3. benchmark on our desktop hardware running an Intel-Core i5-9600K CPU and analyse scaling behaviour.

2.1. What is OpenMP

OpenMP (Open Multi-Processing) is an *application-programming interface* that adds compile-time `#pragma` directives, run-time library calls, and environment variables to C, C++, and Fortran in order to express *shared-memory parallelism* [3].

3. Implementation Overview

3.1. Phase 1: Construction

A fixed-seed `mt19937` produces $\mathcal{N}(0,1)$ coefficients filling `hyperplanes_[L][k][dim]` (one Gaussian normal per hash bit). No data vectors are touched, so this phase is $\mathcal{O}(Lkd)$ and runs once.

3.2. Phase 2: Index Build

The algorithm maps every base vector into each table; the resulting bucket lists hold *IDs only*, keeping RAM low.

```
// ------------- build phase -------------
for (int id = 0; id < (int)base.size(); ++id)
    for (int t = 0; t < L_; ++t)
        tbl_[t][hash_vec(base[id],t)].push_back(id);
```

Figure 1. Index build ($\mathcal{O}(NLkd)$).

3.3. Phase 3: Query

Listing 2 shows signature generation; together with the heap-based re-ranking loop it forms the end-to-end query path.

```
// ---------- k-bit random-hyperplane hash ----------
uint64_t hash_vec(const Vec& v, int t) const {
    uint64_t h = 0;
    for (int j = 0; j < k_; ++j)
        if (dot(v, hyperplanes_[t][j]) > 0)
            h |= 1ULL << j;      // set bit j
    return h;
}
```

Figure 2. Signature computation (k dot products).

3.4. Sequential and Parallel Driver Code

The main program simply loops over all queries in either a for-loop (sequential) or a `#pragma omp` region (parallel).

```
// ----------- sequential loop -----------
for (size_t i = 0; i < queries.size(); ++i)
    ans_seq[i] = index.query(queries[i], topK);
```

Figure 3. Single-threaded evaluation.

```
// ----------- OpenMP parallel -----------
omp_set_num_threads(nthr);
#pragma omp parallel for schedule(static)
for (size_t i = 0; i < queries.size(); ++i)
    ans_par[i] = index.query(queries[i], topK);
```

Figure 4. Thread-parallel evaluation.

Data structures.

- **Hyperplanes** `hyperplanes_[L][k][dim]`: Gaussian random.
- **Hash tables** `tbl_[L]`: `unordered_map` from the k-bit key (`uint64_t`) to a vector of record IDs.
- **Base data** kept unchanged; the index stores pointers only.

Sequential vs. parallel. After `build()` the index is read-only; each thread runs the same `query()` function on its own query vector without locks.

4. Experimental Setup

Dataset. N = 1,000,000 base and Q = 10,000 query vectors drawn from *SIFT1M* [**?**].
Parameters. $L = 12$ tables, $k = 16$ bits/table, $k_{NN} = 10$ neighbours.
Hardware. IntelCore i5-9600K (6C/6T, 3.7 GHz), 16 GB DDR4-2400; Windows 10, `g++ 15.1.0`, `OpenMP 5.0`.
Metric. Wall-clock time via `std::chrono`. Reported the mean value of 10 runs.

5. Results

Mode	/time(ms	Speedup	Parallel Efficiency
Sequential	176062		
Parallel (2 threads)	96212	1.83x	91,5%
Parallel (4 threads)	54465	3.23x	80,75%
Parallel (6 threads)	42513	4.14x	69%
Parallel (8 threads)	45878	3.84x	48%
Parallel (12 threads)	43472	4.05x	33,75 %
Parallel (18 threads)	44829	3.93x	21,83%

Table 1. LSH timing on the i5-9600K for 10k queries.

5.1. Runtime Observations

Table 1 summarises wall–clock time for $Q = 10\,000$ queries under varying thread counts.

1. **Near-linear scaling up to 6 threads.** Moving from 1 to 6 threads reduces latency from 176s to 42.5s, a $4.14\times$ speed-up and 69% parallel efficiency.
2. **Diminishing returns beyond the core count.** With 8–18 threads the run time *increases*; the oversubscription forces context switches, incurs scheduler overhead, and adds contention on shared caches and memory bandwidth.
3. **Practical guideline.** For this hardware, setting `OMP_NUM_THREADS=6` maximises throughput; larger values hurt performance while offering no recall benefit.

6. Conclusion

We implemented a `C++17` program that realises random-hyperplane Locality-Sensitive Hashing for 128-D vectors and exposes both sequential and OpenMP-parallel query paths. The code answers $10\,000$ queries in a $1\,000\,000$ dataset in 176s on a single core of an IntelCore i5-9600K and drops to 42,5s at six threads, a $4.1\times$ speed-up that saturates the physical core count. Additional threads beyond six degrade performance, confirming that the workload is bound by memory bandwidth once all cores are busy.

References

[1] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. *VLDB*, 1999.

[2] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor. *FOCS*, 2008.

[3] OpenMP Architecture Review Board. OpenMP application programming interface v5.0. `https://www.openmp.org`, 2018.

[4] LSH example code. `https://github.com/gamboviol/lsh?tab=readme-ov-file`, 2018.